Assigment- 5

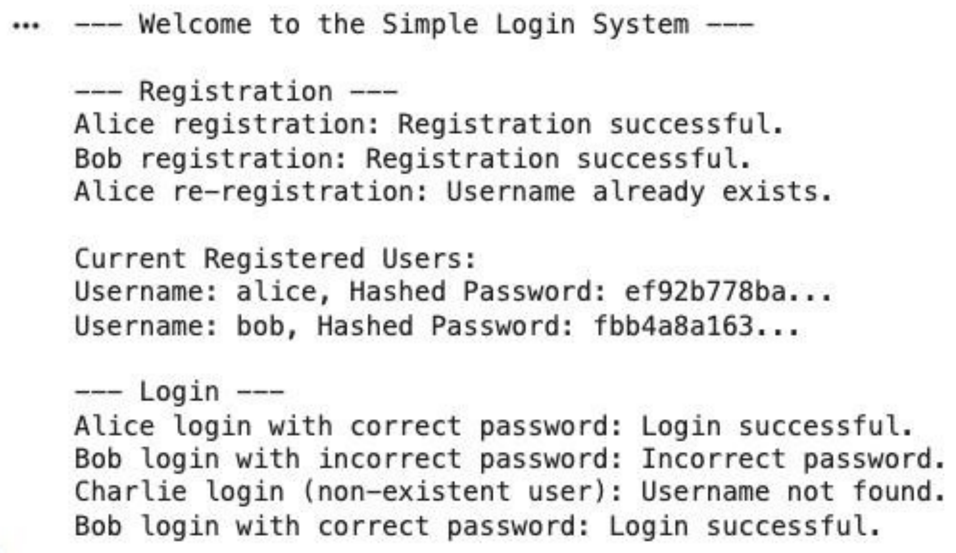
L.Aravind
2303A52028
Batch-38

Task 1: Privacy and Data Security in AI-Generated Code
Scenario
AI tools can sometimes generate insecure authentication logic.
Task Description
Use an AI tool to generate a simple login system in Python.
Analyze the generated code to check:
• Whether credentials are hardcoded
• Whether passwords are stored or compared in plain text
• Whether insecure logic is used

