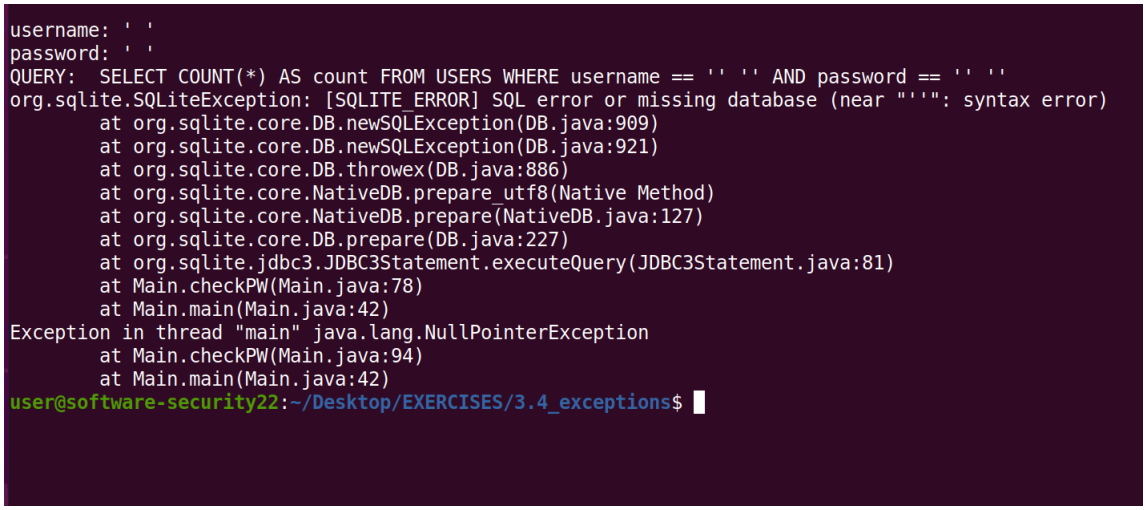
CS 542 – Introduction to Software Security
Exercise on Exception Handling Vulnerability

Binhao Chen (bchen276@wisc.edu), Steven Yang (yang558@wisc.edu)

Due: September 29 at 2:30pm

1 1: Screenshots showing the input(s) used for the attack, and the output(s) you got from the system.



```
username: ' '
password: ' '
QUERY:  SELECT COUNT(*) AS count FROM USERS WHERE username == '' '' AND password == '' ''
org.sqlite.SQLiteException: [SQLITE_ERROR] SQL error or missing database (near "'''": syntax error)
        at org.sqlite.core.DB.newSQLException(DB.java:909)
        at org.sqlite.core.DB.newSQLException(DB.java:921)
        at org.sqlite.core.DB.throwex(DB.java:886)
        at org.sqlite.core.NativeDB.prepare_utf8(Native Method)
        at org.sqlite.core.NativeDB.prepare(NativeDB.java:127)
        at org.sqlite.core.DB.prepare(DB.java:227)
        at org.sqlite.jdbc3.JDBC3Statement.executeQuery(JDBC3Statement.java:81)
        at Main.checkPW(Main.java:78)
        at Main.main(Main.java:42)
Exception in thread "main" java.lang.NullPointerException
        at Main.checkPW(Main.java:94)
        at Main.main(Main.java:42)
user@software-security22:~/Desktop/EXERCISES/3.4_exceptions$
```

Figure 1: input(s) used for the attack, and the output(s) you got from the system.

