

# SQL INJECTION



Group #7



SQL

Elena Donini  
Linda Michelotti  
Michele Benolli  
Davide Cunial

# LAB OBJECTIVES

The main objective of this lab is to understand how SQL INJECTION works.

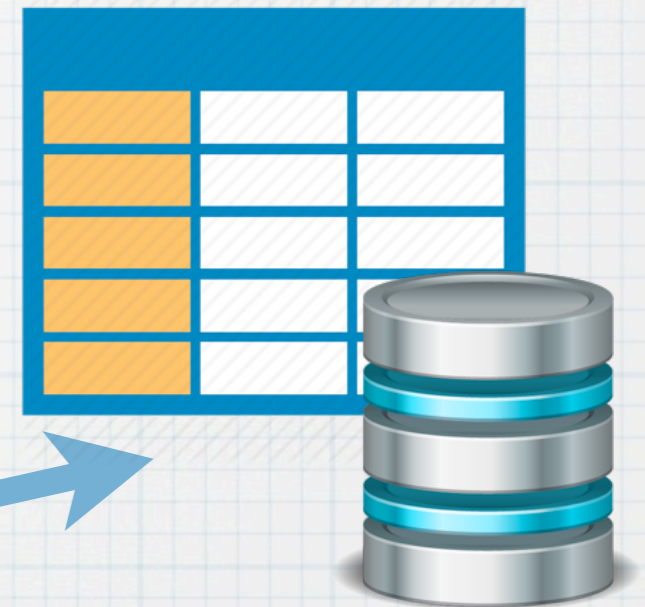
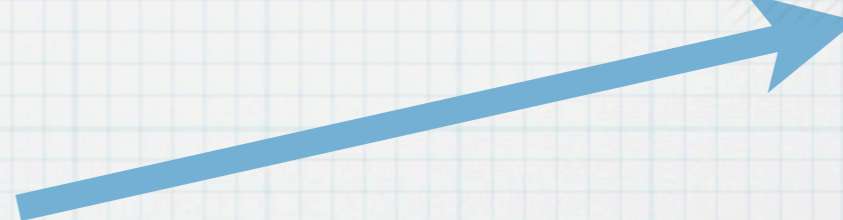
The laboratory is divided into the following sections:

- \* Introduction to SQL
- \* Introduction to SQL injection
- \* Exercises

# WEB and DATABASE



Everyday we access to a website that requires a login.



Web servers use databases to store and retrieve data

# SQL



- \* SQL stands for Structured Query Language.
- \* It is a standard language for accessing and manipulating databases.
- \* The aim of SQL includes data insertion, query, update and delete, schema creation and modification, and data access control.

# QUERY

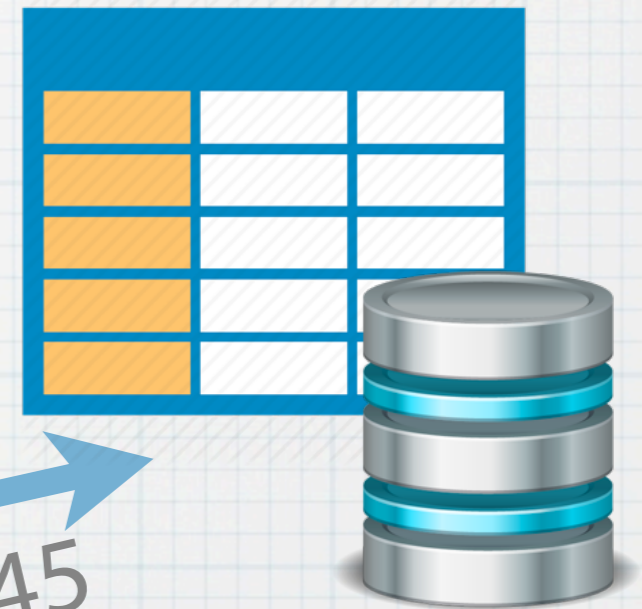


User: Jerry  
Password: 12345



```
SELECT * FROM USERS  
WHERE USER = 'JERRY' AND  
PASSWORD = '12345';
```

User: Jerry  
Password: 12345



# SQL Injection

- \* SQL Injection is a technique where 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.
- \* One of the most common type of vulnerabilities in real world is SQL Injection.



# Lab structure

- \* For this lab we built a web page with the basic theory and some exercises.
- \* Before starting, you have to sign up and login.

SQL Injection Lab

Login

Login SQL Injection Lab

Username

Password

Login

If you've never been here, you can sign up!

# Lab structure


SQL Injection Lab

[Ranking](#) [Logout](#)

## SQL Injection Laboratory

Welcome to our laboratory, Abcd.

This symbol indicates theory and exercises about SQL.

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	1200/1200	✓
4	SQL Injection Union	0/1000	-
5	Blind SQL injection	0/1000	-



# Lab structure

SQL Injection Lab


Ranking

Logout

## SQL Injection Laboratory

Welcome to our laboratory, Abcd.

This is the list of exercises about SQL injection.

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	1200/1200	✓
4	SQL Injection Union	0/1000	—
5	Blind SQL injection	0/1000	—

# Lab structure





SQL Injection Lab

Ranking Logout

## SQL Injection Laboratory

Welcome to our laboratory, Abcd.

These are the exercises that you have to do or that you have not completed yet.

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	1200/1200	
4	SQL Injection Union	0/1000	-
5	Blind SQL injection	0/1000	-

# Lab structure

SQL Injection Lab


Ranking

Logout

## SQL Injection Laboratory

Welcome to our laboratory, Abcd.

These are the completed exercises.

