

Mini Project Report

On

PASSWORD HASHING WITH SALT TECHNIQUE

Submitted by

202IS027 Tridiv Kumar Rabha

202IS020 Rubika Pradhan

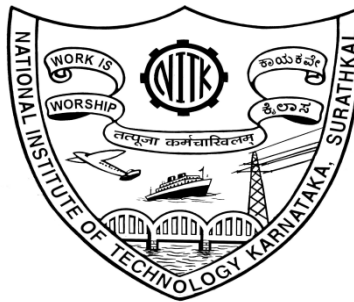
Under the Guidance of

Dr. P. Santhi Thilagam

Dept. of Computer Science & Engineering,

NITK, Surathkal

Date of Submission: 24.5.2021



Department of Computer Science & Engineering
National Institute of Technology Karnataka, Surathkal.
2020-2021

Abstract

Keeping passwords in plain text is not a great solution because the attacker directly owns all users' passwords in plain text. No one, including website/database administrators, should have access to the user's simple text password. Therefore in this project, we would explain the main principles of secure storage (hash, salt, pepper, iteration) and highlight their importance for resisting password recovery methods. Finally, we would proceed with a reliable hash function for secure storage.

Hashing passwords is the standard approach to storing passwords securely. A "Hash" is a one-way function that generates a representation of the password. A user's password is taken, and using a key known to the site; the hash value is derived from the combination of both the password and the key, using a set algorithm. To verify a user's password is correct, it is hashed, and the value compared with that stored on record each time they log in. Hashes are impossible to convert back into plain text, but you don't need to convert them back to break them. Once you know that a particular string converts to a specific hash, you know that any instance of that hash represents that string. It does not take much computational power to generate a table of hashes of combinations of letters, numbers and symbols. Once you have this store of hashes, you can then compare the hash you want to crack and see if it matches. Once you find a match, you know the password. Exposing a hash can be made difficult by adding salt to it. Salts create unique passwords, even in the instance of two users choosing the same passwords. Salts help us mitigate hash table attacks by forcing attackers to re-compute them using the salts for each user. We cannot convert a hashed value into a password directly. Still, you can work out the password if you continually generate hashes from passwords until you find one that matches a so-called brute-force attack.

Declaration

We solemnly declare that the project reports “**Password Hashing Using Salt Technique**” is based on our work carried out during our study under the supervision of **Dr. P. Santhi Thilagam**. We assert the statements made and conclusions drawn as an outcome of our research work. We further certify that

- The work contained in the report is original and has been done by me under the general supervision of my supervisor.
- The work has not been submitted to any other Institution for any other degree, diploma, a certificate in this university or the any other University of India or abroad.
- We have followed the guidelines provided by the university in writing the report.
- Whenever we have used materials (data, theoretical analysis, and text) from other sources, we have given due credit to them in the text of the report and given their details in the references.

Tridiv Kumar Rabha(202IS027)
Rubika Pradhan(202IS020)

Certificate

This is to certify that the report entitled **PASSWORD HASHING WITH SALT TECHNIQUE** submitted by **Tridiv Kumar Rabha** and **Rubika Pradhan** to the **NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA** in partial fulfillment of the requirements for the award of the Degree of Master of Technology in Computer Science and Information Security is a bonafide record of the project work carried out by them under my guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Supervisor: Dr. P. Santhi Thilagam
Dep. Of Computer Science and Engineering
National Institute of Technology, Karnataka
Surathkal, Mangalore - 575025

Date: 24th May, 2021

Contents

Abstract	i
Declaration	ii
Certificate	iii
1 Introduction	1
2 Implementation	1
2.1 Technologies used	1
2.2 Application Overview	1
2.3 Rainbow table attack	3
3 Conclusion	4
References	4

1 Introduction

Technology changes fast. Increasing the speed and power of computers can benefit both the engineers trying to build software systems and the attackers trying to exploit them. Some cryptographic software is not designed to scale with computing power. As explained earlier, the safety of the password depends on how fast the selected cryptographic hashing function can calculate the password hash. A fast function would execute faster when running in much more powerful hardware.

