Experiment No. 4

Aim: To implement a secure password storage using hashing algorithm like bcrypt or PBKDF2

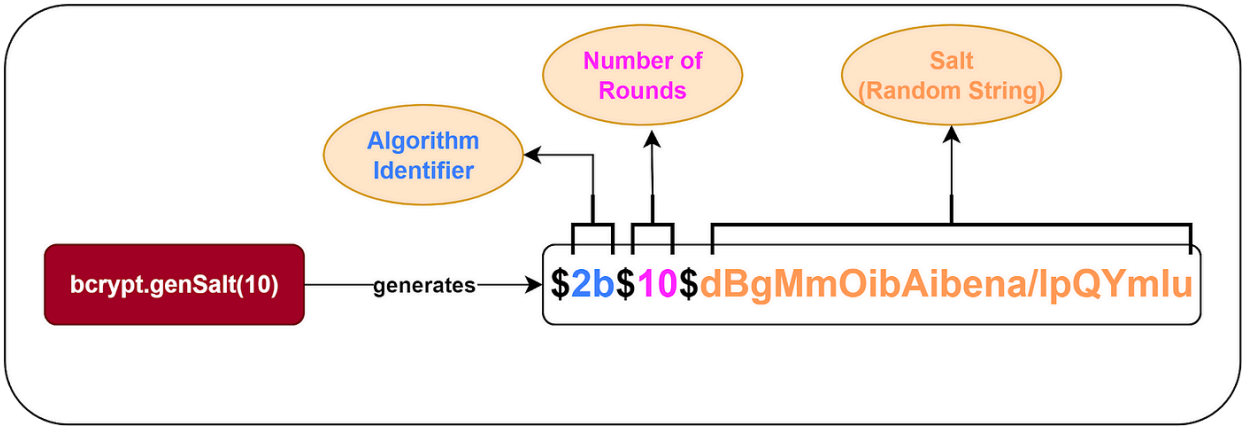
Learning Objective: To implement a secure password storage using hashing algorithm like bcrypt or PBKDF2 in python programming language.

Theory:

Storing user passwords correctly is one of the most critical aspects of web security. Simply storing them as plain text is extremely dangerous, as a single database breach would expose every user's password. The correct approach involves using a **one-way cryptographic hash function** with specific properties.

- **What is Hashing?** Hashing is a process that converts an input of any size (like a password) into a fixed-size string of characters, known as a **hash**. This process is **one-way**, meaning it's computationally infeasible to reverse the process and get the original password from its hash. However, simple fast hashes like SHA-256 are not sufficient for passwords because attackers can use pre-computed lists of hashes (called **rainbow tables**) to find matches quickly.
- **The Importance of Salting** To defeat rainbow table attacks, we use a **salt**. A salt is a unique, random string of data that is generated for each user and added to their password *before* it gets hashed. This salted password is then hashed and stored. Since every user has a different salt, two users with the same password will have completely different hashes, making pre-computation attacks useless. The salt is stored alongside the hash in the database.
- **Slow Hashes are Strong Hashes: bcrypt and PBKDF2** Modern password security relies on algorithms that are **deliberately slow**. Attackers try to guess passwords by hashing billions of possibilities per second (a brute-force attack). By using an algorithm like **bcrypt** or **PBKDF2 (Password-Based Key Derivation Function 2)**, we can introduce a "work factor" or "cost". This cost factor makes the hashing process take a significant amount of time (e.g., 100 milliseconds), making brute-force attacks prohibitively expensive and slow for an attacker. **bcrypt** is often preferred for password

storage because it has built-in salt generation and is designed to be resistant to hardware acceleration attacks (e.g., using GPUs).



Procedure / Implementation:

This procedure outlines the steps to create a secure password storage and verification system using the bcrypt library in Python.

1. **Set Up the Environment** First, ensure you have Python installed on your system. Then, open your terminal or command prompt and install the bcrypt library, which provides all the necessary functions for secure hashing and verification.
2. **Implement User Registration Logic** Write a Python function that handles user registration. This function should accept a plain-text password, generate a salt using bcrypt, hash the password with the salt, and store the resulting **full hash** (which includes the salt and cost factor) in a simulated database.
3. **Implement User Login Logic** Create a second Python function for user login. This function should take a plain-text password from a login attempt, retrieve the user's stored hash from the database, and use bcrypt's built-in check function to securely compare the two.
4. **Demonstrate and Verify the System** Write a simple command-line interface to test the system. Register a new user with a password, then attempt to log in. First, test with the correct password to see a successful verification, and then test with an incorrect password to confirm that the login fails.

Here is the complete Python code for the experiment. Save it as secure_storage.py and run it from your terminal.

```python
import bcrypt
import getpass # To hide password input in the terminal

user_database = {}

def register_user():
    """Handles the user registration process."""
    username = input("Enter a username to register: ")
    if username in user_database:
        print("❌ Username already exists. Please choose another.")
        return

    password = getpass.getpass("Enter a password: ")

    password_bytes = password.encode('utf-8')

    hashed_password = bcrypt.hashpw(password_bytes, bcrypt.gensalt())

    user_database[username] = hashed_password
    print(f"✅ User '{username}' registered successfully!")
    print(f"   Stored Hash: {hashed_password.decode()}")

def login_user():
    """Handles the user login and password verification process."""
    username = input("Enter your username to log in: ")
    if username not in user_database:
        print("❌ Login failed: User not found.")
        return

    password = getpass.getpass("Enter your password: ")

    password_bytes = password.encode('utf-8')

    stored_hash = user_database[username]
    if bcrypt.checkpw(password_bytes, stored_hash):
        print("✅ Login successful! Welcome.")
    else:
        print("❌ Login failed: Incorrect password.")

if __name__ == '__main__':
    print("--- Secure Password Storage Demo ---")
    register_user()
    print("\n--- Now, let's try to log in ---")
    login_user()
    print("\n--- Let's try to log in again with the wrong password ---")
    login_user() # You'll be prompted for user/pass again
```

Learning Outcome: Upon completing this experiment, you will understand why storing plain-text passwords is a critical security flaw and the importance of using salted hashing. You will be able to implement a secure user registration and login system using a modern, slow hashing algorithm like bcrypt to protect user credentials.

Conclusion: This experiment demonstrates that the only secure way to store passwords is to never store them at all, but rather to store a strong, salted, and slow hash. By using a library like bcrypt, we protect user credentials even if the database is compromised, as reversing the hash to find the original password is computationally infeasible. The process of salting defeats pre-computed rainbow tables, while the slowness of the algorithm makes brute-force attacks impractical, providing a robust defense for modern applications.

Name:

Class: BE-CSE

Roll No.:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				