Level	Level name	Your reward	Completed
Theory	SQL Try your code!	Knowledge	
1	String SQL Injection	750/1000	<input checked="" type="checkbox"/>
2	Numeric SQL Injection	600/800	<input checked="" type="checkbox"/>
3	Bypass Login	1200/1200	<input checked="" type="checkbox"/>
4	SQL Injection Union	0/1000	<input type="checkbox"/>
5	Blind SQL injection	0/1000	<input type="checkbox"/>

# Lab structure

SQL Injection Lab

Ranking Logout

## SQL Injection Laboratory

Welcome to our laboratory, Abcd.

For every exercise there is a score. If you ask for the hint the score decreases.

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	1200/1200	✓
4	SQL Injection Union	0/1000	—
5	Blind SQL injection	0/1000	—

# SQL THEORY

---

# SQL

- \* A database contains tables identified by a name (e.g. "User"). The data or information of the database are stored in these tables.
- \* The actions you need to interact with a relational database are done with SQL statements.

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	

# SQL SELECT

- \* The **SELECT** statement is used to select data from a database.
- \* The column names that follow the select keyword determine which columns will be returned in the results.
- \* You can select as many column names as you like, or you can use a \* to select all columns.

```
SELECT column_name, column_name FROM  
table_name;
```

```
SELECT * FROM table_name;
```

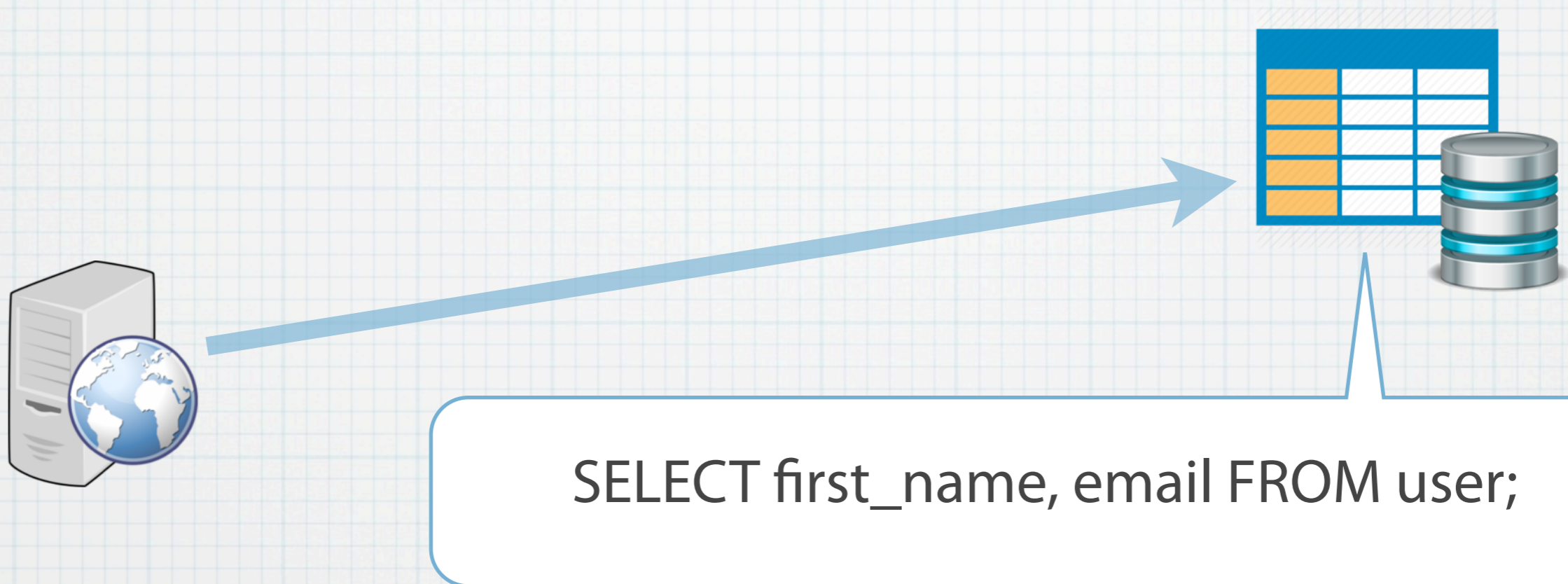
# SQL SELECT

- \* Try to select from the **user** table the columns of the names and emails.



# SQL SELECT

- \* Try to select from the **user** table the columns of the names and emails.



first\_name

email

Giulia

giulia.verdi@studenti.unitn.com

Matteo

matteo.bianchi@studenti.unitn.it

Mario

mario.rossi@studenti.unitn.it

# SQL WHERE

- \* The **WHERE** clause specifies which data values or rows will be returned, based on the criteria described after the keyword where.
- \* There are different conditional selection used in where clauses, for example:
  - = Equal
  - > Greater than
  - < Less than
  - >= Greater than or equal
  - <= Less than or equal
  - <> Not equal to

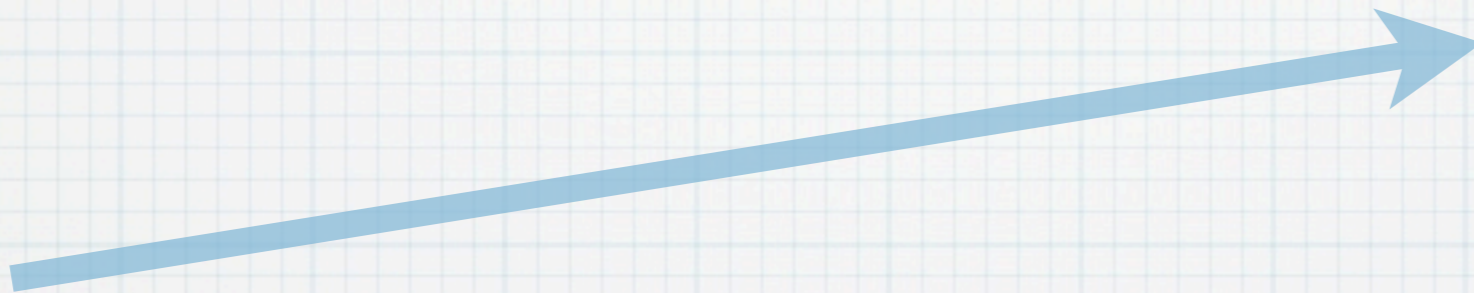
```
SELECT column_name FROM table_name WHERE  
column_name operators value;
```

# SQL WHERE

- \* Try to select from the **user** table the name of the Computer Science student.

