Wondough Bank Coursework (CS263 Cyber Security)

Progress report

Joe Moore, u1917702

Department of Computer Science

University of Warwick

1 Introduction

Having penetration tested the Wondough bank website, a number of security flaws were subsequently found. The ten most critical vulnerabilities were then fixed as well as automated tests devised such that it were possible to entirely ensure the robustness of the new system. Below each of the vulnerabilitiesare otulined alongside the relevent fix and the tests devised to guarantee the success of my improved system.

2 Vulnerability: SQL Injection Attack

Description: The user data of the bank members is stored in a database, and as such upon clicking the login button the bank performs a simple SQL lookup query in order to determine if the username exists in the database. However this does not make use of *prepared statements* but rather constructs the SQL query using string concatenation. This allows the user to use the common technique of including an *OR* followed by a comment to finish the line to gain access. This is demonstrated below:

```
Username inputted: John

SQL: SELECT * FROM users WHERE username='John' LIMIT 1;

Username inputted: blank' OR 1=1;--

SQL: SELECT * FROM users WHERE username='blank' OR 1=1;--' LIMIT 1;
```

As can be seen in the query generated in the second instance, an entirely new SQL query has been created with the **- -** used to comment out the rest of the line. Since 1=1 will always be true, whether the username exists in the database is irrelevant since the statement will return the first user in the database regardless. This is a dangerous prospect as may allow a hacker access without relevant information and as such needs to be addressed.

Testing the Vulnerability: The Vulnerability gives access for all of the following usernames entered:

```
blank' OR 1=1;--
blank' OR 'x'='x';--
blank' OR '*;--
blank' OR TRUE;--
```

Where 1 can be replaced with any number as well as 'x' and 'blank' any string. This can also be used to get any value from any table and since tokens are not time dependent, access to these would allow the hacker to gain full access to an account. This therefore is a serious security flaw.

Mitigation: The issue caused by this flaw is purely as a result of using string concatenation to generate the query. The use of sql prepared statements can be implemented in order to deny such an attack. A prepared statement works by compiling the statement template before later binding values for the parameters of the statement template. As such the statement knows when the query ends and the use of ' to signify the end of the username or - - to turn the remainder of the statement to a comment both no longer work. As such the attack no longer works. A prepared statement works as follows:

Rather than creating and executing the query as follows:

```
PreparedStatement stmt = null;
String query = "SELECT * FROM users WHERE username='"
      + username + "' LIMIT 1;";
stmt = this.connection.Statement();
ResultSet rs = stmt.executeQuery(query);
```

It is done as follows:

```
PreparedStatement stmt = null;
String query = "SELECT * FROM users WHERE username=? LIMIT 1;";
```

```
    stmt = this.connection.prepareStatement(query);
    stmt.setString(1, username);
    ResultSet rs = stmt.executeQuery();
```

This is the change I made to the `DbConnection.java` file in order to mitigate this attack. Below is the automated test I designed in order to test the changes appropriately extinguished the vulnerability. Therefore I designed an automated test to run automatically on the startup of the server to check this vulnerability was not a threat. It can be seen at `test1.java`. It works by calling the get user function with all four examples of SQL injection commands given previously. If none of them return a user then it passes else it fails. Below is the important check.

```
private String runSQLInjection(){
    String[] testSQLInjections = {"blank' OR 1=1;--",
    "blank' OR 'x'='x';--", "blank' OR '*;--", "blank' OR TRUE;--"};
    DbConnection db = Program.getInstance().getDbConnection();
    try {
        for (int i = 0; i <= 3; i++) {
        if (db.getUser(testSQLInjections[i]) instanceof WondoughUser)
            return "FAILED";
        }
    }
    catch (SQLException e) {
        return "FAILED" + e.toString();
    }
    return "PASSED";
}
```

This test passed upon starting the server after implementing my fix and as such I consider this vulnerability solved.

3 Vulnerability: Sending more money than balance

Description: Upon further penetration testing it was discovered that it was possible to send money from the account logged in on to another of a larger amount than the account's balance. This may, upon first glance appear to be a bug rather than necessarily a vulnerability. However, from the previous section we have displayed proof of concept that it is possible for a hacker to gain access to someone's account without the relevant information. As such he could feasibly send any amount (even ludicrous sums like £1 billion) from this bank account at Wondough bank to his bank account with another bank. His bank likely does not have this issue and he has just granted himself huge amounts of money illegally.

Testing the Vulnerability: In order to test this vulnerability I wanted to test to see different things the flaw could enable. It is possible to send transactions to yourself however I do not see how useful this is as a vulnerability to an attacker. However I then decided to add a new user to the database in order to carry out further tests.

