Document présentation

Description	Ce document est une présentation générale sur les attaques par injections SQL. Il fait partie du premier rendu (version 1.0) de l'UE MSSI. Il sera accompagné de deux rendus de présentations portant sur les attaques par failles XSS (version 2.0) et XSRF (version 3.0).

Document certification

	Name	*Fonction*
Group Authors	DJEBIEN Tarik RAKOTOBE Eric STIENNE Rudy	Étudiants Miage IPI NT
Decidor	JODRY Christophe	Enseignant MSSI

Documents Package Version history

Package Version	*Topic*	*Author*
1.0	Introduction et Attaque par injection SQL.	Djebien Tarik
2.0	Initialisation du document et Attaque par Faille XSS.	Sitraka Rakotobe Eric
3.0	Attaque par Faille XSRF et conclusion	Stienne Rudy

INTRODUCTION

Avec la multiplication et la diversification des usages du Web, notamment pour des sites marchands ou de façon plus générale pour des transactions financières, sont apparus de nouveaux types d'attaques qui en exploitent les faiblesses de conception.
D'autre part, des attaquants que l'on pourrait nommer les hackers informatiques exploitent dans les sites webs des zones que leurs auteurs croient avoir protégées ou effacées, nous allons donc présenter trois types d'attaques pour les sites web. Ce document présentera donc dans un premier temps les attaques par injection SQL.

INJECTION SQL

Introduction par l'exemple

L'attaque par injection SQL vise les sites Web qui proposent des transactions mal construites dont les résultats sont emmagasinés dans une base de données relationnelle. Elle consiste en ceci: SQL est un langage qui permet d'interroger et de mettre à jour une base de données relationnnelle; les requêtes sont soumises au moteur de la base en format texte, sans être compilées. Une requête typique est construite à partir de champs de formulaire remplis par l'internaute. Si l'auteur du site a été paresseux, il aura construit ses requêtes en insérant directement les textes rédigés par l'internaute, sans en contrôler la longueur ni le format ni le contenu. Ainsi, un utilisateur malveillant informé de cette faille (ou qui la soupçonnerait) peut confectionner un texte tel qu'une fois incorporé à une requête SQL il ait des effets indésirables sur la base de données, par exemple en y insérant directement des ordres du langage, de telle sorte qu'ils soient interprétés.
En voici un exemple, l'instruction suivante construit directement à partir du nom introduit par l'utilisateur une requête SQL innocente, qui extrait de la base des utilisateurs tous les enregistrements qui concernent celui-là en particulier :

*requete := "SELECT * FROM clients WHERE nom = '" + nom_client + "';"*

Soit un attaquant informé de cette faille qui, au lieu d'entrer dans le formulaire un nom valide, introduit la chaîne de caractères suivante :

*x'; DROP TABLE clients; SELECT * FROM secrets WHERE nom LIKE ' %*

La requête finale sera :

*SELECT * FROM clients WHERE nom = 'x'; DROP TABLE clients; SELECT * FROM secrets WHERE nom LIKE ' %';*

avec, comme résultat, la destruction pure et simple de la table **clients** et un accès imprévu à la table **secrets**, dont le nom suggère qu'elle n'est pas destinée à être lue par les internautes.

La parade à ce type d'attaque consiste essentiellement à écrire des programmes moins naïfs, qui vérifient les données introduites par les utilisateurs avant de les utiliser, et en particulier qui éliminent les caractères qui ont une valeur sémantique spéciale pour SQL. Cette recommandation vaut d'ailleurs pour tous les langages de programmation.

Fonctionnement des attaques par injection

Cette famille d'agressions repose sur un unique problème partagé par de nombreuses technologies de manière persistante : il n'existe aucune séparation stricte entre les instructions d'un programme et les données que saisit un utilisateur. Par conséquent, là où le développeur n'attendait que d'inoffensives données, des attaquants peuvent insidieusement placer des instructions enjoignant au programme de réaliser des actions de leur choix.

Pour réaliser une attaque par injection, il faut réussir à placer, dans des saisies classiques, des données interprétées comme des instructions. Le succès de l'opération repose sur trois éléments :

- Identifier la technologie sur laquelle repose l'application web. Les attaques par injection dépendent beaucoup du langage de programmation ou du matériel impliqués. Pour cela, on peut explorer la situation ou se contenter de tester les attaques par injection les plus communes. On peut deviner quelles technologies sont mises en oeuvre en inspectant les pieds de page, les textes en cas d'erreur, le code source HTML, et employer des outils comme Nessus, Nmap, THC-Amap ou d'autres encore.

- Établir la liste de toutes les saisies utilisateur possibles. Dans certains cas (formulaires HTML), elles seront évidentes. Toutefois, il existe bien d'autres manières d'interagir avec une application web : manipuler des saisies cachées dans les formulaires, modifier certains en-têtes HTTP (comme les cookies), voire des requêtes AJAX fonctionnant de manière invisible en arrière-plan. A peu près toutes les données intervenant dans les requêtes HTTP GET et POST peuvent être considérées comme des saisies utilisateur. Le recours à un mandataire (proxy) web tel que WebScarab, Paros ou Burp pourra aider à repérer toutes les saisies utilisateur dans une application web.

- Trouver la saisie utilisateur vulnérable. Cela peut sembler difficile, mais les pages d'erreur des applications trahissent parfois bien des secrets en la matière.

Rien de tel qu'un exemple pour exposer les attaques par injection. L'injection SQL ci-après illustre bien le sujet, tandis que les cas de figure qui la suivent détaillent la situation en présence de tel ou tel langage ou matériel précis.