2 Your commented code for the mitigation.

```java

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

/**
 * Main execution class for exceptions 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 == null || username.toLowerCase().equals("exit"))
                break;
            String password = terminal.readLine("password: ");

            // check username and password
            if (checkPW(username, password))
                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
     */
```

```java
    private static boolean checkPW(String username, String password) {
        // declare database resources
        Connection c = null;
        Statement statement = null;
        ResultSet results = null;

        boolean success = true;

        String sqlQuery = "SELECT COUNT(*) AS count FROM USERS WHERE username == '" + username + "' AND password == '"
                + password + "'";

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

            // check for the username/password in database
            statement = c.createStatement();
            results = statement.executeQuery(sqlQuery);

            // if no user with that username/password, return false
            if (results.getInt("count") == 0)
                return false;

        } catch (SQLException ex) {
            // sql error, debug info:
            // System.err.print("QUERY:\t");
            // System.err.println(sqlQuery);
            // ex.printStackTrace(System.err);

            // If there is a SQLException, then we set the
            // boolean variable 'success' to be false so that
            // the output will not show the login is successful.
            // This is the third problem with error handling that is fixed now.
            success = false;

            // Here we report the generic error message to the user
            // on the error stream.
            System.err.println("Login failed, username or password is invalid.");
        }
        // cleanup sql objects
        try {
            results.close();
        } catch (SQLException ex) {
        }

        // Here we add this catch clause to handle the
        // java's NullPointerException that crashes the program.
        catch (NullPointerException e) {
        }
        try {
            statement.close();
        } catch (SQLException ex) {
        }
        try {
            c.close();
        } catch (SQLException ex) {
        }

        return success;
    }
}
```

3 Screenshots showing the attack input(s) used above and fixed output(s) after applying the mitigation for the vulnerability.

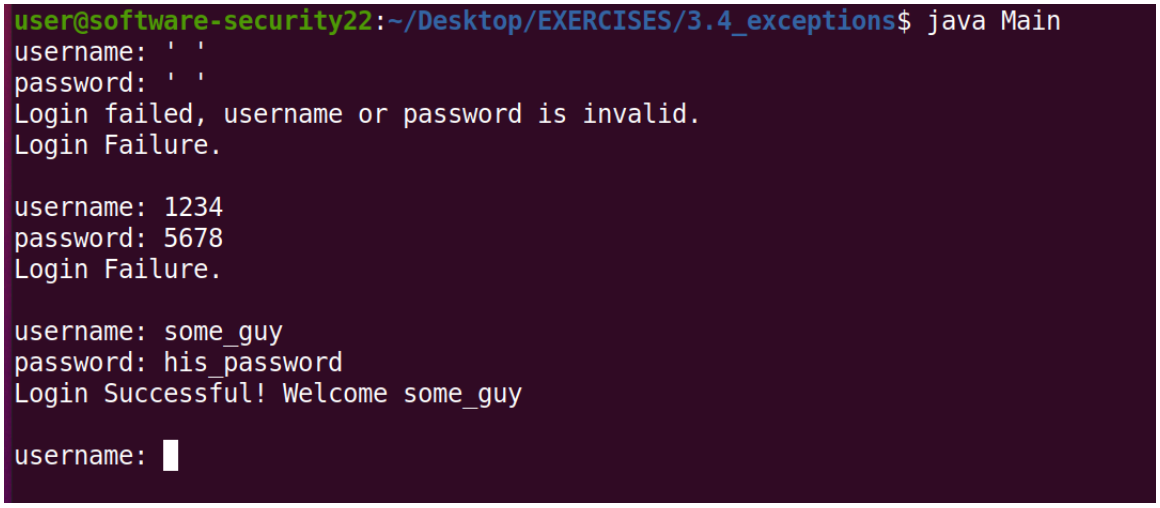


Figure 2: input(s) used for the attack, and the output(s) you got from the system.

4 A explanation on your attack and your mitigation.

Attack: To be able to trigger a SQLException, we need to pass in a malicious user input at the terminal to make the SQL query invalid, more specifically, to cause a syntax error. We can pass in ' ' (two single quotation marks with a blank inside) to cause the query invalid (or we can add anything within the two single quotation marks ' '). Then the program will trigger a SQLException (org.sqlite.SQLiteException) and we will be able to see information about the program, how the SQL query was executed, and database structure.

Mitigation: We first mitigate the unhandled NullPointerException by adding one more catch block after the results.close(). Then we mitigate the information leak in our SQLException handling by only printing out a generic error message saying that the login has failed. Any other information will not be revealed to the user. Lastly, we also need to handle the "Login Successful!" message even if the login actually failed. We can handle it by changing the success back to false after the SQLException has been caught. Therefore, the false will be returned after all exceptions have been caught and "Login Failure." will be printed out instead.

5 Explain which is that second Exception, and why it is generated

The second exception is the NullPointerException in java. When a SQLException is triggered, the 'results' variable (an instance of Class ResultSet) will not store anything since the query is invalid. Then after the SQLException is caught, the results.close() will cause a NullPointerException.