Finvest Holdings
Security Software & Analysis

SYSC 4810 Assignment

Nathan MacDiarmid

101098993

December 4th, 2023

Table of Contents

Problem 1

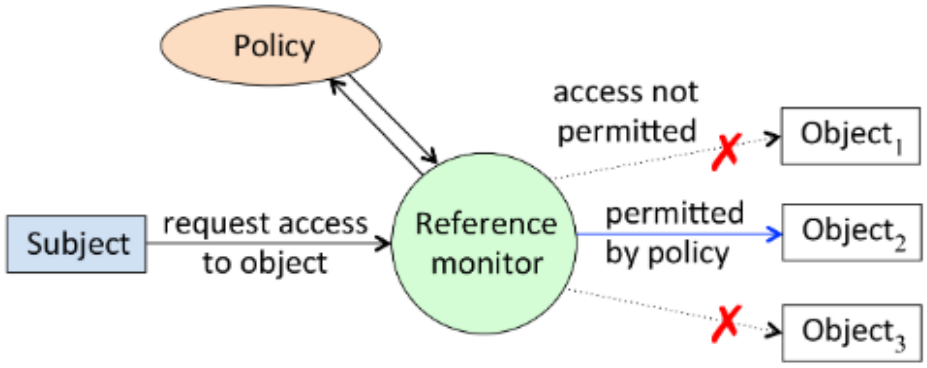
a) The access control model that the Finvest Holdings security system will be the RBAC control model. This is because RBAC sets permissions based on roles which are like what is described in Part 1 Section 2 – Context. RBAC also allows for the expandability of permissions via additions of different roles, making it very scalable.

b) The access control representation that the Finvest Holdings security system will be is an access control matrix. This is because each of the roles have different access to various permissions and an access control matrix allows for the setting of permissions for each role separately. Furthermore, each row can be used as a capabilities list, depicting the permissions of each role, and each column can be used as an access control list, depicting how many roles have access to a selected permission.

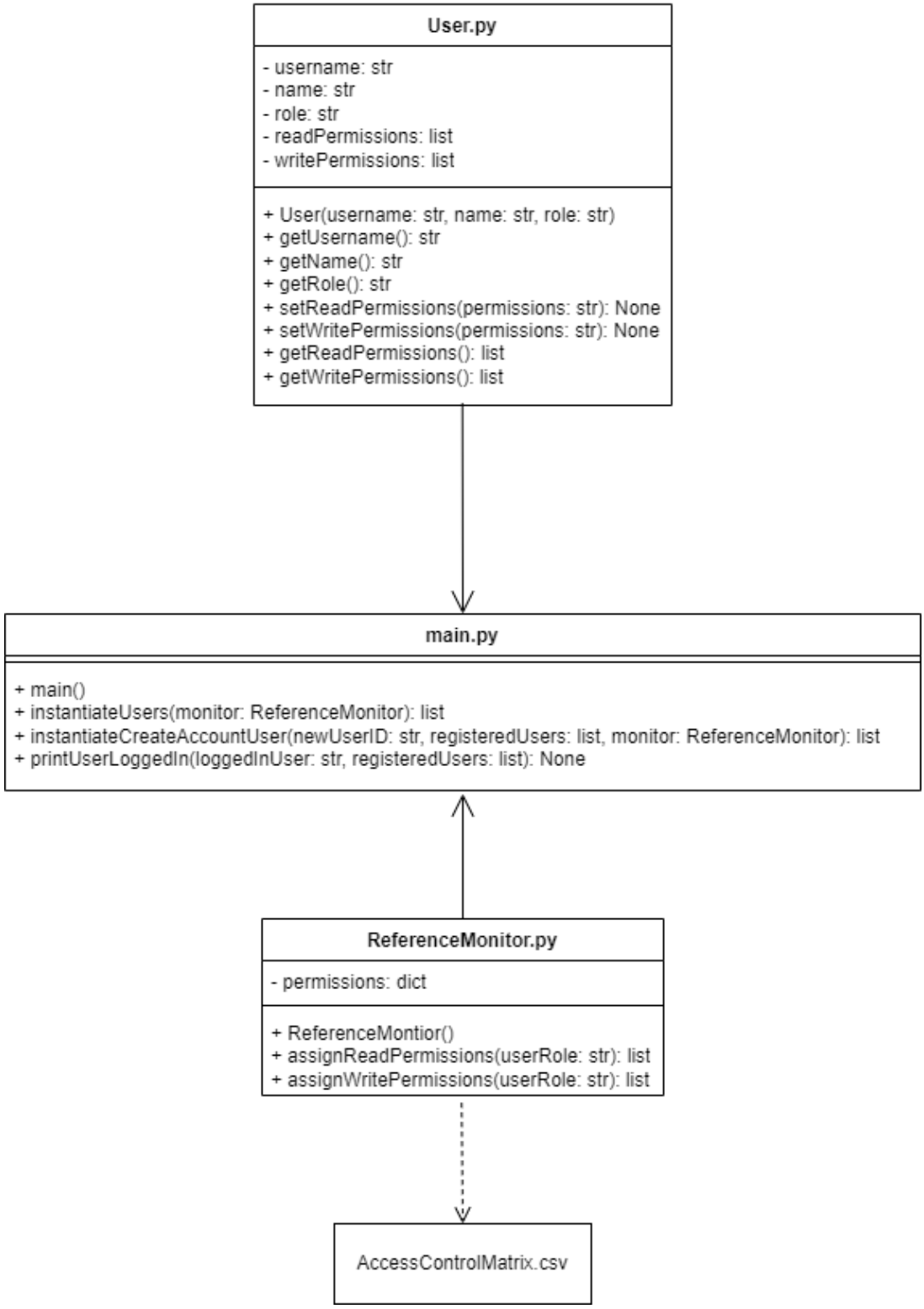
c) In my access control matrix, a role can only have read, write, or no permissions at all. Since there didn't seem to be any programs that required running in the description as a role would either view the information or edit that information, I chose to only allow for the options of reading and writing. Upon checking if the User is a Teller, the User will be notified of the times they are allowed to access the program.

Roles	Account balance	Investment portfolio	FA contact details	FI contact details	FP contact details	Clients account balance	Clients investment portfolio	Money market instruments	Private consumer instruments	Derivatives trading	Interest instruments	Clients account access	Validate investment portfolio modifications
Client	R	R	R	-	-	-	-	-	-	-	-	-	-
Premium Client	R	RW	-	R	R	-	-	-	-	-	-	-	-
Financial Planners	-	-	-	-	-	RW	RW	R	R	-	-	-	-
Financial Advisors	-	-	-	-	-	RW	RW	-	R	-	-	-	-
Investment Analysts	-	-	-	-	-	RW	RW	R	-	R	R	R	R
Technical Support	-	-	-	-	-	R	R	-	-	-	-	RW	RW
Tellers	-	-	-	-	-	R	R	-	-	-	-	-	-
Compliance Officers	-	-	-	-	-	R	R	-	-	-	-	-	R

d) The access control mechanism was implemented in an object-oriented way such that it models the example of a reference model implementation that was given in class in slides Ch5_2[3]. The example given in class and a UML diagram of the access control mechanism is depicted below.



