Fiche pratique sur la prévention des injections SQL

Introduction

Cette fiche pratique vous aidera à prévenir les failles d'injection SQL dans vos applications. Elle définit ce qu'est l'injection SQL, explique où ces failles se produisent et propose quatre options pour se défendre contre les attaques par injection SQL. Les attaques par injection SQL sont courantes pour les raisons suivantes :

1. les vulnérabilités liées à l'injection SQL sont très courantes, et
2. La base de données de l'application est une cible fréquente pour les attaquants, car elle contient généralement des données intéressantes/critiques.

Qu'est-ce qu'une attaque par injection SQL ?

Les pirates peuvent utiliser l'injection SQL sur une application si celle-ci comporte des requêtes dynamiques de base de données qui utilisent la concaténation de chaînes et des entrées fournies par l'utilisateur. Pour éviter les failles d'injection SQL, les développeurs doivent :

1. Cessez d'écrire des requêtes dynamiques avec concaténation de chaînes ou
2. Empêchez l'inclusion d'entrées SQL malveillantes dans les requêtes exécutées.

Il existe des techniques simples pour prévenir les vulnérabilités liées aux injections SQL, qui peuvent être utilisées avec pratiquement tous les langages de programmation et tous les types de bases de données. Bien que les bases de données XML puissent présenter des problèmes similaires (par exemple, les injections XPath et XQuery), ces techniques peuvent également être utilisées pour les protéger.

Anatomie d'une vulnérabilité typique à l'injection SQL

Une faille d'injection SQL courante en Java est décrite ci-dessous. Comme son paramètre « customerName » non validé est simplement ajouté à la requête, un pirate peut entrer du code SQL dans cette requête et l'application prendrait le code du pirate et l'exécuterait sur la base de données.

```
String query = « SELECT account_balance FROM user_data WHERE user_name = »
             + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... ); ResultSet
    results = statement.executeQuery( query );
}

...
```

Défenses principales

- **Option 1 : utilisation de déclarations préparées (avec requêtes paramétrées)**
- **Option 2 : utilisation de procédures stockées correctement construites**
- **Option 3 : Validation des entrées par liste blanche**
- **Option 4 : FORTEMENT DÉCONSEILLÉ : échapper toutes les entrées fournies par l'utilisateur**

Option de défense n° 1 : déclarations préparées (avec requêtes paramétrées)

Lorsque les développeurs apprennent à écrire des requêtes de base de données, ils doivent être informés qu'ils doivent utiliser des instructions préparées avec liaison de variables (également appelées requêtes paramétrées). Les instructions préparées sont simples à écrire et plus faciles à comprendre que les requêtes dynamiques, et les requêtes paramétrées obligent le développeur à définir d'abord tout le code SQL, puis à passer chaque paramètre à la requête ultérieurement.

Si les requêtes de base de données utilisent ce style de codage, la base de données fera toujours la distinction entre le code et les données, quelle que soit la saisie de l'utilisateur. De plus, les instructions préparées garantissent qu'un pirate ne peut pas modifier l'intention d'une requête, même si des commandes SQL sont insérées par un pirate.

Exemple de déclaration préparée Java sécurisée

Dans l'exemple Java sécurisé ci-dessous, si un pirate saisissait l'identifiant utilisateur « `tom » ou « 1 » = « 1` », la requête paramétrée rechercherait un nom d'utilisateur correspondant littéralement à la chaîne e`tom' ou « 1 » = « 1` ». Ainsi, la base de données serait protégée contre les injections de code SQL malveillant.

L'exemple de code suivant utilise `PreparedStatement` , l'implémentation Java d'une requête paramétrée, pour exécuter la même requête de base de données.

```
// Cela devrait VRAIMENT être validé aussi
String custname = request.getParameter("customerName");
// Effectuer une validation des entrées pour détecter les attaques
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query ); pstmt.setString( 1,
custname);
ResultSet résultats = pstmt.executeQuery( );
```

