University of Trento

Network Security Course

SQL INJECTION

Professor:
Luca ALLODI

Students:
Linda MICHELOTTI
Elena DONINI
Michele BENOLLI
Davide CUNIAL

Academic Year 2015-2016

Contents

Chapter 1

Introduction

The aim of our project is to propose an interactive and funny laboratory, where students can learn about SQL and SQL injection and at the same time enjoy those hours. For this reason, the structure of the lab is based on a game with many and short exercises, each one characterized with a different score that depends on the degree of difficulty.
The lab experience is divided in three parts:

1. Introduction to SQL
2. Introduction to SQL injection
3. Facultative exercises on SQL injection

The goal of the first part is to permit everyone (the ones that have already studied SQL and the ones that will see SQL for the first time) to understand the topic and how a query works. In the introduction it is explained the relationship between a web page and the database and how they communicate, what is SQL, how a query works and the basic principle of SQL injection. After this short introduction, there is some theory about SQL: is explained how to create a SELECT query and are illustrated statements with WHERE, LIKE, UNION and NULL.
The second part is composed by some mandatory exercises about SQL injection, that permit to finish the lab and access the ranking. These exercises are sorted by complexity and cover the basic types of SQL injection, from the string injection type to the second order one. The first three exercises - string, numeric and union injection - exploit the absence of validation in the parameters inserted by the user. Following exercises are blind and second order injection, where prepared statements are used and it is exploited another type of vulnerability.
The last part consists of elective exercises about SQL injection, that permit to improve the score. These are more complex than the previous ones, because they combine the features exploited in the mandatory exercises, but all have a hint that helps to solve them. There are four exercises: the first one is based on string injection, the second one analyses the function addslashes(), very used on the defense side because permits to escape single quotes; the following level shows how to overcome the function addslashes() with the use of multiple-byte characters. The last exercise is based on a second order injection joined with an UNION statement.

1.1 Laboratory Structure

An entire website was built for the practical part of the laboratory, with the basic theory and exercises. Before starting the students have to sign up and log in; this permits us to manage the session giving them a variable that is associated to the numbers of exercises completed and to the score achieved. Naturally this part of the website is secure against SQL injection.

The following figure 1.1 shows how the log in page looks like.

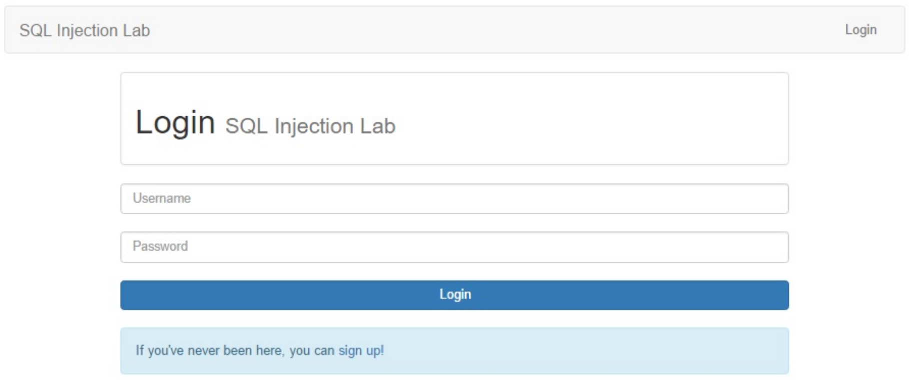


Figure 1.1: Login page

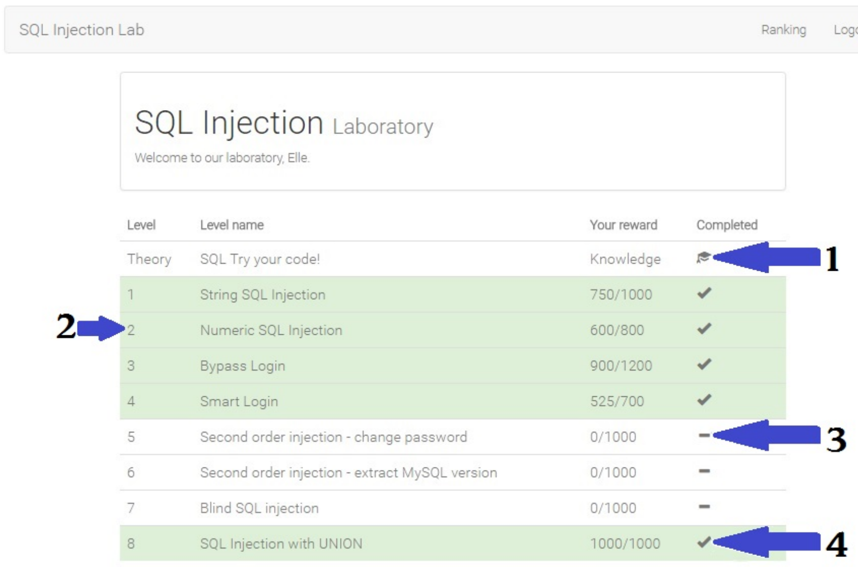
After the log in, it appears an index with the list of the exercises with all the features associated: theory classification, score of each level, specification of the type of attack, solved and not solved exercises. The following figure shows how the web page is composed: in particular, according to figure 1.2:

1. This hat stands for theory exercises, that have no score because are used in order to practice with SQL and so the word 'knowledge' substitutes the value of the exercise.

2. Enumerated list of SQL injection exercises;the theory exercise, instead, is define by the label 'Theory'.

3. Row stands for the exercises not yet completed.

4. Tic stands for the exercises completed. These are characterized by a score gained according to the difficulty of the injection and also to the hint usage.


