TASK 3 SECURE CODING

Why Secure Coding?**

- Security vulnerabilities can lead to **data breaches** and **unauthorized access**.
- **SQL Injection, Plaintext Passwords, and Lack of Input Validation** are common risks.
- Secure coding ensures **confidentiality, integrity, and availability** of user data.

Bandit Scan Results**

- Used **Bandit** (a static code analyzer) to detect security flaws in the code.
- Bandit found **SQL injection vulnerabilities** in string-based query construction.

Bandit Report Summary:
✅ **Issue 1:** Possible SQL injection vector (Line 10)
✅ **Issue 2:** Possible SQL injection vector (Line 22)
✅ **Severity: Medium**
✅ **Confidence: Low**

Example from Bandit Output:

```
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through
string-based query construction.
   Severity: Medium   Confidence: Low
   CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
   Location: test.py:10:12
```

Original Code - Vulnerabilities**

```
import sqlite3
```

```
def create_user(username, password):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()

    # Even worse: Using string concatenation without escaping
    query = "INSERT INTO users (username, password) VALUES ('" + username + "', '"
+ password + "')"
    cursor.execute(query)  # Directly executing user input

    conn.commit()
    conn.close()

def authenticate(username, password):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()

    # Another bad practice: Using unescaped user input in SQL query
    query = "SELECT * FROM users WHERE username = '" + username + "' AND password =
'" + password + "'"
    cursor.execute(query)  # SQL Injection possible here

    user = cursor.fetchone()
    conn.close()
    return user  # Returns user object instead of a boolean
```

Security Issues in the Code**

1. **SQL Injection Risk**
   - Using string concatenation in SQL queries exposes the database to injection attacks.
2. **Plaintext Password Storage**
   - Passwords should be **hashed**, not stored in plaintext.
3. **Lack of Parameterized Queries**
   - Makes the system vulnerable to direct SQL injections.
4. **No Proper Error Handling**
   - If an error occurs (e.g., username already exists), the program might crash.

Secure Implementation - Fixes**

```
import sqlite3
import hashlib
import os

def hash_password(password):
    salt = os.urandom(16)
    hashed_pw = hashlib.pbkdf2_hmac('sha256', password.encode(), salt, 100000)
    return salt + hashed_pw

def verify_password(stored_password, provided_password):
    salt = stored_password[:16]
    stored_hash = stored_password[16:]
    new_hash = hashlib.pbkdf2_hmac('sha256', provided_password.encode(), salt,
100000)
    return new_hash == stored_hash

def create_user(username, password):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                        id INTEGER PRIMARY KEY AUTOINCREMENT,
                        username TEXT UNIQUE NOT NULL,
                        password BLOB NOT NULL)''')
    hashed_password = hash_password(password)
    try:
        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
(username, hashed_password))
        conn.commit()
    except sqlite3.IntegrityError:
        print("Username already exists!")
    finally:
        conn.close()

def authenticate(username, password):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute("SELECT password FROM users WHERE username = ?", (username,))
    record = cursor.fetchone()
```

```
    conn.close()
    return record and verify_password(record[0], password)
```

Security Improvements**

✅ **Uses Parameterized Queries** – Prevents SQL Injection.
✅ **Hashes Passwords with PBKDF2-HMAC-SHA256** – Prevents password leaks.
✅ **Adds Exception Handling** – Prevents crashes due to duplicate entries.
✅ **Implements Secure Password Verification** – Compares stored hashes securely.

Key Takeaways**

- Never store passwords in plaintext. Always use **secure hashing**.
- Use **parameterized queries** to protect against SQL injection.
- Implement **proper error handling** to prevent system crashes.
- Use **static code analyzers like Bandit** to detect vulnerabilities early.
- Security is an ongoing process – always **review and update your code**!
