Database Administration Security

Cybersecurity Department

Course Code: ***CBS 214***

Practical Lecture 4: SQL Injection

Halal Abdulrahman Ahmed

Lecture Outlines

- What is SQL injection (SQi)?
- Comparison between normal & malicious SQL statements
- Implementations of SQL Injection Query (General Example)

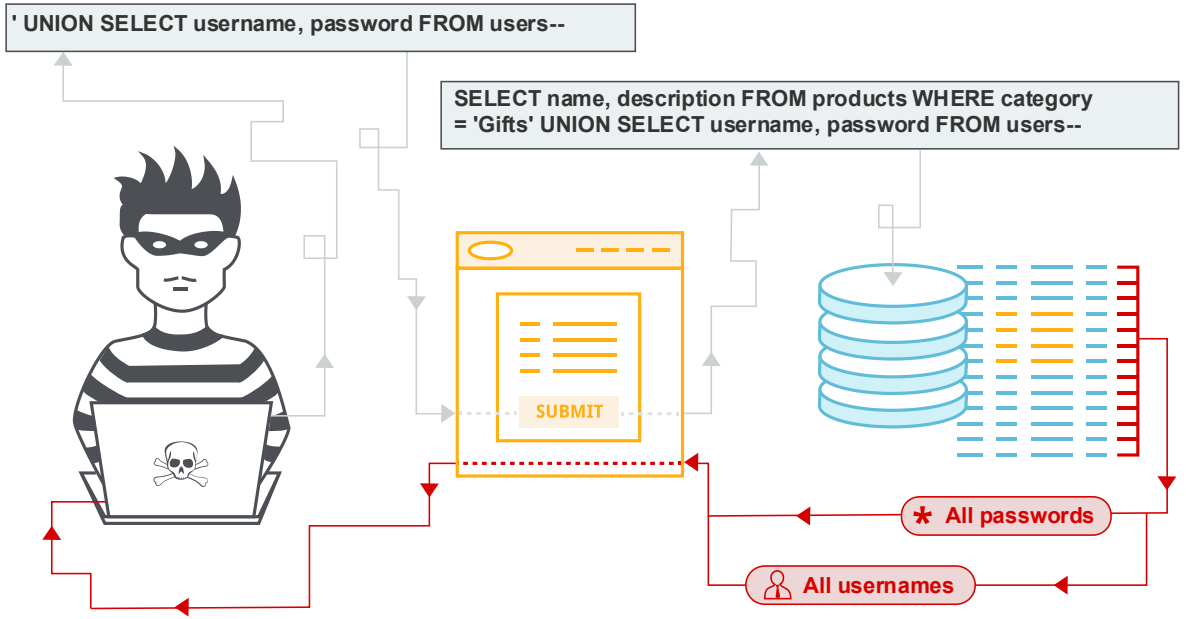
Learning Outcomes

By the end of this lecture, students will be able to:

- **Define** SQL Injection and explain why it is a security threat.
- **Compare** normal and malicious SQL statements and identify the difference.
- **Build** both unsafe and safe stored procedures in Microsoft SQL Server.
- **Apply** parameterized queries to prevent SQL injection attacks.

What is SQL injection (SQi)?

- Structured Query Language (SQL*) Injection is a code injection technique used to modify or retrieve data from SQL databases. By inserting specialized SQL statements into an entry field, an attacker is able to execute commands that allow for the retrieval of data from the database, the destruction of sensitive data, or other manipulative behaviors.