SQL Injection Lab Ranking Logout

SQL Injection Laboratory

Welcome to our laboratory, Elle.

Level	Level name	Your reward	Completed
Theory	SQL Try your code!	Knowledge	🎓
1	String SQL Injection	750/1000	✓
2	Numeric SQL Injection	600/800	✓
3	Bypass Login	900/1200	✓
4	Smart Login	525/700	✓
5	Second order injection - change password	0/1000	–
6	Second order injection - extract MySQL version	0/1000	–
7	Blind SQL injection	0/1000	–
8	SQL Injection with UNION	1000/1000	✓

Figure 1.2: Index page of the website

1.2 Laboratory Setup

In the Malware Laboratory, there isn't internet connection and so it's not possible to access the website built. In order to overcome to this issue, we took advantage of the LAN network already present which is composed by the laptops of the lab. In other words, one of those PCs was chosen and used at the server side, instead others laptops were used by the students at the client side. In order to implement the configuration explained above, it's employed a Linux virtual machine with bridge configuration and a static IP address. Linux virtual machine contains all the website files and also MySQL structure which permits us to manage both the students and the exercises completed. Bridge configuration allows the users at the client side to access the website inside the virtual machine using the correct IP address. Forcing a static IP configuration in the Linux system, the IP address of the VM can be set freely and so is possible to interact with our website without issues.
We chose a Linux machine because permits to build a LAMP stack model which is the acronym of the four components: Linux operating system, the Apache HTTP Server, the MySQL relational database management system and the PHP programming language. Software phpMyAdmin is used in order to interact with the system.

Chapter 2

Theory about SQL

Every dynamic website has to interact with a database in order to extract the information needed for the interaction with the users. An example is the login that is required to identify the user and subsequently access to a personal section of the website. Databases are used in order to store and retrieve data. A web server can interact with a relational database through SQL statements. An example is the login process, required to identify the user. In order to store and retrieve data, web servers use databases with whom interact through SQL statements as shown in figure 2.1.

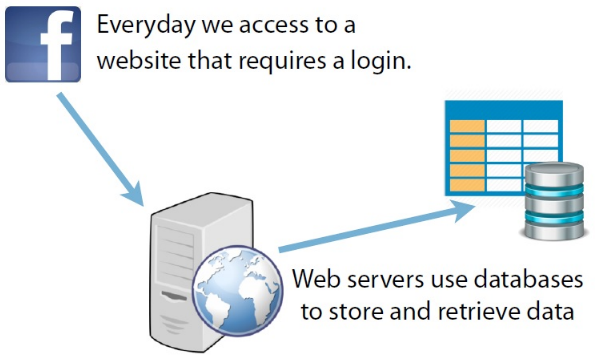

Everyday we access to a website that requires a login.
Web servers use databases to store and retrieve data

Figure 2.1: Example interaction user-server-database

2.1 SQL Language

SQL stands for Structured Query Language and it is a standard language for accessing and manipulating databases. The main actions that can be executed are data insertion, selection, update and delete, schema creation and modification, and data access control. In other words, SQL statements permit to interact with a database and manipulate, insert, extract data of the user. In the case of a login, the user sends to the server its credentials, username and password, and the server - in order to control them - has to interact with the database where this information is stored. This is done through a SELECT query which asks to the database if there is someone saved with that particular combination of name and password.

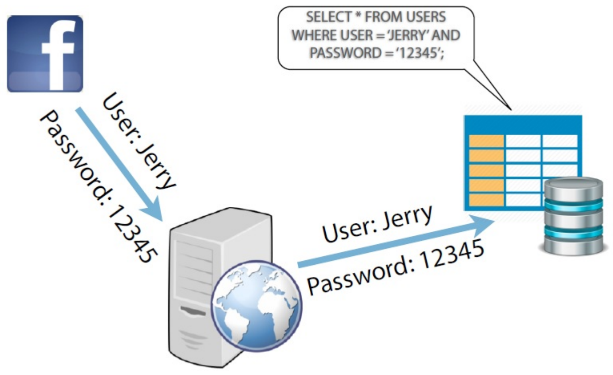


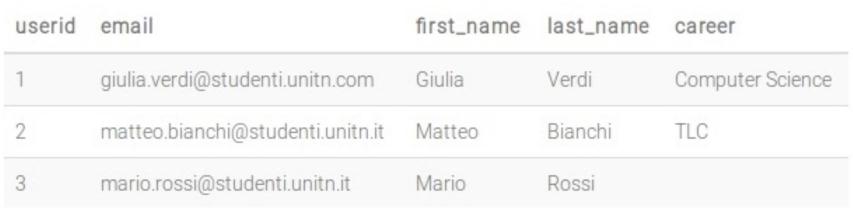
Figure 2.2: Example of a query

The code associated with this request is:

```
SELECT *
FROM table_name
WHERE username='Jerry' AND password='12345'
```

2.1.1 SELECT Query

Data and information are stored in particular structures called tables, that are identified by a name (e.g. "User" in figure 2.3) and are contained in databases.



userid	email	first_name	last_name	career
1	giulia.verdi@studenti.unitn.com	Giulia	Verdi	Computer Science
2	matteo.bianchi@studenti.unitn.it	Matteo	Bianchi	TLC
3	mario.rossi@studenti.unitn.it	Mario	Rossi	

