Vulnerable App Penetration Testing Report

Mobile Hacking Lab

Food Store Challenge

Litio7

March 22th 2026

Version v1.0

Contents

1 Mobile Hacking Lab Food Store Challenge

1.1 Introduction

This challenge is centered around a fictitious ”Food Store” app, highlighting the critical security flaw of SQL Injection (SQLi) within the app’s framework.

1.2 Objective

Your mission is to manipulate the signup function in the ”Food Store” Android application, allowing you to register as a Pro user, bypassing standard user restrictions.

1.3 Scope

Application	**Plataform**
com.mobilehackinglab.foodstore	`Android`

FILE INFORMATION
File Name `com.mobilehackinglab.foodstore.apk`
Size `9.54MB`
MD5 `ddeaf207bc7ffb4ca89326492eb9ac06`
SHA1 `be7f460da3dfae9e6ccbe5bf12cbcca631d22dcb`
SHA256 `9b3f9e03772d037d055933d26da94a4063c159f23971228e1a10e8fed8552ae5`
APP INFORMATION
App Name `Food Store`
Package Name `com.mobilehackinglab.foodstore`
Main Activity `com.mobilehackinglab.foodstore.LoginActivity`
Target SDK `34` **Min SDK** `27` **Max SDK**
Android Version Name `1.0` **Android Version Code** `1`

2 High-Level Summary

An external (black box) penetration test was executed to assess the security posture of Food Store from March 22th 2026 to March 23th 2026. 1 High severity issues were found. It is highly recommended to address the High vulnerability as soon as possible as the vulnerabilities are easily found through basic reconnaissance and exploitable without much effort.

2.1 Recommendations

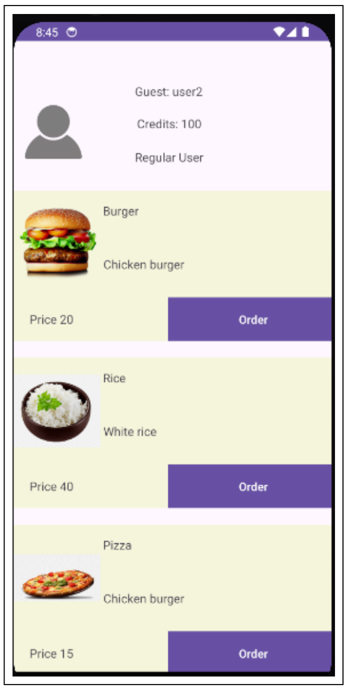
We recommends patching the vulnerabilities identified during the testing to ensure that an attacker cannot exploit these systems in the future. One thing to remember is that these systems require frequent patching and once patched, should remain on a regular patch program to protect additional vulnerabilities that are discovered at a later date.

2.2 Identified Vulnerabilities

The following table defines estimated levels of severity and corresponding CVSS score range used throughout the document to assess vulnerability and risk impact.

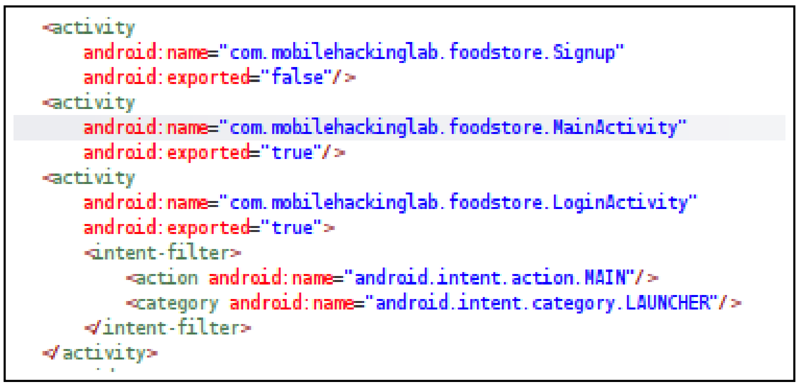
Target Name	**CVSS 3.1**	**Page**
com.mobilehackinglab.foodstore	7.8	5

3 Application Analysis



When you open the app, a **Login** form appears where you can enter a username and password. You can also register a new user via the **Signup** option. Once logged in, the system assigns the **"Regular User"** role and grants 100 **Credits** to spend on products.

