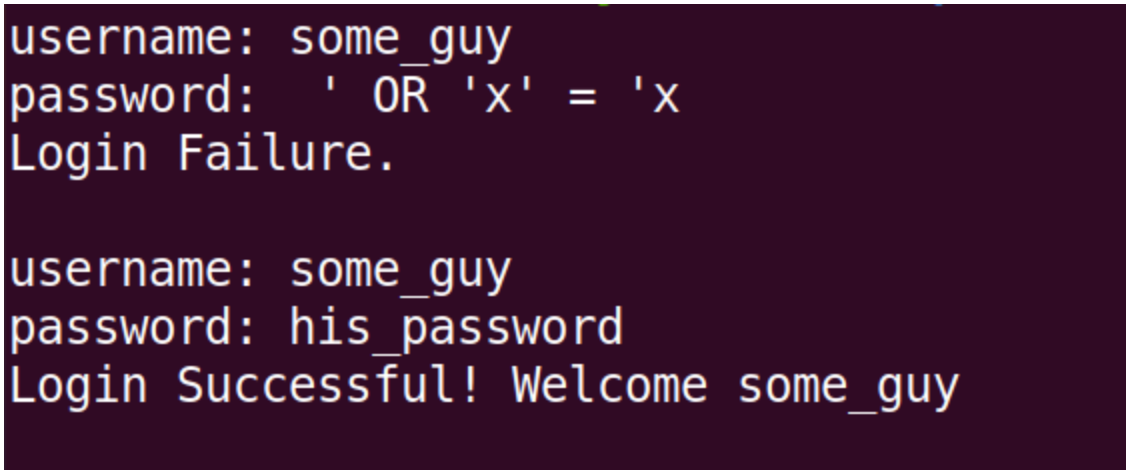
JAVA

```
username: some_guy
password:  ' OR 'x' = 'x
Login Successful! Welcome some_guy
```

```
import java.io.Console;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;

/**
 * Main execution class for sql_injection exercise. Prompts user for username
 * and password to lookup in the accompanying sqlite3 database.
 *
 * @author Joseph Eichenhofer
 *
 */
public class Main {

	private static final String DB_URL = "jdbc:sqlite:users.db";

	/**
	 * Prompt user for username and password. Displays login success or failure
	 * based on lookup in user database.
	 *
	 * @param args
	 *          n/a
	 */
	public static void main(String[] args) {
		Console terminal = System.console();

		if (terminal == null) {
			System.out.println("Error fetching console. Are you running from an IDE?");
			System.exit(-1);
		}

		while (true) {
			// get username and password from user
```