Figure 2.3: Example of table User

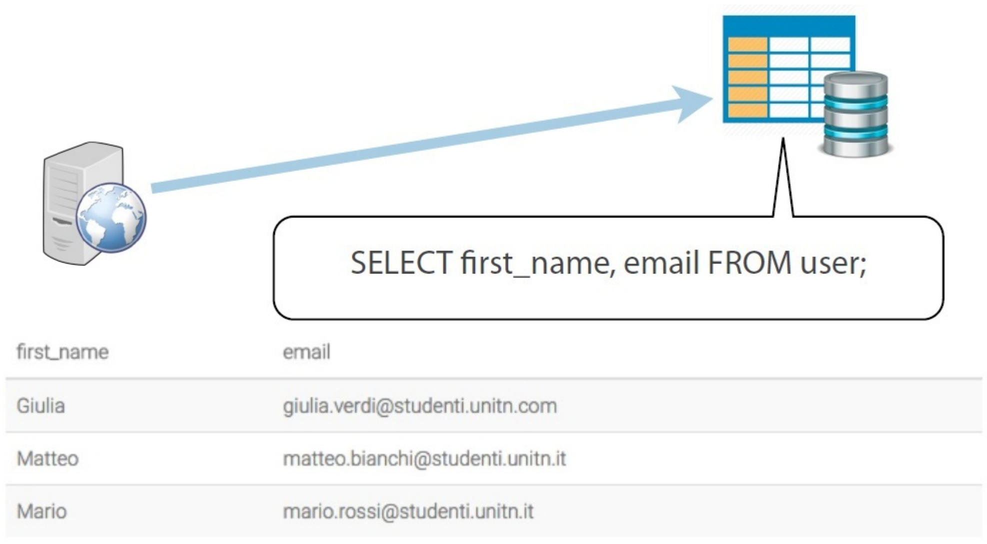
In order to extract the data from a table is used the statement SELECT where the column names that follow the select keyword determine which columns will be returned in the results.

```
SELECT column_name
FROM table_name
```

It's possible select an arbitrary number of columns, or also all the columns of the table using the star character (*) as shown in the following line of code.

```
SELECT * FROM table_name
```

EXERCISE Try to select from the user table the columns of the names and emails.


SELECT first_name, email FROM user;

first_name	email
Giulia	giulia.verdi@studenti.unitn.com
Matteo	matteo.bianchi@studenti.unitn.it
Mario	mario.rossi@studenti.unitn.it

Figure 2.4: Exercise SELECT query

2.1.2 WHERE Condition

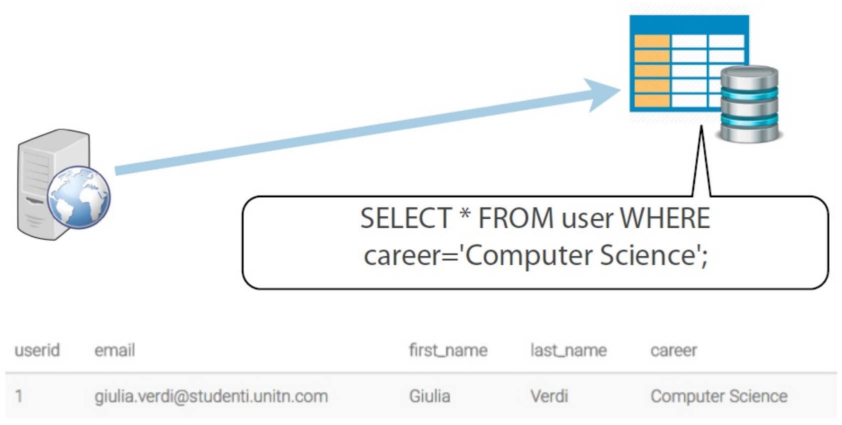
The WHERE clause permits to specify which data values or rows will be returned basing on the criteria described after the keyword where.

```
SELECT column_name
FROM table_name
WHERE column_name operator value;
```

There are different conditional selections used in where clauses, some examples are the following:

- $=$ $\longrightarrow$ Equal;
- $>$ $\longrightarrow$ Greater than;
- $<$ $\longrightarrow$ Lesser than;
- $\geq$ $\longrightarrow$ Greater than or equal to;
- $\leq$ $\longrightarrow$ Lesser than or equal to;
- $<>$ $\longrightarrow$ Not equal to.

EXERCISE Try to select from the user table the name of the Computer Science student.



```
SELECT * FROM user WHERE
career='Computer Science';
```

userid	email	first_name	last_name	career
1	giulia.verdi@studenti.unitn.com	Giulia	Verdi	Computer Science

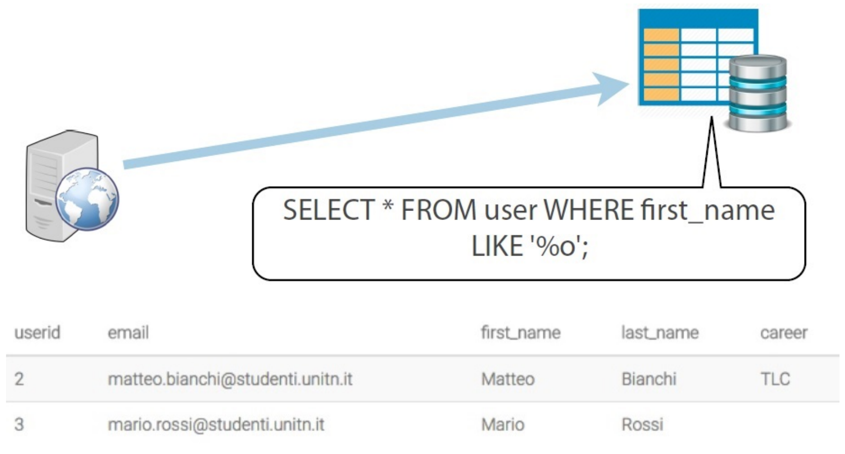
Figure 2.5: Exercise with WHERE condition