AccessControlMechanism



In the example, the Subject requests permission access from the Reference Monitor and the Reference Monitor assigns some or no permissions based on a Policy. In my access control mechanism, the User is a Subject requesting permissions from the ReferenceMonitor that is determined by the AccesesControlMatrix. Each User contains a username, a name, a role, readPermissions and writePermissions which are all private attributes and can only be accessed via getter and setter methods. The username is a unique name only associated to that User and treated like a user ID. The name is the Users personal name, and the Users role is the permissions role such as Client, Premium Client, etc. The readPermissions and writePermissions contained the appropriate permissions that were sent back from the ReferenceMontior.

User.py > User > __init__

You, 41 minutes ago | 1 author (You)

```python
class User:
    def __init__(self, username, name, role) -> None:
        self.__username = username
        self.__name = name
        self.__role = role
        self.__readPermissions = []
        self.__writePermissions = []      You, 41 minutes ago • Uncommitted changes

    def getUsername(self) -> str:
        return self.__username

    def getName(self) -> str:
        return self.__name

    def getRole(self) -> str:
        return self.__role

    def setReadPermissions(self, permissions) -> None:
        self.__readPermissions = permissions

    def setWritePermissions(self, permissions) -> None:
        self.__writePermissions = permissions

    def getReadPermissions(self) -> list:
        return self.__readPermissions

    def getWritePermissions(self) -> list:
        return self.__writePermissions
```

The Reference Monitor in the example controls what Subjects get access to what permissions based on a Policy. In my access control mechanism, the ReferenceMonitor reads from the AccessControlMatrix file and stores all the permissions in a private dictionary attribute called permissions. The ReferenceMonitor is the only file that can access the AccessControlMatrix and has two methods that control what permissions a User receives called assignReadPermissions() and assignWritePermissions(). With the ReferenceMonitor being the only file that can access the AccessControlMatrix, it adds an extra layer of security to the system by extrapolating the matrix to only one entry point; and since no other files have direct access, the risk of another file changing or modifying permissions decreases dramatically.

Like their names suggest, the methods in the ReferenceMonitor assign a User their read and write permissions respectfully. They each do this by receiving a User's role that's passed into the method and returning a list of permissions based on that specific role. If the User role that's passed does not exist, or has no permissions, these methods return an empty list, ensuring that User's are only passed the proper rules.

The main file is where everything is instantiated, and different User's get their defined roles based on the ReferenceMonitor's AccessControlMatrix. One instance of the ReferenceMonitor is instantiated as well as the list of User's that were provided in the assignment description. These User's are only used as an example. The main file contains four methods, instantiateUsers(), instantiateCreateAccountUser(), printUserLoggedIn(), and main().

The instantiateUsers() method takes in a reference to a ReferenceMonitor instance, creates instances of all the Users, and sets the Users' permissions based on the AccessControlMatrix from the ReferenceMonitor instance. The instantiateCreateAccountUser() takes in a newUserID which is a string representing the ID of a new User that isn't registered in the system yet. It also takes in a list of registeredUsers, and a reference to a ReferenceMonitor instance. The monitor instance is used to set this new User's permissions. Every new User is automatically assigned the

‘Client’ role as it has the lowest level of permissions given in the assignment description. The printUserLoggedIn() method takes in a loggedInUser which is the username of the User that successfully logged in, and a list of registeredUsers. The printUserLoggedIn() method prints out all the appropriate information that a User should receive upon logging in given the assignment description. This includes the User’s username, role, readPermissions and writePermissions. It also includes special permissions if the User’s role is a ‘Teller’. Lastly, the main() method takes no inputs and calls all three of these previously defined methods. The main() method also instantiates a ReferenceMonitor instance and keeps a list of registeredUsers in the system. Furthermore, it is the only method that is called globally in the entire program and renders the entire system.

main.py > setUserPermissions

You, 7 minutes ago | 1 author (You)

```python
import UI
import User
import ReferenceMonitor

def instantiateUsers() -> list:
    mischa = User.User("mischa", "Mischa Lowery", "Client")
    veronica = User.User("veronica", "Veronica Perez", "Client")

    winston = User.User("winston", "Winston Callahan", "Teller")
    kelan = User.User("kelan", "Kelan Gough", "Teller")

    nelson = User.User("nelson", "Nelson Wilkins", "Financial Advisor")
    kelsie = User.User("kelsie", "Kelsie Chang", "Financial Advisor")

    howard = User.User("howard", "Howard Linkler", "Compliance Officer")
    stefania = User.User("stefania", "Stefania Smart", "Compliance Officer")

    willow = User.User("willow", "Willow Garza", "Premium Client")
    nala = User.User("nala", "Nala Preston", "Premium Client")

    stacy = User.User("stacy", "Stacy Kent", "Investment Analyst")
    keikilana = User.User("keikilana", "Keikilana Kapahu", "Investment Analyst")

    kodi = User.User("kodi", "Kodi Matthews", "Financial Planner")
    malikah = User.User("malikah", "Malikah Wu", "Financial Planner")

    caroline = User.User("caroline", "Caroline Lopez", "Technical Support")
    pawel = User.User("pawel", "Pawel Barclay", "Technical Support")

    return [mischa, veronica, kelan, nelson, howard, stefania, pawel, winston,
            willow, nala, stacy, keikilana, kodi, malikah, caroline, kelsie]
```

main.py > instantiateUsers

```
82      return [mischa, veronica, kelan, nelson, howard, stefania, pawel, winston,
83              willow, nala, stacy, keikilana, kodi, malikah, caroline, kelsie]
84
85  def instantiateCreateAccountUser(newUserID, registeredUsers, monitor) -> list:
86      userFound = False
87      for i in registeredUsers:
88          if (newUserID == i.getUsername()):
89              userFound = True
90      if (not userFound and newUserID != ""):
91          newUser = User.User(newUserID, newUserID, "Client")
92          registeredUsers.append(newUser)
93          newUser.setReadPermissions(monitor.assignReadPermission(newUser.getRole()))
94          newUser.setWritePermissions(monitor.assignWritePermission(newUser.getRole()))
95      return registeredUsers
96
97  def printUserLoggedIn(loggedInUser, registeredUsers) -> None:
98      for user in registeredUsers:
99          if (loggedInUser == user.getUsername()):
100             print("Username (ID): " + user.getUsername())
101             print("Role: " + user.getRole())
102             if (user.getRole() == "Teller"):
103                 print("As a Teller, you can only access this account from 9am to 5pm")
104             print("\nRead Permissions:")
105             if (user.getReadPermissions() == []):
106                 print("No read permissions")
107             else:
108                 for i in user.getReadPermissions():
109                     print(i)
110             print("\nWrite Permissions:")
111             if (user.getWritePermissions() == []):
112                 print("No write permissions")
113             else:
114                 for j in user.getWritePermissions():
115                     print(j)
116
117 def main():
118     monitor = ReferenceMonitor.ReferenceMonitor()
119     ui = UI.UI()
120
121     registeredUsers = instantiateUsers(monitor)
122
123     loggedInUser = ui.renderUI()
124
125     registeredUsers = instantiateCreateAccountUser(loggedInUser, registeredUsers, monitor)
126
127     printUserLoggedIn(loggedInUser, registeredUsers)
128
129 main()
```

The reason that I chose to create a User class and a ReferenceMonitor class separately was because it keeps the AccessControlMatrix away from the User. By not giving the User direct access to the AccessControlMatrix, I added another layer of security to the system by only providing the User with what it needs which is the given permissions.

e) To test the AccessControlMatrix, I wrote unit tests that assert that each of the roles return the specifically provided permissions as described in the AccessControlMatrix. I also tested that invalid roles did not receive any permissions as well as no input. All AccessControlMatrix tests are found in the AccessControlMatrixTests.py file and can be run by running the file itself, or as a test suite by running UnitTests.py.

