COMP 530 Data Privacy and Security
Homework 4 Report

Due January 9th

Ismayil Ismayilov

Question 1: Authentication

(B) What are the passwords of DigitalCorp users Alice, Bob, Charlie and Harry?

- **Alice** has password **maganda**
- **Bob** has password **gangsta**
- **Charlie** has password **claire**
- **Harry** has password **grenade**

(C) Does your dictionary attack from parts (A) and (B) work? Why or why not?

The initial dictionary attack does not work since the attacker does not know the salt that was used for each user's password before it was hashed. Even if the attacker obtains a salt that a certain user's password was hashed with, he would have to recompute the entire attack table just for that user.

(D) The file salty-digitalcorp.txt contains users Dave, Karen, Faith and Harrison. Devise an attack to find their passwords. Verbally describe your attack strategy and discuss whether this attack requires less computation or more computation than what you did in parts (A) and (B).

As I mentioned in the answer to the previous question, there is a need to recompute the attack table for every salt. The attack strategy is given as follows:

1. Iterate over all (username, salt, hash) triples in `salty-digitalcorp.txt`.
2. Construct an attack table for each such triple by iterating over all passwords textttrockyou.txt, appending the salt to a given password and hashing it.
3. The user's password is obtained by looking up the user's hash in the newly constructed attack table

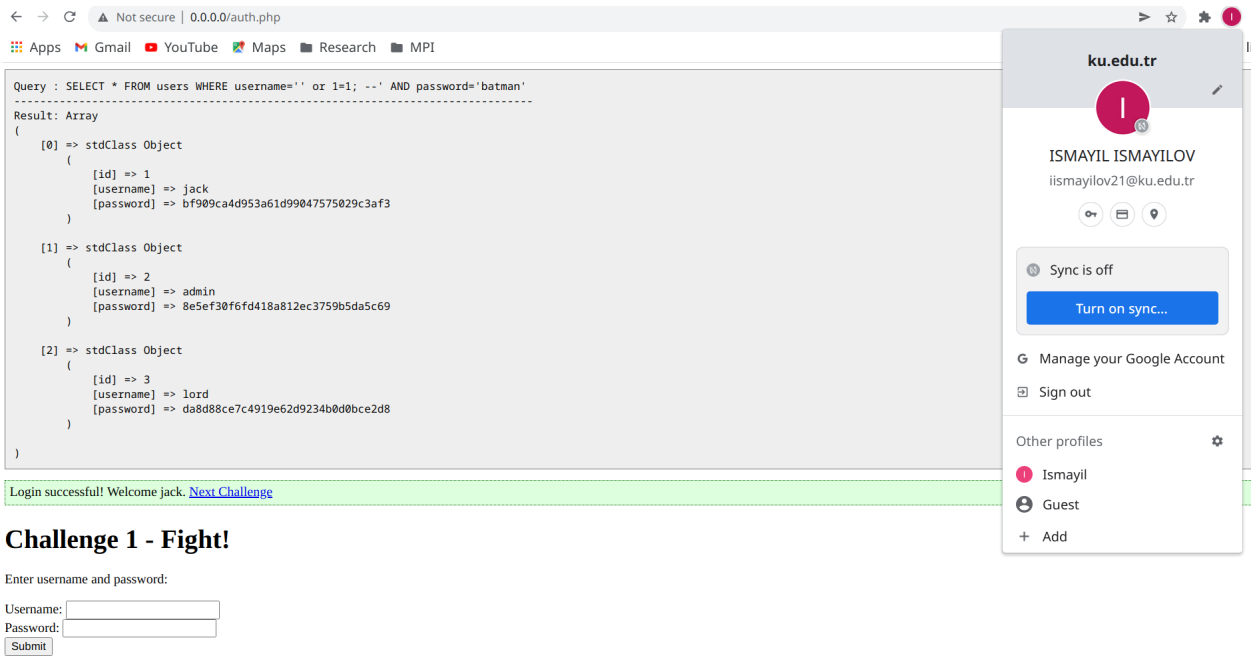
(E) Implement your new attack. Submit its source code as well as the true passwords of Dave, Karen, Faith and Harrison.