Expected Output
• AI-generated login code

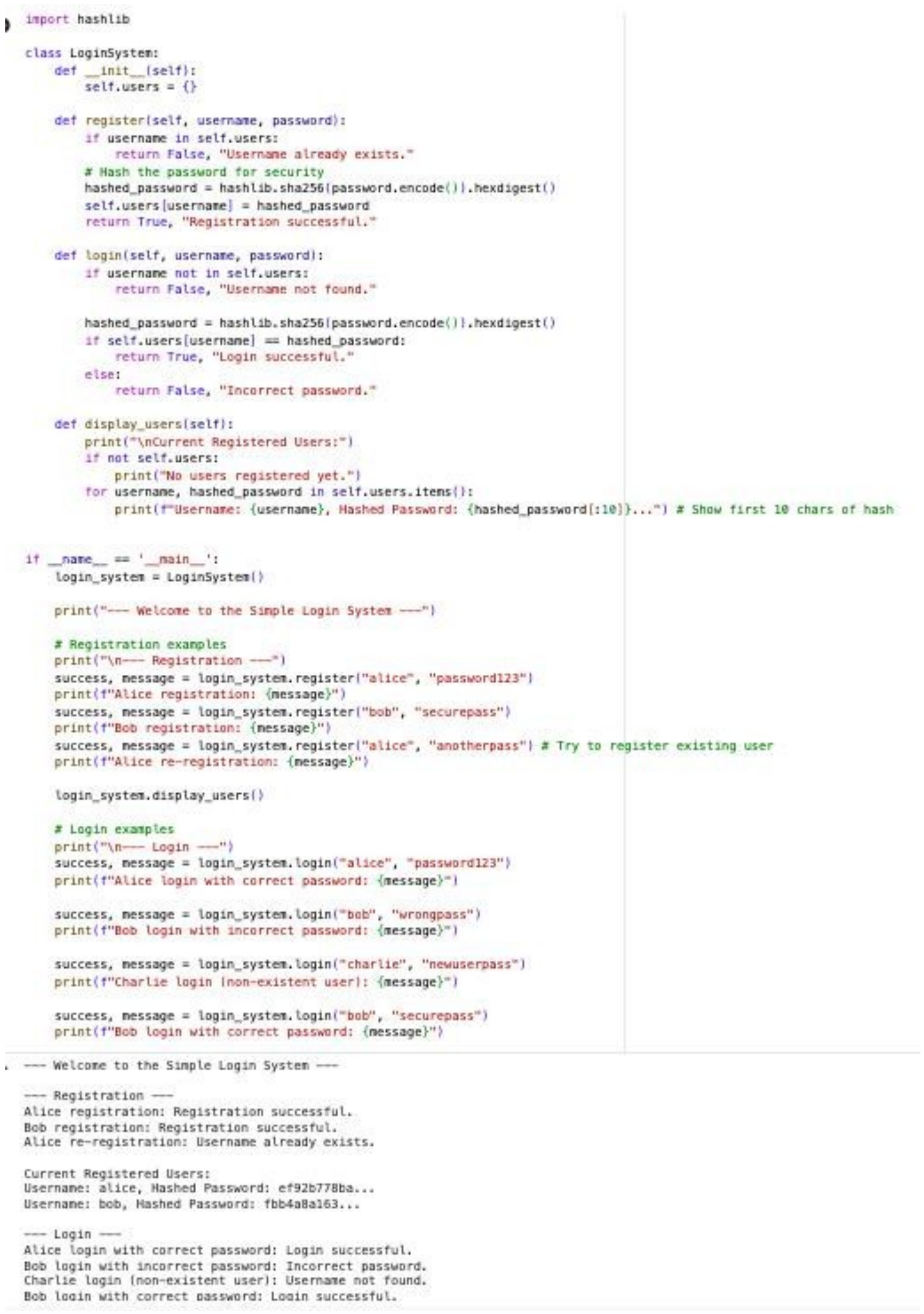
```
import hashlib

class LoginSystem:
def __init__(self):
self.users = {}

def register(self, username, password):
if username in self.users:
return False, "Username already exists."
# Hash the password for security
hashed_password = hashlib.sha256(password.encode()).hexdigest()
self.users[username] = hashed_password
return True, "Registration successful."

def login(self, username, password):
if username not in self.users:
return False, "Username not found."

hashed_password = hashlib.sha256(password.encode()).hexdigest()
if self.users[username] == hashed_password:
return True, "Login successful."
else:
```

```
return False, "Incorrect password."

def display_users(self):
print("\nCurrent Registered Users:")
if not self.users:
print("No users registered yet.")
for username, hashed_password in self.users.items():
print(f"Username: {username}, Hashed Password: {hashed_password[:10]}...") # Show first
10 chars of hash


if __name__ == '__main__':
login_system = LoginSystem()
print("--- Welcome to the Simple Login System ---")

# Registration examples

print("\n--- Registration ---")
success, message = login_system.register("alice", "password123")
print(f"Alice registration: {message}")
success, message = login_system.register("bob", "securepass")
print(f"Bob registration: {message}")
success, message = login_system.register("alice", "anotherpass") # Try to register existing
user
print(f"Alice re-registration: {message}")

login_system.display_users()

# Login examples
print("\n--- Login ---")
success, message = login_system.login("alice", "password123")
print(f"Alice login with correct password: {message}")

success, message = login_system.login("bob", "wrongpass")
print(f"Bob login with incorrect password: {message}")

success, message = login_system.login("charlie", "newuserpass")
print(f"Charlie login (non-existent user): {message}")

success, message = login_system.login("bob", "securepass")
print(f"Bob login with correct password: {message}")
```



```
•••  --- Welcome to the Simple Login System ---

     --- Registration ---
     Alice registration: Registration successful.
     Bob registration: Registration successful.
     Alice re-registration: Username already exists.

     Current Registered Users:
     Username: alice, Hashed Password: ef92b778ba...
     Username: bob, Hashed Password: fbb4a8a163...

     --- Login ---
     Alice login with correct password: Login successful.
     Bob login with incorrect password: Incorrect password.
     Charlie login (non-existent user): Username not found.
     Bob login with correct password: Login successful.
```