AccessControlMatrixTests.py > ...

You, 1 hour ago | 1 author (You)

```python
import ReferenceMonitor        You, 1 hour ago • Added tests and working on report

def testNoRole():
    print('Running No Role tests')
    monitor = ReferenceMonitor.ReferenceMonitor()
    readPermissions = monitor.assignReadPermission('')
    writePermissions = monitor.assignWritePermission('')
    assert(readPermissions == [])
    assert(writePermissions == [])
    print('Passing No Role tests\n')

def testInvalidRole():
    print('Running Invalid Role tests')
    monitor = ReferenceMonitor.ReferenceMonitor()
    readPermissions = monitor.assignReadPermission('Invalid')
    writePermissions = monitor.assignWritePermission('Invalid')
    assert(readPermissions == [])
    assert(writePermissions == [])
    print('Passing Invalid Role tests\n')

def testClient():
    print('Running Client tests')
    monitor = ReferenceMonitor.ReferenceMonitor()
    readPermissions = monitor.assignReadPermission('Client')
    writePermissions = monitor.assignWritePermission('Client')
    assert(readPermissions == ['Account balance', 'Investment portfolio', 'FA contact details'])
    assert(writePermissions == [])
    print('Passing Client tests\n')
```

Problem 2

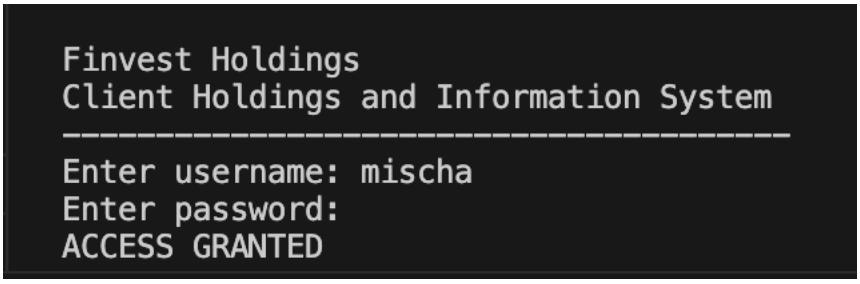
a) Since I used Python, there is a library called hashlib that can be imported from the Python API [1]. The hashlib library contains various hash functions that are one way, contain second-preimage resistance, and collision resistance which are all three properties that a proper hash function should have. The Secure Hash Algorithm (SHA) hash functions vary from 224 bits to 512 bits. Although SHA-1 has known security vulnerabilities, SHA-2 does not. SHA-2 are the updated hash functions that eliminate these vulnerabilities once again create an unhashable function. I used the SHA512 hash function as it uses the most bits to hash a given string, resulting in the most possible number of combinations of bits to hash that string. This makes it the most difficult hash function to decrypt as it contains the highest number of bits.

b) The password record structure of the information that is stored in the passwd.txt file follows the format “username salt hashed salt and password combination”. The password file does not contain permissions of a User, nor the User’s role itself. This is because passwd.txt is simply used as a verification tool to verify that User’s username and password match the given login information. I did this because it adds another additional layer of security to the system specifically regarding offline attacks. Since the passwd.txt file only contains a username, salt, and salt and password combination, an attacker looking for a specific account with specific permissions will be unable to know which ones are which. This deters the attacker more because if they were somehow able to figure out the hashed password, they wouldn’t know what type of permissions they would be getting, especially if they’re looking for a ‘Premium Client’ for example that is able to edit a User’s portfolio.

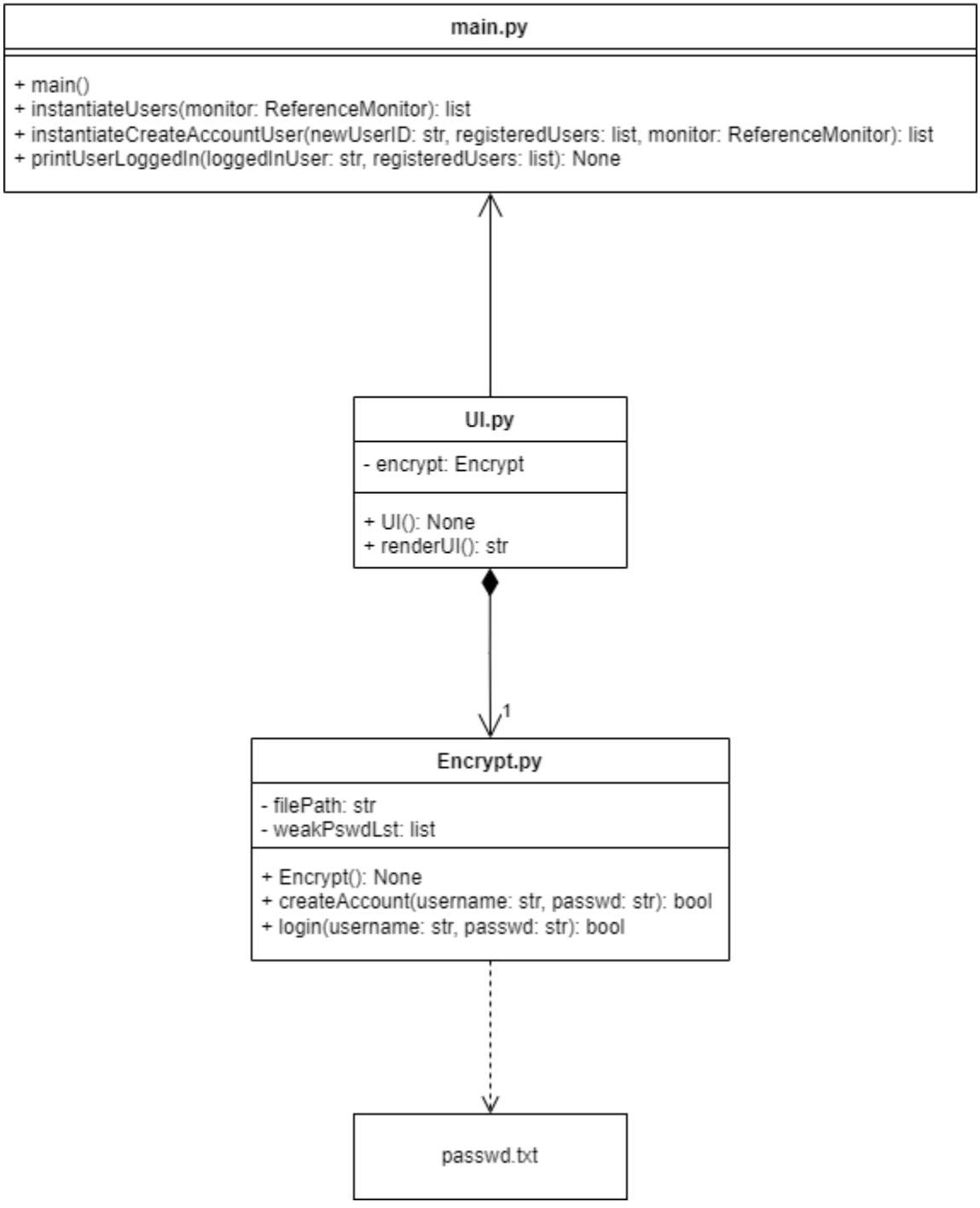
Below is a screenshot of the passwd.txt file with all the Users from the assignment description loaded in. For testing purposes, all the usernames are the names listed and all their passwords

are ‘IloveCats2!’. As you can see, all the password hashes are different, even with the password being the same. This proves that the SHA512 hash function that was chosen contains all three properties that are needed in a hash function.

c) The password file is where a generic user interface began. Since Problem 4 contains instructions on designing a user interface to log Users in and this problem needed abilities to get and store passwords with other information, I decided to implement a basic UI so that I could pass my own information into passwd.txt. This allowed for manual testing during the development of the password file, ensuring that what I stored in the file could be properly retrieved later. The basic UI that I used, and the UML diagram of password file portion of this assignment are depicted below.