- **Dave** has password **kitten**
- **Karen** has password **karen**
- **Faith** has password **bowwow**
- **Harrison** has password **pomegranate**

Question 2: SQL Injection

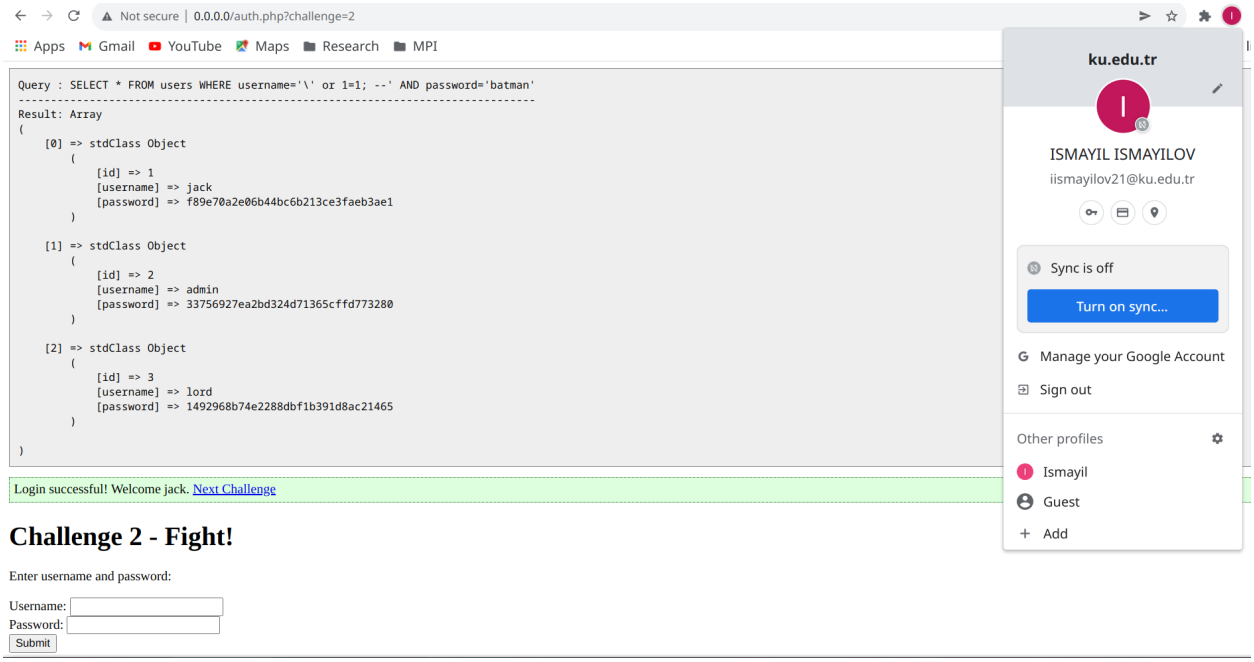
Part 1 (Authentication Based Attacks)