Exemple de déclaration préparée sécurisée C# .NET

Dans .NET, la création et l'exécution de la requête ne changent pas. Il suffit de passer les paramètres à requête à l'aide de la méthode `Parameters.Add()` comme indiqué ci-dessous.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?"; try {
  OleDbCommand command = new OleDbCommand(query, connection); command.Parameters.Add(new
  OleDbParameter("customerName", CustomerName
Name.Text));
  OleDbDataReader reader = command.ExecuteReader();
```

```
    // …
} catch (OleDbException se) {
    // gestion des erreurs
}
```

Bien que nous ayons présenté des exemples en Java et .NET, pratiquement tous les autres langages (y compris Cold Fusion et Classic ASP) prennent en charge les interfaces de requêtes paramétrées. Même les couches d'abstraction SQL, telles que le langage de requête Hibernate (HQL) qui présente le même type de problèmes d'injection (appelés injection HQL), prennent également en charge les requêtes paramétrées :

Hibernate Query Language (HQL) Prepared Statement (paramètres nommés) Exemple

```
// Il s'agit d'une instruction HQL non sécurisée
Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+use
// Voici une version sécurisée de la même requête utilisant des paramètres nommés
Query safeHQLQuery = session.createQuery("from Inventory where productID=:product
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Autres exemples de requêtes préparées sécurisées

Si vous avez besoin d'exemples de requêtes préparées/langages paramétrés, notamment Ruby, PHP, Cold Fusion, Perl et Rust, consultez la fiche de référence sur la paramétrisation des requêtes ou ce site.

En général, les développeurs apprécient les instructions préparées, car tout le code SQL reste dans l'application, ce qui rend les applications relativement indépendantes de la base de données.

Option de défense n° 2 : procédures stockées

Bien que les procédures stockées ne soient pas toujours à l'abri des injections SQL, les développeurs peuvent utiliser certaines constructions de programmation de procédures stockées standard. Cette approche a le même effet que l'utilisation de requêtes paramétrées, à condition que les procédures stockées soient mises en œuvre de manière sécurisée (ce qui est la norme pour la plupart des langages de procédures stockées).

Approche sécurisée des procédures stockées

Si des procédures stockées sont nécessaires, l'approche la plus sûre pour les utiliser consiste à demander au développeur de créer des instructions SQL avec des paramètres qui sont automatiquement paramétrés, à moins que le développeur ne fasse quelque chose qui s'écarte largement de la norme. La différence entre les instructions préparées et les procédures stockées réside dans le fait que le code SQL d'une procédure stockée est défini et stocké dans la base de données elle-même, puis appelé à partir de l'application. Étant donné que les instructions préparées et les procédures stockées sécurisées sont tout aussi efficaces pour prévenir les injections SQL, votre organisation doit choisir l'approche qui lui convient le mieux.

Quand les procédures stockées peuvent augmenter les risques

Parfois, les procédures stockées peuvent augmenter les risques lorsqu'un système est attaqué. Par exemple, sur MS SQL Server, vous disposez de trois rôles principaux par défaut : `db_datareader` `db_datawriter` et

`db_owner` . Avant l'utilisation des procédures stockées, les administrateurs de bases de `db_datareader` `db_datawriter` données accordaient

o

u des droits à l'utilisateur du service web, en fonction des besoins.

Cependant, les procédures stockées nécessitent des droits d'exécution, un rôle qui n'est pas disponible par défaut. Dans certaines configurations où la gestion des utilisateurs a été centralisée, mais est limitée à ces 3 rôles, les applications web
devrait fonctionner comme `db_owner` pour que les procédures stockées puissent fonctionner. Naturellement, cela signifie que si
un serveur est piraté, l'attaquant dispose de tous les droits sur la base de données, alors qu'auparavant, il ne disposait peut-être que d'un accès en lecture.