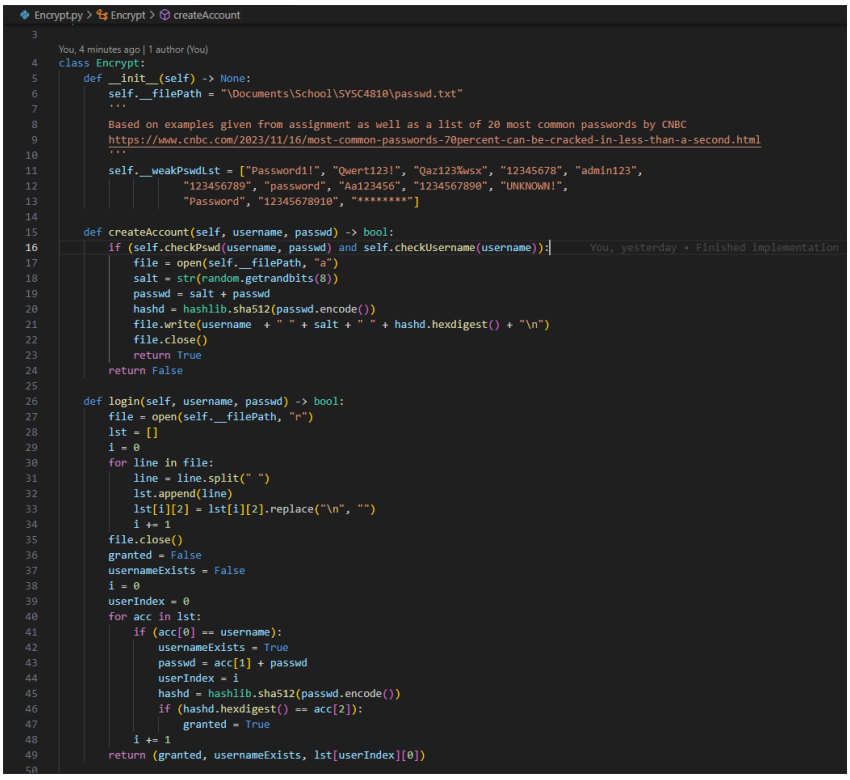
Password File



The passwd.txt file is only accessible by an Encrypt class. This is like the ReferenceMonitor and AccessControlMatrix scenario from Problem 1, by extrapolating the passwd.txt to only one access point, the Encrypt class, it dramatically reduces the possibility of other files changing or incorrectly accessing parts of the passwd.txt file. Additionally, the Encrypt class contains two private attributes called filePath and weakPswdLst. The filePath is the string of the path to the passwd.txt file, there are no getters or setters for this attribute so that it is more difficult for an attacker to change the file path to a possible bad passwd.txt file. The weakPswdLst is a list of commonly used passwords that should be avoided, which will be touched on more in Problem 3 c.

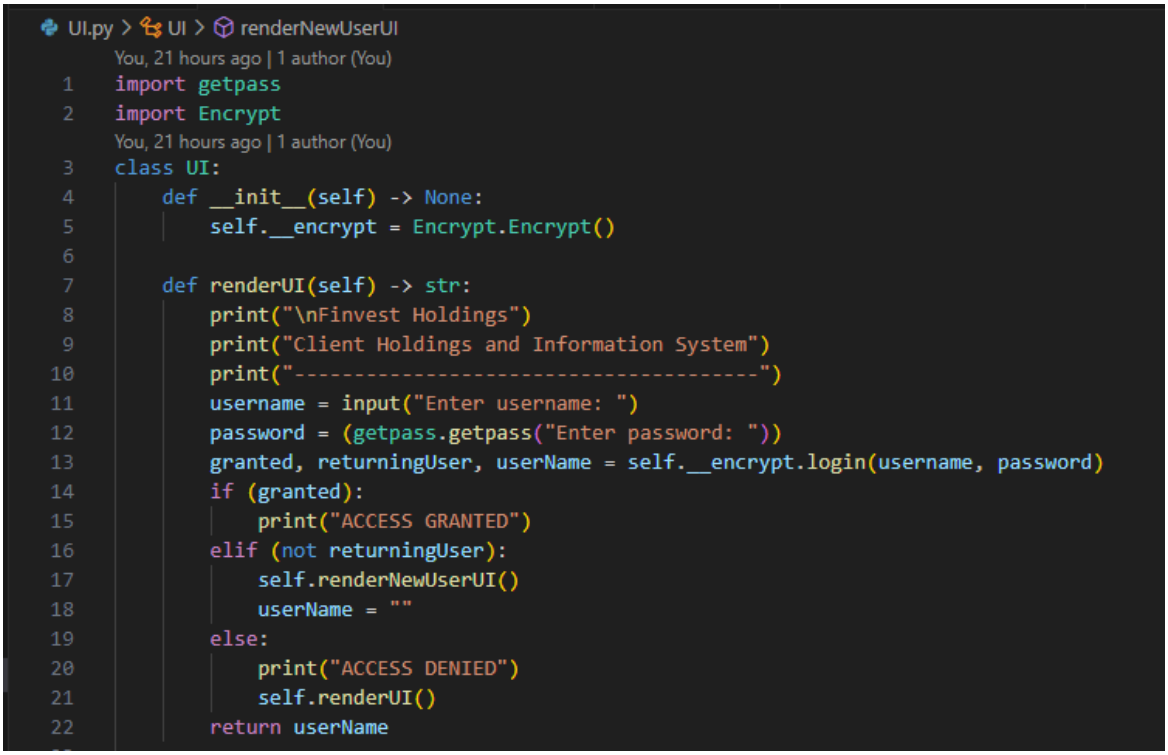
The Encrypt class also contains a createAccount() method as well as a login() method that allows for new information to be passed to the passwd.txt file as well as previously stored information to be retrieved. The createAccount() stores information in the passwd.txt file by taking in a username and password, generating a random salt value from 0 -> 2^8, concatenating the salt with the given password, hashing the concatenated password with SHA512 and storing the hashed password with the given username and the generated salt value in the 'user salt concatenated password' pattern.