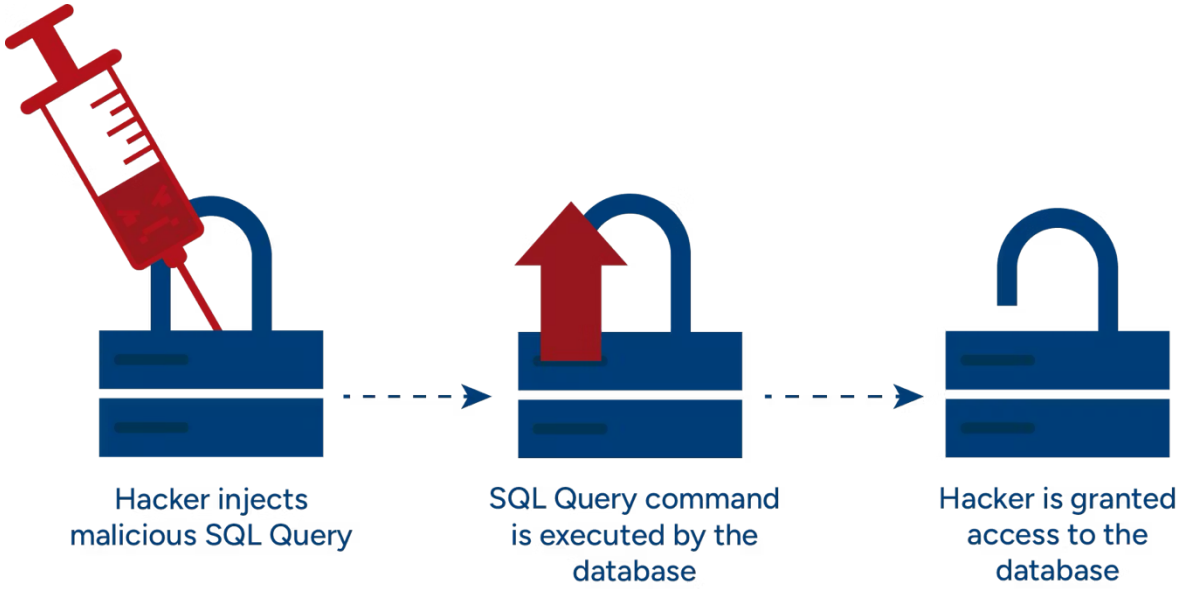


What is SQL injection (SQi)?

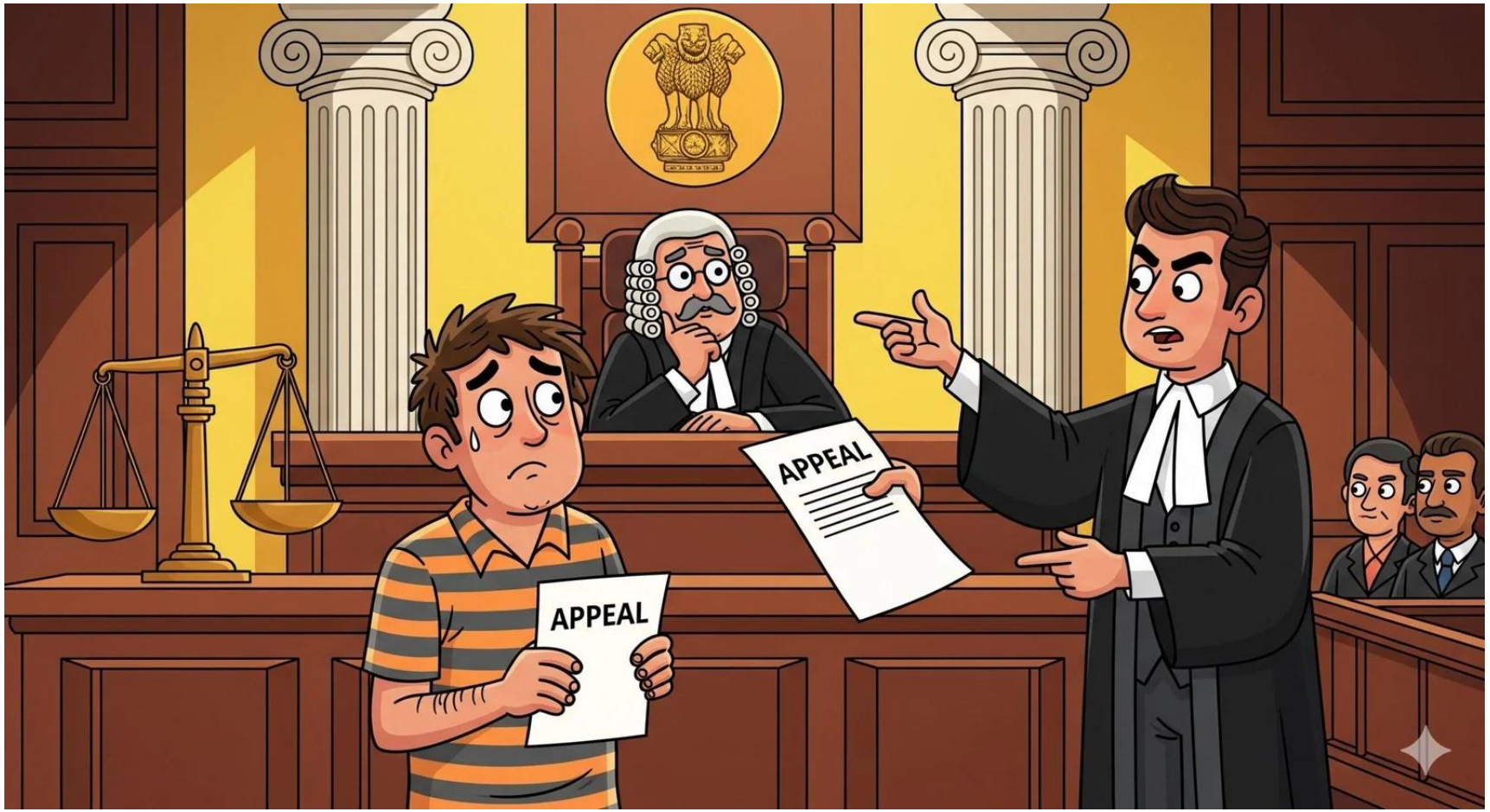
- Using SQL injection, attackers can perform unauthorized database commands on a victim's SQL database.
- With the proper SQL command execution, the unauthorized user is able to spoof the identity of a more privileged user, make themselves or others database administrators, tamper with existing data, modify transactions and balances, and retrieve and/or destroy all server data.