2.1.3 LIKE Operator

Another operator is LIKE, which can be used in a WHERE clause to search for a specific pattern in a column. Like is a very powerful operator that allows you to select only rows that that contain strings which are "like" what you specify.

```
SELECT column_name
FROM table_name
WHERE column_name LIKE pattern;
```

EXERCISE Try to select from the user table the names ending with letter 'o'. The sign "%" is a substitute for zero or more characters, instead "_" is a substitute for a single character.



```
SELECT * FROM user WHERE first_name
LIKE '%o';
```

userid	email	first_name	last_name	career
2	matteo.bianchi@studenti.unitn.it	Matteo	Bianchi	TLC
3	mario.rossi@studenti.unitn.it	Mario	Rossi	

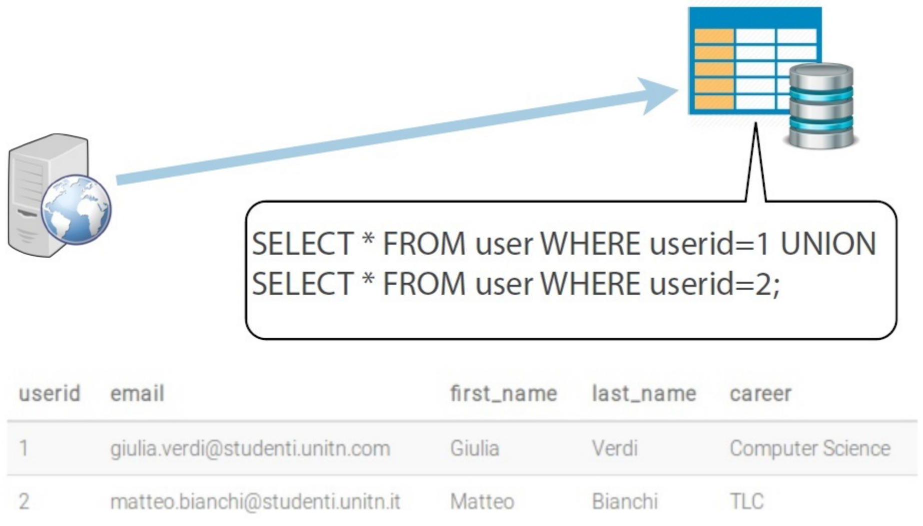
Figure 2.6: Exercise with LIKE operator

2.1.4 UNION Statement

The UNION operator is used to combine the result-set of two or more SELECT statements. Notice that each SELECT statement within the UNION must have the same number of columns and the columns must have the same type.

```
SELECT column_name_1 FROM table_name_1
UNION
SELECT column_name_2 FROM table_name_2;
```

EXERCISE Using the union operator, select records of userid 1 and userid 2.



```
SELECT * FROM user WHERE userid=1 UNION
SELECT * FROM user WHERE userid=2;
```

userid	email	first_name	last_name	career
1	giulia.verdi@studenti.unitn.com	Giulia	Verdi	Computer Science
2	matteo.bianchi@studenti.unitn.it	Matteo	Bianchi	TLC

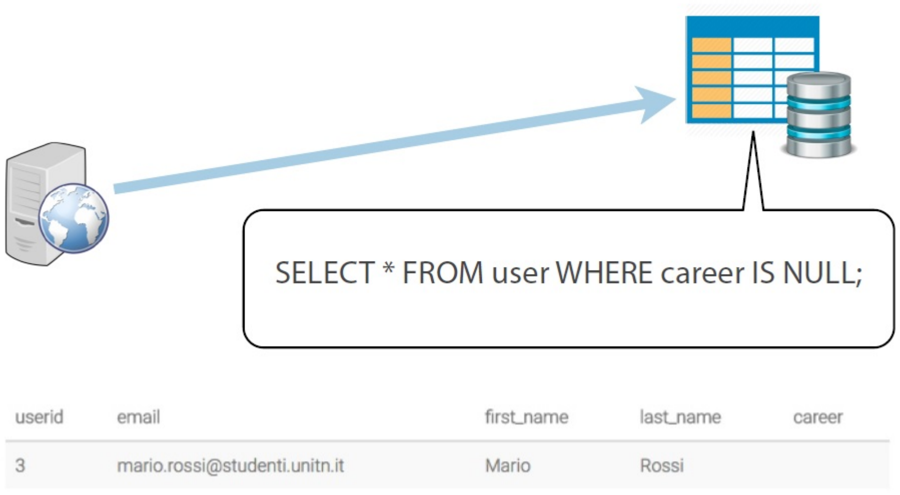
Figure 2.7: Exercise with UNION statement

2.1.5 NULL Value

If a column in a table is optional, we can insert a new record or update an existing record without adding a value to this column. This means that the field will be saved with a NULL value. NULL values are treated differently from other values. NULL is used as a placeholder for unknown or inapplicable values. It is possible to test for NULL values with IS NULL.

```
SELECT column_name
FROM table_name
WHERE column_name IS NULL;
```

EXERCISE Try to select only the records with NULL values in the career column.



```
SELECT * FROM user WHERE career IS NULL;
```

userid	email	first_name	last_name	career
3	mario.rossi@studenti.unitn.it	Mario	Rossi	

Figure 2.8: Example with NULL value.