# SQL WHERE

- \* 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

# SQL LIKE

- \* Another operator is **LIKE**: it is 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 contain strings which are "like" what you specify.

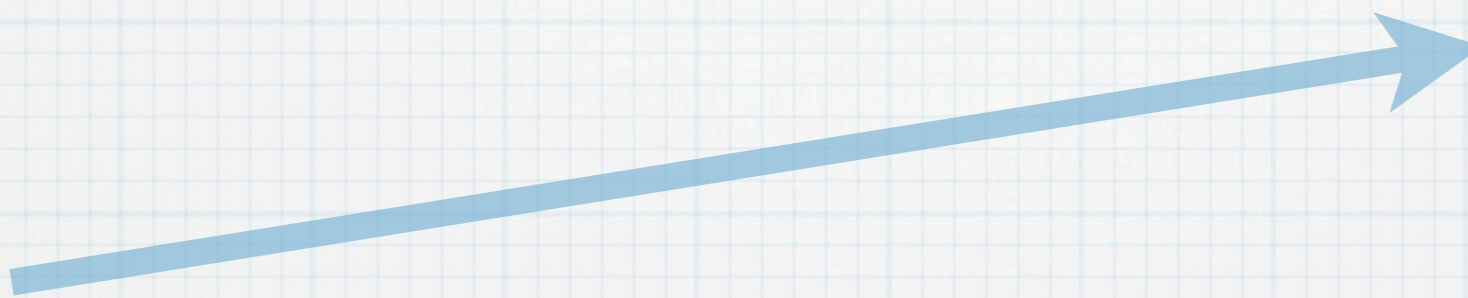
```
SELECT column_name FROM table_name WHERE  
column_name LIKE pattern;
```

# SQL LIKE

- \* 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.