What is SQL injection (SQi)? (cont.)

- In modern computing, SQL injection typically occurs over the Internet by sending malicious SQL queries to an API endpoint provided by a website or service (more on this later in theoretical class). In its most severe form, SQL injection can allow an attacker to gain root access to a machine, giving them complete control.

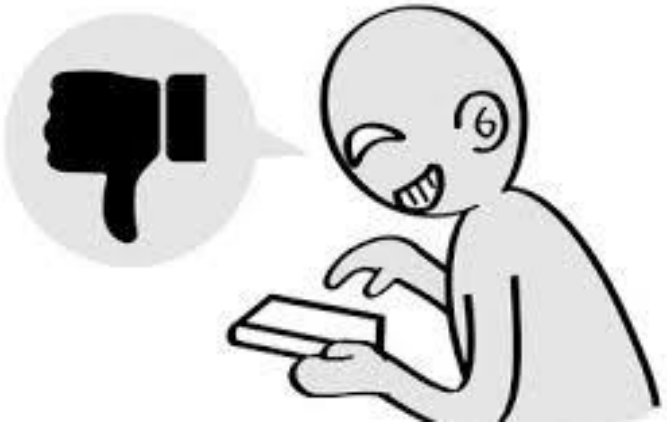


How does a SQL injection attack work?



A system expects normal input (like a username), but an attacker adds extra meaning to it. Because the system doesn't separate plain data from instructions, it treats the input as a command and follows it.

Comparison between normal & malicious SQL statements



Normal SQL Query

A normal SQL query is used to retrieve data from a database based on user input. For example, if a website asks a user to enter their student ID, the system takes that input and creates a query to find matching data. The goal is simple: “find this user and return their information.”

Example:

```
SELECT * FROM students WHERE studentId = 117;
```

In this case, the database searches for the student with ID 117 and returns their record.

How This Applies to Microsoft SQL Server

- The same concept works in Microsoft SQL Server. It uses SQL (Structured Query Language) to communicate with databases and retrieve data. When a user enters a value (like a student ID), the application sends a query to the SQL Server to get the requested information.