Les risques liés à l'attaque par injection SQL

Popularité :	**8**
Simplicité :	**8**
Impacts :	**9**
Confidentialité	*9*
Disponibilité	*9*
Intégrité	*9*
Évaluation du risque :	**9**

Le spectre des possibilités ouvertes par l'injection SQL est vaste : contournement de l'authentification, récupération de codes de cartes de crédits ou prise de contrôle complète de la base de données sur un serveur distant.

SQL, le langage de requêtes structuré, standard de fait pour l'accès aux bases de données, sous-tend désormais la plupart des applications web nécessitant le stockage de données persistantes; il est donc probable qu'il anime aussi vos sites préférés. A l'instar de celle de bien de langages, la syntaxe de SQL mêle instructions de bases de données et saisies utilisateur. Si le développeur n'y prend garde, ces dernières pourront être interprétées comme des instructions et donner à un attaquant distant la possibilité de réaliser les opérations de son choix sur la base de données.

Imaginons ainsi une application web simple imposant une authentification. Un écran de connexion y demandera un identifiant et le mot de passe associé, transmis par l'utilisateur à travers une requête HTTP, suite à quoi le programme les confrontera à la liste des identifiants et mots de passe acceptables - il s'agit en général d'une table placée dans une base de données SQL.

Un développeur pourra créer cette table à l'aide de l'instruction SQL suivante :

```
CREATE TABLE user_table (
        id INTEGER PRIMARY KEY,
        username VARCHAR(32),
        password VARCHAR(41)
);
```

Ce code SQL produit une table de trois champs. Le premier (id) stocke un numéro d'identifiant référençant un utilisateur authentifié dans la base de données. Le deuxième (username) précise son identifiant, arbitrairement limité à 32 caractères au plus. Quant au dernier, password, il renfermera un hachage (hash) du mot de passe correspondant, car c'est une mauvaise idée de stocker ce genre d'informations en clair.

La fonction SQL **PASSWORD()** réalise cette opération. Sous le SGBD MySQL, elle produit un résultat de 41 caractères par exemple.

Pour authentifier un utilisateur, il suffit simplement de comparer les valeurs transmises à chacun des enregistrements de cette table. Si l'un correspond aux deux à la fois, l'opération est réussie et le système sait quel numéro d'identifiant attribuer au candidat.
Supposons que l'utilisateur ait proposé l'identifiant "MorduDeMssi15 " et le mot de passe monMotDePasse. On trouvera l'identifiant comme suit :

```
SELECT id FROM user_table WHERE username = 'MorduDeMssi15' AND password =
PASSWORD('monMotDePasse')
```

Si un tel utilisateur existe dans cette table de la base de données, cette commande SQL renverra le numéro d'identifiant associé. Dans le cas contraire, elle restera muette, signifiant par là que l'opération a échoué et n'a renvoyé aucun résultat.

Il semble donc aisé d'automatiser cette procédure. Prenons l'extrait de code Java que voici, lequel reçoit l'identifiant et le mot de passe d'un utilisateur qu'il tente ensuite d'authentifier à l'aide d'une requête SQL :

```
String username = req.getParameter("username");
String password = req.getParameter("password");
String query = "SELECT id FROM user_table WHERE " +
        "username = ' " + username + " ' AND " +
        "password = PASSWORD( ' " + password + " ' ) " ;

ResultSet rs = stmt.executeQuery(query);
int id = -1; // La valeur -1 signale un utilisateur non authentifié

while (rs.next())
        id = rs.getInt("id");
```

Les deux premières lignes extraient les saisies de la requête HTTP. La suivante construit la requête SQL, qui est ensuite exécutée; son résultat est alors collecté dans la boucle while(). Si un couple de valeurs correspond dans la table, son numéro d'identifiant est renvoyé. Dans le cas contraire, la variable id reste la valeur -1, dénotant un utilisateur non authentifié.

On pourrait croire qu'avec ce code l'utilisateur est authentifié si, et seulement si, l'identifiant et le mot de passe proposés sont reconnus …

...Et l'on aurait tort ! Rien n'empêche un attaquant d'injecter des commandes SQL dans les champs username et password pour modifier le sens de la requête SQL exécutée.

Revenons sur celle-ci :

String query = “SELECT id FROM user_table WHERE “ +
“username = ‘ “ + username + “ ‘ AND “ +
“password = PASSWORD(‘ “ + password + “ ‘) “ ;

Ce code s’attend à trouver des données dans les champs username et password.
Toutefois, rien n’empêche un attaquant d’y placer les caractères de son choix. Que se passe-t-il en cas de saisie de ‘OR 1=1 -- pour le nom d’utilisateur, avec le mot de passe x ?

La chaîne de requête devient alors :

SELECT id FROM user_table WHERE username = ‘ ’ OR 1=1 -- ‘AND password = PASSWORD(‘x’)

En syntaxe SQL, le tiret double ‘--’ introduit un commentaire; le programme ne tient donc pas compte du tout ce qui suit, cette requête est alors équivalente à :

SELECT id FROM user_table WHERE username = ‘’ OR 1=1

Voilà une instruction SELECT qui fonctionne bien différemment : elle renverra des numéros d’identifiants dans le cas où l’identifiant est une chaîne vide (‘’) ou bien si un est égale à un. Cette dernière condition étant toujours vérifiée, elle produira tous les numéros d’identifiants de la table user_table, dont le dernier sera retenu.

Dans cette situation, l’attaquant a placé les instructions SQL (‘OR 1=1 --) en lieu et place des données dans le champ username.