# SQL LIKE

- \* 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	

# SQL UNION

- \* 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 * FROM table_name_1 UNION SELECT *  
FROM table_name_2;
```

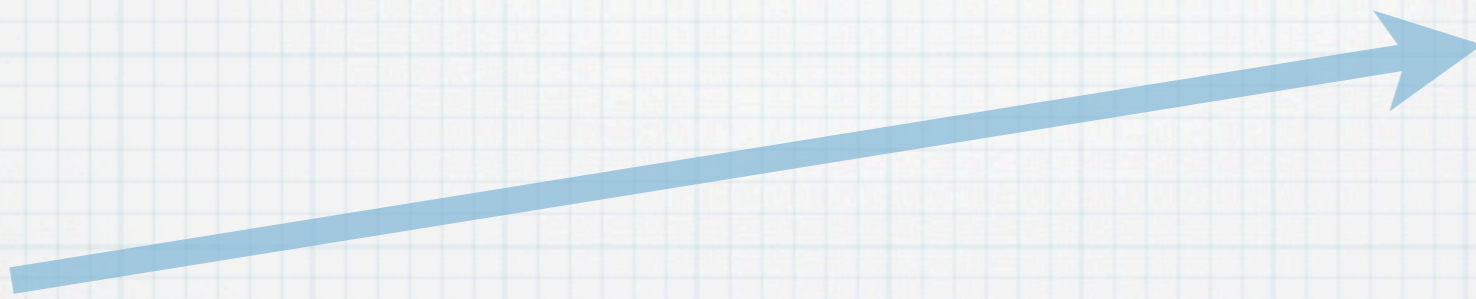


# SQL UNION

- \* Using the union operator, select records of userid 1 and userid 2.

# SQL UNION

- \* 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

# SQL NULL

- \* 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**.

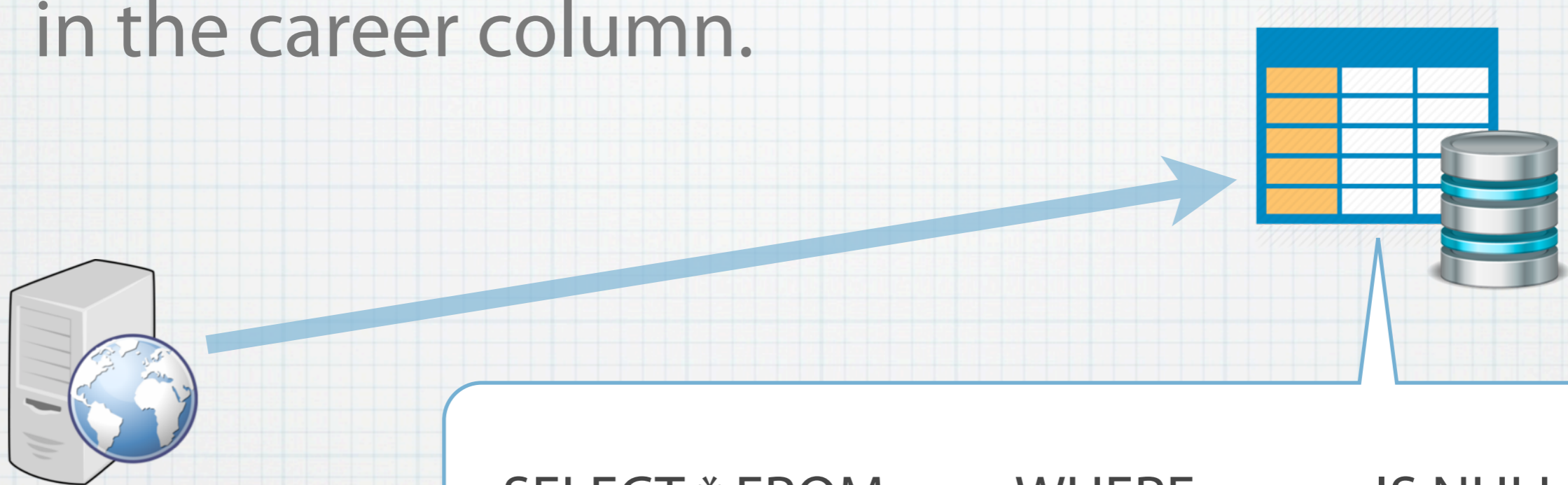
```
SELECT * FROM table_name WHERE column_name  
IS NULL;
```

# SQL NULL

- \* Try to select only the records with NULL values in the career column.

# SQL NULL

- \* 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	

It seems to be secure...  
but it is not!

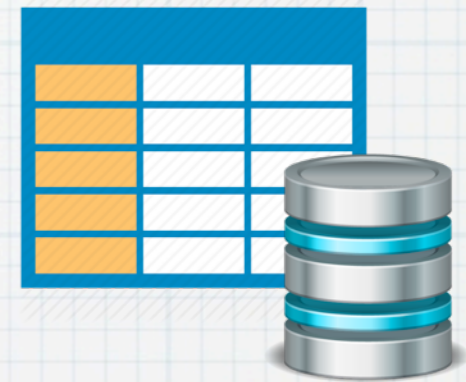
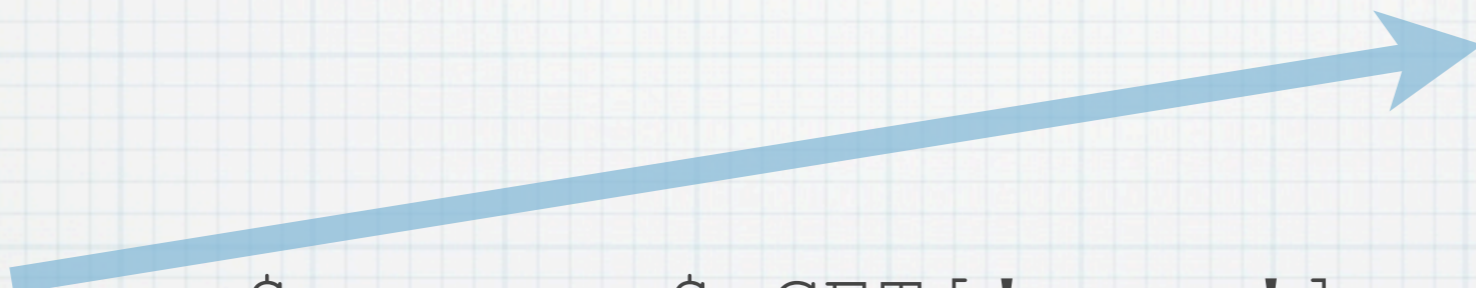


# SQL INJECTION THEORY

---

# STRING SQL INJECTION

- \* SQL injections based on poorly filtered strings are caused by user input that is not filtered to escape characters.

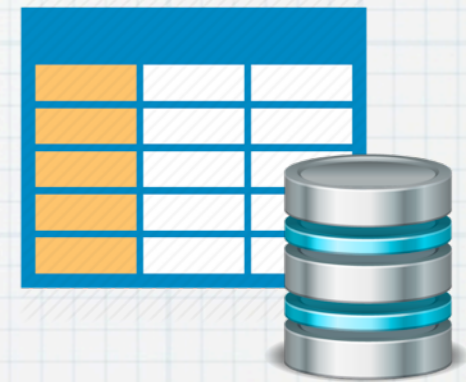