Chapter 3

SQL Injection

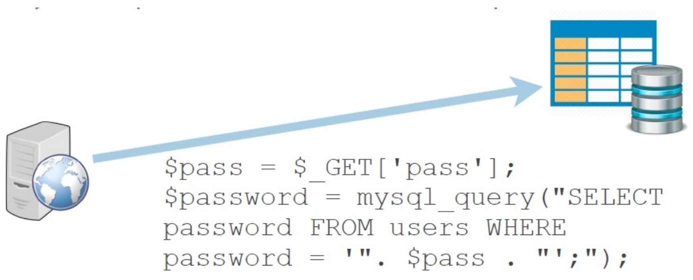
SQL Injection is a technique by which malicious users can inject SQL commands into an SQL statement via web page input. Injected SQL commands can alter SQL statements and compromise the security of a web application and the information stored in databases. In fact, the goal of this attack is to allow the client attacker to access the database and the information stored with read, write or delete privileges. There are many types of attacks, usually poorly filtered or not correctly escaped queries are exploited, derived from a particular insertion of malicious text into some input fields. One common vulnerability exploits poorly filtered strings which are not filtered for escape characters. A user can input a variable that can be passed on as an SQL statement, resulting in a database input manipulation by the end user.

Another type of vulnerability exploited is incorrect type handling, which occurs when an input is not checked for type constraints; for example when is used a numeric ID field that is numeric, but there is no filtering in place to check that the user input is numeric. With php, the function `is_numeric()` should always be used when the field type is explicitly supposed to be a number, in order to filter the input.There are many other types of injection, explained in the following pages. Due to the fact that there are many points of vulnerability between a webpage and a database, SQL injection is one of the most common hacking techniques nowadays. In the OWASP (Open Web Application Security Project) "2013 Top 10 List", database injection is at the first place because of their frequent occurrences and the easiness of the exploits.

Threat Agents	**Attack Vectors**	**Security Weakness**	**Technical Impacts**	**Business Impact**
Application Specific	Exploitability Easy	Detectability Avg, Prevalence Common	Impact Severe	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.	Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

3.1 String SQL Injection

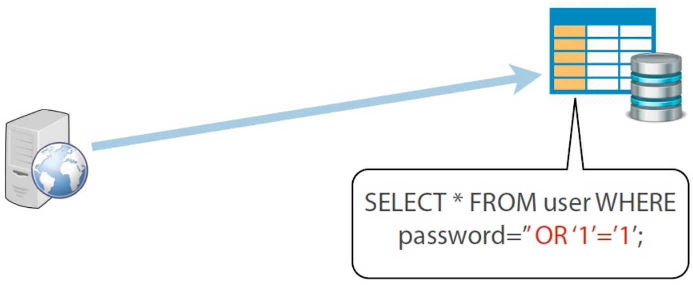
SQL injections based on poorly filtered strings are caused by user input that is not filtered to escape characters, this means that user can input a variable that can be passed on as an SQL statement. Let us consider the example, where a web server interacts with a server with the query shown in the figure 3.1.



```
$pass = $_GET['pass'];
$password = mysql_query("SELECT
password FROM users WHERE
password = '". $pass . "';");
```

Figure 3.1: Example without validation

Password inserted by the user is taken and put in the query. The webpage interacts with the database in order to check if the password is included in it. The parameter added by the user is not filtered, so the text of an injection may look something like: `’ OR ’1’ = ’1`. Because of the OR statement in the SQL query, the check for `password = $var` is insignificant, because 1 is equal to 1. The query will return TRUE, resulting in a positive login.



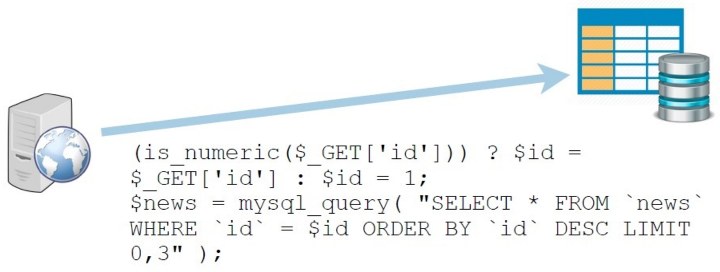
```
SELECT * FROM user WHERE
password=''OR '1'='1';
```

Figure 3.2: Example of string SQL injection

EXERCISE This is the text of the proposed exercise:
Let’s begin from us. We are an egocentric group of developers, so we designed a table with our names and our emails. If you put my name, Michele (the most egocentric of the group), in the input field, my email is returned. You have to find a way to print all the rows at once, or at least more than one.

3.2 Numeric SQL Injection

Numeric SQL injection occurs when an input is not checked for type constraints. An example of this would be an ID field that is numeric, but there is no filtering in place to check that the user input is only numeric. If it is possible the insertion of a string in place of a number, a SQL injection attack may be done. Let us consider the query in figure 3.3, where it's employed the function isnumeric() to filter the data in input. In fact this function checks if the data inserted by the user is a number: returns TRUE if it's a number, FALSE otherwise. This kind of filtering will assure that the ID field is always numeric.



```
(is_numeric($_GET['id'])) ? $id =
$_GET['id'] : $id = 1;
$news = mysql_query( "SELECT * FROM `news`
WHERE `id` = $id ORDER BY `id` DESC LIMIT
0,3" );
```

Figure 3.3: Example of code with numeric validation

EXERCISE This is the text of the proposed exercise:
The table user contains some names and emails. Each person in the table has a unique incremental numeric id. The following input field allows you to insert the id value in order to get the email. You have to find a way to print all the rows at once, or at least more than one.

3.3 SQL Injection with UNION

The UNION operator is used to combine the result-set of two or more SELECT statements. As a constraint, each SELECT statement within the UNION must have the same number of columns and the data type of the columns has to be the same or similar. Union can be very useful in the contest of an injection attack. We can take for example a harmful query which extract some information like `account_number, first_name, last_name, email` from a table called `user`, entering a numeric `user_id`. The query code is the following:

```
SELECT account_number, first_name, last_name, email
FROM user
WHERE userid= $user_id;
```

Using a UNION statement, it is possible to extract data from two different tables. It does not matter whether the entered ID is valid or not, because the second portion of the query (the injected part) is the only one really needed. An important theoretical concept to remember is that the joined results of the two queries must have the same number of columns.