Choix d'un code approprié d'injection SQL

Pour parvenir à injecter du SQL, l'attaquant doit transformer le code existant du développeur en instructions valides. Il est un peu difficile de travailler en aveugle, aussi proposons-nous des requêtes généralement couronnées de succès :

- ' OR 1=1 --
- ') OR 1=1 --

Par ailleurs, de nombreuses applications web sont très disertes en matière de rapport d'erreurs et autres informations de débogage. C'est ainsi qu'une tentative gratuite optant pour ' OR 1=1 --' produit souvent des renseignements très instructifs.

Error executing query : You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELECT (title,body) FROM blog_table WHERE cat='**OR 1=1--'** at line 1

Voilà un message d'erreur qui reprend l'intégralité de l'instruction SQL. Dans ce cas de figure, on constate que la base de données attendait un entier et non une chaîne, aussi l'injection OR 1=1 --, non précédée d'une apostrophe, produira le résultat escompté.

Avec la plupart des bases de données SQL, un attaquant peut placer d'un seul trait de nombreuses instructions, pour peu que la syntaxe de chacune soit correcte. Pour le code suivant, nous avons montré que la valeur **' OR 1=1 --** pour username et x pour password renvoyait le dernier utilisateur :

```
String query = "SELECT id FROM user_table WHERE " +
        "username = ' " + username + " ' AND " +
        "password = PASSWORD(' " + password + " ' ) ";
```

Cependant, il est possible d'insérer d'autres instructions. Ainsi, l'identifiant que voici :
'OR 1=1; DROP TABLE user_table; --

transformerait la requête comme suit :

SELECT id FROM user_table WHERE username = ' ' OR 1=1 ; DROP TABLE user_table;
-- ' AND password = PASSWORD('x');

laquelle équivaut à :

SELECT id FROM user_table WHERE username = '' OR 1=1;
DROP TABLE user_table;

Cette commande réalise une instruction SELECT syntaxiquement correcte, avant de détruire la table des utilisateurs user_table par l'instruction SQL DROP.

Les attaques par injection ne sont pas toujours aveugles. De nombreuses applications web sont développées à l'aide d'outils Open Source. Pour réussir plus facilement vos attaques par injection, téléchargez gratuitement des versions complètes ou d'évaluation des produits et mettez en place votre propre système de test. Si vous parvenez à mettre celui-ci en défaut, il y a fort à parier que le même problème se pose sur toutes les applications employant le même outil/ framework.

Prévention contre l'injection SQL

Le principal problème demeure que les chaînes ne sont pas correctement échappées ou que les types de données ne sont pas contraints. Pour éviter une injection SQL, on commencera par contraindre autant que possible ces derniers (si une saisie représente un entier, on la traitera comme telle à chaque fois qu'on y réfère). Par ailleurs, on échappera systématiquement les saisies des utilisateurs. Pour se protéger contre l'attaque donnée en exemple, il aurait simplement suffi d'échapper l'apostrophe (“ ‘ “) et la barre oblique inversée (“ \ “) en les précédant d'une barre oblique inversée (respectivement, “ \ ‘ “ et “ \\ “).
Parfois, le remède est toutefois bien plus complexe; nous vous recommandons par conséquent de rechercher la fonction d'échappement adaptée à votre base de données.

De loin la meilleure solution consiste à faire appel à des requêtes préparées.
Initialement destinées à optimiser les connecteurs de base de données, elles séparent strictement, à un très bas niveau, les instructions SQL des données issues des utilisateurs.
En d'autres termes, un emploi correct des requêtes préparées évite une fois pour toutes d'interpréter toute saisie utilisateur comme des instructions SQL.

Protection des injections SQL dans le langage Java.

Prévention avec les spécifications JEE

- Avant d'exécuter la requête SQL, il faut effectuer une validation des données, pour supprimer des caractères spéciaux.
- Pour cela, il faut utiliser des requêtes préparées.

```
01.  PreparedStatement  preparedStatement=conn.prepareStatement("SELECT * FROM  usercheck where username=?")
     ;
02.  preparedStatement.setString(1, user);
```

Les méthodes setXXX() s'occupent de la validation des typages des données et de l'échappement des caractères spéciaux.

Consultez leurs javadocs respectives pour chaque type primitifs setString(), setFloat(), etc...

Dorénavant, si on tente une injection SQL de la forme inputFormData' OR 1 = 1
On aura alors une requête SQL générée de la forme suivante :

SELECT * FROM usercheck where username = 'inputFormData\' OR 1 = 1'

tous les caractères spéciaux seront automatiquement échappés.

Lorsque l'on utilise des ORM de persistence des données basées sur les spécifications JPA tel que Hibernate, TopLink, EclipseLink, il ne faut pas croire que les injections SQL ne sont pas possible.

Prévention avec les ORM (JPA)

Pour prévenir le code Java de ce genre d'attaque via les outils d'ORM, j'utilise les requêtes nommées au lieu des simples requêtes. En effet, les requêtes nommées utilisent un mécanisme interne qui utilise les requêtes préparées JEE alors que les simples requêtes utilisent les "normal Statement" en JEE.

Requête simple en JPA :

```
01.  String q="SELECT r FROM  User r where r.userId='"+user+"'";
02.   Query query=em.createQuery(q);
03.   List users=query.getResultList();
```

c'est pourquoi, il faut privilégier les requêtes nommées comme ceci :

```
01.  Query query=em.createNamedQuery("User.findByUserId");
02.   query.setParameter("userId", user);
03.   List users=query.getResultList();
```

em étant le gestionnaire d'entité persistente "Entity Manager" JPA.

Exemples et protections des injections SQL dans le langage Php.