The salt is a pollutant to the raw data (here the password), producing two different hashes from the same data. The salt is unique for each user and is composed of a random sequence. It increases the chance that a password is unique and, therefore, a hash has never been used. The advantages of salt are multiple:

- It is almost impossible to find hash directly on the internet if it is salted. However, the salt must be long enough and random.
- Rainbow tables do not work with salted hash.
- As said before, two users with the same password will not have the same hash if salt is used. After breaking a hash, password cracking software (hashcat, Johntheripper) looks to see if it is not present for another user.

bcrypt is a hash function created by Niels Provos and David Mazières. It is based on the Blowfish encryption algorithm and was presented at USENIX in 1999. Since this algorithm dates back to 1999, it has shown its robustness over time, where some algorithms like Argon2 only exist since 2015. The hash computed by bcrypt has a predefined form:

`$2y$11$SXAXZyioy60hbnyme0J9.ulscXwUFMhbvLaTxAt729tGusw.5AG4C`

- **Algorithm:** This one can take several versions depending on the version of bcrypt (`2`, `$2a$`, `$2x$`, `$2y$` and `$2b$`)
- **The cost:** The number of iterations in the power of 2. For example, the iteration is 11, and the algorithm will do 211 iterations (2048 iterations).
- **Salt:** Instead of storing the salt in a dedicated column, it is directly stored in the final hash.
- **The hashed password**

Since bcrypt stores the number of iterations, this makes it an adaptive function because it can increase the number of iterations, and therefore it is longer and longer. Despite its age and the evolution of computing power, this allows it to be still robust against brute force attacks. The following benchmark shows that it takes 23 days for hashcat to compute the totality of rockyou hashes.

Interestingly, the passwords `azerty` and `matrix`, being very weak passwords and present at the top of the list, were found during the short time (2 hours) that the software worked in the example.

2 Implementation

2.1 Technologies used

- Node JS
- MongoDB
- Ejs
- Express JS

2.2 Application Overview

A user can register with a username and password in our application. The authenticated user can also see his profile details in our application. We have stored the registration credentials in a database. Passwords provided by the users are stored in hashed form with salting.

Activities Google Chrome May 24 9:57 PM 27°C 17.90 KB/s 99%

Document localhost:3000/web/register Apps Downloads Gmail YouTube Cloud Storage... Reading list

Password Salting Home Register Login

Registration

Email/Username

Password

Confirm password

Register

Already have an account? [Login here](#)