The login() method allows for the retrieval of information from the passwd.txt file. It does this by taking in a username and password, like createAccount(), but instead it reads the file line by line, looking for a match on the given username. If it finds a username that matches the given username, login() concatenates the stored salt attached to that username to the given password, hashes that with SHA512 and compares it to the hashed password attached to the username that is stored in passwd.txt. If the password hashes match, the username is then retrieved from passwd.txt and returned. There will be more explanation on this in Problem 4 c.



Any of the inputs that are going into the createAccount() or login() methods come from the UI class. This was my manual testing entry point for this problem as it allowed me to pass and retrieve Users from the passwd.txt file. The UI class contains one attribute called encrypt which is an instance of the Encrypt class. It is a private attribute with no getters and setters. It also contains one method called renderUI() that takes no inputs.

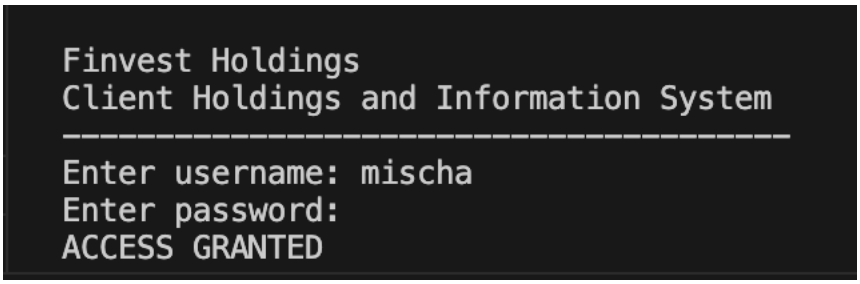
The renderUI() method uses the built-in input() functionality from Python and requests input from the user via terminal. A username and password variable are used locally to store this input and then passed to the login() method from the Encrypt class. Depending on the return value of login(), the user can then call the createAccount() method from the Encrypt class or retry the login() method. If the username and password combination are successful, the username is then passed to main to handle assigning the proper permissions to that User. More information on this will be discussed in Problem 4 c.



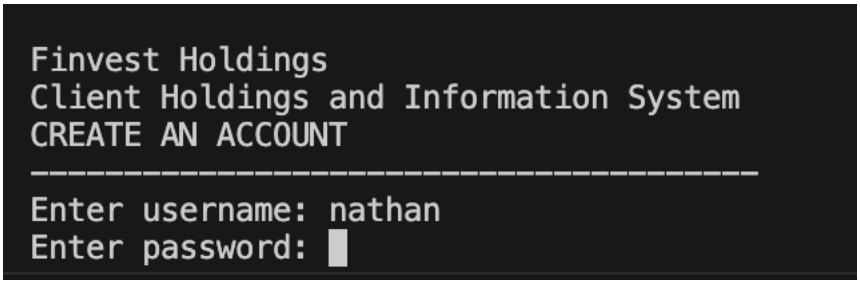
d) During the development process, I tested the password file by initially passing in different username and password pairs and saving them to the passwd.txt file. I then implemented the ability to hash the password, including the hexdigest() function so that it wasn't saved as a hashed object but instead a string of letters and numbers. After, I implemented a random bit generator to generate a random salt value to be attached to the password before its hashed. I then tested that this salt and password combo would save correctly. I then made sure that I could retrieve the salt password combo from passwd.txt. Lastly, I made sure that when I appended the plaintext salt from passwd.txt to the same plaintext password that I stored, they matched. Since I developed the test cases after the entire implementation was finished, the test cases from Problem 3 d and Problem 4 d will subsume these test cases.

Problem 3

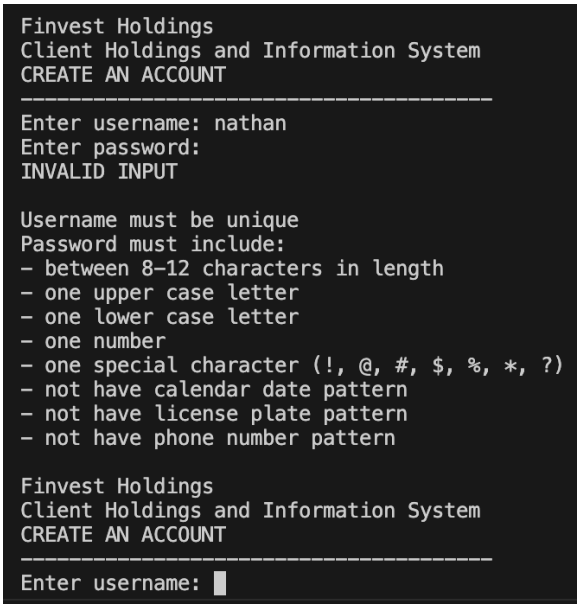
a) The simple user interface is depicted identically to what was asked in the assignment description. The first and only text that the user sees upon trying to login to the system is 'Finvest Holdings Client Holdings and Information System'. It also prompts the user to enter a username followed by a password. If the username and password pair match a user, 'ACCESS GRANTED' is printed, otherwise, if the username exists and the password doesn't match, 'ACCESS DENIED' is printed. This will differ slightly in Problem 4 c when implementing the actual login capabilities. The password is not shown to protect one's input which is done using the getpass Python library [2]. The Python library getpass contains no known security flaws and does not store input anywhere. All of this is rendered and printed out by the UI class.



If the given username doesn't exist, the enroll user interface is displayed. It contains the 'Finvest Holdings' and 'Client Holdings and Information System' text like the login page, with the addition of the 'CREATE ACCOUNT' text. This again prompts the User for a username and password pair.



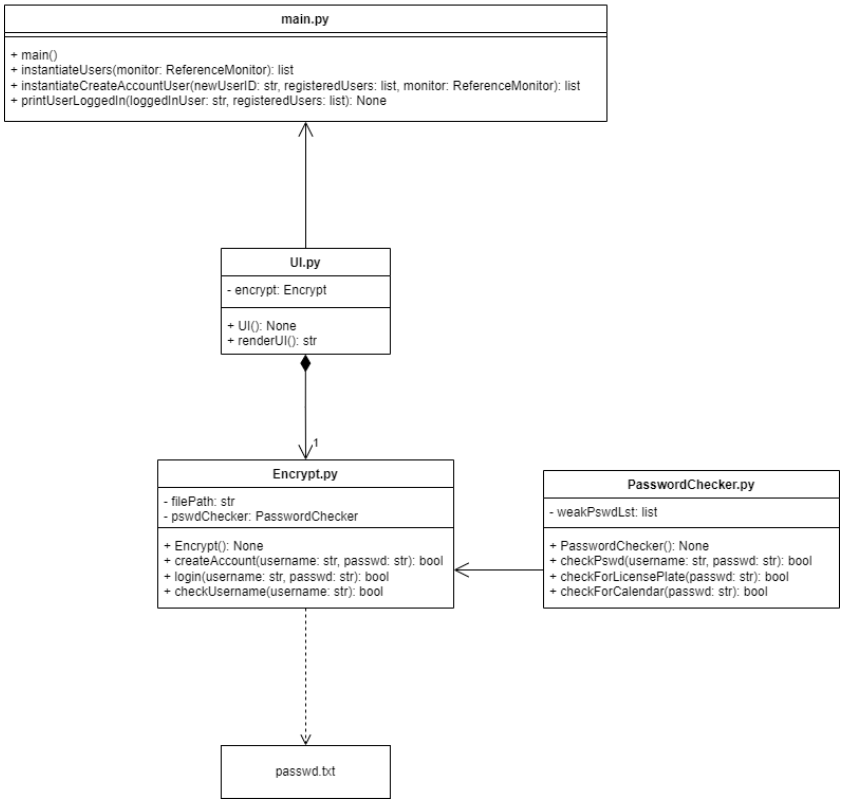
If the username already exists or the password does not meet the requirements, it will print out 'INVALID INPUT', as well as 'Username must be unique' and 'Password must include:' followed by all the password requirements as depicted in the assignment description and seen in Problem 3 b. Once the user account has been created, the program ends and must be restarted for the User to be able to login with the username and password pair they provided. This is illustrated in Problem 4 c.