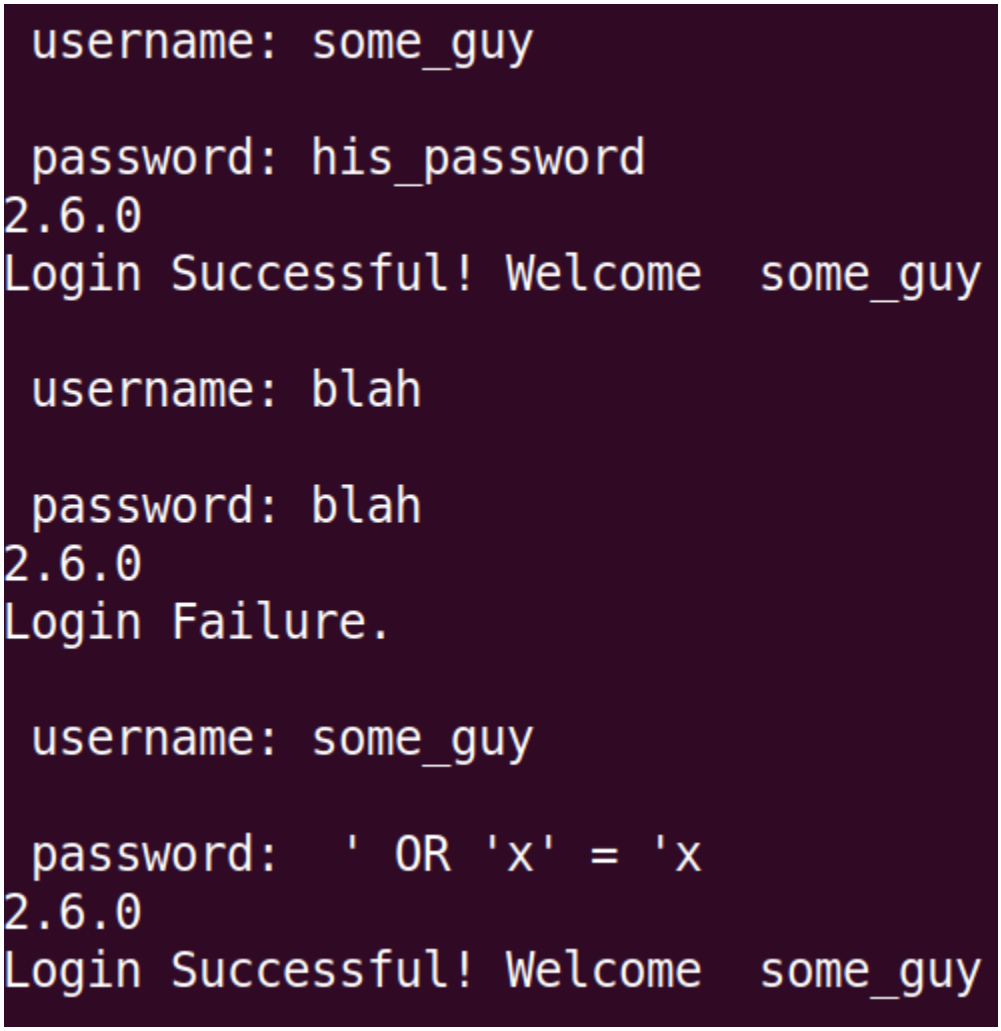
```
            String username = terminal.readLine("username: ");
            if (username.toLowerCase().equals("exit"))
                  break;
            String password = terminal.readLine("password: ");

            // check username and password
            boolean loginSuccess = false;
            try {
                  loginSuccess = checkPW(username, password);
            } catch (SQLException e) {
                  System.out.println("Database Error.");
                  // printing the stack trace only leads to heartbreak, the user doesn't
                  // need to see all that secret info or could use for attack
                  // e.printStackTrace();
            }

            if (loginSuccess)

                  System.out.println("Login Successful! Welcome " + username);
            else
                  System.out.println("Login Failure.");

            // separate iterations for repeated attempts
            System.out.println();
      }
}

/**
 * Connect to the sample database and check the supplied username and password.
 *
 * @param username
 *          username to check
 * @param password
 *          password to check for given username
 * @return true iff the database has an entry matching username and password
 * @throws SQLException
 *          if unable to access the database
 */
private static boolean checkPW(String username, String password) throws SQLException {
      // declare database resources
      Connection c = null;
      // declare prepared statement instead of basic Statement
      PreparedStatement pstmt = null;
      ResultSet results = null;
```

```
        try {
            // connect to the database
            c = DriverManager.getConnection(DB_URL);

            // check for the username/password in database
            // Use prepared statement to avoid parsing user input
            pstmt = c.prepareStatement("SELECT COUNT(*) AS count FROM USERS
WHERE username == ? AND password == ?");
            // put sanitized user input into prepared statement in order to execute clean query
            pstmt.setString(1, username);
            pstmt.setString(2, password);
            results = pstmt.executeQuery();

            // if no user with that username/password, return false; otherwise must be true
            if (results.getInt("count") == 1)
                return true;
            else
                return false;

        } finally {
            // release database resources (ignore any exceptions including null pointer)
            try {
                results.close();
            } catch (Exception e) {
            }
            // close the prepared statement
            try {
                pstmt.close();
            } catch (Exception e) {
            }
            try {
                c.close();
            } catch (Exception e) {
            }
        }
    }
}
```



The initial attack was a SQL injection where I forced the WHERE check to be true by adding an OR statement that was obviously true by default; ‘x’ =’x’. The SQL query, being unchecked, unsanitized, and unprepared meant that even though my input was nonsense to be searching, the program automatically parsed it and integrated my malicious code as part of the query. The mitigation to this was in two parts. First, I had to stop the console from printing the stack trace whenever a SQLException was thrown because a) normal users have no need to see that and b) an attacker could use that sort of excessive information to launch an attack like I did. The second part of the mitigation was utilizing SQL’s prepared statements ability to prevent SQL Injections of any kind from happening again. This works because the input that I give is not immediately and automatically parsed by the program because, as w3schools put it, “If the original statement template is not derived from external input, SQL injection cannot occur.” Also, by changing the requirement for there to be not zero results to exactly one result is an added barrier against SQL injection attacks because an OR based SQL injection attack could lead to many, many results returned rather than just 1 and give a logged in status.

