# Injection SQL, quelques exemples

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 19 mai 2009

http://www.bortzmeyer.org/sql-injection.html

---

L'injection SQL est une technique d'attaque utilisée contre les bases de données SQL dès lors qu'elles sont accessibles de l'extérieur, via une interface qui ne permet normalement **pas** de taper des commandes SQL quelconques. Cette technique est ancienne, banale, et, comme vient de le rappeler l'attaque contre plusieurs registres de noms de domaine <http://www.bortzmeyer.org/attaques-registres-noms.html>, elle est largement utilisée.

Je ne vais pas faire un tutoriel complet sur l'injection SQL, il en existe plusieurs en ligne comme la page de Wikipédia ou une bonne discussion sur StackOverflow <http://stackoverflow.com/questions/332365/xkcd-sql-injection-please-explain>. Je veux juste montrer quelques exemples.

Pour tous les cas qui suivent, on lance le programme en ligne de commande, ce qui offre évidemment bien plus de possibilités à l'attaquant. Mais les attaques par injection SQL sont parfaitement possibles dans d'autres circonstances, et sont en général pratiquée via une page Web.

D'abord, le principe : lorsqu'un programme fait une requête SQL qui contient des **paramètres** spécifiés par l'utilisateur, l'injection SQL consiste à mettre dans ces paramètres du code SQL, avec des caractères qui déclencheront l'injection. L'exemple archétypal est fourni par le célèbre dessin de xkcd <http://xkcd.com/327/>.

Commençons avec un programme Python qui reçoit un paramètre sur la ligne de commandes et l'inclut dans une requête SQL `SELECT`. Ce programme se connecte à une base de données PostgreSQL. La table est créée ainsi :

```
CREATE TABLE students (id SERIAL UNIQUE NOT NULL,
           name TEXT UNIQUE NOT NULL,
           birthdate DATE DEFAULT now());

INSERT INTO students (name) VALUES ('durand');
INSERT INTO students (name) VALUES ('toto');
INSERT INTO students (name) VALUES ('martin');
```

Le programme Python est :

```
import psycopg
import sys

name = sys.argv[1]
connection = psycopg.connect("dbname=essais")
# Il y a une subtilité avec l' "autocommit", je vous laisse la découvrir
cursor = connection.cursor()
cursor.execute("SELECT * FROM Students WHERE name='%s'" % name)
for tuple in cursor.fetchall():
    print tuple
```

Exécuté, ce programme donnera :

```
% python test1.py toto
(2, 'toto', <mx.DateTime.DateTime object for '2009-05-18 00:00:00.00' at b7e51aa0>)
```

Si on fait enregistrer les requêtes par PostgreSQL (`log_statement = 'all'`), on voit bien ce qui a été envoyé au SGBD :

```
2009-05-18 21:47:58 CEST LOG:  statement: SELECT * FROM students WHERE name='toto';
```

Maintenant, si un méchant fait une injection SQL, il tapera, par exemple :

```
% python test1.py "toto'; DROP TABLE Students; SELECT '"
('',)
```

et la table `Students` sera détruite. Le SGBD avait reçu :

```
2009-05-18 21:39:37 CEST LOG:  statement: SELECT * FROM students WHERE name='toto'; DROP TABLE students; SEL
```

et a donc, après le premier `SELECT`, exécuté l'instruction `DROP`...

Évidemment, dans ce cas, l'« attaquant » a accès au SGBD et peut faire n'importe quelle requête directement. Mais imaginons que le code Python ci-dessus aie été executé via una page Web, par exemple parce qu'il fait partie d'une application WSGI <http://www.bortzmeyer.org/wsgi.html>. Alors, le code sera exécuté avec les privilèges du compte qui fait tourner le WSGI (par exemple `apache`) et l'attaquant n'aura pas de compte du tout. Dans ce cas, l'injection SQL lui permettra de faire ce qu'il ne pouvait pas faire autrement. Elle représente donc une sérieuse faille de sécurité.

Comment empêcher l'injection SQL ? Une première réaction est de filtrer la requête pour en retirer les caractères « dangereux » :

```
import re

if re.search("[';]", name):
    raise InjectionSQL
```

mais cette méthode est peu sûre : il est difficile de s'assurer qu'on a prévu **tous** les caractères dangereux (surtout si `name` est en Unicode). La norme SQL étant ce qu'elle est, chaque SGBD a souvent les siens (par exemple, la barre oblique inverse est un caractère spécial pour certains SGBD). Il est plus sûr de procéder en sens inverse. Plutôt que d'interdire les méchants, n'autoriser que les bons :

```
if not re.search("^[a-z0-9]+$", name):
    raise InjectionSQL
```

Ici, l'expression rationnelle `^[a-z0-9]+$` teste que la variable est composée **entièrement** de lettres ASCII minuscules et de chiffres.