```
$pass = $_GET['pass'];  
$password = mysql_query("SELECT  
password FROM users WHERE  
password = '". $pass . "'");
```



# STRING SQL INJECTION

- \* SQL injections based on poorly filtered strings are caused by user input that is not filtered to escape characters.



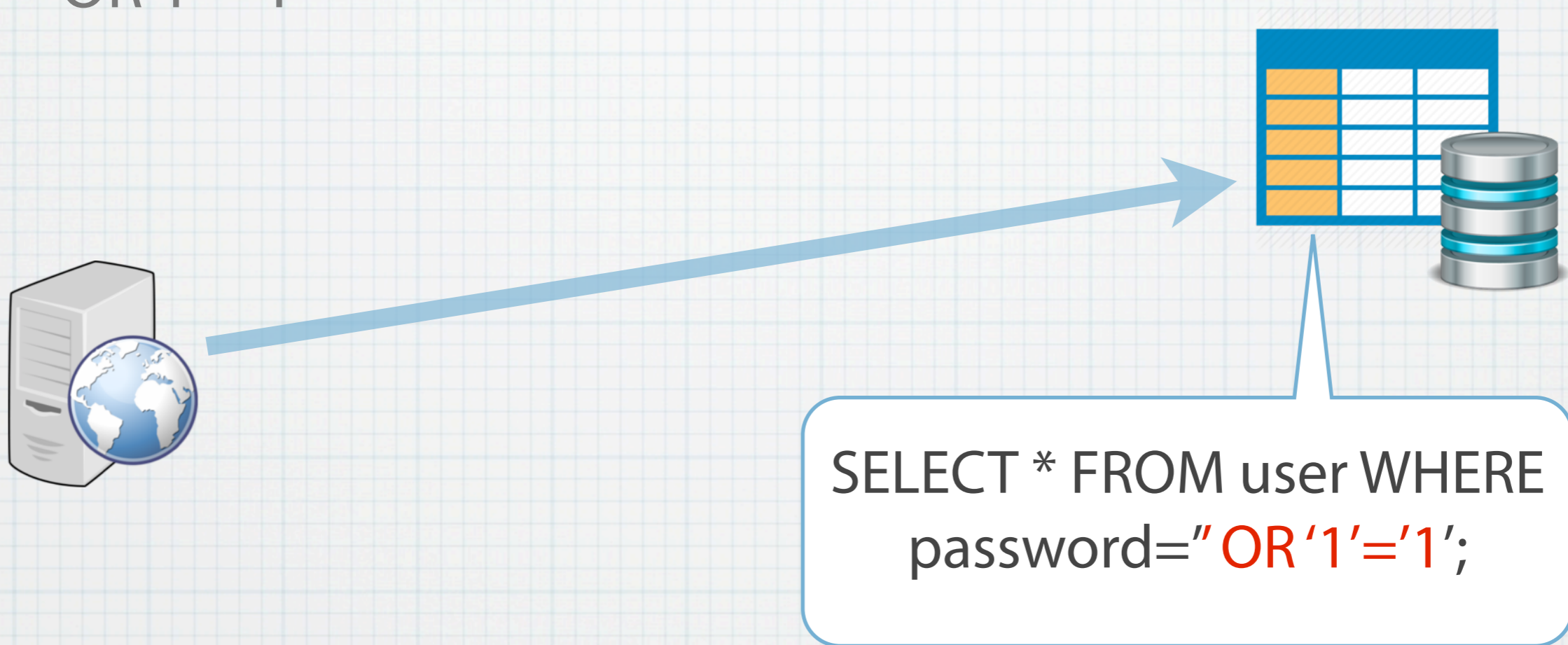
```
$pass = $_GET['pass'];  
$password = mysql_query("SELECT  
password FROM users WHERE  
password = '" . $pass . "'");
```

Take the password  
inserted from the user

Give the password to the  
database, which checks if the  
password is included in it.

# STRING SQL INJECTION

- \* For this reason, 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 as 1 does equal 1.
- \* The query will return TRUE, resulting in a positive login.

# String SQL Injection



Text of 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.

# Bypass Login



Text of exercise:

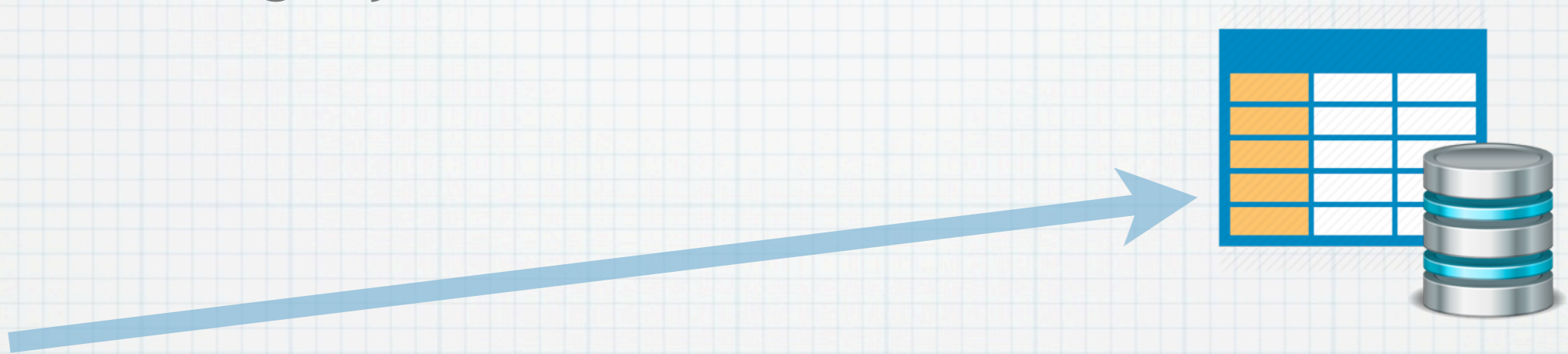
- \* The following login form has some vulnerabilities. Try to get access with your inexistent combination of username and password!

# NUMERIC SQL INJECTION

- \* Incorrect type handling based SQL injections occur 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.

# NUMERIC SQL INJECTION

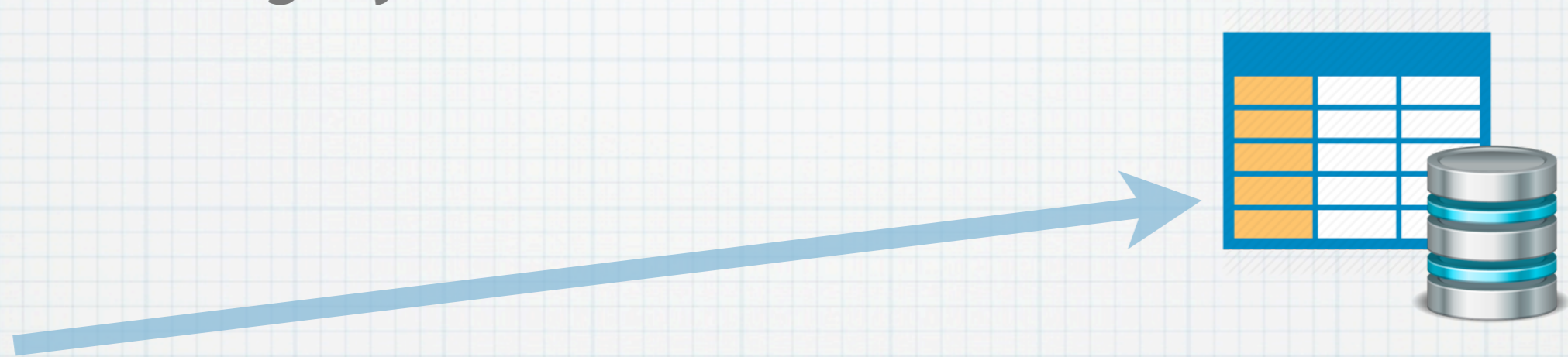
- \* An example of code that will not be subject to incorrect type handling injection is:



```
(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" );
```

# NUMERIC SQL INJECTION

- \* An example of code that will not be subject to incorrect type handling injection is:

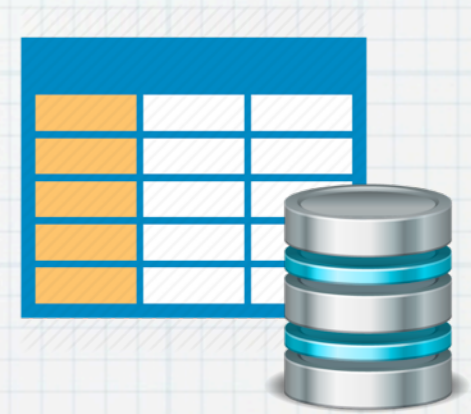
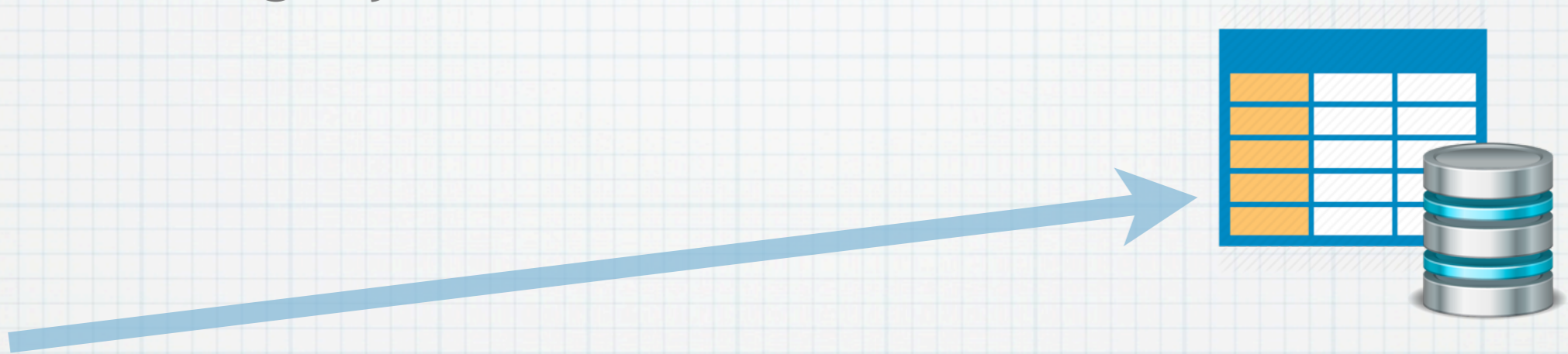


```
(is_numeric($_GET['id'])) ? $id =  
$_GET['id'] : $id = 1;  
$news = mysql_query( "SELECT * FROM `news`  
WHERE `id` = $id ORDER BY `id` DESC LIMIT  
0,3" );
```

checks that `$_GET['id']` is a number

# NUMERIC SQL INJECTION

- \* An example of code that will not be subject to incorrect type handling injection is:



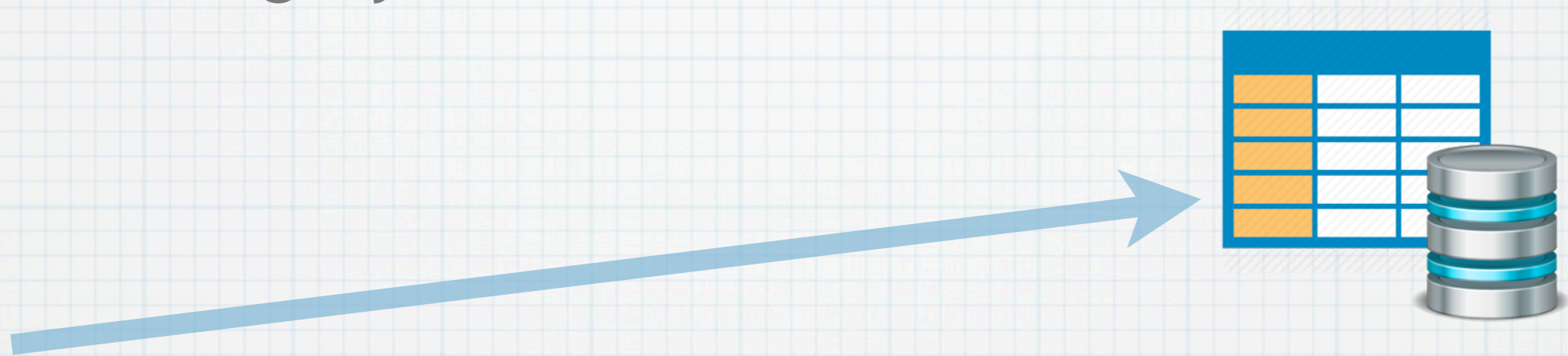
```
(is_numeric($_GET['id'])) ? $id =  
$_GET['id'] : $id = 1;  
$news = mysql_query( "SELECT * FROM `news`  
WHERE `id` = $id ORDER BY `id` DESC LIMIT  
0,3" );
```

if TRUE returns \$id = \$\_GET['id'],  
and if FALSE sets \$id to 1.



# NUMERIC SQL INJECTION

- \* An example of code that will not be subject to incorrect type handling injection is:



```
(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" );
```

- \* This kind of filtering will assure that the ID field is always numeric.

# Numeric SQL Injection



Text of 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.

# Second Order Injection

- \* **Second Order Sql injection** occurs when user submitted values contain SQL injection attacks that are stored in the database, instead of getting executed immediately.
- \* Data coming from the database are trusted as they are without validation with escaping or filtering function. Developers should filter the data either coming from users or the retrieved from the database.

# Second Order Injection

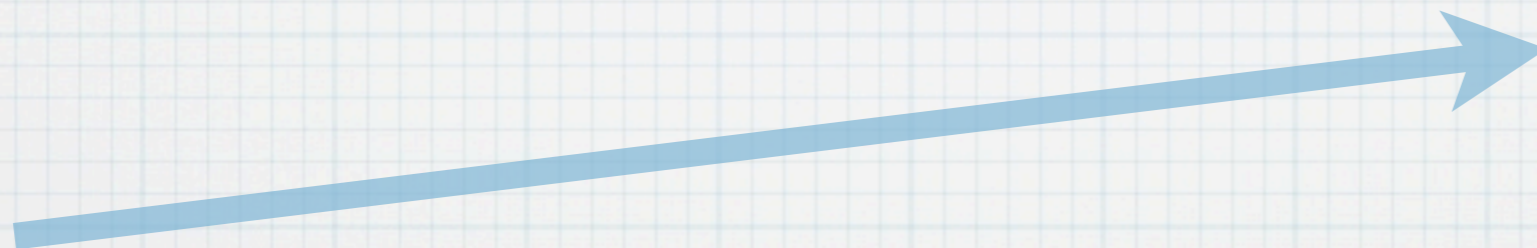


Text of exercise:

- \* The aim of this exercise is to change the password of an user already registered. 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 it password to 'newpass'. Create a new user with an appropriate first name which permits you to inject a malicious. First Name in the database, for example «Elena'; #».

# Second Order Injection

- \* The second part of the attack is to change the password of Elena with «newpass».
- \* You have to click on the button to change password and insert your attacker data.



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

# SQL INJECTION EXERCISES

---

# SQL Injection with UNION



Text of 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 to concatenate another query.



# SQL Injection with Union

- \* For this exercise we use UNION to merge columns from two different tables.
- \* First by entering a userid in the input box we have information about: account\_number, first\_name, last\_name and email.
- \* We can guess that the query will be something like:

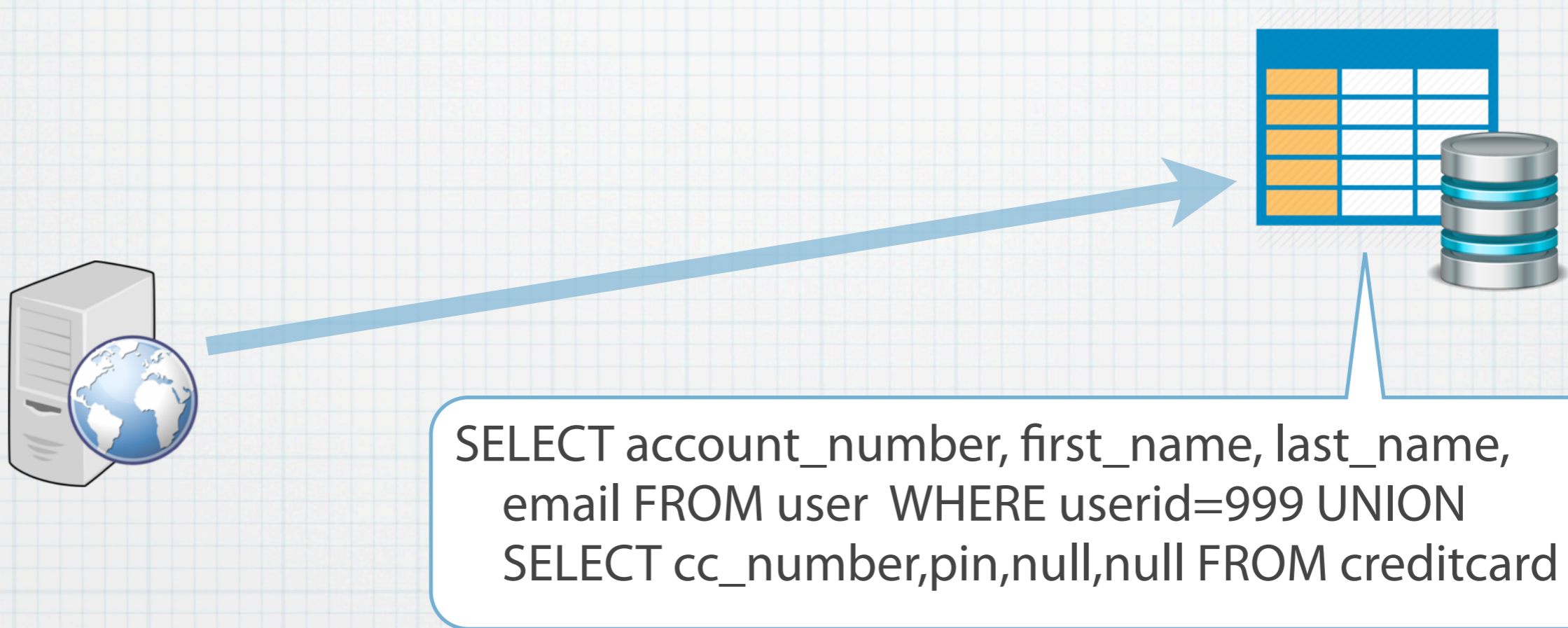


