SYSC 4810 – Assignment 3

Emma Maddock, 100996472

Problem 1 – Access Control Mechanism

Since SecVault Inc. presents a system whose various levels/types of accesses rely heavily on the specific position of each employee, it makes most sense to use a **Role-Based Access Control (RBAC)** model. This allows us to grant the appropriate accesses to each user or employee that allows them to access banking services or do their job while maintaining proper security by not giving them more than they need (the design principle of least privilege). This is especially important in a sector where confidentiality and integrity are of utmost importance. Enforcing least privilege allows for a client's financial details to remain private, and in general allows only people in more senior and trusted roles access to mechanisms that affect the integrity of money flows.

To implement this RBAC model, an **Access Control Matrix (ACM)** will be used, which maps each user to a role, then each role is assigned a set of accesses with respect to each object. This choice again enforces least privilege and allows us to assign privileges to users based on their roles within SecVault.

Roles

Users

	Client	Premium Client	Teller	Tech Support	Financial Advisor	Financial Planner	Investment Analyst	Compliance Officer
Mischa Lowery	✓							
Nick Perez		✓						
Nelson Wilkins			✓					
Seth Macleod				✓				
Willow Garza					✓			
Stacy Kent						✓		
Kodi Matthews							✓	
Caroline Lopez								✓

Figure 1: An example of the User-Role Matrix in the proposed SecVault Inc. RBAC scheme

	Client Information	*Client Account*	*Client Account Balance*	*Client Investment Portfolio*	*Financial Advisor Details*	*Financial Planner Details*	*Investment Analyst Details*	*Private Consumer Instruments*	*Money Market Instruments*	*Derivatives Trading*	*Interest Instruments*
Client		*Password access*	*view*	*view*	*view*						
Premium Client		*Password access*	*view*	*view, modify*	*view*	*view*	*view*				
Teller			*view (restricted)*	*view (restricted)*							
Tech Support	*view*	*Request access*									
Financial Advisor			*view*	*view, modify*				*view*			
Financial Planner			*view*	*view, modify*				*view*	*view*		
Investment Analyst			*view*	*view, modify*				*view*	*view*	*view*	*view*
Compliance Officer			*view*	*view, validate changes*							

Figure 2: An example of the Role-Object Access Matrix in the proposed SecVault Inc. RBAC scheme

```
//creating the role-object matrix as depicted in the report
permissions = new ArrayList<String[]>();
String[] clientPermissions;
String[] premiumClientPermissions;
String[] tellerPermissions;
String[] techSupportPermissions;
String[] fAdvisorPermissions;
String[] fPlannerPermissions;
String[] iAnalystPermissions;
String[] cOfficerPermissions;

clientPermissions = new String[]{"","","VIEW","VIEW","VIEW","","","","","",""};
premiumClientPermissions = new String[]{"","","VIEW,MODIFY","VIEW","VIEW","VIEW","VIEW","","","","",""};
tellerPermissions = new String[]{"","","VIEW","VIEW","","","","","","","",""};
techSupportPermissions = new String[]{"VIEW","REQUEST ACCESS","","","","","","","","","",""};
fAdvisorPermissions = new String[]{"","","VIEW","VIEW,MODIFY","","","","","VIEW","","","",""};
fPlannerPermissions = new String[]{"","","VIEW","VIEW,MODIFY","","","","","VIEW","VIEW","","",""};
iAnalystPermissions = new String[]{"","","VIEW","VIEW,MODIFY","","","","","VIEW","VIEW","VIEW","VIEW","VIEW"};
cOfficerPermissions = new String[]{"","","VIEW","VIEW,VALIDATE CHANGES","","","","","","","","",""};

permissions.add(clientPermissions);
permissions.add(premiumClientPermissions);
permissions.add(tellerPermissions);
permissions.add(techSupportPermissions);
permissions.add(fAdvisorPermissions);
permissions.add(fPlannerPermissions);
permissions.add(iAnalystPermissions);
permissions.add(cOfficerPermissions);
```

Figure 3: RBAC + ACM Implementation in Prototype

The RBAC scheme and ACM in the prototype was implemented by creating essentially a 2D matrix, exactly as depicted in Figure 2 above. The ArrayList of permissions included one entry for each role. Each entry was an array of Strings stating the permission for each Object, again exactly as above in Figure 2.