Identification of security risks



```python
import hashlib

class LoginSystem:
    def __init__(self):
        self.users = {}

    def register(self, username, password):
        if username in self.users:
            return False, "Username already exists."
        # Hash the password for security
        hashed_password = hashlib.sha256(password.encode()).hexdigest()
        self.users[username] = hashed_password
        return True, "Registration successful."

    def login(self, username, password):
        if username not in self.users:
            return False, "Username not found."

        hashed_password = hashlib.sha256(password.encode()).hexdigest()
        if self.users[username] == hashed_password:
            return True, "Login successful."
        else:
            return False, "Incorrect password."

    def display_users(self):
        print("\nCurrent Registered Users:")
        if not self.users:
            print("No users registered yet.")
        for username, hashed_password in self.users.items():
            print(f"Username: {username}, Hashed Password: {hashed_password[:10]}...") # Show first 10 chars of hash


if __name__ == '__main__':
    login_system = LoginSystem()

    print("--- Welcome to the Simple Login System ---")

    # Registration examples
    print("\n--- Registration ---")
    success, message = login_system.register("alice", "password123")
    print(f"Alice registration: {message}")
    success, message = login_system.register("bob", "securepass")
    print(f"Bob registration: {message}")
    success, message = login_system.register("alice", "anotherpass") # Try to register existing user
    print(f"Alice re-registration: {message}")

    login_system.display_users()

    # Login examples
    print("\n--- Login ---")
    success, message = login_system.login("alice", "password123")
    print(f"Alice login with correct password: {message}")

    success, message = login_system.login("bob", "wrongpass")
    print(f"Bob login with incorrect password: {message}")

    success, message = login_system.login("charlie", "newuserpass")
    print(f"Charlie login (non-existent user): {message}")

    success, message = login_system.login("bob", "securepass")
    print(f"Bob login with correct password: {message}")
```

```
--- Welcome to the Simple Login System ---

--- Registration ---
Alice registration: Registration successful.
Bob registration: Registration successful.
Alice re-registration: Username already exists.

Current Registered Users:
Username: alice, Hashed Password: ef92b778ba...
Username: bob, Hashed Password: fbb4a8a163...

--- Login ---
Alice login with correct password: Login successful.
Bob login with incorrect password: Incorrect password.
Charlie login (non-existent user): Username not found.
Bob login with correct password: Login successful.
```

Security Risk Analysis of the Login System

Based on the analysis of the `LoginSyste` class, particularly the `registe` and `logi` methods,

here are the identified security vulnerabilities:

1. **Lack of Password Salting**: While `hashlib.sha256` is used to hash passwords, there is no salt implemented. This makes the system vulnerable to rainbow table attacks. An attacker with access to the hashed passwords could use precomputed hash tables to find the original passwords, especially for common passwords.
2. **Username Enumeration**: The error messages returned by the `logi` method are different for a non-existent username ("Username not found") versus an incorrect password for a valid username ("Incorrect password"). This allows an attacker to

enumerate valid usernames present in the system, which is the first step in many attack scenarios.

3. **Brute-Force Attack Vulnerability**: There are no mechanisms in place to prevent or mitigate brute-force attacks. An attacker could repeatedly try different passwords for a given username without any rate limiting or account lockout, eventually guessing the correct password.
4. **No Input Validation/Password Policy**: The system does not enforce any password complexity requirements (e.g., minimum length, use of special characters, numbers). This allows users to set weak passwords, making them easier to guess. Similarly, there's no validation for usernames.
5. **Storage of Passwords (via `self.user` dictionary)**: The system stores user data, including hashed passwords, in a simple in-memory dictionary. While this is acceptable for a simple demonstration, in a real-world application, this would need to be a secure database. The in-memory storage means data is lost when the application

   restarts. More critically, if the application is compromised, the `self.user` could be directly accessed.