Détournement de clause WHERE

On a l'habitude de formater nos requêtes SQL de cette manière :

```php
<?php
    $requete = "SELECT * FROM membre
    WHERE  pseudo = '".$_GET['pseudo']."'
    AND password = '".$_GET['password']."' ";
?>
```

Notez que j'ai utilisé $_GET mais que l'injection SQL fonctionne tout aussi bien avec $_POST : ne vous croyez donc pas à l'abri en mettant method = "post" en attribut de vos balises de formulaire. C'est souvent une idée préconçue mais qui s'avère totalement fausse car il est fort possible de modifier les en-têtes HTTP pour transmettre des données sous forme $_POST, donc méfiez-vous ! Imaginons un instant que je transmette ceci :

```php
<?php
    $_GET['password'] = " ' OR 1 = '1 ";
?>
```

Ma requête devient :

```php
<?php
    $requete = "SELECT * FROM membre
    WHERE  pseudo = '".$_GET['pseudo']."'
    AND password = '' OR 1 = '1' ";
?>
```

Ainsi, vous pouvez entrer n'importe quel mot de passe, il sera toujours valide puisque vous admettrez que 1 est toujours égal à 1. Il suffit donc de fournir un pseudo valide et la requête est vraie !
De cette manière, il est possible de récupérer facilement les informations sur les membres du site.

Détournement de la clause DELETE

Vous commencez à avoir peur de votre code PHP ? C'est normal mais attendez de voir ce que nous réserve le détournement d'une clause DELETE : c'est peut-être ce qu'il y a de pire !

Prenons un exemple de requête :

```php
<?php
    $requete = "DELETE FROM membre
    WHERE id = '".$_GET['id']."' ";
?>
```

Voilà ce que je peux transmettre en paramètre :

```php
<?php
    $_GET['id'] = "1 ' OR id > '0  ";
?>
```

La requête donne donc :

```php
<?php
    $requete = "DELETE FROM membre
    WHERE id = '1' OR id > '0' ";
?>
```

Cette requête détruira donc toutes les entrées contenues dans votre table membre ! Vous aurez perdu tous vos membres en 10 ms montre en main. Et encore, cette requête pourrait être beaucoup plus dangereuse.

Les requêtes multiples : mysqli_multi_query()

L'interface *mysqli* propose d'exécuter plusieurs requêtes en une seule. Je vous conseille très vivement de ne jamais le faire à moins d'être totalement sûr que les paramètres envoyés seront valides !
Certes, il y a quelques avantages pour le traitement des résultats. Cependant, du point de vue de la sécurité, elles sont à proscrire !

Prenons un exemple concret :

```php
<?php
    $requete = "SELECT pseudo FROM membre
    WHERE id = '".$_GET['id']."' ";
?>
```

Voilà le paramètre que nous pouvons transmettre :

```php
<?php
    $_GET['id'] = "1'; DROP TABLE membre ";
?>
```

Notre requête devient :

```php
<?php
    // Connexion à la BDD
    $link = mysqli_connect("localhost", "my_user", "my_password", "world");

    // Formatage de la requête
    $requete = "SELECT pseudo FROM membre
    WHERE id = '1'; DROP TABLE membre ";

    // Exécution de la requête
    mysqli_multi_query($link, $requete) or exit(mysqli_error);
?>
```

Votre table membre vient de rendre l'âme.

Cette petite partie des exemples d'attaques par injection SQL en PHP nous aura permis de faire une initiation aux dangers de ces attaques dans ce langage et nous allons à présent voir comment s'en protéger efficacement dans ce langage de programmation !

Les magic quotes : la fausse bonne idée...

Il nous faut donc protéger les données issues du formulaire **avant** de les entrer dans notre requête sinon on s'expose à de gros risques...

Nous devons échapper nos valeurs pour les rendre inertes et sans danger. Les *magic_quotes* font partie d'une directive de PHP visant à assurer la sécurité des requêtes SQL à son insu en échappant systématiquement les caractères suivant :

- les guillemets simples **'** ;
- les guillemets doubles **"** ;
- les slashes **/** ;
- les caractères ***NULL***.

Pour échapper un caractère, cette directive ajoute des antislashes dans les chaînes qui transitent vers le script PHP.

En fait, elle joue le même rôle que la fonction *addslashes()*.

Regardons ce que donne l'activation de cette directive ensemble :

```php
<?php
    $requete = "SELECT pseudo FROM membre
    WHERE id = '".$_GET['id']."' ";
?>
```

L'ajout du caractère d'échappement permet d'utiliser certains caractères dans notre requête sans qu'elle ne se transforme pour autant en injection.

```php
<?php
    $requete = "SELECT pseudo FROM membre
    WHERE id = '\"1\"mettez l'injection que vous voulez ici\"' ";
?>
```

Cette directive a donc la capacité de bloquer certaines injections SQL mais elle reste très problématique avec l'utilisation d'XHTML.

Pour éliminer ces barres obliques inverses, on a souvent recours à la fonction *stripslashes()* mais il est souvent fastidieux de nettoyer toutes les variables de PHP avant de les afficher.

Par ailleurs, je ne vous l'ai pas encore dit mais les *magic_quotes* ne protègent pas contre toutes les injections SQL car certaines ne nécessitent pas de guillemets.

Bref, je vous invite à les désactiver au plus vite (de toute façon, avec l'avènement de PHP 6, il est très probable qu'elles disparaissent à jamais).

Sécurisé via la fonction mysql_real_escape_string()

Renvoyez donc les guillemets magiques et la fonction *addslashes()* !

Nous allons maintenant parler d'une fonction bien pratique : *mysql_real_escape_string()*.