This was tested by creating a new user with a certain role, logging in, and then seeing if the permissions granted in the menu were the same as in the ACM for that role.

Problem 2 – Password File

The hash function that will employed for the password file will be **256-bit SHA3**, as it's the most recent iteration of the **Secure Hash Algorithm (SHA),** and has a different structure than SHA-1 and -2, which were first implemented in 1995 and 2002, respectively. 256 bits is also the longer option, which provides a degree of extra security over its 224-bit counterpart. MD5 was not considered as it was stated to be vulnerable, and SecVault had specifically requested that no algorithm have known vulnerabilities.

The salt will be 32 bytes and randomly generated by the Java SecureRandom class, and **prepended** to the user's password, to then be hashed by the SHA-3 algorithm and stored. The 32-byte option for the salt is a concession for performance. Since the SHA-3 algorithm already produces a large hash value that has to be stored, the 32-byte salt over the 64-byte salt will be a small relief for storage and therefore speed. Although less secure, the hash function is already the most secure option and the security will not suffer too much.

The record structure for the file will be as follows:

userID:salt:saltedHash:First Last:roleID

- **userID** – the user's credentials for signing in to the system
- **salt** – the 32-byte salt value
- **saltedHash** – the 256-bit hashed value of the salt prepended to the password
- **First Last** – the user's first and last name
- **roleID** – an ID number that determines the user's role in the system and will be used to grant them certain permissions upon login
    - 01 – client
    - 02 – premium client
    - 03 – teller
    - 04 – tech support
    - 05 – financial advisor
    - 06 – financial planner
    - 07 – investment analyst
    - 08 – compliance officer

```
//salt and hash password
byte[] salt = generateSalt();
String salted = salt + password;

String hashed = hashPassword(salted);

//concatenate into single record
String record = userID + ":" + salt + ":" + hashed + ":" + firstName + " " + lastName + ":" + roleID + "\n";

//store into password file
try {
    FileWriter writer = new FileWriter("pswd.txt",true);
    BufferedWriter bWriter = new BufferedWriter(writer);
    bWriter.write(record);
    bWriter.close();
} catch (IOException e) {
    System.out.println("There was an error writing to the password file.");
}

System.out.println("User record successfully created.");
```

During the **user enrollment** phase of the system, the salt is generated, prepended to the user's selected password, then hashed using the SHA3-256 algorithm. Then, all of the fields collected during enrollment, as well as the salt and the hashed salted password, are written on a new line in the *pswd.txt* password file.

Figure 4: Writing to the Password File in the Prototype

During the **login** phase of the system, the first read of the password file is when the system is looking for a match for the entered username.

Once a username match is found, the system then takes the salt in that record, prepends it to the entered password, hashes it using the same algorithm (SHA3-256), then compares that against the hash value in the record.

The password file was tested by creating a brand new user, which should populate the file, and then immediately logging in as that user, with a few wrong password attempts to make sure it's properly comparing the hashes.

```
//reading the password file and retrieving the salt and the hash based off the username
try {
    reader = new BufferedReader(new FileReader("pswd.txt"));
    record = reader.readLine();

    while ((searchingUsername) && (record != null)) {
        //dissect first part of record to see if user ID is a match
        //copy into a string until first colon reached
        String userID = "";
        for (int i=0;i<record.length();i++) {
            Character curr = record.charAt(i);
            userID += curr;
            if (curr == ':') {
                break;
            }
        }
        userID = userID.substring(0,  userID.length() - 1);

        if(userID.equals(usernameInput)) {
            searchingUsername = false;
        } else {
            searchingUsername = true;
            record = reader.readLine();
        }

    }

    reader.close();

} catch (IOException e) {
    System.out.println("There was an error reading the password file.");
}
```