b) The proactive password checker has a few different parts to it, making sure that each of the required portions for creating the password are met. The password checker requires passwords to be 8-12 characters in length and must include at least one upper-case letter, one lower-case letter, one numerical digit, and one special character from the set {!, @, #, $, %, *}. The password checker also contains a blacklist of commonly used weak passwords that was found in an article written by CNBC, a news platform [3]. The password checker also protects against calendar dates, phone numbers, and license plate formats. A password contains a calendar date if it contains six numerical digits one after another. This also inadvertently protects against phone numbers since a phone number is nine numerical digits and a calendar format will already be detected and not allowed upon the sixth digit. The password checkers protects against Ontario specific license plate numbers in the format WXYZ123 where WXYZ are any letter, upper-case or lower-case and 123 are any numerical digit. If any of these formats are detected anywhere within the provided potential password, including beginning, middle, or end, the password is not allowed. Lastly, the password cannot be the same as the username.

c) The proactive password checker is implemented as its own PasswordChecker class. I decided to do this as it separates the concerns keeping the single responsibility principle in mind. The PasswordChecker contains all the methods necessary to check the given password format so that all the requirements in Problem 3 b are met. Below is the updated UML diagram on the passwd.txt side of main.py that depicts all the methods inside PasswordChecker and how it relates to the rest of the program.

Password File



The Encrypt class now has a PasswordChecker instance variable. This is used to call the methods from PasswordChecker directly in the Encrypt class. As described in Problem 2 c, the username and password are already being passed from the UI to the Encrypt class. With the PasswordChecker addition, the Encrypt class simply passes the username and password one step further to the checkPswd() method in PasswordChecker to run the password checks. The username and password are not saved in the PasswordChecker and are only passed as attributes, avoiding any possible security risks. A Boolean is returned by checkPswd() to the Encrypt class, returning true if the password can be used and false otherwise. This can be seen on line 11 in the code snippet below.

```
Encrypt.py > Encrypt > createAccount

You, 20 minutes ago | 1 author (You)
1  import hashlib
2  import random
3  import PasswordChecker
4
   You, 20 minutes ago | 1 author (You)
5  class Encrypt:
6      def __init__(self) -> None:
7          self.__filePath = "passwd.txt"
8          self.__pswdChecker = PasswordChecker.PasswordChecker()
9
10     def createAccount(self, username, passwd) -> bool:
11         if (self.__pswdChecker.checkPswd(username, passwd) and self.checkUsername(username)):
12             file = open(self.__filePath, "a")
13             salt = str(random.getrandbits(8))       You, last week • Added password file and enrol users
14             passwd = salt + passwd
15             hashd = hashlib.sha512(passwd.encode())
16             file.write(username  + " " + salt + " " + hashd.hexdigest() + "\n")
17             file.close()
18             return True
19         return False
```

This next photo inside the Encrypt class depicts the functionality of checkUsername() method that verifies that the given username is unique. This method is in the Encrypt class and not the PasswordChecker as Encrypt should be the only one to be able to access the passwd.txt file. This limits the number of potential security risks.

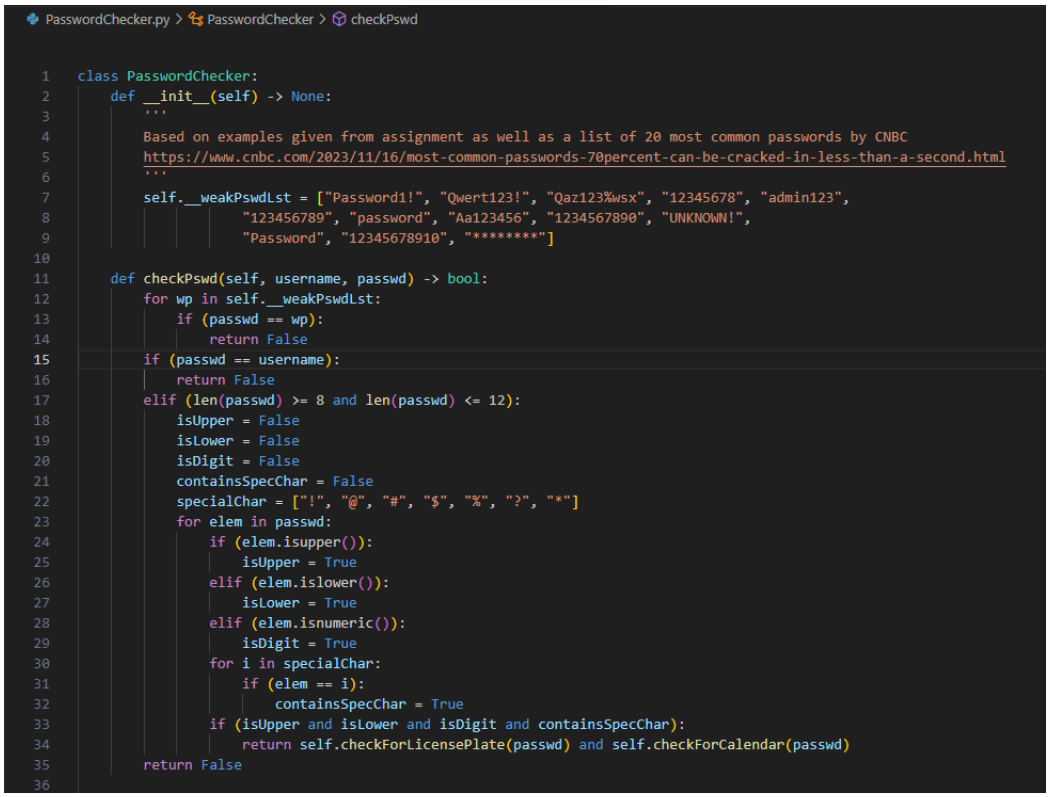
```
20
21     def login(self, username, passwd) -> bool:
22         file = open(self.__filePath, "r")
23         lst = []
24         i = 0
25         for line in file:
26             line = line.split(" ")
27             lst.append(line)
28             lst[i][2] = lst[i][2].replace("\n", "")
29             i += 1
30         file.close()
31         granted = False
32         usernameExists = False
33         i = 0
34         userIndex = 0
35         for acc in lst:
36             if (acc[0] == username):
37                 usernameExists = True
38                 passwd = acc[1] + passwd
39                 userIndex = i
40                 hashd = hashlib.sha512(passwd.encode())
41                 if (hashd.hexdigest() == acc[2]):
42                     granted = True
43             i += 1
44         return (granted, usernameExists, lst[userIndex][0])
45
46     def checkUsername(self, username) -> bool:
47         file = open(self.__filePath, "r")
48         for line in file:
49             line = line.split(" ")
50             if (line[0] == username):
51                 return False
52         return True       You, 6 days ago • Finished password checker
```

The PasswordChecker class contains one private attribute called weakPswdLst which is the list of weak passwords from CNBC described in Problem 3 b. PasswordChecker has three methods called checkPswd(), checkForLicencePlate(), and checkForCalendar(). All these methods are part

of the password checker implementation. The createAccount() method now passes the username and password to checkPswd() to perform the password checks described in Problem 3 b. The reason both the username and password are passed to the checkPswd() method is to verify that the username and password are not the same.