```
SELECT
account_number, first_name,
last_name, email FROM user
WHERE userid=$id
```



# SQL Injection with Union

- \* With a UNION statement we can also insert data from another table into our output, by entering something like:



- \* It doesn't matter whether the ID we enter is valid or not since we are only interested in the second part of the query.

# Blind SQL Injection

- \* Here the SQL Injection is called BLIND because we cannot see the output for our query.
- \* Blind SQL injection is a type of SQL Injection attack that **asks the database true or false questions** and determines the answer based on the applications response.

# Blind SQL Injection



Text of exercise:

- \* Here you can check whether an `account_number` associated with a user is valid or not (eg. 1515 is valid, 1234 is not). The objective of this exercise is to also discover the pin associated with the `cc_number` 1111222233334444 (it's a String!). 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.

# Blind SQL Injection

- \* AND/OR statements are very useful to concatenate additional code to our query, for example:  
1515 AND 1=1 will return true, while  
1515 AND 1=2 will return false!
- \* Instead of asking if 1=1 to solve this exercise we can ask something about the pin associated with our cc\_number = '1111222233334444'.

# Second Order Injection



Text of exercise:

- \* The aim of this exercise is to extract meaningful data from the database using a second order injection. In particular it's possible to exploit the fact that the first name value extracted from the query is used in another query without validation.
- \* We will have to inject a SQL query in the first name field; make sure that the query is correctly formed and then use «Selection of data» to run the injected code.
- \* For example to extract the version of the system used, try to put «select version()».

# Second Order Injection

- \* The goal of the first part of the attack is to inject some malicious code in the database during the login part. In particular, in the First Name field it has to be put the code string that permit to extract the version of the system used.
- \* The goal of the second part of the attack is to run the injected code by the selecting form.
- \* Query ran in the db in the select form is:  
`SELECT*FROM datastore WHERE first_name = 'first_name_extracted';`

# Second Order Injection

- \* In this way were able to exploit the vulnerability to run an arbitrary query on the database.
- \* The way that the application was vulnerable to second order SQLi is very unlikely in real world and this was only used to demonstrate the exploitation of this vulnerability.

# The defense

- \* As we have noticed, one big security issue comes from the **single quote** character.
- \* In PHP the function `str_replace` can be used to **replace all the occurrences of ' in input with "**, sanitizing in this way the string. But there are other problematic characters.
- \* The PHP function `string addslashes ( string $str )` returns a string with **backslashes before characters that need to be escaped**. These characters are single quote ('), double quote ("), backslash (\) and NUL (the NULL byte).



# Addslashes



Text of 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.

# Addslashes

```
<?php
$input = "" OR TRUE;
echo $input . " This is not safe in a database query.<br>";
echo addslashes($input) . " This is safe in a database query.";
$query = "SELECT email FROM user WHERE first_name = ".$input."";
?>
```

- \* Case without `addslashes()`, all the mail in the table are selected.

```
SELECT email FROM user WHERE first_name= ' "" ' OR TRUE;
```

- \* Case with `addslashes()`, the mail corresponding to / user is selected. This means that no mail is selected.

```
SELECT email FROM user WHERE first_name= ' "/" / ' OR TRUE;
```

# Addslashes

- \* As can be seen from the example, `addslashes()` function permits to escape the power of the apostrophe by putting a backslash before it.
- \* Also the double quotes is escaped by putting the a backslash before it.
- \* This permits to validate the string in input avoiding malicious code.

# Other defenses

Less used:

- \* **sprintf()** can be used with conversion specifications to ensure that the dynamic argument is treated the way it's suppose 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 quote\_style parameter, allows the use of ENT\_QUOTES, which will convert both double and single quotes. This will work in the same sense as addslashes() and mysql\_real\_escape\_string() in regards to quotation marks, however, instead of prepending a backslash, it will use the HTML entity of the quotation mark.

# Smart Login



Text of 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 charset, common in China. Wonderful, this is the case! The way in which we can circumvent `addslashes()`'s protection is using multibyte characters.

# Smart Login

- \* In this case, the vulnerability exploited permits to avoid the validation done by the function `addslashes()` and so to inject malicious code in the query.
- \* The vulnerability found regard the GBK mapping code of the characters. Usually a letter is encoded in 8 bit, this means that we can represent 256 unique value.
- \* In some alphabet (as GBK) are employed multibyte character encoding to express more than 256 letters. But 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.

# Smart Login

- \* The exercise **Smart Login** makes visible that `addslashes()` is not enough. With a smart use of multi-byte characters we can construct a single quote exploiting the way in which `addslashes` works.
- \* How is possible to make our code secure against SQL Injection? There are many ways, one is using Prepared Statements.

# Prepared Statement

- \* Character validation is not always the best choice in SQL defense because it's difficult to escape all the possible malicious code.
- \* The optimum way to avoid SQL injection is to use Prepared Statement that permits to parametrize queries and where all the inserted string are evaluated as characters and so not ran.



# Prepared Statement

- \* Character validation is not always the best choice in SQL defense because it's difficult to escape all the possible malicious code.
- \* The optimum way to avoid SQL injection is to use Prepared Statement that permits to parametrize queries and where all the inserted string are evaluated as characters and so not ran.

# Prepared Statement

// query

```
$query = "SELECT * FROM level WHERE name=:name";
```

// template preparation

```
$stmt = $db->prepare($query);
```

// Variable definition for placeholder substitution

```
$stmt->bindValue(':name', $_SESSION["username"]);
```

// instruction execution

```
$result = $stmt->execute();
```

Thank you!

---