PYTHON



VULNERABLE CODE

```
import sqlite3
from sqlite3 import Error
import os

def create_connection(db_file):
    """ create a database connection to a SQLite database """
    conn = None
    try:
        conn = sqlite3.connect(db_file, uri = True)
        print(sqlite3.version)
        return conn
    except Error as e:
        print(e)
```

```
    return conn

def checkPW(u, p):
    # declare result set, result set list, set logged in to false by default,
    # generate a connection to the database, and create a null cursor
    rs = None
    rs_list = 0
    logged_in = False
    conn = create_connection(os.path.join("file:mydb", "pythonsqlite.db"))
    c = None
    #stmt = "SELECT * FROM sqlite_master WHERE tbl_name = 'users'"
    stmt = "SELECT COUNT(*) AS count FROM users WHERE login = '" + u + "' AND password = '" + p +
"'"
    try:
        c = conn.cursor()
        rs = c.execute(stmt)
        rs_list = [i for i in rs]
    except Error as e:
        print(e)
    if (rs_list[0][0] == 0):
        logged_in = False
    else:
        logged_in = True
    conn.close()
    return logged_in

if __name__ == '__main__':

    while 1:
      username = input("\n username: ")
      if username == "exit":
        quit()
      password = input("\n password: ")

      loginSuccess = False
      try:
        loginSuccess = checkPW(username, password)
      except:
        print(loginSuccess)

      if (loginSuccess):
```

```
 username: some_guy

 password: his_password
2.6.0
Login Successful! Welcome  some_guy

 username: blah

 password: blah
2.6.0
Login Failure.

 username: some_guy

 password:  ' OR 'x' = 'x
2.6.0
Login Successful! Welcome  some_guy
```

REPAIRED CODE

```
import sqlite3
from sqlite3 import Error
import os

def create_connection(db_file):
    """ create a database connection to a SQLite database """
    conn = None
    try:
        conn = sqlite3.connect(db_file, uri = True)
        print(sqlite3.version)
        return conn
    except Error as e:
        print(e)

    return conn
```

```
def checkPW(u, p):
    # declare result set, result set list, set logged in to false by default,
    # generate a connection to the database, and create a null cursor
    rs = None
    rs_list = 0
    logged_in = False
    conn = create_connection(os.path.join("file:mydb", "pythonsqlite.db"))
    c = None
    #stmt = "SELECT * FROM sqlite_master WHERE tbl_name = 'users'"
    stmt = "SELECT COUNT(*) AS count FROM users WHERE login = '" + u + "' AND password = '" + p +
"'"
   try:
        c = conn.cursor()
        # prepared statement to thwart SQL injection attacks
        rs = c.execute("""
            SELECT COUNT(*) AS count
            FROM users
            WHERE login = ? AND password = ?""", (u, p))
        rs_list = [i for i in rs]
    except Error as e:
        print(e)
    # No longer check if no results, result set should have exactly 1 possible answer
    if (rs_list[0][0] != 1):
        logged_in = False
    else:
        logged_in = True
    conn.close()
    return logged_in

if __name__ == '__main__':

    while 1:
      username = input("\n username: ")
      if username == "exit":
        quit()
      password = input("\n password: ")

      loginSuccess = False
      try:
        loginSuccess = checkPW(username, password)
      except:
        print(loginSuccess)

      if (loginSuccess):
```

The initial attack was a SQL injection where I forced the WHERE check to be true by adding an OR statement that was obviously true by default; ‘x’ =’x’. The SQL query, being unchecked, unsanitized, and unprepared meant that even though my input was nonsense to be searching, the program automatically parsed it and integrated my malicious code as part of the query. The mitigation to this was in two parts. First, I had to stop the console from printing the stack trace whenever a SQLException was thrown because a) normal users have no need to see that and b) an attacker could use that sort of excessive information to launch an attack like I did. The second part of the mitigation was utilizing SQL’s prepared statements ability to prevent SQL Injections of any kind from happening again. This works because the input that I give is not immediately and automatically parsed by the program because, as w3schools put it, “If the original statement template is not derived from external input, SQL injection cannot occur.” Also, by changing the requirement for there to be not zero results to exactly one result is an added barrier against SQL injection attacks because an OR based SQL injection attack could lead to many, many results returned rather than just 1 and give a logged in status.