De tels tests sont très sûrs mais souvent trop restrictifs. Ici, si on veut enregistrer une personne nommé O'Reilly ou D'Alembert, on est coincés. Une telle approche « tout ce que je ne connais pas est interdit » mène souvent à des restrictions injustifiées <http://www.bortzmeyer.org/arreter-d-interdire-des-a
html>. Pour combattre l'injection SQL, il est donc préférable d'utiliser une autre méthode. Il y en a deux bonnes, faire fabriquer la requête par la bibliothèque d'accès, ou bien par le SGBD (requêtes **préparées**), et pas directement par le programmeur. On va illustrer la première en Python et la seconde en C.

Voici un exemple où la requête est fabriquée par la bibliothèque d'accès au SGBD, ici psycopg :

```
name = sys.argv[1]
connection = psycopg.connect("dbname=essais")
cursor = connection.cursor()
cursor.execute("SELECT * FROM students WHERE name=%(username)s;",
               {'username': name})
for tuple in cursor.fetchall():
    print tuple
```

Un tel code est protégé contre les injections SQL. La bibliothèque psycopg <http://initd.org/pub/software/psycopg/>, en voyant `%(username)s` va le remplacer par la valeur de l'entrée `username` du dictionnaire passé en second paramètre, effectuant toutes les transformations nécessaires pour que les caractères dangereux <http://www.bortzmeyer.org/caracteres-dangereux.html> soient neutralisés (et ces logiciels connaissent mieux les règles que vous, faites leur confiance). Ici, PostgreSQL a vu (regardez bien les apostrophes) :

```
2009-05-18 21:51:01 CEST LOG:  statement: SELECT * FROM students WHERE name='toto''; DROP TABLE students; SELECT
```

Et la table n'est plus détruite, PostgreSQL a simplement cherché un utilisateur de nom `"toto''; DROP TABLE students; SELECT ''"` et ne l'a évidemment pas trouvé.

Dans tous les cas, il faut noter que le seul fait d'utiliser un langage de programmation de haut niveau ne protège pas, contrairement à ce qui se passe avec d'autres attaques comme les débordements de tampons. Les attaques par injection SQL concernent tout le monde.

Dans certains cas, le typage peut aider à empêcher l'injection SQL. Si on cherche la table selon un paramètre numérique :

```
id = int(sys.argv[1])
connection = psycopg.connect("dbname=essais")
connection.set_isolation_level(0)
cursor = connection.cursor()
cursor.execute("SELECT * FROM students WHERE id=%i;" % id)
for tuple in cursor.fetchall():
    print tuple
```

Alors, il n'y a pas de risque. la conversion de l'argument `sys.argv[1]` en entier suffit à garantir qu'il n'y aura pas de caractères dangereux. Mais les chaînes de caractères sont, elles, toujours vulnérables.

Un petit retour sur l'exemple Python sûr. Si on veut éviter de construire un dictionnaire pour le passer en second paramètre, on peut utiliser le dictionnaire prédéfini renvoyé par `globals()` et on utilise alors directement ses variables Python, ici `name` :

```
cursor.execute("SELECT * FROM students WHERE name=%(name)s;",
               globals())
```

On peut aussi faire varier la syntaxe, cette forme, qui n'utilise plus les noms des paramètres, uniquement leur position, est également sûre, on n'utilise pas l'opérateur d'interpolation de Python (le %) :

```
cursor.execute("SELECT * FROM students WHERE name=%s;",
               (name,))
```

La lecture de la spécification de l'API Python pour les bases de données, PEP 249 <http://www.python.org/dev/peps/pep-0249/> est recommandée si on veut tous les détails de l'association des paramètres avec leur valeur. Le style `%(name)s` est nommé `pyformat`, celui avec les positions numériques est `numeric` (mais n'est pas géré par psycopg, chaque bibliothèque peut choisir son style de paramètres, afficher `MABIBLIOTHEQUE.parmastyle` permet de savoir quelle est le style de MABIBLIOTHEQUE). Par exemple, avec le module d'accès à SQLite, PySQLite <http://docs.python.org/library/sqlite3.html>, on écrit souvent `cursor.execute("INSERT INTO table VALUES ('hello world', ?, ?)", (user, message))`, les points d'interrogation étant remplacés par les paramètres, avec échappement des caractères dangereux.

