Injections SQL

# Comment les autoriser

Le titre peut surprendre mais non, nous ne nous sommes pas trompés. Nous avons utilisé des versions récentes des langages web HTML, CSS, PHP ce qui fait, que nativement et en plus, en utilisant de bonnes fonctions sécuritaires, nous n’autorisions malheureusement pas les injections SQL.

1. Tout d’abord, si la directive **magic\_quotes\_gpc** (GET/POST/COOKIE) du php.ini est activée sur le serveur web, alors PHP protège automatiquement les chaînes en échappant par un antislash certains caractères spéciaux (apostrophes, guillemets…). La chaîne : « J’aime le chocolat. » devient une fois postée et échappée par le php : « J\’aime le chocolat. » ce qui rend neutralise l’usage de caractères spéciaux servant aux injections. Ici, l’antislash protège de l’apostrophe. Dans les anciennes versions de PHP, nous étions obligés, par mesure de sécurité, d’appliquer nous même une fonction **addslashes()**. Il faut bien sur éviter désormais d’échapper à nouveau au risque de finir avec des chaînes comme celle-ci : « J\\\\\’aime le chocolat. »

L’initiative de protection de la nouvelle version de PHP n’a pas fait que des adeptes au sein de la communauté malgré sa volonté de fermer les brèches de sécurité, c’est probablement en partie pourquoi la version 6 de PHP désactivera cette directive forcant ainsi les développeurs à se re-protéger, par eux-mêmes cette fois-ci.

Il est donc généralement recommandé de désactiver nous-même dès à présent cette directive. En haut des pages php, on peut ajouter :

**<?php**

*// Désactivation des magic\_quotes\_gpc*

[ini\_set](http://www.php.net/ini_set)('magic\_quotes\_gpc', 0);

**?>**

Ce qui n’est pas toujours efficace car seule la configuration initiale d php est généralement prise en compte. Il est donc souhaitable de la modifier dans le fichier **php.ini**. Or, ce fichier, selon l’hébergement que l’on possède, n’est pas toujours accessible. C’est le cas par exemple pour nous, avec OVH. Une solution alternative se présente, il est possible de modifier les valeurs de register\_globals, zend\_optimizer, php\_ver ET magic\_quotes dans le fichier .**htaccess** présent à la racine de notre arborescence. Nous avons donc ajouté à ce fichier une ligne désactivant la directive : SetEnv MAGIC\_QUOTES 0.

Pour les hébergeurs autorisant moins de flexibilité, il existe toujours la solution de secours qui consiste à créer une fonction php qui effectue le travail inverse de cette directive :

function unescape($text)   
{   
  if(get\_magic\_quotes\_gpc())   
  {   
    $text = stripslashes($text);   
  }   
  return($text);   
}   
# usage   
echo "<p>You entered: " . unescape($\_POST['input']) . "</p>\n";

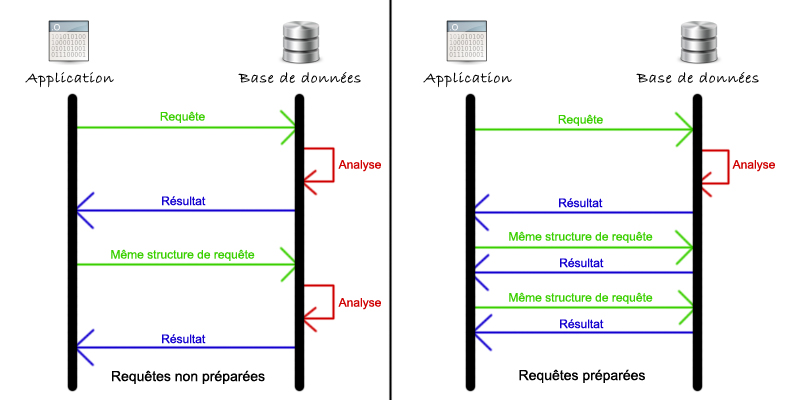
1. Maintenant que le problème de l’échappement automatique lié à notre version PHP est réglé, il en reste encore un. Nous allons maintenant parler d’un sujet clef concernant les injections SQL, les extensions permettant à PHP de se connecter à MySQL ! Il en existe 3, l’extension MySQL, mysqli et PDO. La première, l’extension MySQL est ancienne, elle n’est plus en développement mais en maintenance. Elle est inclus dans php 5 mais désactivée et ne supporte que peut des fonctionnalités récentes utilisées pour la sécurité comme les **commandes préparées** ! Alors que, mysqli et PDO sont toujours en développement actif et supportent de nombreuses fonctionnalités dont celle-ci !

