Intel Unnati Industrial Training

Protecting User Password Keys at REST

Kiran Kumar R Garag
1nt21ec066.kirankumar@nmit.ac.in

Nidhi Thakur
1nt21ec093.nidhi@nmit.ac.in

Vidhi Mathur
1nt21ec170.vidhi@nmit.ac.in

INTRODUCTION

In today's digital world, user passwords are the primary defense against unauthorized access to sensitive information. However, storing passwords in plain text on a disk poses a significant security risk. Even a minor breach can expose thousands of passwords, leading to identity theft, financial loss, and reputational damage. This chapter sets the context for the need to protect user password keys when stored on disk, known as "at rest". This chapter introduces the problem, discusses the implications of insecure password storage, and outlines the goals of the project. The importance of data security and the potential consequences of data breaches are emphasized to highlight the relevance of this study in the current cybersecurity landscape. Thus, protecting user password keys at rest is an essential aspect of maintaining trust and confidence among users, as it demonstrates a commitment to safeguarding their sensitive information and preserving their privacy and security.

SOLUTION

1. Password Hashing (using bcrypt):

Problem Addressed: Storing passwords securely without storing the actual password in plaintext.
Solution: Uses the bcrypt library to hash passwords before storing them. Bcrypt is chosen for its adaptive hashing algorithm, which includes features like salting and cost factor to defend against brute-force attacks.

2. Password Encryption:

Problem Addressed: Protecting sensitive hashed password data during storage or transmission.
Solution: Encrypts the hashed password using the Fernet symmetric encryption algorithm provided by cryptography.fernet. This ensures that even if someone gains access to the encrypted data, they cannot easily decrypt and recover the original hashed password without the encryption key.

3. Key Management:

Problem Addressed: Generating and managing encryption keys securely.
Solution: Provides functions (generate_key()) to generate a random encryption key using Fernet. Proper key management is essential for maintaining the confidentiality and security of encrypted data.

4. Verification and Authentication:

Problem Addressed: Ensuring the integrity and authenticity of user passwords during authentication.
Solution: Demonstrates how to decrypt the encrypted hashed password back to its original form and verify it against the user-provided password using bcrypt.verify(). This step ensures that only users with the correct password can successfully authenticate.

FEATURES OFFERED

1. Hashing the Password (using bcrypt):

Feature: Passwords are hashed securely using the bcrypt hashing algorithm.

2. Encrypting the Hashed Password:

Feature: The hashed password is further encrypted using the Fernet symmetric encryption algorithm.

3. Decrypting the Encrypted Data:

Feature: Demonstrates the ability to decrypt the previously encrypted hashed password.

4. Key Management (Generating and Loading the Key):

Feature: Includes functions to generate a new encryption key (generate_key()).

5. Demonstration of the Whole Process:

Feature: The main() function ties everything together by showcasing the entire flow from password hashing, encryption, decryption, to verification.

6. Password Verification:

Feature: Uses bcrypt's verification function (bcrypt.verify()) to compare the original password with the decrypted hashed password.

PROCESS FLOW

1. Key Generation:

The program starts by generating a random encryption key (key) using Fernet.generate_key().

2. Password Hashing:

A sample password ("my_secure_password") is hashed using the bcrypt.hash() function from passlib.hash.bcrypt. This step converts the password into a secure hashed format suitable for storage.

3. Encryption:

The hashed password is encrypted using the Fernet encryption algorithm. This ensures that even if

someone gains access to the encrypted data, they cannot easily decipher the original hashed password without the encryption key.

4. Decryption:

The encrypted hashed password is decrypted back to its original hashed form using the same encryption key. This step demonstrates the ability to reverse the encryption process securely.