Revised secure version of the code

```
import hashlib
import os

class LoginSystem:
def __init__(self):
self.users = {}

def register(self, username, password):
if username in self.users:
return False, "Username already exists."
# Basic password policy: minimum length of 8 characters
if len(password) < 8:
return False, "Password must be at least 8 characters long."

# Generate a unique salt for each user
salt = os.urandom(16).hex() # 16-byte random salt, converted to hex string
# Concatenate salt with password before hashing
# Ensure consistent encoding for both salt and password
salted_password = (salt + password).encode('utf-8')
hashed_password = hashlib.sha256(salted_password).hexdigest()
# Store both the salt and the hashed password
self.users[username] = (salt, hashed_password)
return True, "Registration successful."

def login(self, username, password):
```

dictionary

```
if username not in self.users:
# Prevent username enumeration by returning a generic error
return False, "Invalid username or password."

# Retrieve the stored salt and hashed password
stored_salt, stored_hashed_password = self.users[username]
# Concatenate the retrieved salt with the provided password and hash it
salted_input_password = (stored_salt + password).encode('utf-8')
hashed_input_password = hashlib.sha256(salted_input_password).hexdigest()
# Compare the newly generated hash with the stored hash
if hashed_input_password == stored_hashed_password:
return True, "Login successful."
else:
# Return a generic error for incorrect password as well
return False, "Invalid username or password."

def display_users(self):
print("\nCurrent Registered Users:")
if not self.users:
print("No users registered yet.")
for username, (salt, hashed_password) in self.users.items():
print(f"Username: {username}, Salt: {salt[:8]}..., Hashed Password:
{hashed_password[:10]}...") # Show first 8 chars of salt and 10 of hash


if __name__ == '__main__':
login_system = LoginSystem()

print("--- Welcome to the Simple Login System (Improved Security) ---")

# Registration examples
print("\n--- Registration ---")
success, message = login_system.register("alice", "password123")
print(f"Alice registration: {message}")
success, message = login_system.register("bob", "securepass")
print(f"Bob registration: {message}")
# Test password policy
success, message = login_system.register("charlie", "short")
print(f"Charlie registration (short password): {message}")
```

```
success, message = login_system.register("alice", "anotherpass") # Try to register existing
user
print(f"Alice re-registration: {message}")

login_system.display_users()

# Login examples
print("\n--- Login ---")
success, message = login_system.login("alice", "password123")
print(f"Alice login with correct password: {message}")

success, message = login_system.login("bob", "wrongpass")
print(f"Bob login with incorrect password: {message}")

success, message = login_system.login("charlie", "newuserpass") # Non-existent user
print(f"Charlie login (non-existent user): {message}")
# Try to login with short password (this should fail as charlie wasn't registered)
success, message = login_system.login("charlie", "short")
print(f"Charlie login (non-existent user with short pass attempt): {message}")

success, message = login_system.login("bob", "securepass")
print(f"Bob login with correct password: {message}")
```

Brief explanation of improvements

Improvements Made to Enhance Security:

1.

Password Salting Implementation:

2.
1. **Change:** A unique, randomly generated salt (`os.urandom(16).hex()`) is now created for each user during registration.
2. **Before:** Passwords were hashed directly using `hashlib.sha256` without salting.
3. **Impact:** This prevents rainbow table attacks. Even if an attacker obtains the database of hashed passwords, they cannot use precomputed tables to reverse the hashes, as each hash is unique due to the unique salt. This significantly increases the effort required for an attacker to crack passwords.

3.

Prevention of Username Enumeration:

4.

1. **Change:** The `logi` method now returns a single, generic error message ("Invalid username or password.") regardless of whether the username does not exist or the password for an existing username is incorrect.
2. **Before:** The system provided distinct error messages: "Username not found." for non-existent users and "Incorrect password." for valid users with wrong passwords.
3. **Impact:** This prevents attackers from easily determining which usernames are valid within the system. By obscuring the existence of a username, it makes it harder for attackers to target valid accounts for brute-force or other credential-stuffing attacks.

5\.

Basic Password Policy Enforcement:

6\.

1. **Change:** During registration, the `registe` method now enforces a minimum password length (e.g., 8 characters).

2. **Before:** There was no password policy, allowing users to set weak or short passwords.
3. **Impact:** This encourages users to create stronger passwords, making them more resilient to brute-force attacks and dictionary attacks. While this is a basic policy, it's a fundamental step towards better password hygiene.

7\.

Secure Storage of Salt and Hashed Password:

8\.

1. **Change:** Both the unique salt and the hashed password are now stored together as a tuple `(salt, hashed password)` for each user.
2. **Before:** Only the hashed password was stored.
3. **Impact:** Storing the salt with the hash is crucial because the salt is necessary to re-hash a user's input password during login for comparison. It ensures that the salting mechanism is properly utilized for authentication.