Figure 5: Checking the Password File for a Username Match

```
//get salt and hash from record
//RETRIEVE SALT
int colonCount = 0;
String salt = "";
for (int i=0;i<record.length();i++) {
    Character curr = record.charAt(i);
    if (colonCount == 1) {
        salt += curr;
    }
    if (curr == ':') {
        ++colonCount;
    }
}
salt = salt.substring(0,  salt.length() - 1);

//RETRIEVE HASH
int colonCountTwo = 0;
String recordHash = "";
for (int i=0;i<record.length();i++) {
    Character curr = record.charAt(i);
    if (colonCountTwo == 2) {
        recordHash += curr;
    }
    if (curr == ':') {
        ++colonCountTwo;
    }
}
recordHash = recordHash.substring(0,  recordHash.length() - 1);

//RETRIEVE ROLEID
char roleIDchar = record.charAt(record.length() - 1);
int roleID = Character.getNumericValue(roleIDchar);
```

Figure 6: Retrieving Data from the Password File in the Prototype

pswd.txt

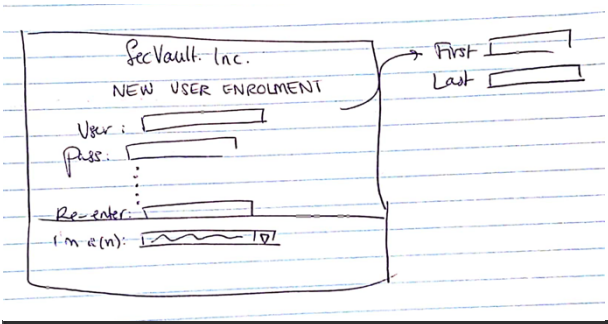
```
emmad2:[B@37a71e93:413fb470a0a5b75a1ee526c69afc7e76e3c4223801cd4d39a98c48f13159085a:emma maddock:5
soy4:[B@37a71e93:9f4c6e4ce885fa7a638a0ab1ca921c02647468cba72155601e717569cf2d21c2:soy sauce:7
```

Figure 7: The Password File pswd.txt Used in the Prototype

Problem 3 – Enrolling Users

The enrollment user interface involves asking the user to enter all of the relevant information the system needs to uphold its security/access control system.

In this case, the design included asking the user to enter the following fields in order to populate a record for the password file:



- First Name
- Last Name
- Username
- Password
- Re-enter Password
- Digit-based Role Selection

The password checker was designed as a number of separate checks, that gives the all clear only if all six of the checks came back positive. The checks as specified by the requirements were:

- 8-12 characters in length
- One upper-case character
- One lower-case character
- One digit
- One special character from the set: {!,@,#,$,%,?,*}
- Is not a match to a common password in the file *commonPswds.txt*

```
SecVault Investments, Inc.
NEW USER ENROLLMENT

What is your first name?
Emma
What is your last name?
Maddock
What username would you like to use?
emmad4
Select a password that meets the following criteria:
        -between 8-12 characters in length
        -include one upper-case character
        -include one lower-case character
        -include one digit
        -include one of the following symbols: !, @, #, $, %, ?, *
oops
Password does not match criteria. Please try again.
```

```
Select a password that meets the following criteria:
        -between 8-12 characters in length
        -include one upper-case character
        -include one lower-case character
        -include one digit
        -include one of the following symbols: !, @, #, $, %, ?, *
Coolio22!
Password meets criteria. Please re-enter password:
Coolio22!
```

```
SecVault Investments, Inc.

Are you a new user? (Y/N):
y
```

The implementation of the user enrollment phase involves the user interacting with prompts in a console (as pictured by the screenshots in this section). The user, upon entering the system, is asked if they're a new user. If yes, then they begin the enrollment process. If no, they advance to the login phase, which will be touched upon in the next section.

