SQL injections

SQL injections are one of the most common (web) vulnerabilities. All SQL injections exercises, found here, use MySQL for back-end. SQL injections come from a lack of encoding/escaping of user-controlled input when included in SQL queries.

Depending on how the information gets added in the query, you will need different things to break the syntax. There are three different ways to echo information in a SQL statement:

- Using quotes: single quote or double quote.
- Using back-ticks.
- Directly.

For example, if you want to use information as a string you can do:

```
SELECT * FROM user WHERE name="root";
```

or

```
SELECT * FROM user WHERE name='root';
```

If you want to use information as an integer you can do:

```
SELECT * FROM user WHERE id=1;
```

And finally, if you want to use information as a column name, you will need to do:

```
SELECT * FROM user ORDER BY name;
```

or

```
SELECT * FROM user ORDER BY `name`;
```

It's also possible to use an integer as string, but it will be slower:

```
SELECT * FROM user WHERE id='1';
```

The way information is echoed back, and even what separator is used, will decide the detection technique to use. However, you don't have this information, and you will need to try to guess it. You will need to formulate hypotheses and try to verify them. That's why spending time poking around with the examples on the liveCD is so important.

Example 1

In this first example, we can see that the parameter is a string, and we can see one line in the table. To understand the server side code, we need to start poking around:

- If we add extra characters like "1234", using `?name=root1234`, no record is displayed in the table. From here, we can guess that the request uses our value in some kind of matching.
- If we inject spaces in the request, using `?name=root+++` (after encoding), the record is displayed. MySQL (by default) will ignore trailing spaces in the string when performing the comparison.
- If we inject a double quote, using `?name=root"`, no record is displayed in the table.
- If we inject a single quote, using `?name=root'`, the table disappears. We probably broke something...

From this first part, we can deduce that the request must look like:

```
SELECT * FROM users WHERE name='[INPUT]';
```

Now, let's verify this hypothesis.

If we are right, the following injections should give the same results.

- `?name=root' and '1'='1`: the quote in the initial query will close the one at the end of our injection.
- `?name=root' and '1'='1' #` (don't forget to encode `#`): the quote in the initial query will be commented out.
- `?name=root' and 1=1 #` (don't forget to encode `#`): the quote in the initial query will be commented out and we don't need the `'` in `'1'='1'`.
- `?name=root' #` (don't forget to encode `#`): the quote in the initial query will be commented out and we don't need the `1=1`.

Now these requests may not return the same thing:

- `?name=root' and '1'='0`: the quote in the initial query will close the one at the end of our injection. The page should not return any result (empty table), since the selection criteria always returns false.
- `?name=root' and '1'='1 #` (don't forget to encode `#`): the quote in the initial query will be commented out. We should have the same result as the query above.
- `?name=root' or '1'='1`: the quote in the initial query will close the one at the end of our injection. `or` will select all results, with the second part being always true. It may give the same result, but it's unlikely, since the value is used as a filter for this example (as opposed to a page only showing one result at a time).
- `?name=root' or '1'='1' #` (don't forget to encode `#`): the quote in the initial query will be commented out. We should have the same result as the query above.

With all these tests, we can be sure that we have a SQL injection. This training only focuses on detection. You can look into other PentesterLab training, and learn how to exploit this type of issues.

Example 2

In this example, the error message gives away the protection created by the developer: `ERROR NO SPACE`. This error message appears as soon as a space is injected inside the request. It prevents us from using the `' and '1'='1` method, or any fingerprinting that uses the space character. However, this filtering is easily bypassed, using tabulation (HT or `\t`). You will need to use encoding, to use it inside the HTTP request. Using this simple bypass, you should be able to see how to detect this vulnerability.

Example 3

In this example, the developer blocks spaces and tabulations. There is a way to bypass this filter. You can use comments between the keywords to build a valid request without any space or tabulation. The following SQL comments can be used: `/**/`. By replacing all space/tabulation in the previous examples using this comment, you should be able to test for this vulnerability.