EXERCISE This is the text of the proposed exercise.
Here you can enter a userid and some information will be displayed in the table. Try to extract other information about `cc_number` and `pin` from the table `creditcard` using UNION to concatenate another query.

SOLUTION The vulnerability exploited in this exercises is the missing validation of the input field. It is possible to insert a string in it, containing a SELECT query with a UNION statement just before. In this example, the table user has four columns, and the table creditcard only two; in order to fill the missing column names, it is possible to use the placeholder NULL. Injecting an appropriate string in the query, it is possible to manipulate the output in order to extract the desired values.

```
SELECT account_number, first_name, last_name, email
FROM User WHERE userid= 999
UNION
SELECT cc_number, pin, NULL, NULL
FROM creditcard;
```

3.4 Blind SQL Injection

Most good production environments do not allow to see the result of a SQL injection in the form of output error messages or extracted database fields. In order to overcome this limitation an attack called blind SQL injection may be done. This type of SQL Injection attack asks to the database true or false questions and determines the answer based on the applications response. In the case of partially blind injections, only slight changes can be seen in the resulting page: for instance, an unsuccessful injection may redirect the attacker to the main page and a successful attack returns a blank page. Totally blind injections do not produce differences in output and it is harder to determine whether an injection is actually taking place.

EXERCISE This is the text of the proposed exercise: Here you can check whether an `account_number` associated to a user is valid or nor (eg. 1515 is valid, 1234 is not). The objective of this exercise is to discover the pin associated with the `cc_number` 1111222233334444. Pins and credit card numbers are stored in another table called pins. For this query we suggest that you use parenthesis! When you are done just enter the pin in the input box to complete.

SOLUTION In order to solve this exercise, it is possible to use the error message given in output by the server query. In particular, the form permits to enter an `account_number` associated with a userID and checks its validity. The only messages shown are: 'valid userID' and 'invalid userID'. The field is not validated with any function, so it is possible to insert malicious code. In particular it is possible to concatenate two query with AND OR statements in order to ask to the database if the pin associated to `cc_number` 1111222233334444 is the one indicated. Remember how work OR and AND statement:

- 1515 AND 1=1 returns true;

- 1515 AND 1=2 returns false.

3.5 Second Order SQL Injection

3.5.1 Prepared Statements

The most operational kind of defense from SQL injection are parametric queries, in other words to use prepared statements that permit to avoid escaping all the possible malicious code. In fact, input string are evaluated as characters and not as code, so not ran. This is an example of using prepared statements:

```
<?php
//query declaration
$_query = "SELECT * FROM table WHERE name=':name' ";

//template preparation
$_stmt = $_name_database->prepare($_query);

// Variable definition for placeholder substitution
$stmt->bindValue(':name', $_username);

// instruction esecution
$_result = $_stmt->execute();
>
```

The aim of the query proposed is to extract all the data associated to the user with name equal to a value inserted by the user. In the query definition we use the placeholder :name in order to identify the place of the user input parameter. Placeholder is substituted later by using blindValue() function that permits to manage the inserted string as character and so to avoid SQL injection. But also this defense which seems quite secure can be exploited through a second order injection attack.
Second Order attack occurs when user submitted values contain malicious code are stored in the database, instead of getting executed immediately. This means that in a second moment, when a query interact with the injected parameters, will execute the code carried. This occurs because data coming from the database are trusted as they are without validation with escaping or filtering function.
Second order SQL injection is very unlikely in real world and this was only used to demonstrate the exploitation of this vulnerability.

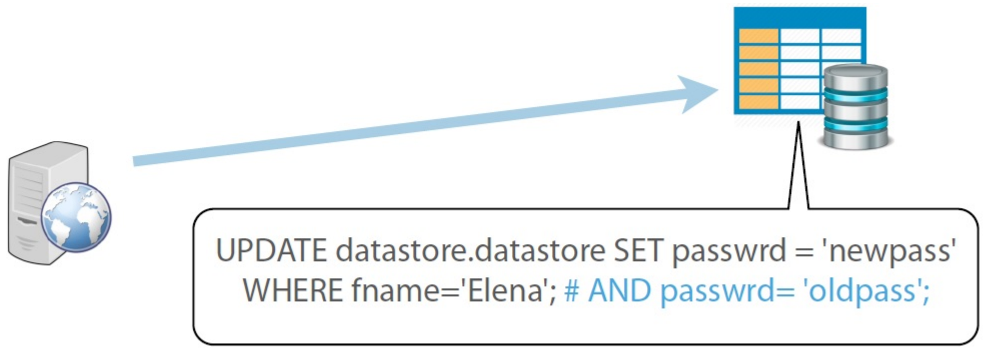
EXERCISE The aim of this exercise is to change the password of an user already registered: this can be done having privileges as administrator but you don't have them! A possible solution, it's to use second order injection: a SQL injection payload is stored in the database and then later used by some other functionality. The name of the user already registered is 'Elena' and the goal is to change her password to 'newpass'.