The checkPswd() method is the main method in PasswordChecker that returns the final Boolean to the createAccount() method in Encrypt, being true if the password is usable and false otherwise. It begins by parsing the weakPswdLst and checking if the given password matches any of those, if it does, checkPswd() returns false and the remainder of the method does not run. The checkPswd() method then checks if the password and username are the same, if they are, checkPswd() returns false. If the username and password are not the same, checkPswd() checks if the password length is between 8 and 12 inclusively. If that's true, checkPswd() loops through each individual character of the password and checks if there's an upper-case and lower-case letter, a number, and a special character from the set {!, @, #, $, %, *}. The checkPswd() method stores if the character has each of these in Boolean expressions for each requirement, i.e., isUpper is changed to true if an upper-case character is found. If all three of these Booleans are true, checkPswd() returns the Boolean result of a call of checkLicensePlate() and checkForCalender(). Otherwise, checkPswd() always returns false, indicating that the password is not usable.

```
23
24      def renderNewUserUI(self) -> None:
25          print("\nFinvest Holdings")
26          print("Client Holdings and Information System")
27          print("CREATE AN ACCOUNT")
28          print("-------------------------------------")
29          username = input("Enter username: ")
30          password = (getpass.getpass("Enter password: "))
31          if (not self.__encrypt.createAccount(username, password)):
32              print("INVALID INPUT\n")
33              print("Username must be unique")
34              print("Password must include:")
35              print("- between 8-12 characters in length")
36              print("- one upper case letter")
37              print("- one lower case letter")
38              print("- one number")
39              print("- one special character (!, @, #, $, %, *, ?)")
40              print("- not have calendar date pattern")       You, last
41              print("- not have license plate pattern")
42              print("- not have phone number pattern")
43              self.renderNewUserUI()
```




```python
class PasswordChecker:
    def __init__(self) -> None:
        '''
        Based on examples given from assignment as well as a list of 20 most common passwords by CNBC
        https://www.cnbc.com/2023/11/16/most-common-passwords-70percent-can-be-cracked-in-less-than-a-second.html
        '''
        self.__weakPswdLst = ["Password1!", "Qwert123!", "Qaz123%wsx", "12345678", "admin123",
                    "123456789", "password", "Aa123456", "1234567890", "UNKNOWN!",
                    "Password", "12345678910", "********"]

    def checkPswd(self, username, passwd) -> bool:
        for wp in self.__weakPswdLst:
            if (passwd == wp):
                return False
        if (passwd == username):
            return False
        elif (len(passwd) >= 8 and len(passwd) <= 12):
            isUpper = False
            isLower = False
            isDigit = False
            containsSpecChar = False
            specialChar = ["!", "@", "#", "$", "%", "?", "*"]
            for elem in passwd:
                if (elem.isupper()):
                    isUpper = True
                elif (elem.islower()):
                    isLower = True
                elif (elem.isnumeric()):
                    isDigit = True
                for i in specialChar:
                    if (elem == i):
                        containsSpecChar = True
                if (isUpper and isLower and isDigit and containsSpecChar):
                    return self.checkForLicensePlate(passwd) and self.checkForCalendar(passwd)
        return False

```

The checkForLicensePlate() method checks for an Ontario license plate format, WXYZ123, anywhere within the password, this could be the beginning the middle or the end. The method returns true if the password does not contain a direct license plate format and false if it does contain the license plate format, indicating that the password is not usable. It first initializes four variables to 0 called letterCount, numberCount, curr, and prev. The letterCount is to keep track of how many letters in a row there are in the password. The numberCount is for how many numbers in a row there are, but this only starts counting after letterCount has reached four, i.e., there are four letters in a row. The curr and prev are simply to keep track of which character index the loop is on while looping through the password with curr being the current character index and prev being the previous character index. Every loop through, checkForLicensePlate() checks if the letterCount is four. If it is, it then checks if the current element is a number, if so, it increments numberCount. Otherwise, the numberCount is reset to zero. If letterCount is not four, checkForLicensePlate() then checks if the current element is a letter, if so, it increments letterCount. Lastly, the loop checks if letterCount and numberCount equal four and three respectfully, if they do, we have a license plate pattern and false is returned, indicating that the password can't be used. Otherwise, curr and prev are incremented and the loop continues. If checkForLicensePlate() doesn't return prematurely, it returns true, indicating that the password is valid and useable.

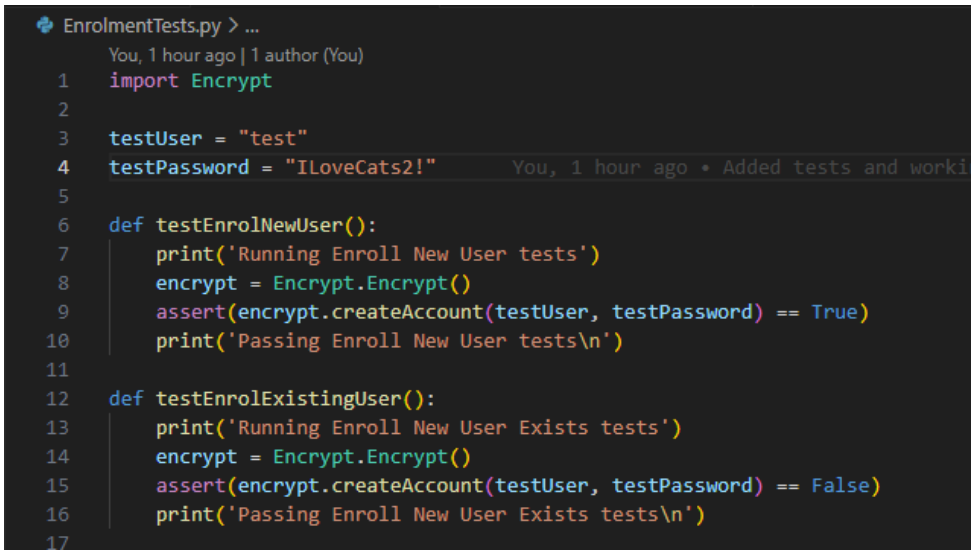
```
36
37      def checkForLicensePlate(self, passwd) -> bool:
38          letterCount = 0
39          numberCount = 0
40          curr = 0
41          prev = 0
42          for elem in passwd:
43              if (letterCount == 4):
44                  if (elem.isnumeric()):
45                      numberCount += 1
46                  elif (passwd[curr].isnumeric() and passwd[prev].isnumeric() and numberCount > 0):
47                      numberCount += 1
48                  else:
49                      numberCount = 0
50              else:
51                  if (elem.isalpha()):
52                      letterCount += 1
53                  elif (passwd[curr].isalpha() and passwd[prev].isalpha() and letterCount > 0):
54                      letterCount += 1
55                  elif (letterCount != 4):
56                      letterCount = 0
57              if (letterCount == 4 and numberCount == 3):
58                  return False
59              curr += 1
60              prev = curr - 1
61          return True
62
```