The user is given guidelines and prompted to enter in all of the information required to create a record in the password file. The system continuously checks if the input is valid as per the constraints discussed. When the system has accepted each input as valid, it creates a record and places it in the password file (as seen previously). Then, the user is brought to the login screen.

```
Select your role with respect to SecVault Investments, Inc. by entering the appropriate number:
        1 - client
        2 - premium client
        3 - teller
        4 - tech support
        5 - financial advisor
        6 - financial planner
        7 - investment analyst
        8 - compliance officer
7
Processing...
User record successfully created.
```

The design for the enrollment phase also includes a security mechanism that forces the user to select a new password once they fail to re-enter the password they initially entered 3 times.

```
if (passwordSecure) {
    while (!passwordMatch) {
        System.out.println("Password meets criteria. Please re-enter password: ");
        passwordTwo = sc.next();

        if (password.equals(passwordTwo)) {
            passwordMatch = true;
            passwordSelection = false;

        } else {
            passwordMatch = false;
            ++passwordMatchTryCounter;
            if (passwordMatchTryCounter == 3) {
                System.out.println("You have tried 3 times to match passwords. Please select a new password."
                passwordSelection = true;
                break;
            }
        }
    }
}
else {
    System.out.println("Password does not match criteria. Please try again.");
    passwordSelection = true;
}
```

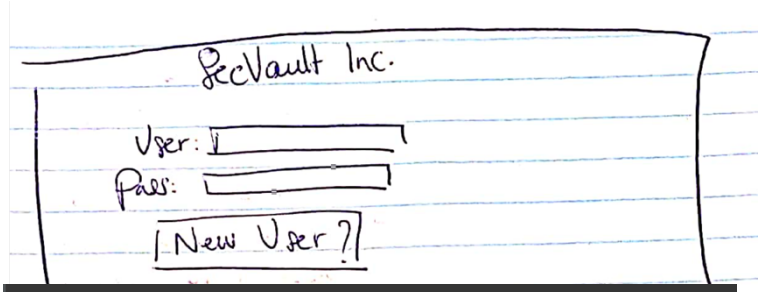
The method checkPasswordCompliance() implements the password checker and returns true if the password meets all of the security criteria, each represented by a separate Boolean.

```
public Boolean checkPasswordCompliance(String password) {
    Boolean passwordSecure = false;
    Boolean checkOne = false; //checking proper length
    Boolean checkTwo = false; //checking for an upper-case
    Boolean checkThree = false; //checking for a lower-case
    Boolean checkFour = false; //checking for a digit
    Boolean checkFive = false; //checking for a symbol from the set
    Boolean checkSix = true; //checking against a file of common passwords

    //array of the special characters to choose from to check against
    char[] specialCharacters = {'!','@','#','$','%','?','*'};

    BufferedReader reader;

    //CHECK ONE
    if ((password.length() >= 8) && (password.length() <= 12)) {
        checkOne = true;
    } else {
        checkOne =false;
    }

    //CHECK TWO
    for (int i=0;i<password.length();i++) {
        Character curr = (Character) password.charAt(i);
        if (Character.isUpperCase(curr)) {
            checkTwo = true;
            break;
        }
    }

    //CHECK THREE
    for (int i=0;i<password.length();i++) {
        Character curr = (Character) password.charAt(i);
        if (Character.isLowerCase(curr)) {
            checkThree = true;
            break;
        }
    }

    //CHECK FOUR
    for (int i=0;i<password.length();i++) {
        Character curr = (Character) password.charAt(i);
        if (Character.isDigit(curr)) {
            checkFour = true;
            break;
        }
    }

    //CHECK FIVE
    for (int i=0;i<password.length();i++) {
        Character curr = (Character) password.charAt(i);

        for (int j=0;j<specialCharacters.length;j++) {
            if (curr == specialCharacters[j]) {
                checkFive = true;
                break;
            }
        }
    }

    //CHECK SIX
    try {
        reader = new BufferedReader(new FileReader("commonPswds.txt"));
        String commonPassword = reader.readLine();
        while (commonPassword != null) {
            if (password == commonPassword) {
                checkSix = false;
                break;
            }
            commonPassword = reader.readLine();
        }
        reader.close();
    } catch (IOException e) {
        System.out.println("There was an error reading the common passwords file.");
    }

    if (checkOne && checkTwo && checkThree && checkFour && checkFive && checkSix) {
        passwordSecure = true;
    } else {
        passwordSecure = false;
    }

    return passwordSecure;
```