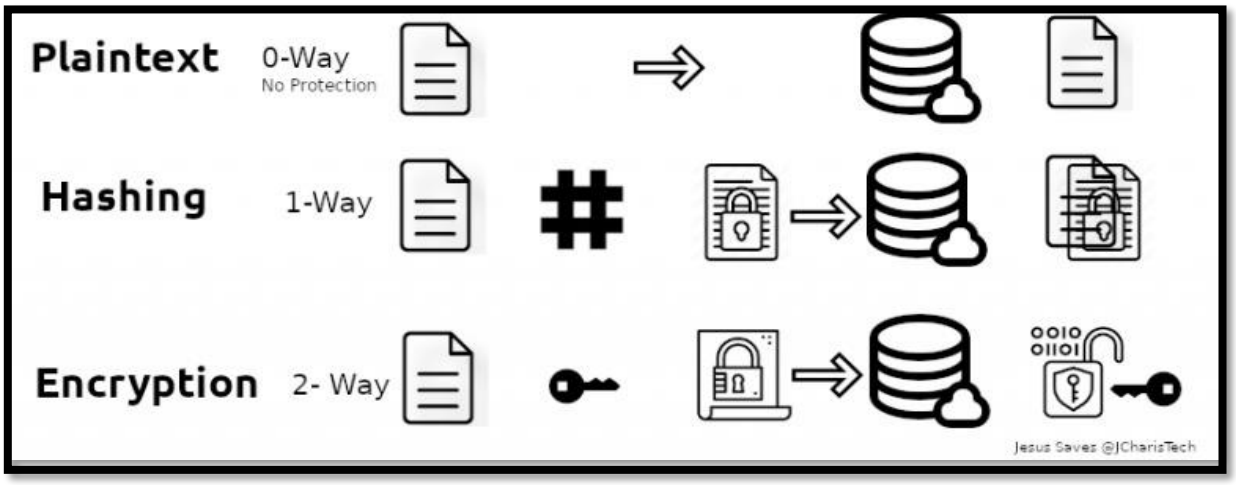
5. Password Verification:

Finally, the decrypted hashed password is verified against the original password using bcrypt.verify(). This step confirms whether the original password matches the decrypted hashed password, ensuring authentication integrity.

6. Output:

Throughout the process, the program prints various outputs to demonstrate each stage: the generated encryption key, the hashed password, the encrypted hashed password, the decrypted hashed password, and the result of the password verification.

ARCHITECHTURE DIAGRAM



- Plaintext

Description: Plaintext is unprotected and can be read by anyone who accesses it.

Protection Level: 0-way (No protection)

Process: The plaintext can be directly stored and retrieved from storage without any modifications.

Symbol: A simple document icon representing readable text.

- Hashing

Description: Hashing is a one-way process that converts plaintext into a fixed-size string of characters, which is typically a hash value.

Protection Level: 1-way (Protected)

Process: The plaintext is transformed into a hash value using a hashing algorithm. This hash value is then stored. Hashing is irreversible, meaning you can't convert the hash back into the original plaintext.

Symbol: A document icon transformed into a hash symbol (#) and stored.

- Encryption

Description: Encryption is a two-way process that converts plaintext into ciphertext using an encryption key. The ciphertext can be decrypted back into the original plaintext using a decryption key.

Protection Level: 2-way (Protected with the ability to revert)

Process: The plaintext is encrypted using an encryption key, resulting in ciphertext. This ciphertext is stored. When needed, the ciphertext can be decrypted back into the original plaintext using a decryption key.

Symbol: A document icon locked with a key, transformed into ciphertext, stored, and then unlocked back into the original document.

TECHNOLOGY USED

- Programming Language:

Python

- Cryptography Libraries:

cryptography.fernet for symmetric encryption (Fernet algorithm)

passlib.hash.bcrypt for password hashing (bcrypt algorithm)

- Python Standard Library:

os module for secure random number generation

TEAM MEMBERS AND CONTRIBUTION

Nidhi Thakur: Implementation of Hashing

Responsible for implementing the hash_password(password) function using bcrypt from passlib.hash.bcrypt.

Ensure passwords are securely hashed and stored.

Kiran Kumar R Garag: Implementation of Encryption

Responsible for implementing the encrypt_data(data, key) and decrypt_data(cipher_text, key) functions.

Utilize cryptography.fernet for symmetric encryption to securely encrypt and decrypt hashed passwords.

Vidhi Mathur: Key Management and Integration

Responsible for implementing the generate_key() function for generating encryption keys.

Manage key storage and integration with hashing and encryption modules.

Ensure overall integration and testing of the complete solution.

CONCLUSION

This project demonstrates a secure method for handling user passwords by combining hashing and encryption. Using bcrypt for password hashing and Fernet for encryption, it ensures that passwords stored on disk are protected against unauthorized access.

Key Achievements:

- Password Security: Passwords are hashed with bcrypt, making them resistant to cracking attempts.
- Data Encryption: Fernet encryption adds an extra layer of protection for hashed passwords.
- Key Management: Secure generation and handling of encryption keys are implemented.
- Modular Design: The code is modular, making it easy to maintain and update.

Practical Implications:

- Secure Storage: Hashed and encrypted passwords can be securely stored in a database.
- User Authentication: Ensures secure verification of user passwords.
- Scalability: The system is designed to be scalable and adaptable.

Future Enhancements:

- Two-Factor Authentication: Add 2FA for increased security.
- Advanced Encryption Techniques: Explore stronger encryption methods.
- Automated Key Rotation: Implement policies for regular key updates.

In summary, this project provides a robust solution for secure password handling using modern cryptographic techniques.