Prototype :

```
string mysql_real_escape_string ( string $unescaped_string [, resource $link_identifier ] )
```

Cette fonction échappe les caractères suivants :

- les guillemets simples **'** ;
- les guillemets doubles **"** :
- les slashes **/** ;
- les caractères ***NULL*** ;
- les caractères suivants : , , **\x00** et **\x1a**.

En gros, elle neutralise tous les caractères susceptibles d'être à l'origine d'une injection SQL. À partir de maintenant, utilisez toujours cette fonction pour sécuriser les chaînes transmises à vos requêtes. Par ailleurs, il faut aussi que vos données soient placées dans des guillemets (simples ou doubles) sinon, il est possible de réinjecter du code (notamment des sous-requêtes grâce aux parenthèses).

Exemple :

```php
<?php
    $pseudo = mysql_real_escape_string($_POST['pseudo']);
    $requete = "SELECT * FROM membre WHERE pseudo = '$pseudo' ";
?>
```

Contrairement à ce qu'on pourrait penser, il est toujours possible de réaliser des injections SQL particulières qui visent notamment à surcharger votre serveur en alourdissant votre requête. Ce type d'injection utilise les caractères **%** et **_**.

Le caractère **%** est souvent utilisé dans MySQL avec la clause *LIKE*. Ce caractère est un joker qui représente n'importe quelle autre chaîne.

```
1 | SELECT id, pseudo, password FROM membre WHERE pseudo LIKE 'a%'
```

Cette requête récupère toutes les informations sur les membres dont le pseudo commence par un **a**.

La fonction addcslashes()

Prototype :

```
string addcslashes ( string $str , string $charlist )
```

Cette fonction souvent méconnue offre la possibilité de se protéger contre le type d'injection soulevée auparavant.

La chaîne à échapper est le premier paramètre à passer à la fonction. Puis, on entre les caractères qui doivent être échappés.

Elle s'utilise comme cela :

```php
<?php
    $chaine = "%salut_";
    // Échappement des caractères % et _
    $chaine = addcslashes($chaine, '%_');
    echo $chaine; // Affiche \%salut\_
?>
```

Je vous propose une fonction qui sécurise vos données avant de les passer dans votre requête. Centraliser le traitement est une bonne habitude à acquérir dès le début.

```php
<?php
    function securite_bdd($string)
    {
        // On regarde si le type de string est un nombre entier (int)
        if(ctype_digit($string))
        {
            $string = intval($string);
        }
        // Pour tous les autres types
        else
        {
            $string = mysql_real_escape_string($string);
            $string = addcslashes($string, '%_');
        }

        return $string;
    }
?>
```

Notre fonction *securite_bdd()* prend un paramètre : une chaîne de caractères quelconque.

Si cette chaîne n'est composée que de nombres, on force la conversion explicite en un entier avec la fonction *intval()*.

Autrement, on échappe la chaîne avec la fonction *mysql_real_escape_string()* puis *addcslashes()*.

Maintenant, je veux qu'à chaque fois que vous récupérez une donnée issue d'un formulaire ou d'une URL, vous lui appliquiez cette fonction avant de faire quoi que ce soit.

Exemple :

```php
<?php
    // Sécurisation des variables
    $pseudo = securite_bdd($_GET['pseudo']);
    $password = securite_bdd($_GET['password']);

    // Formatage de la requête
    $requete = "SELECT * FROM membre
    WHERE  pseudo = '$pseudo'
    AND password = '$password' ";

    // Envoi au serveur en toute sécurité ^^
    mysql_query($requete) or exit(mysql_error());
?>
```

Maintenant, toutes les données qui entrent dans votre BDD doivent être sécurisées de cette manière.

Les requêtes préparées

Les requêtes préparées sont une innovation majeure apportée notamment par MySQLi. Concrètement, elles permettent une augmentation de la sécurité et, dans certains cas, un gain de performances.

En tant que programmeur, vous savez (ou vous vous doutez) qu'il existe plusieurs types de variables :

- les nombres entiers **int** ;
- les nombres réels **float**/**double** ;
- les chaînes de caractères **string** ;
- etc.

La préparation de nos requêtes nous permettra de fixer le type de variable qui doit entrer dans notre requête. De cette manière, la majeure partie des injections SQL sera impossible !

Utilisons l'interface MySQLi pour comprendre le mécanisme de la préparation de requêtes. Il faut bien entendu suivre un processus assez fixe :

- **init()** : création d'un objet commande et association à la connexion ;
- **prepare()** : préparation de la requête avec utilisation des "trous" (placeholder), analyse de la requête puis compilation ;
- **bind_param()** : liaison des paramètres dans l'ordre des marqueurs ;
- **execute()** : exécution de la requête avec la valeur des paramètres envoyés ;
- **store_result()** : transmission de l'intégralité des résultats ;
- **bind_result()** : liaison des résultats à des variables PHP ;
- **fetch()** : équivalent de *mysqli_fetch_array()* ;
- **close()** : fermeture de la commande préparée.

Nous allons employer le style de programmation procédurale pour la compréhension de tous, mais sachez qu'en orienté objet c'est le même mécanisme.

Formatons notre requête SQL :

```php
<?php
    $requete = "SELECT id, pseudo, password
    FROM membre
    WHERE  pseudo = ?
    AND password = ? ";
?>
```

La seule différence avec une requête non préparée, c'est la présence de marqueurs **?**. C'est un marqueur de paramètre qui s'utilise comme une variable. Ce marqueur ne s'écrit pas entre guillemets. Il ne faut donc pas utiliser **'?'** mais bien **?**.