Example 4

This example represents a typical mis-understanding of how to protect against SQL injection. In the 3 previous examples, using the function `mysql_real_escape_string` would have prevented the vulnerability. In this example, the developer used the same logic. However, the value used is an integer and is not echoed between single quote `'`. Since the value is directly put in the query, using `mysql_real_escape_string` does not prevent anything. Here, you only need to be able to add spaces and SQL keywords to break the syntax. The detection method is really similar to the one used for string-based SQL injection. You just don't need the quote at the beginning of the payload.

Another method to detect this is to play with the integer. The initial request is `?id=2`. By playing with the value 2, we can detect the SQL injection:

- `?id=2 #` (`#` needs to be encoded) should return the same thing.
- `?id=3-1` should return the same thing. The database will automatically perform the subtraction and you will get the same result.
- `?id=2-0` should return the same thing.
- `?id=1+1` (`+` needs to be encoded) should return the same thing. The database will automatically perform the addition and you will get the same result.
- `?id=2.0` should return the same thing.

And the following should not return the same results:

- `?id=2+1`.
- `?id=3-0`.

Example 5

This example is really similar to the previous, detection-wise. If you look into the code, you will see that the developer tried to prevent SQL injection by using a regular expression:

```
if (!preg_match('/^[0-9]+/', $_GET["id"])) {

   die("ERROR INTEGER REQUIRED");

}
```

However, the regular expression used is incorrect; it only ensures that the parameter `id` **starts** with a digit. The detection method used previously can be used to detect this vulnerability.

Example 6

This example is the other way around. The developer made a mistake in the regular expression again:

```
if (!preg_match('/[0-9]+$/', $_GET["id"])) {

   die("ERROR INTEGER REQUIRED");

}
```

This regular expression only ensures that the parameter `id` **ends** with a digit (thanks to the `$` sign). It does not ensure that the beginning of the parameter is valid (missing `^`). You can use the methods learnt previously. You just need to add an integer at the end of your payload. This digit can be part of the payload or placed after a SQL comment: `1 or 1=1 # 123`.

Example 7

Another and last example of bad regular expression:

```
if (!preg_match('/^-?[0-9]+$/m', $_GET["id"])) {

  die("ERROR INTEGER REQUIRED");

}
```

Here we can see that the beginning (`^`) and end (`$`) of the string are correctly checked. However, the regular expression contains the modifier `PCRE_MULTILINE` (`/m`). The multiline modifier will only validate that one of the lines is only containing an integer, and the following values will therefore be valid (thanks to the new line in them):

- `123\nPAYLOAD`;
- `PAYLOAD\n123`;
- `PAYLOAD\n123\nPAYLOAD`.

These values need to be encoded when used in a URL, but with the use of encoding and the techniques seen previously you should be able to detect this vulnerability.

Example 8

In this example, the parameter name gives away where it will get echoed in the SQL query. If you look into MySQL documentation, there are two ways to provide a value inside an `ORDER BY` statement:

- directly: `ORDER BY name` ;
- between back-ticks: `ORDER BY `name``.

The `ORDER BY` statement cannot be used with value inside single quote `'` or double quote `"`. If this is used, nothing will get sorted, since MySQL considers these as constants.

To detect this type of vulnerability, we can try to get the same result using different payloads:

- `name` #` (`#` needs to be encoded) should give the same results.
- `name` ASC #` (`#` needs to be encoded) should give the same results.
- `name`, `name`: the back-tick in the initial query will close the one at the end of our injection.

And the following payloads should give different results:

- `name` DESC #` (`#` needs to be encoded).
- `name`` should not give any result, since the syntax is incorrect.

Example 9

This example is similar to the previous one, but instead of back-tick ``` ``

There are other methods that can be used in this case, since we are directly injecting in the request without a back-tick before. We can use the MySQL `IF` statement to generate more payloads:

- `IF(1, name,age)` should give the same results.
- `IF(0, name,age)` should give different results. You can see that the columns are sorted by age, but the sort function compares the values as strings, not as integers (`10` is smaller than `2`). This is a side effect of `IF` that will sort values as strings if one of the column contains a string.