Example in code:

```
studentId = getRequestString("studentId");
lookupStudent = "SELECT * FROM students WHERE studentId = " + studentId;
```

SQL Injection Query (General Example)

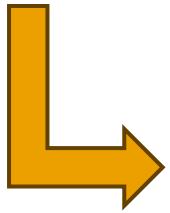
- In this example, an attacker enters a malicious input instead of a normal student ID. Instead of typing a number, they include additional SQL logic:

Malicious input:

```
117 OR 1=1
```

The application takes this input and builds the SQL query without validating it.

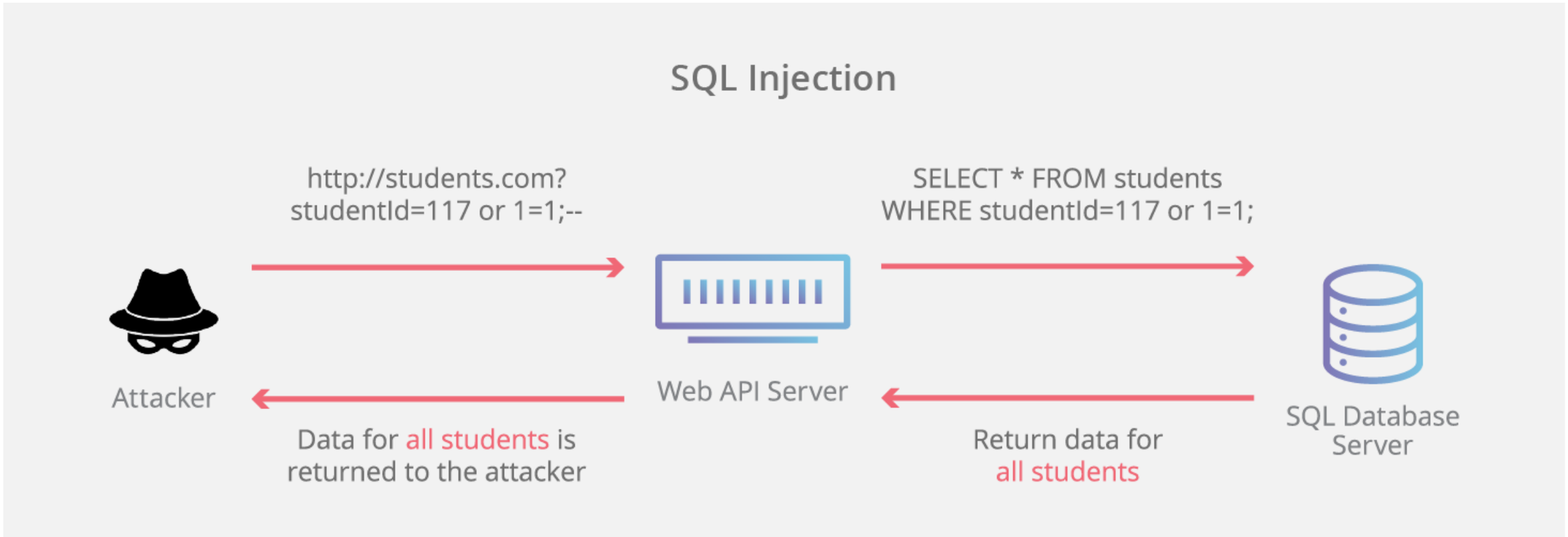
```
SELECT * FROM students WHERE studentId = 117 OR 1=1;
```



Normally, the query searches for a specific student ID. However, the added condition OR 1=1 is always true. This means the database will return **all records** from the students table instead of just one.

How This Applies to Microsoft SQL Server

- This behavior can occur in Microsoft SQL Server because it executes SQL statements exactly as they are written. When a web application sends a query to SQL Server, the database does not know whether the input came from a trusted source or a user form.
- If user input is directly inserted into a query without proper validation or protection, Microsoft SQL Server will execute it as part of the SQL command.
- For example, instead of returning a single student record, the query may be modified to include always-true conditions such as OR 1=1. In this case, SQL Server processes the full condition and returns all rows from the table instead of just one record.
- As a result, sensitive information stored in the database can be exposed.