Voyons à présent le processus de commande préparée :

```php
<?php
    // Connexion à la BDD
    $connexion = mysqli_connect("localhost", "my_user", "my_password", "world");

    // Création de l'objet commande
    $prepa = mysqli_stmt_init($connexion);

    // Préparation de la requête : envoi à la base
    mysqli_stmt_prepare($prepa, $requete);

?>
```

Récupérons à présent les données issues du formulaire :

```php
<?php
    $pseudo = $_GET['pseudo'];
    $password = $_GET['password'];
?>
```

Maintenant, il ne nous reste plus qu'à forcer le typage de nos variables et à les transmettre à la requête.
Nous allons utiliser des lettres qui correspondent à un type particulier de variable. Je vais vous présenter les lettres principales que nous utiliserons :

Lettre	Type
s	string
i	int
b	BLOB

Le type BLOB correspond à des données brutes comme un fichier ou une image.

Nous allons passer en paramètres les variables en forçant leur type :

```php
<?php
    mysqli_stmt_bind_param($prepa, 'ss', $pseudo, $password);
?>
```

Expliquons le code ci-dessus.

Tout d'abord, on passe en paramètre notre requête préparée **$prepa** puis on indique le type de variables qu'elle doit recevoir. Nous utilisons deux fois la lettre **s** car on transmet les variables **$pseudo** et **$password** qui doivent correspondre à des chaînes de caractères.

Nous devons encore exécuter la requête et traiter les résultats :

```php
<?php
    // Exécution de la requête
    $resultat = mysqli_stmt_execute($prepa);

    // Récupération des résultats
    mysqli_stmt_store_result($prepa);

    // On lie les résultats à des variables PHP
    mysqli_stmt_bind_result($prepa, $id, $pseudo, $password);

    // Affichage des résultats
    mysqli_stmt_fetch($prepa);
    echo 'Salut ' . htmlentities($pseudo);
    echo 'ID : ' . $id;
    echo 'Password ' . $password;

    // Fermeture de la requête
    mysqli_stmt_close($prepa);

?>
```

Récapitulation de la procédure :

```php
<?php
    // Formatage de la requête
    $requete = "SELECT id, pseudo, password
    FROM membre
    WHERE  pseudo = ?
    AND password = ? ";

    // Connexion à la BDD
    $connexion = mysqli_connect("localhost", "my_user", "my_password", "world")

    // Création de l'objet commande
    $prepa = mysqli_stmt_init($connexion);

    // Préparation de la requête : envoi à la base
    mysqli_stmt_prepare($prepa, $requete);

    // Récupération des variables
    $pseudo = $_GET['pseudo'];
    $password = $_GET['password'];

    // Transfert des paramètres
    mysqli_stmt_bind_param($prepa, 'ss', $pseudo, $password);

    // Exécution de la requête
    $resultat = mysqli_stmt_execute($prepa);

    // Récupération des résultats
    mysqli_stmt_store_result($prepa);

    // On lie les résultats à des variables PHP
    mysqli_stmt_bind_result($prepa, $id, $pseudo, $password);

    // Affichage des résultats
    mysqli_stmt_fetch($prepa);
    echo 'Salut ' . htmlentities($pseudo);
    echo 'ID : ' . $id;
    echo 'Password ' . $password;

    // Fermeture de la requête
    mysqli_stmt_close($prepa);

?>
```

Vous le voyez, c'est quand même assez fastidieux d'écrire une requête préparée. Mais rappelez-vous que c'est une des méthodes les plus sûres pour envoyer des requêtes au serveur. Pour une utilisation plus facile des requêtes préparées, je recommande **PDO** (PHP Data Object) mais le principe reste le même.

Pour les curieux, voilà comment on prépare et on exécute une requête avec **PDO** :

```php
<?php
    // $dbh correspond à une connexion avec une base de données (MySQL, PostgreSQL, Oracle...)
    $stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
    $stmt->bindParam(':name', $name);
    $stmt->bindParam(':value', $value);

    // Insertion d'une ligne
    $name = 'one';
    $value = 1;
    $stmt->execute();
?>
```

Pour info, j'utilise maintenant **PDO** pour gérer les échanges avec mes bases de données et j'essaye au maximum de préparer mes requêtes. Sinon pour échapper les chaînes transmises il faut utiliser la méthode **PDO::quote()**.

Protection des injections SQL dans le langage .Net.

Ces recommandations sont applicable pour

- ASP.NET version 1.1
- ASP.NET version 2.0

Introduction

Je vais vous montrer un certain nombre de façons d'aider à protéger votre application ASP.NET d'attaques d'injection de SQL. L'injection de SQL peut arriver quand une application utilise la saisie pour construire des déclarations de SQL dynamiques ou quand il utilise des procédures stockées pour se connecter à la base de données. Des mesures de sécurité conventionnelles, comme l'utilisation de SSL et IPSEC, ne protègent pas votre application d'attaques d'injection de SQL. Des attaques d'injection de SQL fructueuses permettent aux utilisateurs malveillants d'exécuter des commandes dans la base de données d'une application.
Les contre-mesures incluent l'utilisation d'une liste de caractères acceptables pour contraindre la saisie, l'utilisation des requêtes SQL paramétrées pour l'accès de données et l'utilisation d'un rôle privilégié spécifique avec une limitation des permissions dans la base de données.
L'utilisation de procédures stockées avec le SQL paramétré est l'approche recommandée parce que les paramètres SQL sont “type safe”. Des paramètres SQL “type safe” peuvent aussi être utilisés avec le SQL dynamique. Dans des situations où le SQL paramétré ne peut pas être utilisé, envisagez d'utiliser la technique d’échappement des caractères.

Limitation des données saisies

Vous devriez valider toute la saisie à vos applications ASP.NET pour le type, la longueur, le format et la portée. En contraignant la saisie utilisée dans vos requêtes d'accès aux données, vous pouvez protéger votre application .NET de l'injection de SQL.