The user enrollment/password checker was tested by using the console to create a new user and then checking the password file to make sure the record is accurate to what was entered. As for the password checker, during the enrollment process, a variety of passwords were tried with all checks true except for one, to make sure each part of the check was working properly.

Problem 4 – Login Users

For the login phase of the system, the user is asked to enter a username and a password. This is done via console prompts and inputs, same as the user enrollment.



The system first checks to see if there's a match for the username in the password file (as explained previously). If there's a match, then the system holds that record and asks for a password. Once the user inputs a password, the system then extracts the salt and the hash from the corresponding password file record. The salt is prepended to the user's input password, then hashed using the same SHA3-256 algorithm. The user then compares the hash value generated from the input to the stored hash value in the record. If there's a match, the user is granted access into the system. If there is no match, the user is denied access.

```
SecVault Investments, Inc.
USER LOGIN

Username:
emmad
That username doesn't exist in the database. Try entering a different one.

Username:
emmad4
Password:
hola
Wrong password. Try again.
Password:
Coolio22!
Login success. Welcome!
```

```
SecVault Investments, Inc.
Hello Emma Maddock, Investment Analyst.
WHAT WOULD YOU LIKE TO DO?

1: VIEW Client Account Balance
2: VIEW,MODIFY Client Investment Portfolio
3: VIEW Private Consumer Instruments
4: VIEW Money Market Instruments
5: VIEW Derivatives Trading
6: VIEW Interest Instruments
```

If access is granted, the user's Role ID is extracted from the record and they are brought to a screen which displays all of their allowed actions based on their access, granted by the ACM based off the Role ID in their record. The action menu screen also displays their name and role title, based off fields in the record. Each user will see a different menu based off the accesses granted to their particular role.

```
SecVault Investments, Inc.
Hello Halle Berry, Premium Client.
WHAT WOULD YOU LIKE TO DO?

1: VIEW,MODIFY Client Account Balance
2: VIEW Client Investment Portfolio
3: VIEW Financial Advisor Details
4: VIEW Financial Planner Details
5: VIEW Investment Analyst Details
```

If the user is registered as a teller (roleID = 3), they will not be allowed to login outside of the hours 9am-5pm.

```
SecVault Investments, Inc.
Hello hannah goo, Tech Support.
WHAT WOULD YOU LIKE TO DO?

1: VIEW Client Information
2: REQUEST ACCESS Client Account
```