Lors du développement de notre site de test, nous avons instinctivement utilisé une extension récente, pratique et sécuritaire, PDO ! **PDO** signifie **Php Data Object**, c’est une extension compilée de php qui sert de couche d’abstration des fonctions d’accès aux bases de données. Il ne fonctionne pas avec n’importe quelle base de données mais pour celles compatibles, il propose des fonctions d’accès universelles. Ce qui s’avère très pratique et facile d’utilisation. PDO possède de nombreux pilotes, IBM, Informix, MySQL, ODBC, Postgre SQL, SQLite … or il est encore en développement, d’autres sont donc à venir.

Après cette brève introduction de PDO, voici le cœur de ce qui nous interesse, les **requêtes préparées**. Leur principe est de soumettre un moule de requête et de définir les endroits (avec des « palce holders ») où l’on insèrera nos valeurs dynamiques dans la requête. Un place holder représente une seule valeur.

Ensuite, le SGBD va préparer c’est-à-dire interpréter compiler et stocker temporairement ce qui lui sera utile lors de l’execution de la vraie requête. La dernière étape consiste à associer les valeurs voulues (provenant généralement de variables) à leurs places holders. Le SGBD va alors savoir qu’il insère des valeurs et non des commandes. De ce fait les injections SQL ne fonctionnent plus ou, à minima pas pour un simple select ! Lors de la phase d’insertion en base d’élément, il subsiterait une faille, l’interprétation des valeurs ajoutées.

L’inconvénient principal lors du développement et de l’utilisation de PDO et de ses requêtes préparées, c’est la phase de débugguage … on n’a pas accès à la valeur compilée de notre requête dans le code php. Ensuite, l’écriture d’une requête devient plus longues et lors de l’execution d’une requête unique les performances sont moindres. Mais cette fonctionnalité permet de gagner du temps sur les requêtes executées plus d’une fois et enfin, **on gagne au niveau de la sécurité vu que les requêtes préparées sépare les données de la structure de la requête** !



Donc, pour autoriser les injections SQL, on a du désactiver l’échappement automatique de PHP5 et ne plus utiliser de requêtes préparées de PDO. Nous avons enfin pu tester des injections SQL au niveau notamment de l’identification donnant accès à l’intranet d’administration de notre site web !

Les utilisateurs existants dans la base de données sont :

|  |  |  |  |
| --- | --- | --- | --- |
| **Login** | so | ludo | ju |
| **Mot de passe** | 123456 | 654321 | 456123 |

Nous avons écrit volontairement les mots de passe ici, en clair, mais ils ont été haché via la fonction **sha1** mise à disposition par le langage PHP. L’inconvénient de sécurité étant que le PHP est exécuté côté serveur donc les mots de passe transitent en clair à cause de l’absence de connection sécurisée (https). Si nous souhaitions corriger cela, on pourrait imaginer l’utilisation côté client grâce au javascript de la fonction de hachage, on pourrait même utiliser un sel pour la session ou l’enregistrement en base.

