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.

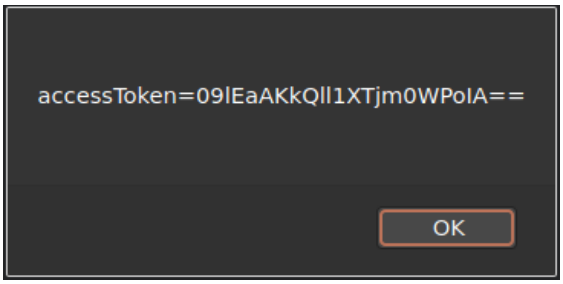

Figure 1: The popup window displayed when a user clicks *List transactions*

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

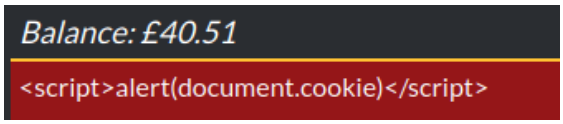

Figure 3: The transaction description as interpreted by the browser

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 Vulnerability: Multiple users with duplicate usernames

Description: The fourth vulnerability I discovered was the ability for multiple users to have the same username. This is a fundamental issue. The username of an account needs to be its unique identifier that allows a program to identify an individual specific user. If we allow multiple users to share a username then what will happen is that upon one of those users signing the program will return the id of the first user in the database with that username. This means that either if they have by chance chosen a bad and matching password that access to another users finances could be granted to another. Otherwise it will be impossible for a user to sign in and to access their finances, which is without doubt a serious issue.

Testing the Vulnerability: To test this vulnerability I first uncommented the section of code that creates a new user. This then created the user *test4User* twice, with passwords *password* and *password2* respectively. I then signed in to

the website, as *test4User@wondoughbank.com*, which I was only able to do using the password *password*, since upon getting the user the program was returning the first instance in the database. Therefore in a real world scenario the second user with password *password2* would be entirely unable to access the website indefinitely.

Mitigation: Solving this vulnerability occurred in the `createUser()` function within `DbConnection`. A check needed to be applied here to ensure that the username being used to create the new user did not already exist in the database. To do this the `createUser()` was modified such that it was no longer a `void` but rather returned a boolean value. True would be if the user had been created and otherwise false. Therefore at the beginning of create user we call `getUser` on the username and if it doesn't return null then that username already exists in the database and as such the function should return false and go no further.

```
if(getUser(user.getUsername()) != null){
    System.out.println("User already exists");
    return false;
}
```

Although I manually tested this in order to ensure its rigorousness an automated test was devised. This test performed 2 actions that both needed to be completed in order for the test to pass. Firstly a new user needed to be created:

```
WondoughUser testuser1 = new WondoughUser(1,
    "test4User@wondoughbank.com");
testuser1.setSalt(securityConfiguration.generateSalt());
testuser1.setHashedPassword(securityConfiguration.pbkdf2("password",
    testuser1.getSalt()));
testuser1.setIterations(securityConfiguration.getIterations());
testuser1.setKeySize(securityConfiguration.getKeySize());

if((connection.createUser(testuser1)) == false) return "FAILED";
```

As can be seen above if the user was not created then the test would fail. After this the test then attempted to create a second user with the same name directly after. The code for this was identical since we wanted the users to be the same. However, this time if the user was created the test failed:

```
if((connection.createUser(testuser2)) == true) return "FAILED";
```

Then if all that did not happen the test passed. The reason for having the condition of making the first user be created in order to pass was to ensure that the database connection was working since without this condition the test would pass if it just couldn't connect. Finally I performed some clearing up of the database such that the test data was removed since it did not represent a real system user and it also needed to be removed so the test would work again next time:

```
String query ="DELETE FROM users"
              +"WHERE username='test4User@wondoughbank.com'";
Statement stmt = connectionToDelete.createStatement();
stmt.executeUpdate(query);
return "PASSED";
```

This test passed every time and as such this vulnerability can be considered thoroughly mitigated.

6 Vulnerability: PBKDF2 Iterations

Description: The password hashing in the wondough website uses PBKDF2. This pseudo random hashing algorithm has a field called 'iterations', which is the number of times the PRF will be applied to the password when deriving the key. The number iterations was deemed to need to be 1000 as early as 2000, and A Kerberos standard in 2005 recommended 4096 iterations. Therefore the current iteration set as 1 is an entirely unacceptable amount[1].

Password Complexity	Entropy estimate	1000 iterations	10000 iterations
Comprehensive8	33	4 hours 46 min	47 hours
8 random lowercase	37	12 hours	5 days
8 random letters	45	123 days	3.5 years
8 random characters	52	325 years	3250 years

The use of a salt has allowed the ability to use precomputed hashes (rainbow tables) to be reduced. It also prevents multiple passwords being tested together.

Testing the Vulnerability: A hacker could use an SQL injection attack (as previously demonstrated is possible on the wondough system) to retrieve hashed passwords of users. This means that given a low iteration he could much more easily and in a shorter time brute force the password. A brute force can take as little as 4 hours for 1000 iterations as seen in the above table and consequently the average time to brute force 1 iteration is dangerously small. As such it is clear that the severity of this vulnerability is alarming. Since the likelihood of a brute force working is technically 100%, although often unachievable time periods, by reducing these time periods to mere hours it would allow a hacker to potentially gain access to huge amounts of account passwords and by extension their money. As such the threat is severe.

Mitigation: In line with current information I have decided in order to ensure that password are not likely to be brute forced in an reasonable time, to increase the iteration size to 10,000[2]. This should mean given even a basic 8 lowercase character password it would take (roughly) 5 days to brute force. This is a reasonable length of time given that users should have to have a wide range of characters in their password. I updated the `security.json` file which the program uses to determine its security configuration.

```
{
"iterations": "10000",
"keySize": "16"
}
```

Then to design automatic testing I created a new user the same way as I had done for vulnerability 4. For this I checked his iteration size was exactly 10,000 and if it was that meant the update to `security.json` had correctly enhanced this security and meant that the chance of this vulnerability working being 100% was no longer the case. Therefore a severe improvement.

7 Vulnerability: All tokens set to user 0

Description:

Testing the Vulnerability:

Mitigation:

References

[1] Parameter choice for pbkdf2. `https://cryptosense.com/blog/parameter-choice-for-pbkdf2/`. Accessed: 2020-12-8.

[2] A. F. Iuorio and A. Visconti. Understanding optimizations and measuring performances of pbkdf2. In I. Woungang and S. K. Dhurandher, editors, *2nd International Conference on Wireless Intelligent and Distributed Environment for Communication*, pages 101–114, Cham, 2019. Springer International Publishing.