Task 2: Bias Detection in AI-Generated Decision Systems
Scenario
AI systems may unintentionally introduce bias.
Task Description
Use AI prompts such as:
• “Create a loan approval system”
• Vary applicant names and genders in prompts
Analyze whether:
• The logic treats certain genders or names unfairly

- **Approval decisions depend on irrelevant personal attributes**

Suggest methods to reduce or remove bias.
Expected Output
• Python code generated by AI

```
class LoanApprovalSystem:
def ____init____(self):
# Define basic approval thresholds and rules self.MIN_CREDIT_SCORE = 600
self.MIN_INCOME_FOR_LOAN_AMOUNT_RATIO = 0.3 # income should be at least 30% of loan amount

def evaluate_loan(self, credit_score, income, loan_amount, employment_status): # Rule 1: Credit score must meet the
minimum requirement
if credit_score < self.MIN_CREDIT_SCORE:
return "Rejected", "Reason: Credit score too low."

# Rule 2: Income must be sufficient relative to the loan amount
# For simplicity, we'll say income must be at least 30% of the loan amount if income < loan_amount *
self.MIN_INCOME_FOR_LOAN_AMOUNT_RATIO:
return "Rejected", "Reason: Insufficient income relative to loan amount."

# Rule 3: Employment status must be 'employed' if
employment_status.lower() != 'employed':
return "Rejected", "Reason: Applicant is not employed."

# If all rules are passed, approve the loan return "Approved", "Reason:
All criteria met."


if ____name___ == '____main___':
loan_system = LoanApprovalSystem()

print("--- Loan Application Evaluation ---")

# Sample Applicant 1: Should be Approved applicant1 = {
"credit_score": 720,
"income": 60000,
"loan_amount": 100000, "employment_status":
"employed"
}
decision, reason = loan_system.evaluate_loan(**applicant1) print(f"\nApplicant 1 (Credit:
{applicant1['credit_score']}, Income:
${applicant1['income']:.2f}, Loan: ${applicant1['loan_amount']:.2f}, Employment:
{applicant1['employment_status']}): {decision} - {reason}")
```

```
# Sample Applicant 2: Rejected (Low Credit Score) applicant2 = {
"credit_score": 550,
"income": 70000,
"loan_amount": 150000, "employment_status":
"employed"
}
decision, reason = loan_system.evaluate_loan(**applicant2) print(f"\nApplicant 2 (Credit:
{applicant2['credit_score']}, Income:
${applicant2['income']:.2f}, Loan: ${applicant2['loan_amount']:.2f}, Employment:
{applicant2['employment_status']}): {decision} - {reason}")

# Sample Applicant 3: Rejected (Insufficient Income) applicant3 = {
"credit_score": 680,
"income": 20000,
"loan_amount": 100000, "employment_status":
"employed"
}
decision, reason = loan_system.evaluate_loan(**applicant3) print(f"\nApplicant 3 (Credit:
{applicant3['credit_score']}, Income:
${applicant3['income']:.2f}, Loan: ${applicant3['loan_amount']:.2f}, Employment:
{applicant3['employment_status']}): {decision} - {reason}")

# Sample Applicant 4: Rejected (Unemployed) applicant4 = {
"credit_score": 650,
"income": 50000,
"loan_amount": 80000, "employment_status":
"unemployed"
}
decision, reason = loan_system.evaluate_loan(**applicant4) print(f"\nApplicant 4 (Credit:
{applicant4['credit_score']}, Income:
${applicant4['income']:.2f}, Loan: ${applicant4['loan_amount']:.2f}, Employment:
{applicant4['employment_status']}): {decision} - {reason}")

# Sample Applicant 5: Approved (Edge case for income ratio) applicant5 = {
"credit_score": 600,
"income": 30000,
"loan_amount": 100000, "employment_status":
"employed"
}
decision, reason = loan_system.evaluate_loan(**applicant5) print(f"\nApplicant 5 (Credit:
{applicant5['credit_score']}, Income:
${applicant5['income']:.2f}, Loan: ${applicant5['loan_amount']:.2f}, Employment:
{applicant5['employment_status']}): {decision} - {reason}")
```