Practical 5

Aim: Write a python code to identify SQL injection vulnerabilities and mitigate them.

Task 1: Identify SQL Injection Vulnerability

The SQL injection vulnerability is present in the get_user_data_vulnerable function. The function constructs a SQL query by directly embedding user input into the query string without proper sanitization, which allows an attacker to manipulate the SQL query

Code:

```
import sqlite3


def create_db():
    """Create a SQLite database and populate it with sample data."""
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL,
            email TEXT NOT NULL
        )
    ''')
    cursor.execute('''
        INSERT INTO users (username, email) VALUES
        ('Arpit', 'd22amtics@gmail.com'),
        ('user1', 'user1@example.com'),
        ('user2', 'user2@example.com')
    ''')
    conn.commit()
    conn.close()
```

```
def delete_all_entries(table_name):
    """Delete all entries from a specified table."""
    conn = sqlite3.connect('example.db')  # Replace with your database path
    cursor = conn.cursor()
    cursor.execute(f"DELETE FROM {table_name}")
    conn.commit()
    conn.close()
def get_user_data_vulnerable(vulnerable_input):
    """Retrieve user data using a vulnerable SQL query."""
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{vulnerable_input}'"
    cursor.execute(query)
    results = cursor.fetchall()
    conn.close()
    return results
```

For example, if an attacker provides input like `"Arpit' --"`, the query becomes:

sql

```
SELECT * FROM users WHERE username = 'Arpit' --'
```

Task 2: Mitigate Using Prepared Statements

To mitigate the SQL injection vulnerability, you should use parameterized queries (prepared statements). This approach ensures that user input is treated as data rather than executable code. The get_user_data_secure function demonstrates

Here's how to securely retrieve user data:

```
import sqlite3

def create_db():
    """Create a SQLite database and populate it with sample data."""
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL,
            email TEXT NOT NULL
        )
    ''')
    cursor.execute('''
        INSERT INTO users (username, email) VALUES
        ('Arpit', 'd22amtics@gmail.com'),
        ('user1', 'user1@example.com'),
        ('user2', 'user2@example.com')
    ''')
    conn.commit()
    conn.close()


def delete_all_entries(table_name):
    """Delete all entries from a specified table."""
    conn = sqlite3.connect('example.db')  #Replace with your database path
    cursor = conn.cursor()
    cursor.execute(f"DELETE FROM {table_name}")
```

```
    conn.commit()
    conn.close()
def get_user_data_secure(safe_input):
    """Retrieve user data using a secure, parameterized query."""
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ?"
    cursor.execute(query, (safe_input,))
    results = cursor.fetchall()
    conn.close()
    return results
```

Task 3: Mitigate Using Prepared Statements

1. Vulnerable Code Execution:

If you run the main() function with the vulnerable query input ("Arpit' --"), it will likely cause an error or unintended behavior due to the SQL injection. The specific behavior may vary depending on how the SQL query is interpreted by SQLite.

2. Secure Code Execution:

When using the secure query input ("Arpit"), the get_user_data_secure function correctly fetches user data without any risk of SQL injection

Output

```
Vulnerable query results:
(1, 'Arpit', 'd22amtics@gmail.com')

Secure query results:
(1, 'Arpit', 'd22amtics@gmail.com')
```

Table:

```
Initial table contents:
(1, 'Arpit', 'd22amtics@gmail.com')
(2, 'user1', 'user1@example.com')
(3, 'user2', 'user2@example.com')
(4, 'Arpit', 'd22amtics@gmail.com')
(5, 'user1', 'user1@example.com')
(6, 'user2', 'user2@example.com')

Vulnerable query results:
(1, 'Arpit', 'd22amtics@gmail.com')
(4, 'Arpit', 'd22amtics@gmail.com')

Secure query results:
(1, 'Arpit', 'd22amtics@gmail.com')
(4, 'Arpit', 'd22amtics@gmail.com')
```