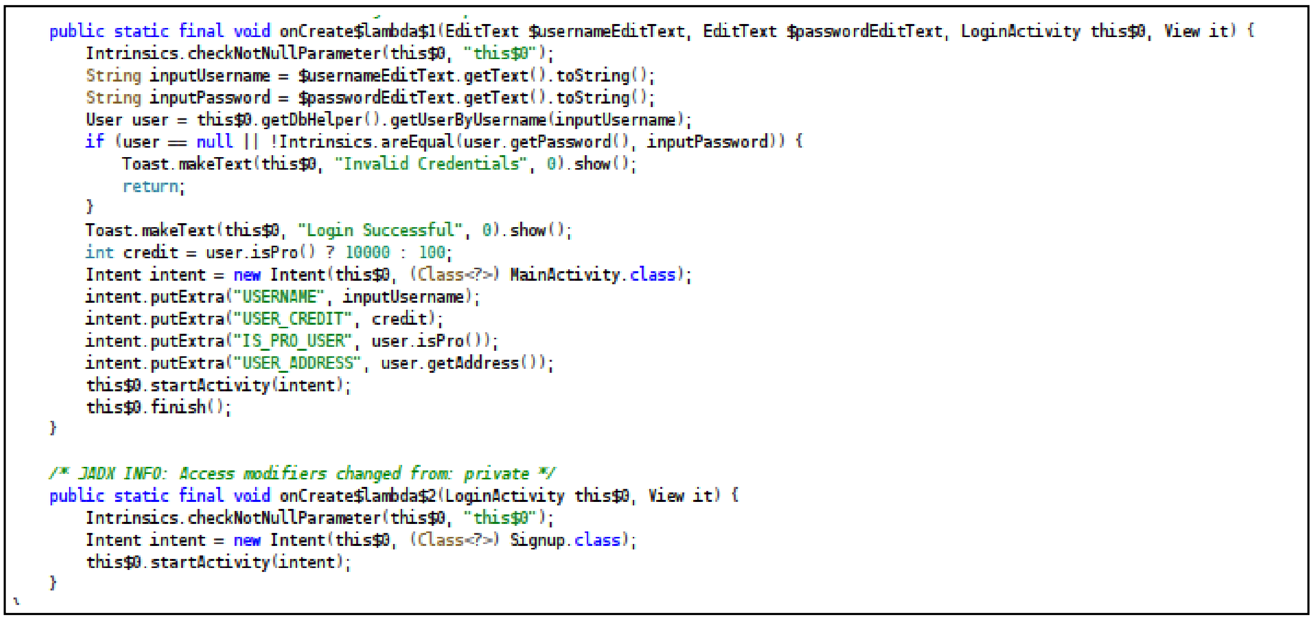
The app doesn't contain much custom code. In the *AndroidManifest*, we find the *Signup*, *MainActivity*, and *LoginActivity* activities. The latter acts as a launcher, so it runs "first" and directs the user there as soon as the app starts.



```
<activity
    android:name="com.mobilehackinglab.foodstore.Signup"
    android:exported="false"/>
<activity
    android:name="com.mobilehackinglab.foodstore.MainActivity"
    android:exported="true"/>
<activity
    android:name="com.mobilehackinglab.foodstore.LoginActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Image 1: AndroidManifest.xml

Looking at the *LoginActivity* class, we can see how it handles the login logic for registered users.



```
public static final void onCreate$lambda$1(EditText $usernameEditText, EditText $passwordEditText, LoginActivity this$0, View it) {
    Intrinsics.checkNotNullParameter(this$0, "this$0");
    String inputUsername = $usernameEditText.getText().toString();
    String inputPassword = $passwordEditText.getText().toString();
    User user = this$0.getDbHelper().getUserByUsername(inputUsername);
    if (user == null || !Intrinsics.areEqual(user.getPassword(), inputPassword)) {
        Toast.makeText(this$0, "Invalid Credentials", 0).show();
        return;
    }
    Toast.makeText(this$0, "Login Successful", 0).show();
    int credit = user.isPro() ? 10000 : 100;
    Intent intent = new Intent(this$0, (Class<?>) MainActivity.class);
    intent.putExtra("USERNAME", inputUsername);
    intent.putExtra("USER_CREDIT", credit);
    intent.putExtra("IS_PRO_USER", user.isPro());
    intent.putExtra("USER_ADDRESS", user.getAddress());
    this$0.startActivity(intent);
    this$0.finish();
}