Et en C ? Les injections SQL existent aussi. Si on exécute l'insertion ainsi (le code C complet est en (en ligne sur `http://www.bortzmeyer.org/files/insert-sql-with-injection.c`), on utilise l'interface libpq <http://www.postgresql.org/docs/current/interactive/libpq.html> de PostgreSQL) :

```
snprintf(sql_command, MAX_SQL_SIZE,
            "SELECT * FROM students WHERE name='%s';", name);
result = PQexec(conn, sql_command);
```

on est tout aussi vulnérable qu'en Python et pour les mêmes raisons. Une solution est de neutraliser (*"to escape"*) les caractères dangereux avec `PQescapeStringConn`. Une méthode souvent plus élégante est d'utiliser des **requêtes préparées** (le code complet est en (en ligne sur `http://www.bortzmeyer.org/files/insert-sql-without-injection.c`)) :

```
params[0] = argv[1];
result = PQprepare(conn, "MyInsertion", "SELECT * FROM students WHERE name=$1;",
             NPARAMS, NULL);
result =
        PQexecPrepared(conn, "MyInsertion", NPARAMS, (const char **) params, NULL,
                       NULL, 0);
```

Comme les requêtes préparées sont gérées par le SGBD, celui-ci peut afficher dans son journal la requête et la valeur des paramètres. Ici, pour une requête « normale » :

```
2009-05-19 09:56:32 CEST LOG:  execute MyInsertion: SELECT * FROM students WHERE name=$1;
2009-05-19 09:56:32 CEST DETAIL:  parameters: $1 = 'toto'
```

et ici pour une tentative d'injection SQL :

```
2009-05-19 09:43:17 CEST LOG:  execute MyInsertion: SELECT * FROM students WHERE name=$1;
2009-05-19 09:43:17 CEST DETAIL:  parameters: $1 = 'toto''; DROP TABLE Students; SELECT '''
```

Bien sûr, rien n'est gratuit, les requêtes préparées ont des avantages (sécurité contre l'injection SQL, possibilité pour le SGBD de passer du temps à les optimiser car il peut espérer qu'elles seront réutilisées) mais aussi des inconvénients puisque, par exemple, le SGBD doit désormais optimiser la requête sans disposer des données et donc le risque d'avoir un plan d'exécution sous-optimal <http://archives.postgresql.org/pgsql-jdbc/2009-05/msg00034.php>. (Merci à Marc Cousin pour ce point.)

Comme avec tous les problèmes de sécurité, il est recommandé d'adopter une **défense en profondeur**. Même si les assaillants ont pris pied sur l'enceinte du château, il faut que le donjon continue à résister. Dans le cas d'un SGBD, cela veut dire que le compte sous lequel se connecte l'application au SGBD doit avoir des privilèges **minimaux**. Par exemple, si l'application ne fait que **lire** la base, il faut uniquement lui donner le privilège `SELECT` et certainement pas `INSERT` ou `DELETE`. Un exemple est le moteur de recherche de ce blog </search>. Comme l'application (écrite en Python et reposant sur PostgreSQL) n'a jamais à modifier les données, l'utilisateur qui exécute la page Web, `apache`, n'a que le droit de lecture. Ainsi, même en cas d'injection SQL sur cette page, les dégâts seront limités. Voici l'affichage des privilèges <http://www.postgresql.org/docs/current/interactive/sql-grant.html> :

```
blog=> \dp blog.articles
                  Access privileges for database "blog"
 Schema |   Name   | Type  |              Access privileges
--------+----------+-------+---------------------------------------------
 blog   | articles | table | {stephane=arwdxt/stephane,apache=r/stephane}
(1 row)
```

qui se lit « `stephane` a tous les droits, `apache` a uniquement le droit de lecture (r pour "*read*") ».

Il existe des outils de tests de vulnérabilités :

- BSQL Hacker <http://www.darknet.org.uk/2008/09/bsql-hacker-automated-sql-injection-fra>,
- SQL Injection <https://addons.mozilla.org/en-US/firefox/addon/6727> (une extension à Firefox), que je trouve bien faite et pratique,
- HackBar <https://addons.mozilla.org/en-US/firefox/addon/3899>, une autre extension Firefox que je n'arrive pas à faire fonctionner,
- SQLmap <http://sqlmap.sourceforge.net/>, un excellent testeur, très riche, en ligne de commandes. Par exemple, `sqlmap -u http://www.bortzmeyer.org/search\?pattern=toto` va automatiquement tester plein de méthodes d'injection SQL sur mon moteur de recherche.
- Sur l'excellent site StackOverflow <http://www.bortzmeyer.org/stack-overflow.html>, on trouve de nombreuses discussions sur l'injection SQL <http://stackoverflow.com/questions/tagged/sql-injection>,
- Plein d'exemples amusants et d'aide en <http://ha.ckers.org/sqlinjection/>.

Pour en savoir plus : http://www.bortzmeyer.org/sql-injection.html