SQL Injection
http://students.com?
studentId=117 or 1=1;--
SELECT * FROM students
WHERE studentId=117 or 1=1;
Attacker
Web API Server
SQL Database
Server
Data for all students is
returned to the attacker
Return data for
all students

SQL Injection (SQLi) - Overview

- SQL injection is an attack that targets weak websites or APIs connected to databases.
- An API is the system that allows a server to receive requests and return data from a database.
- Attackers use automated tools to scan websites for input fields or APIs and test malicious SQL inputs.
- If user input is not properly checked, the database may treat it as a command instead of normal data.
- This can allow attackers to access, modify, or delete sensitive database information.
- Even a single weak input field or API endpoint can expose the entire system.
- It often happens due to tight deadlines, lack of experience, or outdated (legacy) code.
- SQL injection is easy to exploit but also easy to prevent with proper secure coding practices (never trust user input, always validate and parameterize queries).

SQL Injection Lab Project

```
CREATE DATABASE SchoolDB;
GO

USE SchoolDB;
GO
```

Create Database

```
CREATE TABLE Students (
    StudentId INT PRIMARY KEY,
    Name VARCHAR(50),
    Grade VARCHAR(10)
);
GO
```

Create Tables

```sql
INSERT INTO Students VALUES (101, 'Ali', 'A');
INSERT INTO Students VALUES (102, 'Sara', 'B');
INSERT INTO Students VALUES (103, 'Omar', 'A');
INSERT INTO Students VALUES (104, 'Lina', 'C');
GO
```

Insert Data

```sql
SELECT * FROM Students WHERE StudentId = 101;
```

Normal Query
(Safe Behavior)

```
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

creates a stored procedure named GetStudent_Unsafe

defines an **input parameter** for the procedure

start of the procedure body (the code that runs)

```sql
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

Creates a variable named @sql.

- It will store a **SQL query as text**.
- NVARCHAR(MAX) means it can hold a very long string.

```sql
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

- This is the **most important (and dangerous) line**.
- It builds a SQL query by **combining text + user input**
- 'SELECT * FROM Students WHERE StudentId = ' → fixed SQL part
- + @input → user input is added directly

```sql
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

- Executes the SQL query stored in @sql.
- SQL Server takes the full string and runs it as a command
- It does **not check if parts came from user input.**
- SQL Server executes exactly what is inside the string.

```sql
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

End of the procedure

```sql
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(100)
AS
BEGIN
    DECLARE @sql NVARCHAR(MAX);

    SET @sql = 'SELECT * FROM Students WHERE StudentId = ' + @input;

    EXEC(@sql);
END;
GO
```

```sql
EXEC GetStudent_Unsafe '101';
```

Run Procedure (Normal Input), it returns **one student.** Input '101' is passed to the procedure

Why This Procedure is Unsafe?

- It uses **dynamic SQL**
- It directly inserts user input into the query
- It does not validate or restrict input
- SQL Server cannot distinguish:
  - data
  - commands
- As a result the query logic can be changed unexpectedly

Why This Procedure is Unsafe?

- @input → comes from user
- @sql → becomes full SQL command
- EXEC(@sql) → runs it blindly

Unexpected Behavior:

- **Instead of normal query:**

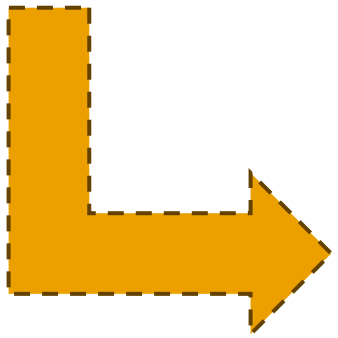
```
SELECT * FROM Students WHERE StudentId = 101;
```

- **The query could behave like:**

```
SELECT * FROM Students WHERE StudentId = 101 OR (condition always true);
```

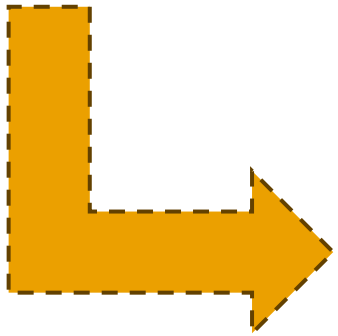
Create Safe Stored Procedure