Exemple de procédure stockée Java sécurisée

L'exemple de code suivant utilise l'implémentation Java de l'interface de procédure stockée (`CallableStatement`) pour exécuter la même requête de base de donnée `sp_getAccountBalance` doit être prédéfinie dans la base de données et utiliser la même fonctionnalité que la
requête ci-dessus.

```
// Cela devrait VRAIMENT être validé
String custname = request.getParameter("customerName"); try {
  CallableStatement cs = connection.prepareCall("{call
sp_getAccountBalance(?)}"); cs.setString(1,
  custname);
  ResultSet results = cs.executeQuery();
  // … gestion du jeu de résultats
} catch (SQLException se) {
  // … journalisation et gestion des erreurs
}
```

Exemple de procédure stockée VB .NET sécurisée

L'exemple de code suivant utilise une `commande SqlCommand` , implémentation .NET de l'interface de procédure stockée, pour exécuter la même requête de base de données. La `sp_getAccountBalance` stored doit être prédéfinie dans la base de données et utiliser la même fonctionnalité que la requête définie ci-dessus.

```
 Essayez
   Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
   command.CommandType = CommandType.StoredProcedure
   command.Parameters.Add(new SqlParameter("@CustomerName",
CustomerName.Text))
   Dim reader As SqlDataReader = command.ExecuteReader() '...
 Catch se As SqlException 'gestion
   des erreurs
 End Try
```

Option de défense n° 3 : validation des entrées par liste blanche

Si vous êtes confronté à des parties de requêtes SQL qui ne peuvent pas utiliser de variables liées, telles que les noms de tables, les noms de colonnes ou les indicateurs d'ordre de tri (ASC ou DESC), la validation des entrées ou la refonte des requêtes constituent la défense la plus appropriée. Lorsque des noms de tables ou de colonnes sont nécessaires, l'idéal est que ces valeurs proviennent du code et non des paramètres utilisateur.

Exemple de validation sécurisée du nom d'une table

AVERTISSEMENT : l'utilisation de valeurs de paramètres utilisateur pour cibler des noms de tables ou de colonnes est le signe d'une conception médiocre et une réécriture complète doit être envisagée si le temps le permet. Si cela n'est pas possible, les développeurs doivent mapper les valeurs des paramètres aux noms de tables ou de colonnes légaux/attendus afin de s'assurer que les entrées utilisateur non validées ne se retrouvent pas dans la requête.

Dans l'exemple ci-dessous, étant donné que `tableName` est identifié comme l'une des valeurs légales et attendues pour un nom de table dans cette requête, il peut être directement ajouté à la requête SQL. Gardez à l'esprit que les fonctions génériques de validation des tables peuvent entraîner une perte de données si les noms de tables sont utilisés dans des requêtes où ils ne sont pas attendus.

```
Chaîne tableName ;
switch(PARAM) :
   case « Valeur1 » : nomTable = « fooTable »
                      ;
                   break;
   cas « Valeur2 » : nomTable = « barTable » ;
                  break ;
   ...

 par défaut : lancer une nouvelle exception InputValidationException (« valeur inattendue fournie »
                                                                    + « pour le nom de la table »)
                                                                    ;
```

Utilisation la plus sûre de la génération dynamique de SQL (DÉCONSEILLÉE)

Lorsque nous disons qu'une procédure stockée est « implémentée de manière sécurisée », cela signifie qu'elle ne comprend aucune génération SQL dynamique non sécurisée. Les développeurs ne génèrent généralement pas de SQL dynamique à l'intérieur des procédures stockées. Cependant, cela est possible, mais doit être évité.

Si cela ne peut être évité, la procédure stockée doit utiliser la validation des entrées ou un échappement approprié, comme décrit dans cet article, afin de s'assurer que toutes les entrées fournies par l'utilisateur à la procédure stockée ne peuvent pas être utilisées pour injecter du code SQL dans la requête générée dynamiquement. Les auditeurs doivent toujours chercher à
pour les utilisations de `exécuter` ou `exec` dans les procédures stockées SQL Server. Des directives d'audit similaires
`sp_execute` ,
sont nécessaires pour des fonctions similaires d'autres fournisseurs. **Exemple de**

génération de requêtes dynamiques plus sûres (DÉCONSEILLÉ)

Pour quelque chose d'aussi simple qu'un ordre de tri, il est préférable que les données fournies par l'utilisateur soient converties en valeur booléenne, puis que cette valeur booléenne soit utilisée pour sélectionner la valeur sûre à ajouter à la requête. Il s'agit d'un besoin très courant dans la création de requêtes dynamiques.

