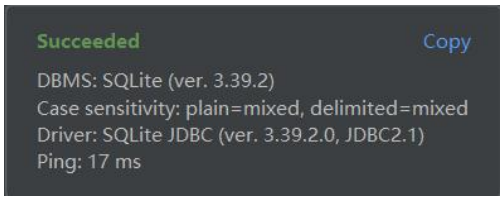
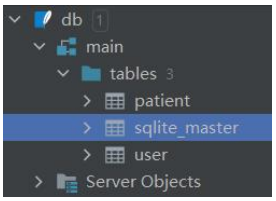
Analysis of Security Flaws

1: In this task, the experiment downloaded and used a jar package called Driver: SQLite JDBC (ver. 3.39.2.0, JDBC2.1) to connect the db.sqlite3, then forming a UI model.



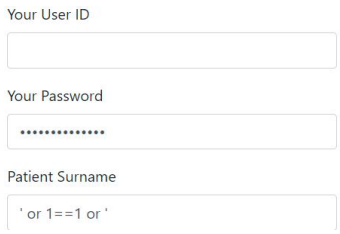


2:After adding compilation to the dependencies of the build.gradle file, this task is successfully executed. compile("org.slf4j:slf4j-simple:1.7.9").

3: There are three pages in total, login,invalid ,details. And an warning page with a link to jetty.

4:Through the test on the login interface, userId actually refers to the user name, and the experiment found that when entering the user name and patient name, you can illegally query or modify the patient information, or even modify the user information, by replacing them with sql codes and injecting sql statement.

Illegal login: input ' or 1==1 or ' into password and Patient Surname frame



Obtain all patient information: ' or 1==1 or '

Patient Details

Surname	Forename	Date of Birth	GP Identifier	Treated For
Davison	Peter	1942-04-12	4	Lung cancer
Baird	Joan	1927-05-08	17	Osteoarthritis
Stevens	Susan	1989-04-01	2	Asthma
Johnson	Michael	1951-11-27	10	Liver cancer
Scott	Ian	1978-09-15	15	Pneumonia

5: By observing login.html, AppServlet code and db.sqilte3 database, the research found that the back end and front end did not do any encryption to the user's information, and the database was stored in plaintext. The back end code was stored in the database without any encryption. At the same time, the front end transmitted the user's information without encrypting it, and still relied on '&' for string splicing, which easily led to data leakage.

The experiment uses an application, Fiddler Classic, which can intercept the front and rear data transmission. It can easily obtain the data in the transmission process without encryption, which will cause serious data leakage.

There is the captured request information:

#	Result	Protocol	Host	URL	Body	Caching
1	200	HTTP	Tunnel to	maxcdn.bootstrapcdn.co...	0	
2	200	HTTP	localhost:8080	/	1,798	
3	200	HTTP	Tunnel to	cdnjs.cloudflare.com:443	0	
4	200	HTTP	Tunnel to	code.jquery.com:443	0	
5	200	HTTP	Tunnel to	www.googleapis.com:443	0	
6	200	HTTP	localhost:8080	/favicon.ico	2,117	

```
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
Origin: http://localhost:8080
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8080/
Accept-Encoding: gzip, deflate, br
Accept-Language: en,zh-CN;q=0.9,zh;q=0.8
Cookie: Idea-96346db1=eda9f48c-b340-46fe-a856-c15d559e2c0a

username=nde&password=wysiwyg0&surname=Davison
```

Implementation of Security Fixes

1: The experiment selects the sql injection described in task4

2: After testing, SQL injection is well prevented and the whole program works well

3:The experiment uses prepareStatement to process SQL statements.

In the scenario of using JDBC directly, if there are spliced SQL statements in the code, injection is likely to occur, but the original code uses the same method as splicing.

The safe writing method is to use parameterized queries, that is, parameter binding (? Placeholder) and PreparedStatement are used in SQL statements.

PreparedStatement is the sub interface of Statement, which can pass in SQL statements with placeholders, and provides a method to supplement placeholder variables.

It should be noted here that the use of PreparedStatement does not mean that injection will not occur. If there are spliced SQL statements before the use of PreparedStatement , injection will still occur.

So the experiment uses Connection, PreparedStatement, ResultSet and SQLException to jointly process SQL statements.

Core modification:

Modification of sql query:

private static final String AUTH_QUERY = "select * from user where username=? and password=?";

private static final String SEARCH_QUERY = "select * from patient where surname like ?";

Part of core code:

Connection connection = null;PreparedStatement statement = null;ResultSet resultSet = null;Object ret = null;try {connection = database;statement =connection.prepareStatement(AUTH_QUERY);

//set param, process the statement and obtain the return value} catch (SQLException e) {e.printStackTrace();} finally {if (statement != null)statement.close();}