# Wondough Bank Coursework (CS263 Cyber Security)

Progress report

**Joe Moore, u1917702**

Department of Computer Science

University of Warwick

# Contents



# 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.

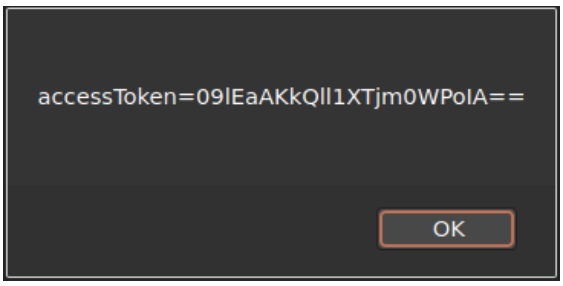


Figure 1: The popup window displayed when a user clicks *List transactions*

As can be seen from this figure, this injection allows a user to display the token.

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:

| | | | | |
|---|---|---|---|---|
| 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

Whilst the website correctly interprets this, printing < > rather than `<` `>` This difference between the string stored by the database compared to the 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.

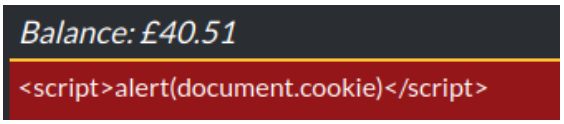


Figure 3: The transaction description as interpreted by the browser

# 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 & keysize

**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[4].

| 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. The use of a keysize as small as 16 is also very dangerous and it is reccomended thata key size of 124 is used [7]

**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[6]. 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": "124"}
```

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 keysize exactly 124 and if it was it 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: Login always as user 0

**Description:** Upon logging into the website as the second user I had created, *hacker@wondoughbank.com*, it became clear that the transactions displayed were

in fact not belonging to the user *hacker@wondoughbank.com*. It was then immediately clear that they belonged to the original user *intern@wondoughbank.com*. This was confirmed when it was observed that the tokens created were all associated with the user whose id was 0.

| | user | requestToken | accessToken |
|---|---|---|---|
| | Fil... | Filter | Filter |
| 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 1 |
| 3 | 0 | 2 | 2 |
| 4 | 0 | 3 | 3 |
| 5 | 0 | 4 | 4 |
| 6 | 0 | 5 | 5 |
| 7 | 0 | 6 | 6 |
| 8 | 0 | 7 | 7 |

Figure 4: The table of tokens showing each was associated with the user 0

This would allow all users access to user 0's account. This is a critically important vulnerability since it could be accidentally exploited by well meaning users as well as hackers. It is also a vulnerability that requires no technical knowledge to exploit and this low skill ceiling makes it of particular threat. If combined with the fact that the bank used to allow the transfer of money from one account to another, attackers could transfer unlimited money easily from the account of the unfortunate user whose id happens to be 0.

**Testing the Vulnerability:** I tested this vulnerability as a hacker would by simply logging in as user 1 (which would then naturally log me in as user 0) and then transferring money to user 1. I watched the money leave my account, which proves the issue since transferring money to myself should cause no change in value to my balance. This would be how a hacker could exploit such a vulnerability but also how a regular user simply not paying attention and thinking they had logged into their account could also cause trouble. This

could make the vulnerability a further threat since an attacker could simply use this as an excuse if they were ever discovered to have used the attack.

**Mitigation:** By using print lines in the `getUSer()` function I discovered the issue. In the below code ids 1 and 2 in lines 1 and 2 returned 1, the correct id. This meant that the issue was not in the fetching of the id from the database. Whilst id 3 printed 0 meaning the issue occured in the creation of the new user from the values from the database.

```
System.out.println("\nID1"+rs.getInt(1));
System.out.println("\nID2"+rs.getInt("id"));
WondoughUser user = new WondoughUser(rs.getInt("id"),
    rs.getString("username"));
user.setHashedPassword(rs.getString("password"));
user.setSalt(rs.getString("salt"));
user.setIterations(rs.getInt("iterations"));
user.setKeySize(rs.getInt("keySize"));
System.out.println("\nID3"+user.getID());
return user;
```

It then transpired that passing the id into the constructor of `WondoughUser` was pointless since the constructor never set the value. Therefore I added the following line to fix the issue. This is the user the tokens are created from and much more causing the root of the issue.

