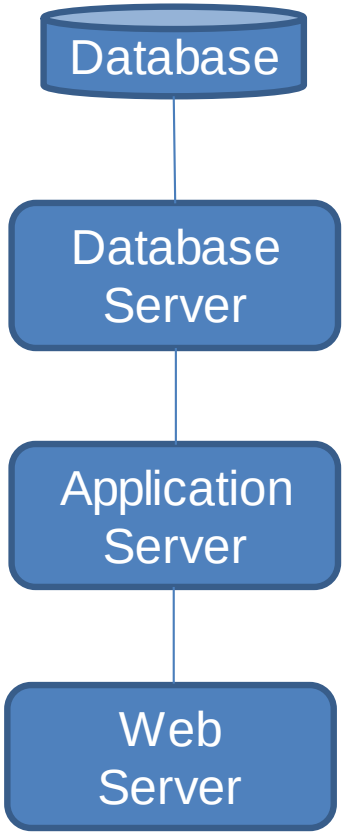
Server Environment

Access – shell vs application

- SQL to create, manipulate, and query relational databases via
  - Database system's shell
  - C, Python, Java programs accessing information from a database
- Database applications may be programmed by Frameworks (Django, Ruby on Rails)
  - Generate calls to database system
- Use case: Java program making SQL commands against MariaDB

Three Tier Architecture

- Database often used in conjunction with middleware (application/web servers)
  - Web Server (View)
  - Application Server (Controller)
  - Database Server (Model)
- Processes in different tiers may on same machine or on different machines
  - Loopback interfaces (127.0.0.1)



JDBC

- Java Database Connectivity
  - API allowing external access to SQL database manipulation and update commands
  - General programming environment with library interface with the database
- Database vendors supports their own communication protocols over TCP
  - Each vendor provide Driver Class for JDBC
- JDBC is written in Java(called TYPE 4) for portability

JDBC…

- import java.sql.*
- Open connection between client and DB server
- Get a connection from connection factory

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost/testdb?"+
        "user=root&password=root");
```

  - Connection to mysql process running on same machine(localhost) on the default port(3306)
  - String after ? is the username and password

JDBC…

- Create a statement object so we can submit SQL statements to the driver

  Statement stmt = conn.createStatement();

- Submit the statement by calling executeQuery method

  rs = stmt.executeQuery("SELECT * FROM T1")

- rs is ResultSet object which encapsulates cursor (pointer) into the rows returned by query
  - next() method used traverse the result set
    - Advances one row

JDBC…

- Use the getInt(col1) and getString(col2) methods of ResultSet object to get current values of column

```
while (rs.next()) {
    int a = rs.getInt("col1");
    String b = rs.getString("col2");
}
```

- Compile javac programname.java
- Run and make sure MYSQL driver jar is in class path
  - java –cp .:/path-to-mysql-jar programname

Ad hoc Queries

- Dynamically alter the string passed to executeQuery or executeUpdate
- SELECT query

```
rs = stmt.executeQuery("SELECT * FROM t1 WHERE a1="+a);
```

- INSERT, UPDATE, DELETE (to execute DDLs, DMLs)

```
rs = stmt.updateQuery("DELETE * FROM t1 WHERE a1="+a);
```

- Cons of using ad hoc queries?

1. Ad hoc Queries – Slow

- Dynamically altering the string passed to executeQuery or executeUpdate will perform slow

```
rs = stmt.executeQuery("SELECT * FROM t1 WHERE a1="+a);
```

- If value of variable(a) is changed 10 times in the application, then ad hoc queries will be compiled by DBMS query compiler 10 times
  - Let's say your application will be called to insert a lot of new records to the database

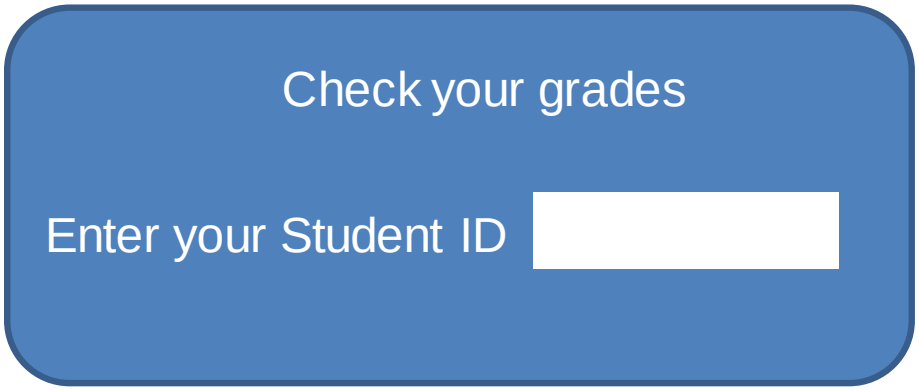
Solution – PreparedStatements