Par exemple :

```
public String someMethod(boolean sortOrder) {
 String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC")
 ...
```

Chaque fois que les données saisies par l'utilisateur peuvent être converties en un type autre que String, tel que date, nombre, booléen, énuméré, etc. avant d'être ajoutées à une requête ou utilisées pour sélectionner une valeur à ajouter à la requête, cela garantit la sécurité de l'opération.

La validation des entrées est également recommandée comme défense secondaire dans TOUS les cas, même lorsque vous utilisez des variables liées, comme indiqué précédemment dans cet article. D'autres techniques permettant de mettre en œuvre une validation rigoureuse des entrées sont décrites dans la fiche de référence sur la validation des entrées.

Option de défense 4 : FORTEMENT DÉCONSEILLÉE : échapper toutes les entrées fournies par l'utilisateur

Dans cette approche, le développeur échappera toutes les entrées utilisateur avant de les intégrer dans une requête. Sa mise en œuvre est très spécifique à la base de données. Cette méthodologie est fragile par rapport à d'autres défenses, et nous NE POUVONS PAS garantir que cette option empêchera toutes les injections SQL dans toutes les situations.

Si une application est créée à partir de zéro ou nécessite une faible tolérance au risque, elle doit être créée ou réécrite à l'aide de requêtes paramétrées, de procédures stockées ou d'un type de mappeur relationnel d'objets (ORM) qui crée vos requêtes à votre place.

Défenses supplémentaires

Au-delà de l'adoption de l'une des quatre défenses principales, nous recommandons également d'adopter toutes ces défenses supplémentaires afin d'assurer une défense en profondeur. Ces défenses supplémentaires sont les suivantes :

- **Privilège minimal**
- **Liste blanche Validation des entrées**

Privilège minimal

Pour minimiser les dommages potentiels d'une attaque par injection SQL réussie, vous devez réduire au minimum les privilèges attribués à chaque compte de base de données dans votre environnement. Commencez par déterminer les droits d'accès dont vos comptes d'application ont besoin, plutôt que d'essayer de déterminer les droits d'accès que vous devez supprimer.

Assurez-vous que les comptes qui n'ont besoin que d'un accès en lecture ne se voient accorder qu'un accès en lecture aux tables auxquelles ils doivent accéder. N'ATTRIBUEZ PAS D'ACCÈS DE TYPE DBA OU ADMIN À VOS COMPTES D'APPLICATION
. Nous comprenons que cela est facile et que tout « fonctionne » lorsque vous procédez ainsi, mais c'est très dangereux.

Réduire au minimum les privilèges des applications et du système d'exploitation

L'injection SQL n'est pas la seule menace qui pèse sur les données de votre base. Les pirates peuvent simplement modifier les valeurs des paramètres parmi celles qui leur sont proposées, pour leur attribuer une valeur non autorisée, mais à laquelle l'application elle-même pourrait avoir accès. Ainsi, en réduisant au minimum les privilèges accordés à votre application, vous diminuerez le risque de tentatives d'accès non autorisées, même lorsqu'un pirate n'utilise pas l'injection SQL dans le cadre de son exploitation.

Pendant que vous y êtes, vous devriez réduire au minimum les privilèges du compte du système d'exploitation sous lequel le SGBD fonctionne. N'exécutez pas votre SGBD en tant que root ou système ! La plupart des SGBD fonctionnent dès leur installation avec un compte système très puissant. Par exemple, MySQL fonctionne par défaut en tant que système sous Windows ! Modifiez le compte OS du SGBD pour lui attribuer des privilèges plus appropriés et restreints.

Détails sur le principe du moindre privilège lors du développement

Si un compte n'a besoin d'accéder qu'à certaines parties d'une table, envisagez de créer une vue qui limite l'accès à cette partie des données et attribuez au compte l'accès à la vue plutôt qu'à la table sous-jacente. N'accordez que très rarement, voire jamais, l'accès à la création ou à la suppression de comptes de base de données.

Si vous adoptez une politique consistant à utiliser des procédures stockées partout et à ne pas autoriser les comptes d'application à exécuter directement leurs propres requêtes, limitez alors ces comptes à l'exécution des procédures stockées dont ils ont besoin. Ne leur accordez aucun droit direct sur les tables de la base de données.