/* JADX INFO: Access modifiers changed from: private */
public static final void onCreate$lambda$2(LoginActivity this$0, View it) {
    Intrinsics.checkNotNullParameter(this$0, "this$0");
    Intent intent = new Intent(this$0, (Class<?>) Signup.class);
    this$0.startActivity(intent);
}
```

Image 2: com.mobilehackinglab.foodstore;LoginActivity

Here, onCreate$lambda$1 retrieves and validates the credentials against the database. Then, if the verification is successful, it displays a **Login Successful** Toast and creates an Intent to *MainActivity* with the user's data.

This section contains several critical and poorly implemented issues. When querying the local database, the password comparison (Intrinsics.areEqual(user.getPassword(), inputPassword)) uses the plaintext values exactly as they are stored CWE-312.

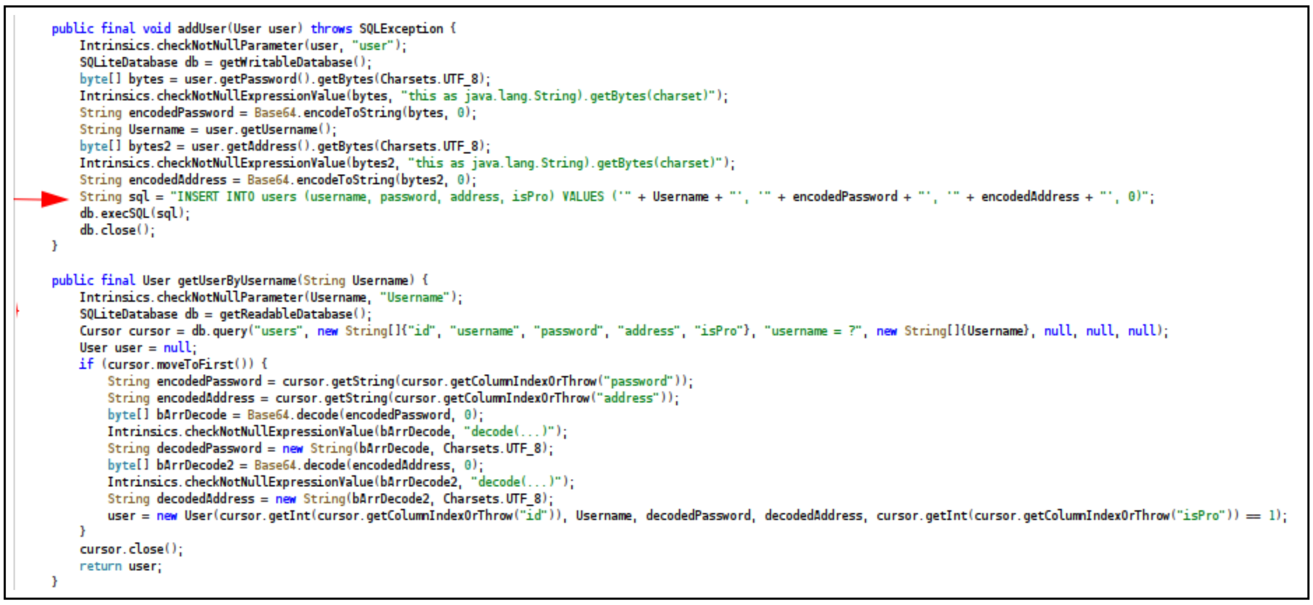
The role logic is also exposed. The credit allocation (user.isPro() ? 10000 : 100) depends solely on a local attribute, allowing it to be manipulated at runtime.

Furthermore, it constructs an Intent that carries and exposes sensitive user information: USER_ADDRESS, IS_PRO_USER, and USER_CREDIT. These values determine both the privilege status and the amount of credits assigned. The current design relies entirely on data sent from the client. Using adb, it is possible to inject arbitrary values into the Intent's extras to force access as a privileged user or directly manipulate the user's credits CWE-602.

```
$ adb shell am start -n com.mobilehackinglab.foodstore/.MainActivity --es USERNAME user --ei USER_CREDIT 10001 --ez IS_PRO_USER true --es USER_ADDRESS "home"
```

Code 1: intent injection

Another vulnerability becomes apparent when analyzing the *DBHelper* class.