```
PreparedStatement pstmt  = conn.preparestatement
    ("SELECT * FROM t1 WHERE a1=? and a2 =?");
pstmt.setString(1,"cs157a");
pstmt.setInt(2,100)
ResultSet rs = pstmt.executeQuery();
```

- DBMS query compiler compiles the query and then reuse it when different values are provided for the ? components

2. Ad hoc Queries – Security

- SQL Injection Use Case 1



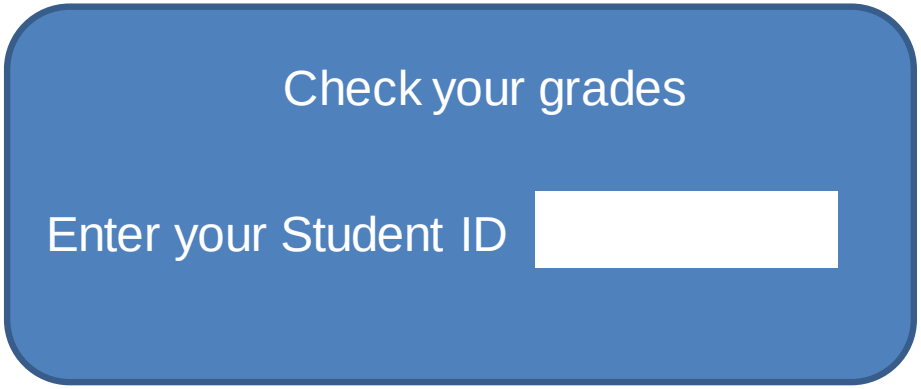
- Query = "SELECT * FROM STUDENT NATURAL JOIN

  TAKES WHERE year=2019 AND

  ID="+ request.form['StudentID']

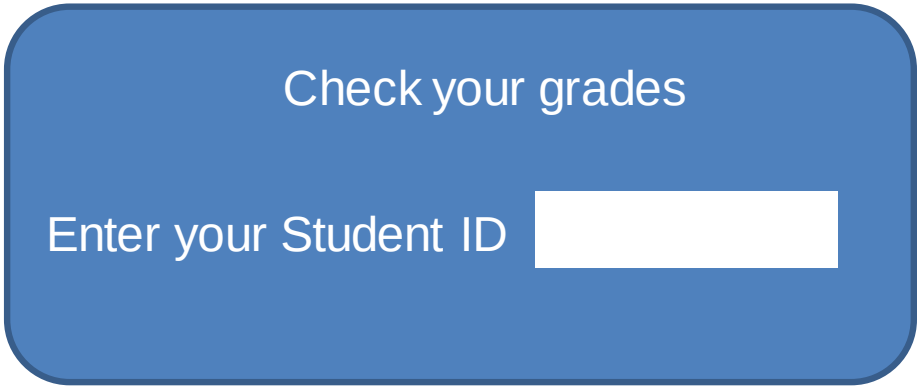
- Injection: set StudentID variable to **0 OR 1=1**
  - Shows grades of all students

SQL Injection - Use Case 2



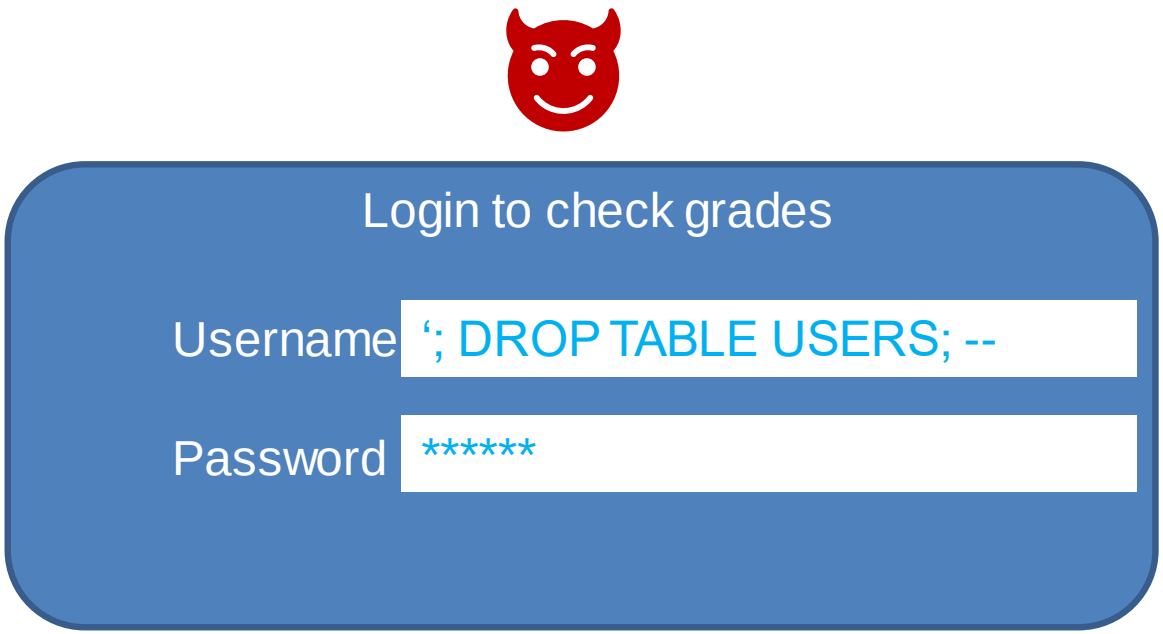