SOLUTION Create a new user with an appropriate first name which permits you to inject a malicious First Name in the database, for example Elena'; #

. The name is injected correctly in the database because the query ran using prepared statement and so apostrophe and hash marker are escaped. This is the query ran:

```
INSERT INTO database (username, first_name, password)
VALUES ('attacker', 'Elena';#  ',␣'Bella');
```

The second part of the attack is done changing the password of attacker with 'newpass': click on the button 'Change Password' and insert your attacker data. The query ran is the following, where the part in blue is commented: even if the password of Elena is unknown, it's possible to change it exploiting the code previously injected.



```
UPDATE datastore.datastore SET passwrd = 'newpass'
WHERE fname='Elena'; # AND passwrd= 'oldpass';
```

Figure 3.4: Second order SQL injection

Chapter 4

Optional Exercises

The last part, instead, consists of elective exercises about SQL injection that permit to improve own score. These are more complex than the previous ones, because they combine the features exploited in the mandatory exercises, but all of them have a hint that helps to achieve. There are four exercises: the first one is based on string injection; the second one analyses function addslashes() very used in defense because permits to escape single quote; the following one shows how to overcome function addslashes() with multiple-byte characters. The last one, instead, is based on a second order injection joined with an UNION statement.

4.1 Bypass Login

This exercise is based on string SQL injection where there is no validation on the input parameters. The main difference respect to previous exercises is due to the fact that in the form there are two fields one regard the email and the other the password. The values put in these two field have HTML constraints: in fact they must be type email and password. This means that the email field there have to be a string with (at) and (dot) characters.

EXERCISE This is the text of the exercises:
The following log in form has some vulnerabilities. Try to get access with your nonexistent combination of username and password!

SOLUTION In order to solve this exercise, a fake email has to put in the email feild, for example **sql@injection.com**. Instead the string injection has to be entered in the password field.

4.2 Addslashes Exercise

Function addslashes() permits to validate the string in input avoiding malicious code: in fact, this function escape the power of the apostrophe and also double quotes by putting a backslash before them. Let us consider the following example where the aim is to extract the data regarding a user identify by the parameter first_name inserted by the user.

```
<?php
$_input = " ' OR TRUE";
$_input //Case 1: not safe in a database query
addslashes($_input) //Case 2: safe in a database query
$query="SELECT * FROM table WHERE name='".$_input."'";
>
```

- Case 1: no validation of the inserted values with addlashes(). By putting the value of the variable $input in the query, all the mail in the table are selected.

  ```
  SELECT * FROM table WHERE name=' ' OR TRUE;
  ```

- Case 2: validation of the values with addlashes() function. Using an escaping function, only the mail corresponding to / user is selected, this means that no mail is selected.

  ```
  SELECT * FROM table WHERE name='\' OR TRUE;
  ```

Other type of defenses used are the following.

- **sprintf()** can be used with conversion specifications to ensure that the dynamic argument is treated the way it's supposed to be treated. For example, if a call for the users' ID number were in the string, %d would be used to ensure the argument is treated as an integer, and presented as a (signed) decimal number.

- **htmlentities()** in conjunction with the optional second parameter, allows the use of ENT_QUOTES, which will convert both double and single quotes. This will work in the same sense as addslashes() in regards to quotation marks, however, instead of prepending a backslash, it will use the HTML entity of the quotation mark.

EXERCISE Let's take a look about security: we have to sanitize our input fields. In PHP we can use a string replacement method or – for example - the function `string addslashes (string $str)`. It returns a string with backslashes before characters that need to be escaped: single and double quote, backslash and the NULL byte. In this level you have to use this wonderful function, in order to prevent the SQL injection attack.

4.3 Smart Login

The vulnerability found regard the GBK mapping code of the character and permits to avoid the validation done by the function addslashes(), so to inject malicious code in the query. Usually a letter is encoded in 8 bit, this means that we can represent 256 unique values, but in some alphabet (as GBK) are employed multi-byte character encoding to express more than 256 letters. If it's not applied a multibyte-aware function (as in our case), it's not possible to determine correctly the beginning and the ending of a string. The following exercise makes visible that addslashes() is not enough to sanitize an input string; in fact, with a smart use of multi-byte characters, we can construct a single quote exploiting the way in which addslashes() works.

EXERCISE This login form simply requires a password to enter in the magic world of the administrators. But is a smart login form the input is sanitized with the function addslashes(), which returns a string with backslashes before characters that need to be escaped. These characters are single quote ', double quote ", backslash and the NULL byte. Try to insert a query with prohibited characters! You can exploit poorly coded websites that make use of addslashes() if their database uses the GBK char-set, common in China. Wonderful, this is the case! The way in which we can circumvent addslashes()'s protection is using multi-byte characters.

SOLUTION The query ran in the php file is the following:

```
<?php
$username = addslashes($username);
$query = "SELECT * FROM users WHERE username = '$username'";
$result = mysqli_query($query);
>
```

In order to exploit the vulnerability, a username like this has to be inserted: **chr(0x87) " .' OR TRUE; –** . The function addslashes() tries to escape the single quote before the OR operator by inserting a backslash (\ or 0x5C), thus, successfully constructing our "grumble" character (0x875C). There is no backslash escaping our single quote now.
Instead of chr(0x87) we would insert any GBK character whose rightmost byte is 0x87. Why? Because the byte sequence would be something like 0x?? + 0x87 of our special GBK character plus 0x5C of addslashes()'s backslash, ending up with 0x??875C which means a successful consumption of that backslash. But why does that special GBK character has to end in 0x87? It does not always have to be like this, just look for any character that ends in 0x5C, look at its first byte, and the character you are looking for is any one that ends in that byte.
Does this happen in UTF-8? Nope; this encoding does not have characters that end in 0x5C.

4.4 Second Order SQL Injection

The aim of this exercise is to extract meaningful data from the database using a second order injection, in particular the version of the system used (MySQL and UBUNTU). That can be done through a second order injection: as explain above, some data, like the value of first name, are extracted from the database without validation. If something is injected in the first name field, it's possible to extract some meaningful information.