The login and action menu were tested by creating a number of users with different Role IDs and logging in as them with their respective passwords. Thenn verifying that they received the correct permissions according to the RBAC + ACM scheme.

```
//generate timestamp
Date dt = new Date();
int hour = dt.toInstant().atZone(ZoneId.systemDefault()).getHour();

//salting the user's password input
String inputSalted = salt + passwordInput;
String inputHashed = hashPassword(inputSalted);

if (inputHashed.equals(recordHash)) {
    if ((roleID != '3') || ((roleID == '3') && ((hour >= 9) && (hour <= 17)))) {
        loginMenu = false;
        username = false;
        password = false;
        System.out.println("Login success. Welcome!");

        populateHomeScreen(roleID,name);
    }
    else {
        System.out.println("You only have access to the system 9am-5pm.");
    }
} else {
    ++attempts;
    if (attempts == 3) {
        System.out.println("Too many attempts. You have been booted from the system.");
        System.exit(0);
    }
    loginMenu = true;
    username = true;
    password = true;
    System.out.println("Wrong password. Try again.");
}
```

```
public void populateHomeScreen(int roleID, String name) {

    int roleIndex = roleID - 1;

    System.out.println("\n\nSecVault Investments, Inc.");
    System.out.println("Hello " + name + ", " + roles[roleIndex] + ".");
    System.out.println("WHAT WOULD YOU LIKE TO DO?\n");

    String[] rolePermissions = permissions.get(roleIndex);

    int menuIndex = 1;
    for (int i=0;i<objects.length;i++) {
        if (rolePermissions[i] != "") {
            System.out.println(menuIndex + ": " + rolePermissions[i] + " " + objects[i]);
            ++menuIndex;
        }
    }

}
```

Problem 5 – Summary and Presentation of Full Prototype

In summary, in response to the needs and requirements set forward by SecVault Investments, Inc., a full prototype solution was provided to accomplish the following:

- Allow users to enrol in the system
    - Constraint: password must follow provide security policy
- Allow users to login to the system
    - Constraint: tellers cannot login outside the hours of 9am-5pm
- Allow users to access certain functions in the system based off their role

The system prototype works in the following way:

1. The user is asked if they are a new user. If yes, continue to step 2, if no, continue to step 4.
2. The user is in the enrollment phase. They enter all of the information required from the system. The system ensures all inputs are valid and that security policies are followed.
3. The system salts and hashes the user's password using the SHA3-256 algorithm, and places a record with the hash and other information into a password file named *pswd.txt*
4. The user is in the login phase. They are asked to first enter a username. The system checks to see if the username exists in the password file *pswd.txt*. Once a match is found, the user is then asked to enter the password associated with that username. The user's input password, along with their respective password file record's salt, is hashed again with the SHA3-256 algorithm. The user's hashed input and the record's hash value are then compared for a match. If there is a match, the user is then granted access to their action menu.
5. The user can see their action menu. Their action menu contains the permissions they have within the system based off their roleID from their record in the password file.

To compile and run the prototype:

1. Download the Eclipse Java environment
2. Unzip the *assignment3_sc.zip* file
3. Open Eclipse and go to New > Java Project
4. Name the Java Project (doesn't matter), click Finish
5. Once the project is created, right click on the project name in the Package Explorer
6. Import > expand General > Projects from Folder or Archive > Next
7. Directory > Select the assignment3 folder where the unzipped contents of the zip file are and hit Open
8. Ensure that the assignment3_sc folder is checked in the box and select Finish
9. Expand the assignment3_sc folder in the Package Explorer on the left, then src, then (default package)
10. Double click on **Login.java**
11. Click on the green circle with the white play button in the center to start the program (or alternatively navigate to Run > Run in the command bar at the top)
12. Type 'y' (without the apostrophes) and then hit enter to create a new user
13. Fill out all the required information and remember your username and password

14. If you see “User record successfully created.”, then open *pswd.txt* to see that your record was placed there
15. Enter your username and password to login
16. You should see a menu with the accesses that correspond to the role ID that you selected when creating your user

The console screenshots in each previous section of this report should serve as a reference to follow along with the above steps for each phase.

Ideally, the system would also include some sort of verification that the user actually belongs to the role that they are trying to register for. If there had been more time, a more elegant user interface would have also been designed for the prototype, rather than a console that continuously takes inputs.

In conclusion, a secure and functioning prototype of the outlined system requested by SecVault Investments, Inc. was designed and implemented using security principles and mechanisms to enforce a security policy and constraints.