Practical Lab Exercise: SQL Injection (SQLi) Preventions

Block 1: Reset & Create Database

```
IF DB_ID('SqlInjectionLab') IS NOT NULL
BEGIN
    ALTER DATABASE SqlInjectionLab SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
    DROP DATABASE SqlInjectionLab;
END;
GO

CREATE DATABASE SqlInjectionLab;
GO

USE SqlInjectionLab;
GO
```

Does this database already exist?

Forces everyone out of the database close any open connections

Block 2: Create Users Table

```
IF OBJECT_ID('dbo.Users', 'U') IS NOT NULL
    DROP TABLE dbo.Users;
GO

CREATE TABLE dbo.Users (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    Username NVARCHAR(50),
    Password NVARCHAR(50)
);
GO
```

Does the Users table already exists?

Block 3: Insert Sample Data

```
IF NOT EXISTS (SELECT 1 FROM dbo.Users)
BEGIN
    INSERT INTO dbo.Users (Username, Password)
    VALUES
    ('admin', 'admin123'),
    ('user1', 'pass1'),
    ('test', 'test123');
END;
GO
```

Block 4: *Classic SQL Injection (In-band)*

- User input is directly concatenated into the query
- This is the **root vulnerability** all other types rely on

```
DECLARE @Username NVARCHAR(100) = 'admin';
DECLARE @Password NVARCHAR(100) = 'admin123';

DECLARE @sql NVARCHAR(MAX);

SET @sql =
'SELECT * FROM dbo.Users WHERE Username = '''
+ @Username + ''' AND Password = ''' + @Password + '''';

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

Create a variable and store “admin”

Create a variable to hold a SQL query

Build a SQL query by combining text + user input

```
SELECT * FROM dbo.Users
WHERE Username = @Username
AND Password = @Password;
```

Prevention

- Use parameterized
- Avoid dynamic SQL

- First query = unsafe (builds SQL using input)
- Second query = safe (uses parameters)

Block 5: Tautology-based SQL Injection

- Condition like OR 1=1 makes query always true
- Used to bypass authentication

```
DECLARE @input NVARCHAR(200);

SET @input = ''' OR 1=1 --';

DECLARE @sql NVARCHAR(MAX);

SET @sql =
'SELECT * FROM dbo.Users WHERE Username = '''
+ @input + '''';

PRINT 'Always-true condition example';
PRINT @sql;
GO
```

OR 1=1 = always TRUE, Database returns all users

Block 6: UNION-based SQL Injection

- Combines original query with another query
- Used to extract additional data

```
DECLARE @input NVARCHAR(200);

SET @input = ''' UNION SELECT name, type FROM sys.objects --';

DECLARE @sql NVARCHAR(MAX);

SET @sql =
'SELECT Username, Password FROM dbo.Users WHERE Username = '''
+ @input + '''';

PRINT 'UNION query example';
PRINT @sql;
GO
```

Block 7: Error-based SQL Injection

- Relies on database error messages
- Errors reveal structure (tables, columns, etc.)

```
BEGIN TRY
    SELECT 1 / 0;
END TRY
BEGIN CATCH
    PRINT 'Error-based behavior example';
    PRINT ERROR_MESSAGE();
END CATCH;
GO
```

Block 8: *Blind SQL Injection (Boolean-based)*

- No direct data returned
- Attacker infers information from TRUE/FALSE responses

```
DECLARE @result BIT;

IF EXISTS (SELECT 1 FROM dbo.Users WHERE Username = 'admin')
    SET @result = 1;
ELSE
    SET @result = 0;

PRINT 'Boolean logic example';
PRINT @result;
GO
```

Checks if the admin exist or not!

Block 9: Blind SQL Injection (Time-based)

- Uses delays (WAITFOR)
- Response time reveals TRUE/FALSE

```
PRINT 'Time delay example';

WAITFOR DELAY '00:00:03';

PRINT 'Finished after delay';
GO
```

Pause execution for 3 seconds

Block 10: *Stacked Queries SQL Injection*

- Executes multiple SQL statements in one input
- Can modify or delete data

```
DECLARE @sql NVARCHAR(MAX);

SET @sql =
'SELECT * FROM dbo.Users; SELECT COUNT(*) FROM dbo.Users;';

PRINT 'Stacked query example';
EXEC(@sql);
GO
```

Two queries in one string

Block 11: Out-of-Band SQL Injection

- Data is sent outside the database (DNS/HTTP)
- Used when direct response is not available

```
PRINT 'External interaction example';
PRINT 'Data could be sent outside the database through network requests';
GO
```