# TASK-3 SECURE CODING REVIEW

## BY-Raunak kumar jha

Let's consider a simple example in Python for a web application that accepts user input and performs a database query. We'll review this code for vulnerabilities related to SQL injection and provide recommendations for improvement.

Language used -PYHTON

```
import sqlite3


def search_users(name):
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE name = '" + name + "'"
    cursor.execute(query)
    users = cursor.fetchall()
    conn.close()
    return users
```

**Vulnerabilities:-**

SQL Injection: The code directly concatenates the user-provided name variable into the SQL query string. This makes the application vulnerable to SQL injection attacks, where an attacker can manipulate the input to execute arbitrary SQL commands.

**Recommendations:-**

**Use Parameterized Queries:** Instead of concatenating user input directly into the query string, use parameterized queries. This separates the SQL code from the user input and prevents SQL injection attacks.

```
def search_users(name):
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE name = ?"
    cursor.execute(query, (name,))
    users = cursor.fetchall()
    conn.close()
    return users
```

**Input Validation:** Validate user input to ensure it meets expected criteria. In this case, validate that the name input contains only valid characters for a username and does not contain any SQL injection payloads.

```
import re


def search_users(name):
    if not re.match("^[a-zA-Z0-9_]+$", name):
        raise ValueError("Invalid name format")
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE name = ?"
    cursor.execute(query, (name,))
    users = cursor.fetchall()
    conn.close()
    return users
```

**Database Permissions:** Ensure that the database user account used by the application has the least privileges necessary to perform its tasks. Avoid using accounts with administrative privileges for routine database operations.

**Error Handling:** Implement proper error handling to gracefully handle exceptions that may occur during database operations. This prevents leaking sensitive information to users in error messages.

**Security Testing:** Conduct thorough security testing, including penetration testing and code reviews, to identify and mitigate vulnerabilities beyond just SQL injection.

## Example-2

**Let's consider another example, this time in PHP, where user input is used in a SQL query without proper sanitization:**

```
<?php


$db = new mysqli('localhost', 'username', 'password', 'database');


if ($db->connect_errno) {
    die('Failed to connect to MySQL: ' . $db->connect_error);
}


function search_users($name) {
    global $db;
    $query = "SELECT * FROM users WHERE name = '" . $name . "'";
    $result = $db->query($query);
    if (!$result) {
        die('Error in query: ' . $db->error);
    }
    $users = $result->fetch_all(MYSQLI_ASSOC);
    return $users;
}


?>
```

**Vulnerabilities:**

**SQL Injection:** As in the previous example, the code concatenates the $name variable directly into the SQL query string, making it vulnerable to SQL injection attacks.

**Recommendations:**

Prepared Statements: Use prepared statements or parameterized queries to separate the SQL logic from the user input. This prevents SQL injection by treating user input as data rather than executable code.

```
function search_users($name) {
    global $db;
    $query = "SELECT * FROM users WHERE name = ?";
    $statement = $db->prepare($query);
    $statement->bind_param('s', $name);
    $statement->execute();
    $result = $statement->get_result();
    $users = $result->fetch_all(MYSQLI_ASSOC);
    return $users;
}
```

Error Handling: Implement proper error handling to deal with database errors gracefully. Avoid displaying sensitive information such as database error messages to end users, as this could aid attackers in exploiting vulnerabilities.

Input Validation: Validate user input to ensure it meets expected criteria before using it in a query. For example, you could check that the $name variable contains only alphanumeric characters and spaces.

```
function search_users($name) {
    global $db;
    if (!preg_match('/^[a-zA-Z0-9\s]+$/', $name)) {
        die('Invalid name format');
    }
    // Proceed with the query
}
```

**Least Privilege Principle:** Ensure that the database user account used by the application has the minimum necessary privileges required for its tasks. Avoid granting unnecessary permissions, such as administrative privileges, to mitigate the impact of potential attacks.

**Security Audits:** Regularly perform security audits, including code reviews and vulnerability assessments, to identify and address potential security issues proactively.

Applying these recommendations will help secure the PHP code against SQL injection vulnerabilities and enhance the overall security of the application