Challenge 1



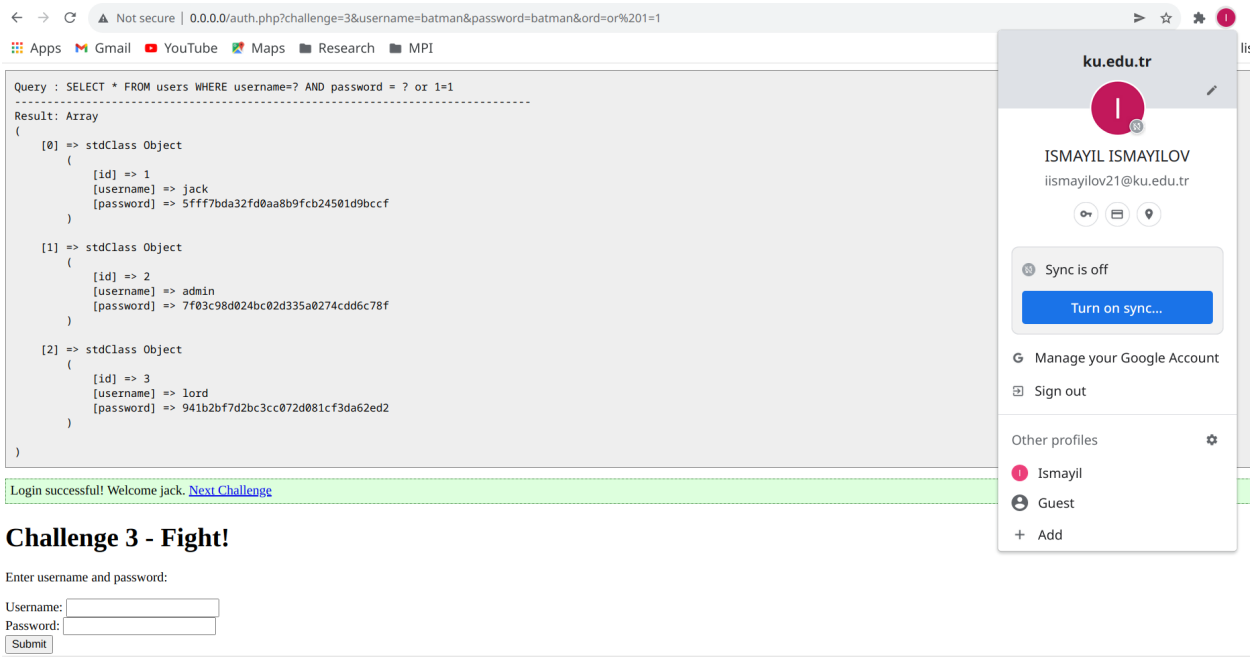
- **Payload** - For the *username*, I input `' or 1=1; --`. For the *password*, any arbitrary string can be used; I use **batman**.
- **Explanation** - The single quote at the beginning of the input to *username*, is used to close the beginning single quote in the *username* input field. After that, I use the tautology `or 1=1` which enables the successful login. Note that while I use `;` after the tautology, I now realize that this is not necessary and could have been omitted. After that, `--` is used to ignore everything that comes after including password validation (which is why any arbitrary password can be used)

Challenge 2



- **Payload** - The payload used for *Challenge 1* successful bypasses this challenge as well. For reference, the payload is `' or 1=1; --` for *username* and `batman` for *password* (though any password can be used)
- **Explanation** - The reason why the payload is successful can be gleaned by looking at the constructed SQL query in the screenshot above. We can see that the single quote at the start of the input to *username* is escaped and instead replaced with `´`. The resulting flow is equivalent to the one presented in *Challenge 1*; the beginning single quote is closed, the tautology is present and everything after is ignored.

Challenge 3



- **Payload** - As stated, for this challenge, the payload is provided through the URL. Any arbitrary strings can be used for *username* and *password*; I used `batman` for both. For the *ord* parameter, I used `or 1=1`. The relevant section of the payload URL is given as &username=batman&password=batman&ord=or 1=1
- **Explanation** - As stated the vulnerability is in the `ORDER BY` clause. The vulnerability can be exploited since the *order* parameter can be provided by the user. The injection works because the tautology `ord 1=1` is used as the input for *ord*.

Part 2 (Union Based Attack)

Challenge 5

- **Payload** - As stated, for this challenge, the payload is provided through the URL. The payload is injected through the *username* parameter. The relevant section of the payload is given as ?username=lord' UNION SELECT S.salary, 0, 0, 0, 0, 0 from salaries S; –
- **Explanation** - First, I identify the individual with the classified salary as the one with the *username* *lord*. Then, by testing multiple `UNION SELECT` statements, I found that 6 is the number of columns returned by the first `SELECT` statement in the source code. After this, I provide the payload to the *username* parameter. The first column of the `SELECT` statement is the salary; the remaining 5 are the padding 0-columns (needed to make `SELECT` statements match). Additionally, I filter the `SELECT` to only include the user whose salary is larger than 100000 using the `WHERE S.salary > 100000`. As can be seen from the screenshot, the user's salary is displayed; in this run, it is 108153.