Privilèges d'administration minimaux pour plusieurs bases de données

Les concepteurs d'applications web doivent éviter d'utiliser le même compte propriétaire/administrateur dans les applications web pour se connecter à la base de données. Différents utilisateurs de bases de données doivent être utilisés pour différentes applications web.

En général, chaque application Web distincte qui nécessite un accès à la base de données doit disposer d'un compte utilisateur de base de données désigné que l'application utilisera pour se connecter à la base de données. De cette façon, le concepteur de l'application peut bénéficier d'une bonne granularité dans le contrôle d'accès, réduisant ainsi autant que possible les privilèges. Chaque utilisateur de la base de données aura alors un accès sélectif uniquement à ce dont il a besoin et un accès en écriture si nécessaire.

À titre d'exemple, une page de connexion nécessite un accès en lecture aux champs nom d'utilisateur et mot de passe d'une table, mais aucun accès en écriture sous quelque forme que ce soit (ni insertion, ni mise à jour, ni suppression). Cependant, la page d'inscription nécessite bien sûr un privilège d'insertion dans cette table ; cette restriction ne peut être appliquée que si ces applications Web utilisent différents utilisateurs de base de données pour se connecter à la base de données.

Renforcer le principe du moindre privilège avec les vues SQL

Vous pouvez utiliser les vues SQL pour augmenter encore la granularité de l'accès en limitant l'accès en lecture à des champs spécifiques d'une table ou à des jointures de tables. Cela pourrait présenter des avantages supplémentaires.

Par exemple, si le système est tenu (peut-être en raison d'exigences légales spécifiques) de stocker les mots de passe des utilisateurs, au lieu de mots de passe hachés avec sel, le concepteur pourrait utiliser des vues pour compenser cette limitation. Il pourrait révoquer tout accès à la table (à tous les utilisateurs de la base de données à l'exception du propriétaire/administrateur) et créer une vue qui affiche le hachage du champ mot de passe et non le champ lui-même.

Toute attaque par injection SQL qui réussirait à voler des informations de la base de données se limiterait au vol du hachage des mots de passe (qui pourrait même être un hachage à clé), car aucun utilisateur de la base de données pour aucune des applications web n'a accès à la table elle-même.

Validation des entrées par liste blanche

En plus d'être une défense primaire lorsque rien d'autre n'est possible (par exemple, lorsqu'une variable de liaison n'est pas légale), la validation des entrées peut également être une défense secondaire utilisée pour détecter les entrées non autorisées avant qu'elles ne soient transmises à la requête SQL. Pour plus d'informations, veuillez consulter la fiche de référence sur la validation des entrées. Procédez avec prudence ici. Les données validées ne sont pas nécessairement sûres à insérer dans les requêtes SQL via la construction de chaînes.

Articles connexes

Fiches pratiques sur les attaques par injection SQL :

Les articles suivants décrivent comment exploiter différents types de vulnérabilités d'injection SQL sur diverses plateformes (que cet article a été créé pour vous aider à éviter) :

- Fiche pratique sur l'injection SQL
- Contourner les WAF avec SQLi - Injection SQL Contourner les WAF

Description des vulnérabilités d'injection SQL :

- Article de l'OWASP sur les vulnérabilités d'injection SQL
- Article de l'OWASP sur les vulnérabilités liées à l'injection SQL

aveugle **Comment éviter les vulnérabilités liées à l'injection**

SQL :

- Article du guide des développeurs OWASP sur la manière d'éviter les vulnérabilités d'injection SQL
- Fiche pratique OWASP qui fournit de nombreux exemples spécifiques à différents langages de requêtes paramétrées utilisant à la fois des instructions préparées et des procédures stockées
- Le site Bobby Tables (inspiré de la bande dessinée en ligne XKCD) propose de nombreux exemples dans différents langages de déclarations préparées et de procédures stockées paramétrées.

Comment examiner le code à la recherche de vulnérabilités liées aux injections SQL :

- Article du guide de révision du code OWASP sur la manière de vérifier le code à la recherche de vulnérabilités d'injection SQL

Comment tester les vulnérabilités aux injections SQL :

- Article du guide de test OWASP sur la manière de tester les vulnérabilités aux injections SQL