- Query = "SELECT * FROM STUDENT NATURAL JOIN TAKES WHERE year=2019 AND ID="+ request.form['StudentID']
- Injection: set StudentID variable to
  - **0 UNION SELECT * FROM instructor**
  - **0 UNION SELECT * FROM creditcards**

SQL Injection - Use Case 3



- Query = "SELECT * FROM STUDENT NATURAL JOIN TAKES WHERE year=2019 AND ID="+ request.form['StudentID']
- Injection: set StudentID variable to
  - **0; DROP TABLE instructor;--**
  - Denial of Service attack

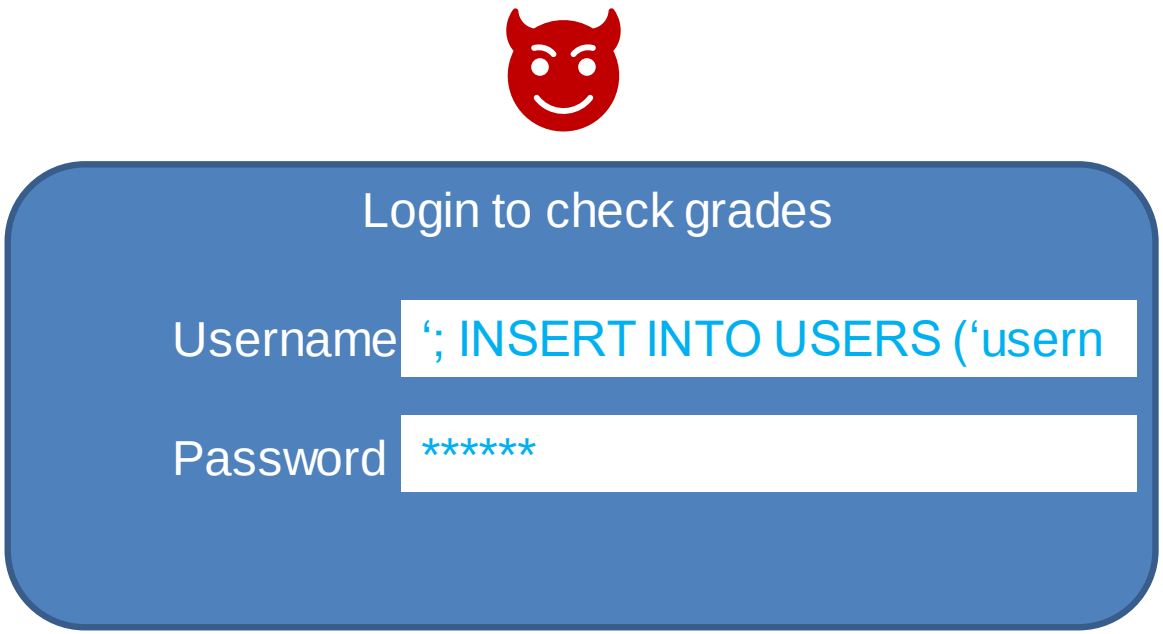
SQL Injection - Use Case 3.1



Web Server will make malicious query:

SELECT password
FROM USERS
WHERE username IS "; DROP TABLE users;--

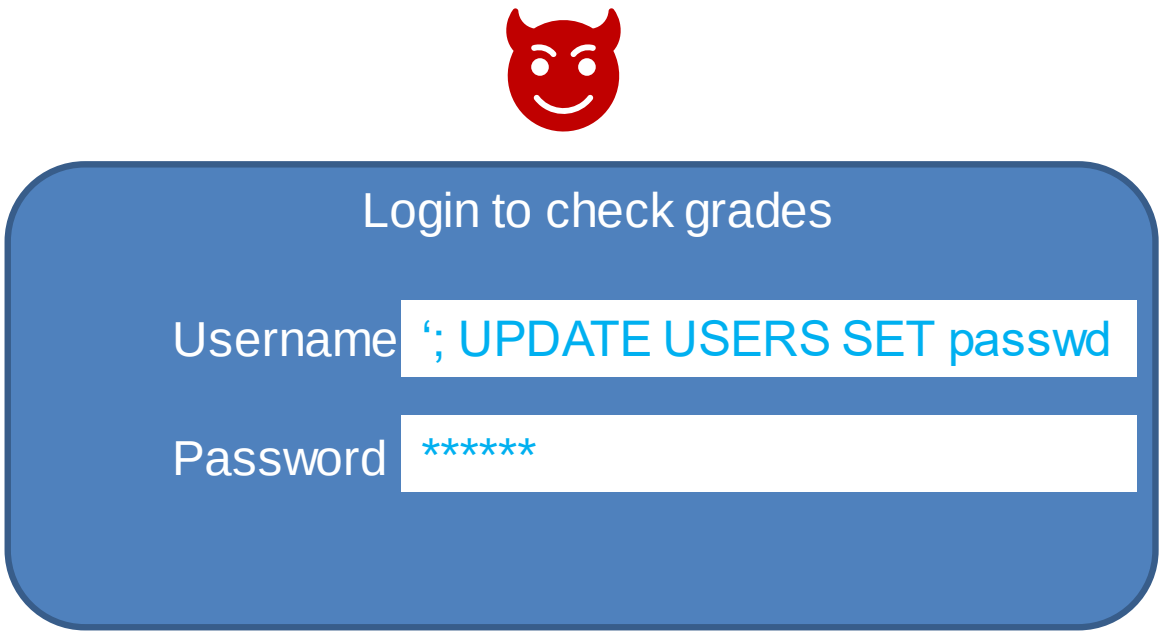
SQL Injection - Use Case 4



Web Server will make malicious query:

SELECT password
FROM USERS
WHERE username IS "; INSERT INTO USERS ('username','password') VALUES('hacker','123456');--

SQL Injection - Use Case 4.1



Web Server will make malicious query:

SELECT password
FROM USERS
WHERE username IS ";UPDATE USERS SET passwd='cracked' WHERE username='admin'--

Solution – SQL Injection

- Input validation
  - Filter semicolons, percent symbols, hyphens, apostrophes
  - Check data types (e.g. make sure it's an integer)
  - Escape quotes for string inputs
- Prepared statements
  - Allow creation of static queries with bind variables
    - Bind variables: ? placeholders guaranteed to be data

SQL Injection