Figure 1: Registration page

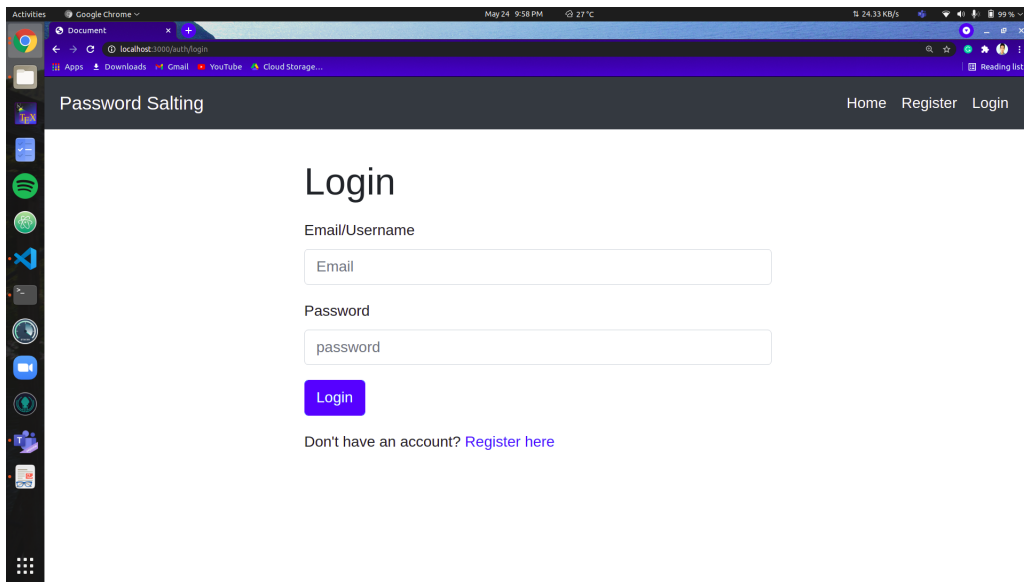


Figure 2: Login page

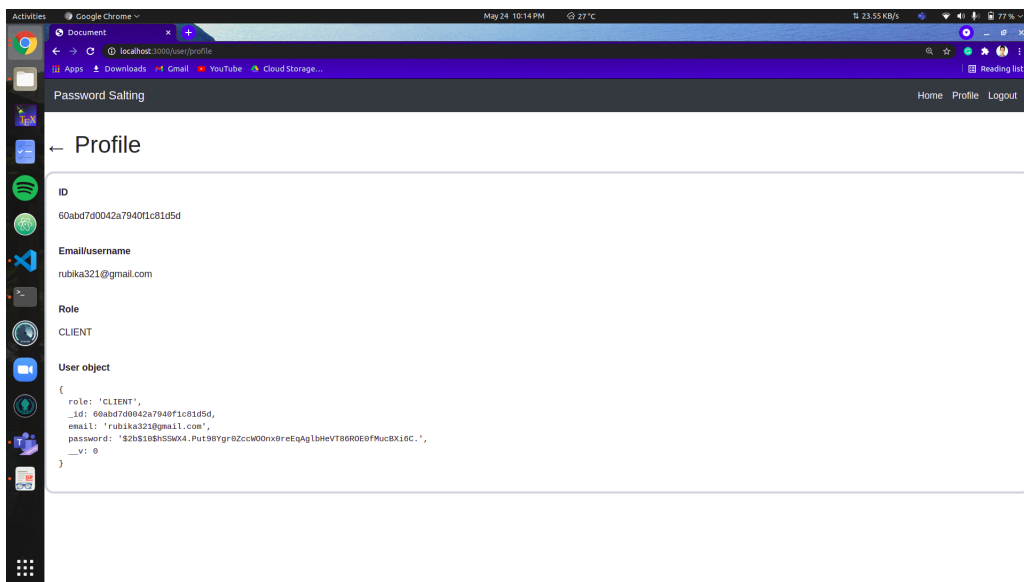
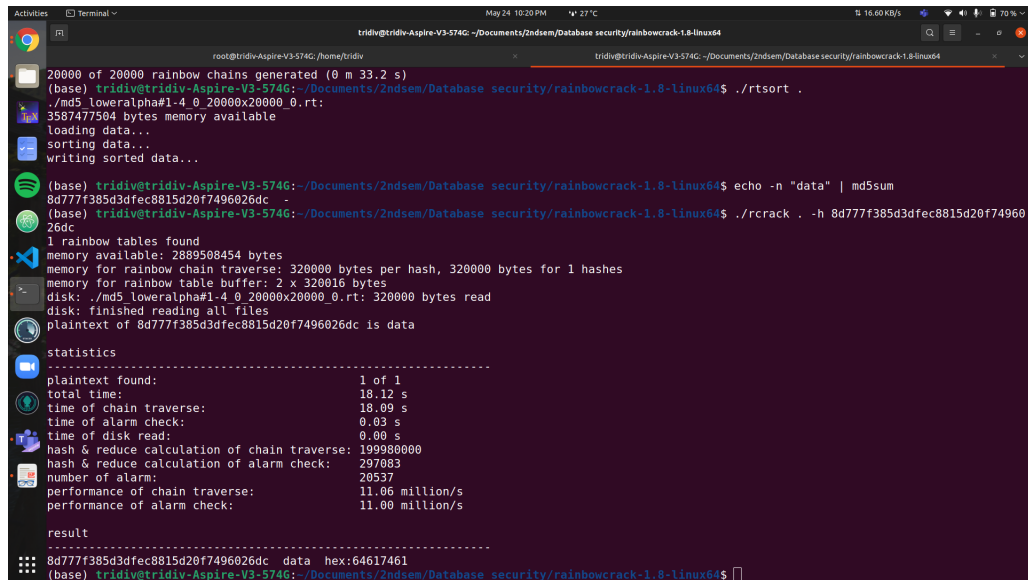


Figure 3: Profile page

2.3 Rainbow table attack

Rainbow tables are a subject that deserves an article on its own. Quickly, it's a data structure that allows retrieving passwords with a good storage/time compromise. This structure has a list of pre-calculated hashes and makes it possible to retrieve a hash in an acceptable time.