```
public final void addUser(User user) throws SQLException {
    Intrinsics.checkNotNullParameter(user, "user");
    SQLiteDatabase db = getWritableDatabase();
    byte[] bytes = user.getPassword().getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes, "this as java.lang.String).getBytes(charset)");
    String encodedPassword = Base64.encodeToString(bytes, 0);
    String Username = user.getUsername();
    byte[] bytes2 = user.getAddress().getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes2, "this as java.lang.String).getBytes(charset)");
    String encodedAddress = Base64.encodeToString(bytes2, 0);
    String sql = "INSERT INTO users (username, password, address, isPro) VALUES ('" + Username + "', '" + encodedPassword + "', '" + encodedAddress + "', 0)";
    db.execSQL(sql);
    db.close();
}

public final User getUserByUsername(String Username) {
    Intrinsics.checkNotNullParameter(Username, "Username");
    SQLiteDatabase db = getReadableDatabase();
    Cursor cursor = db.query("users", new String[]{"id", "username", "password", "address", "isPro"}, "username = ?", new String[]{Username}, null, null, null);
    User user = null;
    if (cursor.moveToFirst()) {
        String encodedPassword = cursor.getString(cursor.getColumnIndexOrThrow("password"));
        String encodedAddress = cursor.getString(cursor.getColumnIndexOrThrow("address"));
        byte[] bArrDecode = Base64.decode(encodedPassword, 0);
        Intrinsics.checkNotNullExpressionValue(bArrDecode, "decode(...)");
        String decodedPassword = new String(bArrDecode, Charsets.UTF_8);
        byte[] bArrDecode2 = Base64.decode(encodedAddress, 0);
        Intrinsics.checkNotNullExpressionValue(bArrDecode2, "decode(...)");
        String decodedAddress = new String(bArrDecode2, Charsets.UTF_8);
        user = new User(cursor.getInt(cursor.getColumnIndexOrThrow("id")), Username, decodedPassword, decodedAddress, cursor.getInt(cursor.getColumnIndexOrThrow("isPro")) == 1);
    }
    cursor.close();
    return user;
}
```

Image 3: com.mobilehackinglab.foodstore;DBHelper

The *addUser* method contains two critical flaws that completely compromise data security.

First, the line String encodedPassword = Base64.encodeToString(bytes, 0); encodes the password using Base64, which serves no cryptographic purpose.

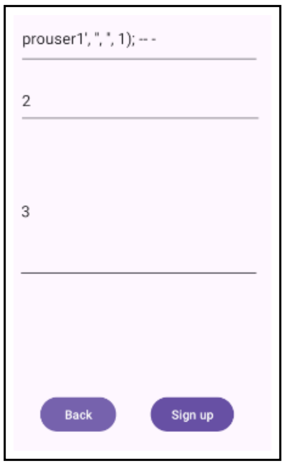
Second, the query construction introduces direct concatenation of user-supplied input from the **Username** into the SQL statement, allowing the original query to be manipulated via SQL injection CWE-89.

```
String sql = "INSERT INTO users (...) VALUES ('" + Username + "', '" + encodedPassword + "', '" + encodedAddress + "', 0)";
db.execSQL(sql);
```

Code 2: sql query

4 Initial Access

In this way, it is possible to modify the query before it reaches the database, forcing the creation of a privileged user. prouser1', '', '', 1); – -



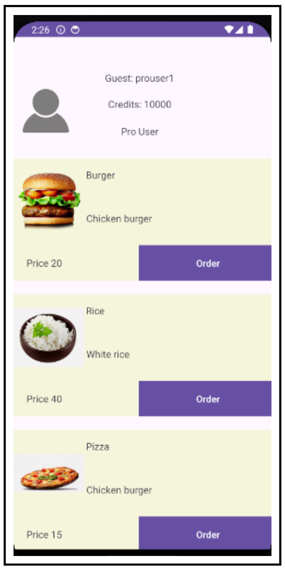
prouser1', '', '', 1); -- -

2

3

Image 4: prouser signup

The payload closes the original string and injects a new instruction where the isPro field takes the value 1. The rest of the query is commented out, so the legitimate data entered by the user is invalidated. And all you need to do is authenticate using the credentials defined in the payload, **username:** "prouser1"/**password:** " ".



2:26

Guest: prouser1

Credits: 10000

Pro User

Burger

Chicken burger

Price 20

Rice

White rice

Price 40

Pizza

Chicken burger

Price 15

Image 5: prouser login