I will demonstrate a unique example of insecure interaction between components. I have chosen one of the most dangerous, SQL Injection. This is only part of what would be larger application and is for demonstration purposes.

**SQL Injection**

A hacker could insert malicious code such as the following for username: 105 OR 1=1

This could make an SQL statement look like this:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

Of course, 1=1 is always true, so this query would return all rows. If the Users table contained usernames and passwords this could be dangerous. This best way to mitigate this vulnerability is by implementing prepared statements.

The general idea of prepared statements is that you give MySQL a template for a query that you want to run and you indicate places where you can fill in the blanks later (indicated by question marks in following example). This way we can pass strings in for values instead of malicious scripts. The argument may be one of four types: integer, double, string, or blob. For this example, we will focus on strings.

Bad technique:

The following example leaves the application open to SQL Injection:

"INSERT INTO subjects (A) VALUES(".$\_POST["a"].")"

The user may need to slightly modify SQL syntax to cause harm. This is as wide open as it gets to being vulnerable to SQL injection.

Good technique:

INSERT INTO subjects

(menu\_name, position, visible)

VALUE

(?,?,?)

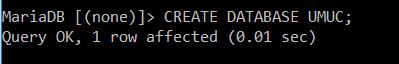
Use MySQL i’s prepare function to execute a statement:

"INSERT INTO subjects (A) VALUES(?)"

The ? serves as a placeholder for the values we will fill in later on.

\*Never use user-provided data to generate these statements. Any user provided data should be passed as parameters to the statement after it has been prepared.

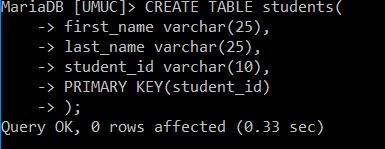
1. Create database UMUC.



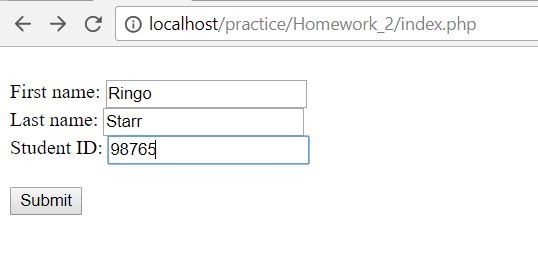
2. Switch to UMUC database.

C:\Users\justi\AppData\Local\Microsoft\Windows\INetCache\Content.Word\2.jpg

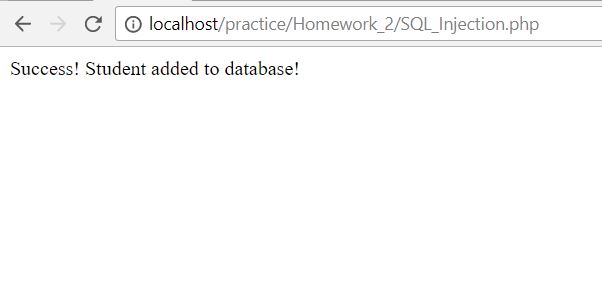
3. Create table named students.



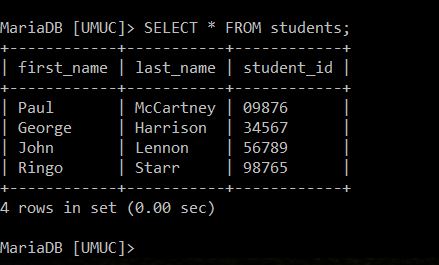
4. Enter student information into form.



5. Submit form with prepared statements. Submission successful.



6. Enter remaining students into database to populate table.



**Vulnerable code snippet**:

//The following INSERT statement is vulnerable. It simply takes whatever the user enters into the form //and inserts it directly into the database. This is as wide open as it gets for SQL injection.

$query = "INSERT INTO students (";

$query .= " first\_name, last\_name, student\_id";

$query .= ") VALUES (";

$query .= " '{$first\_name}', '{$last\_name}', '{$student\_id}' ";

$query .= ")";

**Safe code using prepared statement**:

// The ?’s serve as placeholders for our values, which we have told the program are strings

// hence the ‘sss’.

$stmt = $connection->prepare("INSERT INTO students (first\_name, last\_name, student\_id) VALUES (?, ?, ?)");

$stmt->bind\_param('sss', $\_POST['first\_name'], $\_POST['last\_name'], $\_POST['student\_id']);

$stmt->execute();

$stmt->close();