Many rainbow tables are available online and also we can create on our local system. We created a rainbow table using the md5 function and then found the hash value of the password we want to find in the rainbow table. We were successful in getting the plain text of the hashed password from the rainbow table. We then tried to find the salted hash of the password in our rainbow table, but we could not. Therefore, it is proved that password salting is not susceptible to rainbow table attacks.



```
root@tridiv-Aspire-V3-574G: ~/home/tridiv
20000 of 20000 rainbow chains generated (0 m 33.2 s)
(base) tridiv@tridiv-Aspire-V3-574G:~/Documents/2ndsem/Database security/rainbowcrack-1.8-linux64$ ./rtsort .
./md5_loweralpha#1-4_0_20000x20000_0.rt:
3587477504 bytes memory available
loading data...
sorting data...
writing sorted data...

(base) tridiv@tridiv-Aspire-V3-574G:~/Documents/2ndsem/Database security/rainbowcrack-1.8-linux64$ echo -n "data" | md5sum
8d777f385d3dfec8815d20f7496026dc -
(base) tridiv@tridiv-Aspire-V3-574G:~/Documents/2ndsem/Database security/rainbowcrack-1.8-linux64$ ./rcrack . -h 8d777f385d3dfec8815d20f7496026dc
1 rainbow tables found
memory available: 2889508454 bytes
memory for rainbow chain traverse: 320000 bytes per hash, 320000 bytes for 1 hashes
memory for rainbow table buffer: 2 x 320016 bytes
disk: ./md5_loweralpha#1-4_0_20000x20000_0.rt: 320000 bytes read
disk: finished reading all files
plaintext of 8d777f385d3dfec8815d20f7496026dc is data

statistics
-----
plaintext found:          1 of 1
total time:              18.12 s
time of chain traverse:   18.09 s
time of alarm check:      0.03 s
time of disk read:        0.00 s
hash & reduce calculation of chain traverse: 199980000
hash & reduce calculation of alarm check: 297093
number of alarm:          20537
performance of chain traverse: 11.06 million/s
performance of alarm check: 11.00 million/s

result
-----
8d777f385d3dfec8815d20f7496026dc data hex:64617461
(base) tridiv@tridiv-Aspire-V3-574G:~/Documents/2ndsem/Database security/rainbowcrack-1.8-linux64$
```

Figure 4: Rainbow table attack

3 Conclusion

Hashing passwords alone is not enough. Hackers are aware of many hashing algorithms, and they keep lists, called rainbow tables, of common hashes of common passwords. They have got most if not all the hashes for “pass123 and similarly common passwords. So, if any one of those hashes appears in the list, the entire hashing scheme becomes known, and all the other more unusual passwords become much more easily compromised as well.

That’s where salting comes in. You can’t force users to come up with good passwords on their own – there’s always going to be someone trying to get away with “pass123.” What you can do is make the password better on the machine level. You can add some extra bits to it, called a “salt.” For example, “pass123 becomes “8Ytu9j06pass123.” Much better! You do have to store the salt in your database somewhere, because your user does not have that part memorized. If your system is compromised, that means the salt gets compromised as well. However, the benefit is this: when you run that salted password through a hashing algorithm, it produces a different hash than the one produced by “pass123 on its own. That means that rainbow tables have to contain thousands more entries for each common password plus all the salt possibilities for each common password, making them a much less effective tool for password cracking. Therefore, if you have to store passwords, always make sure they’re hashed and salted. You should still have everyone change their passwords if you get hacked, but this method should give you a bit more time and security.

References

- [1] Diksha.S.Borde, Poonam.A.Hebare, Priyanka.D.Dhanedhar, “Overview Of Web Password Hashing Using Salt Technique”.
- [2] Praveen Gauravaram, “Security Analysis of salt——password Hashes”
- [3] <https://en.wikipedia.org/wiki/Bcrypt>
- [4] <https://www.npmjs.com/package/bcrypt>
- [5] <https://www.educative.io/edpresso/what-is-hashing>