Donc, lorsque l’on veut se connecter à l’intranet, on se connecte avec le login so, ludo ou ju, et le mot de passe correspondant, en théorie. Or, grâce aux injections SQL, on peut usurper l’identité de l’utilisateur de notre choix ! Le principe est simple, on doit saisir un login et un mot de passe. Or, on sait qu’ils seront utilisés dans une requête à une base de données pour vérifier l’égalité des champs avec les lignes présentes en base qui leur correspondent. On peut deviner de quelle type de base de données il s’agit, ici, MySQL, on sait donc qu’il s’agit d’une requête SQL de type SELECT. On va utiliser les valeurs que l’on doit saisir de sorte qu’ils ne soient pas utilisés en temps que « valeurs » mais « commandes ». On détourne le but premier de la requête. On peut en faire ce que l’on souhaite.

Si l’on veut juste accéder à l’intranet, donc s’identifier, on peut faire l’injection la plus connue probablement.

Voici la requête initiale présente dans le code php :

$query = "SELECT \* FROM user WHERE login = '".$this->login()."' AND password = '".$this->pwd()."';";

Il nous suffit dans le WHERE d’obtenir une simple réponse VRAIE pour que, la requête retourne toute la table user !

Pour cela, on se joue des conditions de test en saisissant : **‘ OR 1=1 #**



Cela nous donne la requête suivante envoyée à MySQL :

**SELECT \* FROM user WHERE login = ‘’ OR 1=1 #’ AND password = 'QuelqueChoseHachéeSansImportance’;**

Le fonctionnement est simple, l’apostrophe referme la chaîne de caractère, le OR est ensuite interprété comme l’ajout d’une condition OU puis, la condition elle-même, 1=1 est toujours vraie ! Donc le where retourne vraie pour toutes les lignes de la table user de la base de données ! Il faut noter que # est le caractère marquant le début des commentaires donc tout ce qui suit n’est pas interprété ! CQFD

L’inconvénient d’une injection aussi simple pour accéder à l’intranet, c’est que l’on se connecte en usurpant l’identité du premier tuble présent dans la table user. Or tous les utilisateurs ne possèdent pas forcément le même niveau d’acréditation, les mêmes droits. On peut donc se permettre de paufiner cette injection pour prendre préférentiellement l’identité de l’administrateur !

* Revenons-en à l’aspect sécurité :

Si l’on s’intéresse pour finir plutôt au côté sécurité du problème, il faut donc laisser l’échappement automatique de PHP5 ou s’en charger. Puis, il est plus sécuritaire d’utiliser les requêtes préparées (de PDO par exemple) tout en sachant que la sécurité face aux injections SQL n’est pas garantie seulement par ce type de requête. D’autres paramètres sont à considérer. Dans des scénarios très spécifiques, on peut retrouver des failles.

En effet, pour les **anciennes versions de MySQL** (seulement), les requêtes préparées possèdent une vulnérabilité liée à la fonction utilisée pour l’échappement lors de simulation des requêtes préparées de PDO. Par défaut avec MySQL, PDO simule les requêtes préparées ce qui signifie que côté client, il utilise un sprintf (de la librairie C) associé à **mysql\_real\_escape\_string()**. Or, la fonction d’échappement mysql\_real\_escape\_string n’est pas sans faille. On la compare souvent à la fonction **addslashes()**. Son principal avantage vis-à-vis de addslashes() repose sur le fait qu’elle prend en compte lors de l’échappement du jeux de caractères à considérer ce qui lui permet d’échapper correctement en « casi » toute situtation. En effet, pour échapper correctement elle a besoin de connaître le jeu de caractère utilisé (multi-bits ou non) ce qui lui est normalement transmis par le pointeur de connexion de la base de données. Or sous MySQL, il y a 2 moyens de changer le jeu de caractère. On peut le changer dans le fichier de configuration MySQL (my.cnf) en mettant :

default-character-set=GBK

ou via une requête qui ne nécessite pas d’être administrateur :