EXERCISE The aim of this exercise is to extract some from data from the database: this can be done having privileges as administrator but you don't have them! A possible solution, it's to use second order injection: a SQL injection payload is stored in the database and then later used by some other functionality. Extract the version of the system using this function SELECT VERSION() in the injection.

SOLUTION In this exercise there are two form, the first one permits to create a user and the second one to see the characteristics associated to an user. So it's possible to guess that the first form uses an INSERT/UPDATE query, instead the second uses a SELECT query. This means that it's possible to insert in the first form an UNION statement in order to extract the the version of the system.
Create a new user with an appropriate first name which permits you to inject a malicious First Name in the database. Remember also in this case, that the number of the columns returned by the two queries has to be the same: the first query returns three value (username, first name and password); instead SELECT VERSION() function only one. For example it can be injected aa' UNION SELECT VERSION(),2,'r.
The name is injected correctly in the database because the query ran using prepared statement and so apostrophe and hash marker are escaped. This is the query ran:

```
INSERT INTO database (username, first_name, password)
VALUES ('attacker','aa' UNION SELECT VERSION(),2,'r' ,
 'Bella');
```

The second part of the attack consists to run the injected code using the select form: when someone tries to select the data associated to the user inserted previously, the query ran is the following.

```
SELECT * FROM database
WHERE first_name='aa' UNION SELECT VERSION(),2,'r';
```

The output of this query are the data associated to the username 'aa' and the version of the system.

Chapter 5

Conclusion

The aim of our lab was to transmit something about SQL theory and injection to the students, for this reason the lab is composed by many short exercises. In order to monitor the results of the students and to make the lab funny, at each exercises completed is associate a score depending on the usage of the hint.
In the final ranking, many of the students have reach the maximum score, instead the others achieve something less because of the hints' usage. In conclusion, all the students learned something about programming in SQL and the basis of SQL injection.

During the lab, we saw that all the students try to understand and complete all the exercises by their-selves. In particular, the lab was structured in this way: the exercises regarding SQL theory was done together, one of us was explaining using slides, instead the others were between the student answering to questions.
Instead for the mandatory exercises of SQL injection we chose to adopt another technique: the aim of the exercises and the vulnerability exploited were explained briefly. Then the students completed the exercise using hints and asking to us. When all of them had finished mandatory exercises, the game was on and they tried to solve optional ones by their own using also in this case hint and asking to us.
Analyzing received questions, it's possible to say that the students were very careful and interested in what were explained. As it could expect, there were many well working groups and some that had same difficulty. But we tried to explain and to repeat them the concept many time, this permitted them to complete also the mandatory exercises.

During the lab all the basics types of SQL injection exercises were seen, but not all them are common in real word of SQL injection. In fact in real website, the entered values are always validate and sanitized. This means that only blind and second order injection are frequently used: pay attention that in order to use second order injection you have to know very well the query ran.

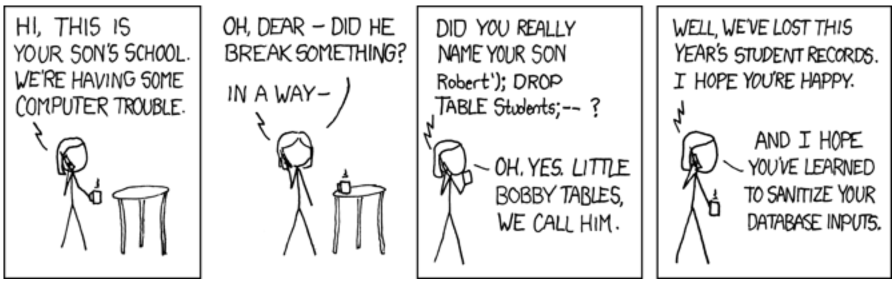

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 5.1: A joke, to conclude with some fun!

References

[1] Slides of Network Security course, prof. Allodi.

[2] LAMP tutorial: https://www.digitalocean.com/community/tutorials/how-to-install-linux-apache-mysql-php-lamp-stack-on-ubuntu-14-04

[3] PhpMyAdmin tutorial: https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-phpmyadmin-on-ubuntu-14-04

[4] SQL theory: http://www.w3schools.com/sql/

[5] PHP theory: http://www.w3schools.com/php/default.asp

[6] Top 10 attacks ranking in 2013: https://www.owasp.org/index.php/Top_10_2013-A1-Injection

[7] SQL Injection: http://www.dis.uniroma1.it/~damore/was/slides/sqlinjectionENG.pdf

[8] SQL Injection: https://www.owasp.org/index.php/SQL_Injection

[9] SQL Injection Exercises: http://www.unixwiz.net/techtips/sql-injection.html

[10] SQL Injection Exercises: http://www.cis.syr.edu/~wedu/seed/Labs/Attacks_SQL_Injection/SQL_Injection.pdf

[11] Numeric SQL injeciton http://php.net/manual/en/function.is-numeric.php

[12] Prepared Statements: http://dev.mysql.com/doc/refman/5.7/en/sql-syntax-prepared-statements.html

[13] Second order injection: https://haiderm.com/second-order-sql-injection-explained-example/

[14] Blind SQL injection: https://www.owasp.org/index.php/Blind_SQL_Injection

[15] Addslashes function: http://php.net/manual/en/function.addslashes.php

[16] Smart Login: `https://epadillas.wordpress.com/2012/12/29/multibyte-sql-injection-mysql-and-php-case-study/`

[17] Second order injection: `http://www.esecforte.com/second-order-sql-injection/`