Mitigation: The process of creating a transaction occurs in the `DbConnection` file in the `createTransaction()` procedure. This feature already includes a check to ensure the amount in the transaction is not negative, as follows:

```
if(amount < 0) {
        return false;
}
```

And as such in a similar vein an additonal check was then added which makes use of the `getTransactions(user)` and `getAccountBalance()` functions to retrieve the user's balance and compare it to the amount. As follows:

```
if(amount > getTransactions(user).getAccountBalance()){
    return false;
}
```

This was then tested manually with inputs larger than the balance, which were all not entered as transactions, followed by the balance itself and values lower than it, all of which the fix permitted. However I then set about automating this testing. The automatic testing seen in `test2.java` creates a new user *testUser*, it then tries to remove his account balance + 1, which should fail if the fix is correct. It then tries the same with his exact account balance, if this works then the fix is perfect. Since both these conditions are always fine, this vulnerability has been correctly fixed.

4 Vulnerability: JavaScript Injection Attack

Description: A second type of injection attack works on this website. It was possible to enter a JavaScript script into the transaction box. This would be stored by the database as the description of the transaction. Consequently when the user clicked *List transactions* each transaction would be listed, and as such each description interpreted in html. This means a description enclosed by `<script></script>` tags will be interpreted as a script and as such executed as the website comes to list the transactions. This allows a hacker to gain valuable information by printing out session data in a pop up window. This security flaw is exasperated by the fact that all transactions are viewable to all users. Thus any script injected will affect all users. This is known as a persistent (or stored) vulnerability. Persistent vulnerabilities are more significant since the malicious script is rendered automatically, therefore there is no requirement to individually target victims or to take them to a third party website.

Testing the Vulnerability: To test this vulnerability a series of JavaScript queries were entered as transactions. One of the most dangerous was the following:

```
<script>alert(document.cookie)</script>
```

Once injected upon the selection of *List transactions* this will open a popup window which will display the users token.
As can be seen from this figure, this injection allows a user to display the token.

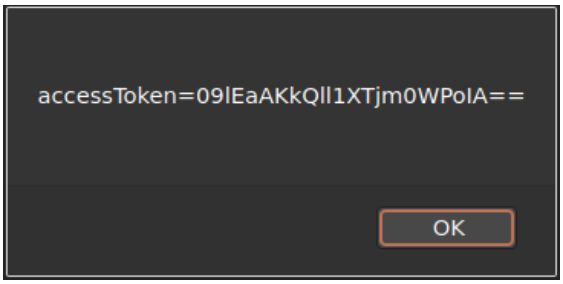

Figure 1: The popup window displayed when a user clicks *List transactions*

This again being particularly dangerous since the tokens are not time dependent on this website. Consequently this can be said to be a major security flaw since it has the possibility of allowing a hacker indefinite access to the banking website.

Mitigation: This program needs to be addressed with the use of *html encoding*. This interprets certain key symbols such as < and > such that the website knows they are simply to be printed and not interpreted as that symbol. As such they need to be stored differently, for example in our example < is stored as `<` and > is stored as `>` Then when the website encounters these new encodings it knows to print < and > rather than when it encounters < and > which means the start of something such as a script. In order to ensure the most rigorous security I imported a library to perform this encoding:

```
import org.apache.commons.lang.StringEscapeUtils;
```

From there the `escapeHtml()` function was used to generate a new html encoded description of the transaction in the `createTransaction()` function, which was then entered into the database in place of the description.

```
String encoded = StringEscapeUtils.escapeHtml(description);
```

This works as required and if you look at the database you'll see the encoded raw text:
Whilst the website correctly interprets this, printing < > rather than `<` `>`
This difference between the string stored by the database compared to the

16	16	0	-1.0	<script>alert(document.cookie)</script>
17	17	1	1.0	<script>alert(document.cookie)</script>

Figure 2: The transaction description as stored by the database

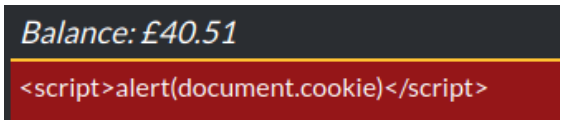

Figure 3: The transaction description as interpreted by the browser

string entered by the user is what has been used to develop an automated test. The test enters a new transaction containing the tags `<script>` and `</script>`. Then it checks to ensure that the value stored by the database does not contain either < or >. Which by ensuring, it can confirm that the threat of JavaScript injection is no longer a concern. See `test3.java` for the full test.

5 Progress

Summarise the progress you have made so far. You can cross-reference other sections (Section 4).

6 Project management

Include a timetable (in 2 week chunks) for the remainder of the academic year, up until the submission deadline.