```
CREATE PROCEDURE GetStudent_Safe
    @id INT
AS
BEGIN
    SELECT * FROM Students WHERE StudentId = @id;
END;
GO
```



- A secure procedure is created
- Input is treated as a **number (INT)**
- No dynamic SQL is used

```
EXEC GetStudent_Safe 101;
```



- SQL Server runs a fixed query
- Only correct data is returned
- Query logic cannot be changed

SQL Injection Lab - Full Script

Step 1

```
-- Create the database
CREATE DATABASE SchoolDB;
GO

-- Switch to the new database
USE SchoolDB;
GO

-- Create the Students table
CREATE TABLE Students (
    StudentId  INT          PRIMARY KEY,
    Name       VARCHAR(50)  NOT NULL,
    Grade      VARCHAR(10)  NOT NULL
);
GO
```

Step 2

```
USE SchoolDB;
GO

INSERT INTO Students (StudentId, Name, Grade)
VALUES
    (101, 'Ali Hassan',    'A'),
    (102, 'Sara Ahmed',    'B'),
    (103, 'Omar Khalid',   'A+'),
    (104, 'Lana Nawzad',   'C'),
    (105, 'Davan Majid',   'B+');
GO

-- Confirm the data was inserted
SELECT * FROM Students;
```

Step 3

```
USE SchoolDB;
GO

-- Normal query: retrieve one student by ID
SELECT * FROM Students
WHERE StudentId = 101;
GO
```

Step 4

```
USE SchoolDB;
GO

-- Creates a procedure that is vulnerable to SQL injection
CREATE PROCEDURE GetStudent_Unsafe
    @input NVARCHAR(MAX)   -- accepts any text (dangerous!)
AS
BEGIN
    -- Builds the SQL query as a string
    DECLARE @sql NVARCHAR(MAX);

    -- User input is directly joined into the query
    SET @sql = 'SELECT * FROM Students WHERE StudentId = '
                + @input;   -- <-- injection point!

    -- Executes whatever string was built
    EXEC(@sql);             -- runs it blindly
END;
GO
```

Step 5

```
USE SchoolDB;
GO

-- Normal input: returns only student 101
EXEC GetStudent_Unsafe '101';
GO

-- The query that gets executed internally:
-- SELECT * FROM Students WHERE StudentId = 101
```

Step 6

```
USE SchoolDB;
GO

-- Malicious input: bypasses the WHERE condition
EXEC GetStudent_Unsafe '101 OR 1=1';
GO

-- The query that gets executed internally:
-- SELECT * FROM Students WHERE StudentId = 101 OR 1=1
-- Because 1=1 is always TRUE, ALL rows are returned!
```

Step 7

```
USE SchoolDB;
GO

-- Secure version: uses a typed parameter, no dynamic SQL
CREATE PROCEDURE GetStudent_Safe
    @id INT   -- input must be a number (INT), nothing else
AS
BEGIN
    -- Fixed query: SQL Server treats @id as data only
    SELECT * FROM Students
    WHERE StudentId = @id;
END;
GO
```

Step 8

```
USE SchoolDB;
GO

-- Normal input: works correctly, returns one record
EXEC GetStudent_Safe 101;
GO

-- Attempt injection: this will FAIL with a type error
EXEC GetStudent_Safe '101 OR 1=1';
GO
-- Error: Conversion failed when converting the nvarchar
-- value '101 OR 1=1' to data type int.
-- SQL Server REJECTS the malicious input entirely.
```

Cleanup - Drop Everything (Optional)

```
USE SchoolDB;
GO

-- Drop the procedures
DROP PROCEDURE IF EXISTS GetStudent_Unsafe;
DROP PROCEDURE IF EXISTS GetStudent_Safe;
GO

-- Drop the table
DROP TABLE IF EXISTS Students;
GO

-- Switch away before dropping the database
USE master;
GO

DROP DATABASE IF EXISTS SchoolDB;
GO
```

References

- Microsoft. (n.d.). *SQL injection*. Microsoft Learn. Retrieved April 27, 2026, from https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver17


Any
Question