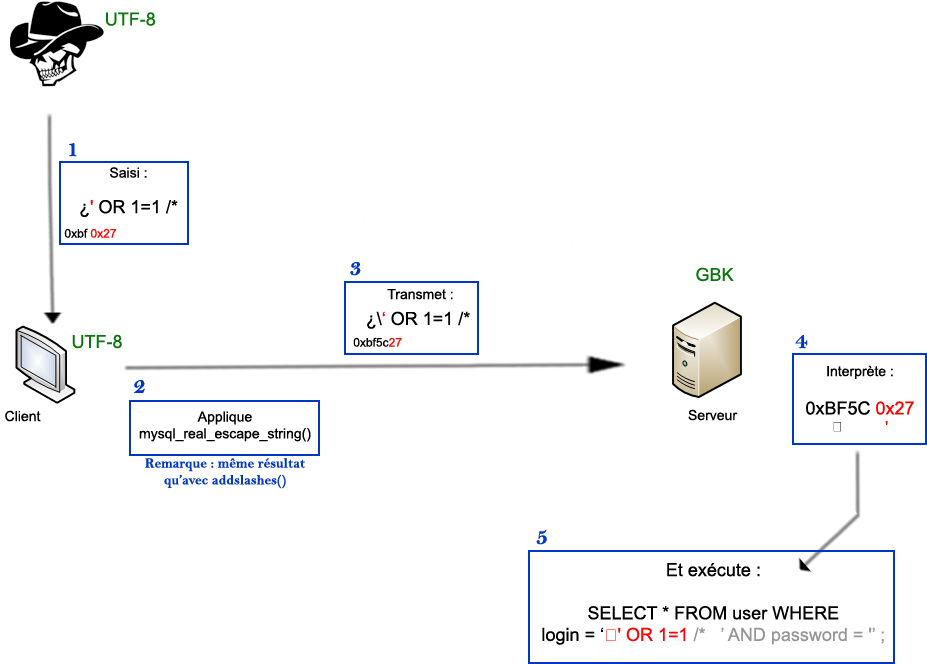
SET CHARACTER SET 'GBK'

Cette dernière méthode est problématique car elle change effectivement le jeu de caractère mais la fonction d’échappement mysql\_real\_escape\_string() continuera d’échapper en se basant sur le jeu de caractère par défaut. De faite, **mysql\_real\_escape\_string()** et **addslashes()** vont fonctionner de manière identique pour cette fois et posséder la même faille.

Nous allons détailler un cas de figure qui semble pertinent, il est décrit sur différents sites internet anglophones mais nous ne l’avons pas testé personnellement. Il nécessite un serveur MySQL attendant par défaut le jeu de caractère GBK.

Le principe exploité par les pirates est la discordance client/serveur qu’il existe entre les différents Jeux de codage de caractère (UTF-8, GBK..), certains d’entre eux n’interprètent pas correctement les caractères multi-bytes. L’astuce clef consiste donc à trouver un caractère lambda (stratégiquement choisi : ¿) et l’associer à une apostrophe. Côté client, l’échappement aura lieu en intercalant un anti-slash entre le caractère et l’apostrophe. Mais côté serveur, cela donnera seulement 2 caractères distincts (et non trois), le premier ayant le code hexadécimal correspondant au caractère lambda concaténé à l’anti-slash (caractère multi-bits existant par exemple en GBK : 뼧) et le second étant l’apostrophe !

Voici un schéma pour illustrer le principe :



Pour éviter ce genre d’injection, on peut régler les divergences de jeux de caractères entre le client et le serveur en informant l’un, l’autre. On peut aussi désactiver la simulation des requêtes préparées par PDO en utilisant :

$pdo->setAttribute(PDO::ATTR\_EMULATE\_PREPARES, false);

L’attribut *PDO::ATTR\_EMULATE\_PREPARES* « Active ou désactive la simulation des requêtes préparées ». Et on peut aussi simplement mettre à jour notre version de MySQL !