En contraignant la saisie, c'est une bonne pratique de créer une liste de caractères acceptables et utiliser des expressions régulières pour rejeter n'importe quels caractères qui ne sont pas dans la liste. Le risque potentiel associé à l'utilisation d'une liste de caractères inacceptables est qu'il est toujours possible de d'oublier un caractère inacceptable en définissant la liste; aussi, un caractère inacceptable peut être représenté dans un format alterné pour passer la validation.

Contraindre les saisies dans les pages web ASP.NET

Commencé par limiter la saisie dans le côté serveur du code pour vos pages Web ASP.NET. Ne comptez pas sur la validation de données côté de client via Javascript parce que l'on peut facilement contourner cela. La validation de données côté client s'utilise seulement pour réduire des allers et retours et améliorer l'expérience utilisateur.

Si vous utilisez le server controls, utilisez le contrôle de validation ASP .NET, comme le **RegularExpressionValidator** et **RangeValidator** qui vont contrôler et contraindre les données en entrées de formulaire de saisie. Si vous utilisez un contrôle de validation standard HTML, utilisez la classe **Regex** du coté serveur.

Exemple :

```
<%@ language="C#" %>
<form id="form1" runat="server">
    <asp:TextBox ID="SSN" runat="server"/>
    <asp:RegularExpressionValidator ID="regexpSSN" runat="server"
                                    ErrorMessage="Incorrect SSN Number"
                                    ControlToValidate="SSN"
                                    ValidationExpression="^\d{3}-\d{2}-\d{4}$" />
</form>
```

Si les données sont obtenues à partir d'un cookie, utilisez une expression régulière comme ceci :

```
if (Regex.IsMatch(Request.Cookies["SSN"], "^\d{3}-\d{2}-\d{4}$"))
{
    // access the database
}
else
{
    // handle the bad input
}
```

Contraindre les saisies dans le code d'accès ASP.NET au données en base.

Dans certaine situation, vous devez founir un système de validation d'accès de vos données, en plus de celui fait au niveau des pages web ASP.NET. Deux situations communes répondant à ce cas particulier sont :

- Le cas **Untrusted clients** : Si la donnée peut provenir d'une ressource non vérifiée ou si vous ne pouvez pas garantir si la donnée a été validée préalablement, ajouter une validation logique dans vos méthode d'accès aux données au sein de vos DAOs (Data Access Object).
- Le cas **Library code** : Si le code d'accès de vos données est imbriqué dans une librairie qui provient d'un framework pour un usage dans de multiples applications, votre code propriétaire doit ajouter vos propres contrôles de validation, car on ne peut pas toujours vérifiée si la librairie utilisée protègent l'accès des données.

L'exemple qui suit montre comment une routine d'accès aux données peut valider ses paramètres en entrées en utilisant une expression régulière.

```
using System;
using System.Text.RegularExpressions;

public void CreateNewUserAccount(string name, string password)
{
    // Check name contains only lower case or upper case letters,
    // the apostrophe, a dot, or white space. Also check it is
    // between 1 and 40 characters long
    if ( !Regex.IsMatch(userIDTxt.Text, @"^[a-zA-Z'./s]{1,40}$"))
      throw new FormatException("Invalid name format");

    // Check password contains at least one digit, one lower case
    // letter, one uppercase letter, and is between 8 and 10
    // characters long
    if ( !Regex.IsMatch(passwordTxt.Text,
                    @"^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$" ))
      throw new FormatException("Invalid password format");

    // Perform data access logic (using type safe parameters)
    ...
}
```

Utilisation des paramètres avec des procédures stockées

L'utilisation de procédure stockée ne prévient pas nécessairement du risque d'attaque par injection SQL. La chose importante à faire est d'utiliser des paramètres avec les procédures stockées. Si vous n'utilisez pas de paramètres, votre procédure stockée est vulnérable aux injections SQL si elle utilisent des entrées non filtrées.

Le code suivant montre comment utiliser **SqlParameterCollection** en appelant une procédure stockée.

```
using System.Data;
using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))
{
  DataSet userDataset = new DataSet();
  SqlDataAdapter myCommand = new SqlDataAdapter(
            "LoginStoredProcedure", connection);
  myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
  myCommand.SelectCommand.Parameters.Add("@au_id", SqlDbType.VarChar, 11);
  myCommand.SelectCommand.Parameters["@au_id"].Value = SSN.Text;

  myCommand.Fill(userDataset);
}
```

Dans cette situation, le paramètre **@au_id** est traitée comme une valeur littérale et non comme un code exécutable. De plus, le paramètre subit une vérification de typage et de longueur.
La valeur d'entrée ne peut dépasser 11 caractères. Si la donnée n'est pas conforme au type ou à la longueur définie par le paramètre, la classe **SqlParameter** lève une exception.

Étudier l'usage de votre application pour les procédures stockées paramétrées

Étant donné que l'usage des procédures stockées paramétrées ne protègent pas nécessairement des attaques par injection SQL, vous devez revoir l'usage définie par ce genre de procédure. Par exemple, la procédure stockée paramétrée suivante possède des failles de sécurités.

```
CREATE PROCEDURE dbo.RunQuery
@var ntext
AS
        exec sp_executesql @var
GO
```

- Cette procédure stockée s'exécute quelle que soit la valeur du paramètre ntext
  Imaginez que la variable **@var** est affectée avec :
  DROP TABLE ORDERS;
  Dans ce cas, la table **ORDERS** sera supprimée.
- Cette procédure stockée s'exécute avec les privilèges **dbo**.
- Le nom de la procédure stockée **RunQuery** est un mauvais choix. Si un hacker est capable de voir le schéma de la base de donnée, il ou elle pourra voir le nom de la procédure stockée. Avec un nom comme **RunQuery,** il peut deviner que cette procédure stockée est utilisée pour exécuter une requête SQL.