```
public WondoughUser(int id, String username) {
    this.username = username;
    this.id = id;//added line
}
```

From here I devised an automated test to call the `getUser` command to test that the user returned shared the username and id of the functions parameters. The test successfully passed each time.

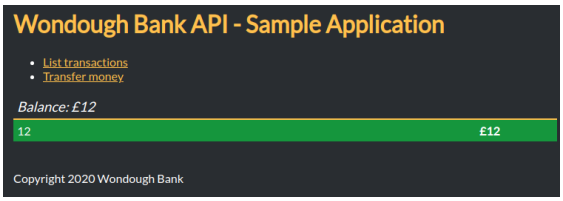


Figure 5: After fixing the constructor the correct balance and transactions were shown

# 8 Vulnerability: Salts aren't random

**Description:** The salt in this website is made dependent on the day, that is for a given day ever instance of `generateSalt()` will return the same value. The idea of a is to combat the use of rainbow tables. However by having a standard static salt for an entire day it is possible to extract and generate a new rainbow table. Since a user's salt is never change it is possible to extract the salt and a rainbow table will work for all users who created their accounts on the same day, and as such makes it easier for an attacker to target multiple users at once [3].

**Testing the Vulnerability:** If a hacker knows the salt is hard-coded based on a particular day, lookup tables and rainbow tables can be built for that salt, to make it easier to crack hashes. An attacker could easily perform an SQL injection with an additional `SELECT salt FROM users` to gain access to the salt for a given user. From there all they need to do is apply the salt to each password guess before they hash it [2]. An attacker could make educated guesses without having access to the salts about the days on which passwords were made. For example if an account is sandwiched by two others with the same hash then those outer two must have the same salt and as such the same salt. As such the account in the middle has been compromised by the fact that the three accounts all share a salt since it is obvious that the middle account must share the salt of the account before and after it.

**Mitigation:** In order for the salts to be secure a salt **must be random and unique per user**. If the salt is short it is possible for an attacker to build a

lookup table for each possible salt. If the salt is only three ASCII characters, there are only 95x95x95 or 857,375 possible salts [3]. That may seem large however if only 1 MB of the most common passwords are stored the entire rainbow table accounting for all possible salts will be under a terabyte. It is true that a good practice is that a salt should be at least 16 bytes [5]. I have gone to 20 bytes to ensure maximum security. I imported the `secureRandom` package to be used in the new method of salt generation, which works as follows:

```
public String generateSalt() {
    SecureRandom randomy = new SecureRandom();
    //generate a random number
    byte bytes[] = new byte[20];
    //assign space for the salt
    randomy.nextBytes(bytes);
    Base64.Encoder encoder = Base64.getEncoder();
    //encode the salt and return the string
    return encoder.encodeToString(bytes);
}
```

The automated test for this vulnerability would generate 100,000 salts and check all of them were different The check works by placing all the salts in a hash set and checking the size of the hash set is equal to the size of the array. It passes every time and as such the vulnerability is extinguished.

# 9 Vulnerability: Tokens are 1-10 and don't expire

**Description:** The access tokens on the website have 2 key issues. Firstly there are only ten eleven tokens availabe (0 through to 10). Secondly the tokens have no expiry date and as such if a hacker were to gain access to a token through for example Javascript injection as seen in vulnerability 3, he could use it to keep permanent access to a user's account even if they changed their password, regardless of how complex the password hashing was made. This of course is a serious security flaw and needs to be addressed. By only having 11 tokens this flaw is made worse since an attacker would only need to try 11

tokens in order to get access.

**Testing the Vulnerability:** To do this I logged in as *hacker@wondoughbank.com* and displayed the transactions. After my fix to the login issue in the previous vulnerability the result was identical to figure 5. I then went into the developer options on my browser to display the cookies:

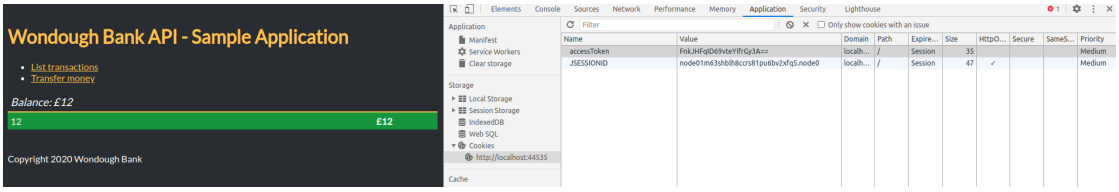


Figure 6: The token visible in the cookies of my session

From here I was able to change the token to one associated with user 0 (from *jxTkX87qFnpaNt7dS+olQw==* to *jxTkX87qFnpaNt7dS+olQw==*)

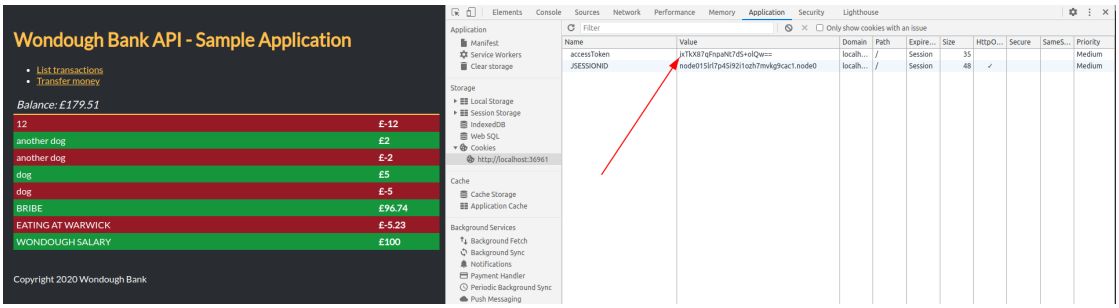


Figure 7: By editing the token I gain access to the intern's account

**Mitigation:** The first issue of the 0-10 occurs in `dbConnection` where the following code is used to create the tokens id:

```
app.setRequestToken(Integer.toString(this.largestRequestToken(), 10));
app.setAccessToken(Integer.toString(this.largestAccessToken(), 10));
```

I have therefore chosen to use the salt generation as the token since this is unique each time and much more complex. It also means that the issue of the access and request token being the same solved.

```
String salt;
salt = Program.getInstance().getSecurityConfiguration().generateSalt();
app.setRequestToken(salt);
salt = Program.getInstance().getSecurityConfiguration().generateSalt();
app.setAccessToken(salt);
```

Since the salt is random each time this allows us to solve the issue of recurring tokens. To solve the issue of the fact the tokens never expire an additional column was added to the database called `expiryDate`, which was a `TimeStamp` and represented the point in time when that key would no longer be valid. To set this I edited the `createApp()` function in `dbConnection`. A fourth value was added to the prepared statement, which is a TimeStamp. From there the current time is stored and 30 minutes added to it. This is to be the value of the `expiryDate` of the access token. Whilst the request token has a date of 1 hour from now. The choice of times was to ensure the security of the website was as rigorous as possible. This choice of time meant there's time to generate new access and request tokens after the access token is expired. The request tokens will expire a little while later and can get purged in a timely manner to avoid accumulation [1]. It also means if a hacker were to gain access to a token it would expire shortly making access limited. However inorder to give the keys different expiry times they had to be stored seperatly, as such the function was altered to create two entries into the database, one with request token as null and one with access token as null.
The expiry times were set as follows:

```
Timestamp now;
now = new Timestamp(System.currentTimeMillis());
now.setTime(now.getTime() + TimeUnit.MINUTES.toMillis(30));
stmt.setTimestamp(4, now);

now = new Timestamp(System.currentTimeMillis());
```

| | user | requestToken | accessToken | expiryDate |
|---|---|---|---|---|
| | Fil... | Filter | Filter | Filter |
| 1 | 1 | *NULL* | qhhDvew4yh758vi4RBrSGT... | 1607603983758 |
| 2 | 1 | PyWJ4pu+PyB3z9uuIst8nI/... | *NULL* | 1607605783768 |

Figure 8: The adapted way of storing tokens

```
now.setTime(now.getTime() + TimeUnit.HOURS.toMillis(1));
stmt.setTimestamp(4, now);
```

Finally a remove tokens function was created that would be called regularly to remove outdated tokens. It checked if the expiry date of the token had been surpassed by comparing it to the current time:

```
long ut1 = System.currentTimeMillis();
String query3 = "DELETE FROM authorised_apps "
                +"WHERE expiryDate < "+ ut1 + ";";
Statement stmt2 = this.connection.createStatement();
int rs = stmt2.executeUpdate(query3);
```

From this an automatic test was devised to ensure all these new criteria were met. It would create temporary tokens, ensure the difference in expiration date, value and then check they would be removed in an hours time. The randomness and greater complecity of the tokens alongside their limited time means this vulnerability has been mitigated as far as possible.

# References

[1] Antipattern: Set a long expiration time for oauth tokens. https://docs.apigee.com/api-platform/antipatterns/oauth-long-expiration. Accessed: 2020-12-9.

[2] The bug chamer: Passwords matter. http://bugcharmer.blogspot.com/2012/06/passwords-matter.html. Accessed: 2020-12-10.

[3] Crackstation: Salted password hashing - doing it right. `https://crackstation.net/hashing-security.htm#salt`. Accessed: 2020-12-10.

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

[5] P. Gauravaram. Security analysis of salt||password hashes. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 25–30, 2012.

[6] 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.

[7] V. A. K. R. G. "Levent Ertaul, Manpreet Kaur. "implementationand performance analysisof pbkdf2, bcrypt, scrypt algorithms".