Utilisation des paramètres avec le SQL dynamique

Si vous ne pouvez pas utiliser de procédure stockée, vous devez utiliser des paramètres lors de la construction des requêtes SQL dynamique. Ce code montre comment utiliser **SqlParamtersCollection** avec le SQL dynamique.

```
using System.Data;
using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))
{
  DataSet userDataset = new DataSet();
  SqlDataAdapter myDataAdapter = new SqlDataAdapter(
         "SELECT au_lname, au_fname FROM Authors WHERE au_id = @au_id",
         connection);
  myCommand.SelectCommand.Parameters.Add("@au_id", SqlDbType.VarChar, 11);
  myCommand.SelectCommand.Parameters["@au_id"].Value = SSN.Text;
  myDataAdapter.Fill(userDataset);
}
```

Utiliser des batchs paramétrés

Une erreur commune se léve lors de la concaténation de plusieurs requêtes SQL pour envoyer un batch de requêtes au serveur de base de donnée d'un seul coup. Dans ce cas de figure vous ne pouvez pas utiliser de paramètres. Mais vous pouvez utiliser cette technique si vous êtes sûre que les noms de paramètres ne sont pas répétés. Vous pouvez facilement faire cela en utilisant des noms de paramètres unique lors de la concaténation des requêtes SQL, comme dans cet exemple :

```
using System.Data;
using System.Data.SqlClient;
. . .
using (SqlConnection connection = new SqlConnection(connectionString))
{
  SqlDataAdapter dataAdapter = new SqlDataAdapter(
       "SELECT CustomerID INTO #Temp1 FROM Customers " +
       "WHERE CustomerID > @custIDParm; SELECT CompanyName FROM Customers " +
       "WHERE Country = @countryParm and CustomerID IN " +
       "(SELECT CustomerID FROM #Temp1);",
       connection);
  SqlParameter custIDParm = dataAdapter.SelectCommand.Parameters.Add(
                                          "@custIDParm", SqlDbType.NChar, 5);
  custIDParm.Value = customerID.Text;

  SqlParameter countryParm = dataAdapter.SelectCommand.Parameters.Add(
                                       "@countryParm", SqlDbType.NVarChar, 15);
  countryParm.Value = country.Text;

  connection.Open();
  DataSet dataSet = new DataSet();
  dataAdapter.Fill(dataSet);
```

```
}
. . .
```

Utiliser des routines d'échappement pour analyser les caractères en entrés

Dans la situation où les requêtes SQL paramétrées ne peuvent être utilisées et vous êtes forcé d'utiliser le SQL dynamique à la place, vous devez contrôler les caractères en entrés qui ont une sémantique spéciales pour le moteur d'exécution SQL du serveur de base de données.
Les caractères spéciaux posent problème uniquement pour le SQL dynamique car ils sont échappés par les méthodes "setters" des requêtes paramétrés SQL.

Vous pouvez utiliser une méthode C# du type :

```
private string SafeSqlLiteral(string inputSQL)
{
  return inputSQL.Replace("'", "''");
}
```

Utiliser des habilitations appropriées pour le compte d'accès à la base de donnée

Votre application doit se connecter à votre base de donnée en utilisant un compte de faible privilège. Si vous utilisez une authentification Windows pour vous connecter, le compte Windows associé doit avoir un faible privilège délégué par le système d'exploitation sous jacent et il doit avoir des privilèges limités pour accéder aux ressources Windows. De plus, le login correspondant au SQL server doit être restreint en permission sur la base de donnée.

- Créer un SQL server login pour le service web. Le service web possède une authentification sur le réseau qui est présenté à la base de donnée de la forme DOMAIN\WEBSERVERNAMES$. Par exemple, si votre domaine est XYZ et que votre serveur web se nomme 123, vous devez créer un identifiant de connexion pour la base de donnée de la forme XYZ\123$.
- Ajouter à ce nouvel identifiant les droits en accès uniquement pour la base de donnée de l'application qui lui est dédiée en créant un USER et en ajoutant un RÔLE à ce USER.
- Ajouter la permission pour faire en sorte que ce RÔLE soit nécessairement requis pour appeler uniquement les procédure stockées ou accéder aux tables requises utiles à

l'application. N'offrez une visibilité qu'aux procédures stockées utilisées au sein de l'application et à l'usage stricte minimal des tables utilisées au sein de l'application.
Par exemple, si l'application ASP .NET effectue uniquement des sélections en base de données et aucune mise à jour, ne donner que des droits en lecture et non en écriture sur les tables. Ceci limitera les risques causés par l'attaquant qui utilisera une méthode par injection SQL.

Désactiver les informations relatives aux erreurs

Utilisez des exceptions structurées pour encapsuler les erreurs potentiellement déclenchées par votre application ASP .NET pour prévenir leur propagation jusqu'au client Web.
Utilisez des logs très détaillés localement sur votre serveur pour maintenir une traçabilité de l'attaque mais renvoyé le minimum d'information et de détails au client.

Si des erreurs sont levées lorsque le client est connectée à la base de donnée, vérifiez que vous ne diffusez uniquement des informations qui ne dévoilent pas votre technologie ASP .NET ou votre version de serveur de base de donnée pour ne pas compromettre la sécurité de celui ci.
Les attaquants utilisent les messages d'erreurs pour analyser le modèle conceptuel de données de votre base de donnée et affiner leur test d'attaque par injection SQL.