The checkForCalendar() method is much less complex than the checkForLicensePlate() method. Following the format for a date being 23/12/06, it is a sequence of six consecutive numbers. So checkForCalendar() simply iterates through the password and checks if the character in the password is a number. If so, it increments a numberCount like checkForLicensePlate(), otherwise it resets to zero. If the numberCount ever reaches six consecutive numbers, checkForCalendar() returns false, indicating that the password is unusable. Otherwise, the curr and prev variables are incremented like checkForLicensePlate() and the loop rolls over. If checkForCalendar() doesn't find six consecutive numbers, it returns true, indicating that the password is usable. Like I mentioned previously, this method also implicitly detects phone numbers and the other popular date format which is 123-456-7890 and 2023/12/06 respectfully. This is because both have more consecutive numbers than six, so the checkForCalendar() method already handles it. I did this to reduce redundant code.

```
62
63      def checkForCalendar(self, passwd) -> bool:
64          numberCount = 0
65          curr = 0
66          prev = 0
67          for elem in passwd:
68              if (elem.isnumeric()):
69                  numberCount += 1
70              elif (passwd[curr].isnumeric() and passwd[prev].isnumeric() and numberCount > 0):
71                  numberCount += 1
72              elif (numberCount != 6):
73                  numberCount = 0
74              if (numberCount >= 6):
75                  return False
76              curr += 1
77              prev = curr - 1
78          return True
```

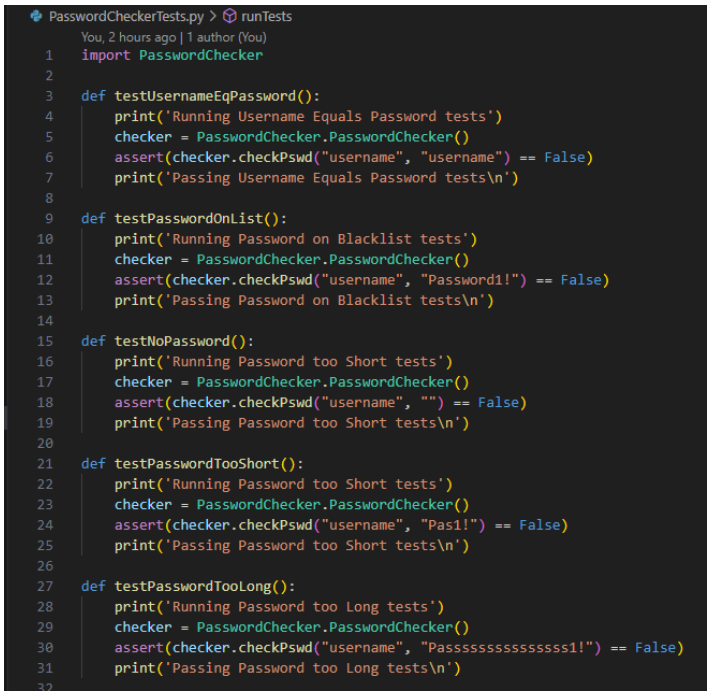
If all these conditions are met, the createAccount() method in the Encrypt class is allowed to save the username and encrypted password in the passwd.txt file.

d) To test the enrollment mechanism, I created a unit test to ensure a new User can be created and saved to the passwd.txt file. I also create a unit test to ensure that a new User cannot be created with the same username as an already existing User. These unit tests can be found in EnrolmentTests.py and can be run by either running the file directly or running the UnitTests.py file that runs all unit tests in a suite.



```
EnrolmentTests.py > ...
      You, 1 hour ago | 1 author (You)
1     import Encrypt
2
3     testUser = "test"
4     testPassword = "ILoveCats2!"       You, 1 hour ago • Added tests and worki
5
6     def testEnrolNewUser():
7         print('Running Enroll New User tests')
8         encrypt = Encrypt.Encrypt()
9         assert(encrypt.createAccount(testUser, testPassword) == True)
10        print('Passing Enroll New User tests\n')
11
12    def testEnrolExistingUser():
13        print('Running Enroll New User Exists tests')
14        encrypt = Encrypt.Encrypt()
15        assert(encrypt.createAccount(testUser, testPassword) == False)
16        print('Passing Enroll New User Exists tests\n')
17
```

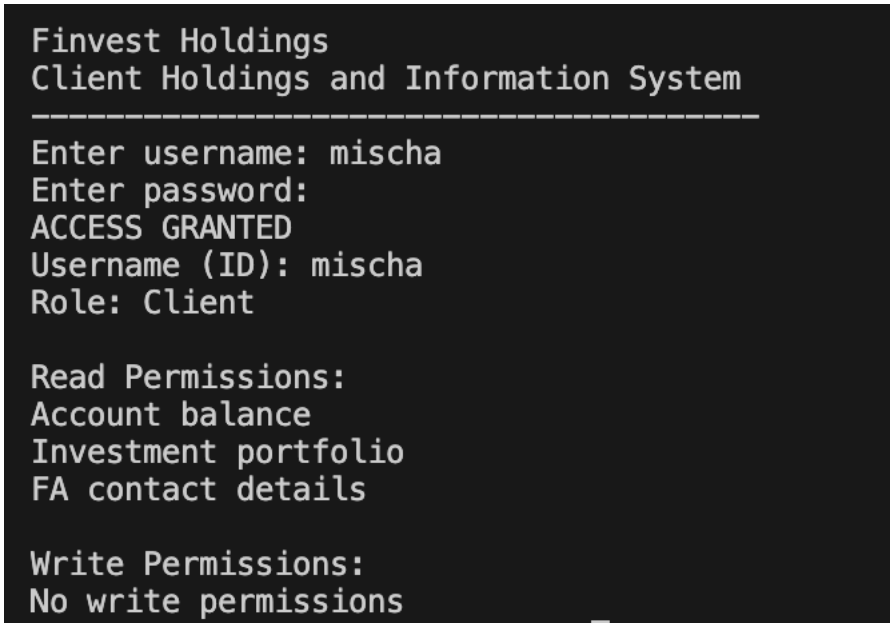
To test the password checker, I created unit tests that check various passwords, ensuring that each of them touch every possible edge of the checkPswd() method. This means that I make sure each of the password requirements described in the assignment are tested, including checking for license plate and calendar formats. I also ensured that passwords matching the proper format can be inputted. These unit tests can be found in PasswordCheckerTests.py and can be run by running the file itself, or by running UnitTests.py as a test suite.



```
PasswordCheckerTests.py > runTests
      You, 2 hours ago | 1 author (You)
1     import PasswordChecker
2
3     def testUsernameEqPassword():
4         print('Running Username Equals Password tests')
5         checker = PasswordChecker.PasswordChecker()
6         assert(checker.checkPswd("username", "username") == False)
7         print('Passing Username Equals Password tests\n')
8
9     def testPasswordOnList():
10        print('Running Password on Blacklist tests')
11        checker = PasswordChecker.PasswordChecker()
12        assert(checker.checkPswd("username", "Password1!") == False)
13        print('Passing Password on Blacklist tests\n')
14
15    def testNoPassword():
16        print('Running Password too Short tests')
17        checker = PasswordChecker.PasswordChecker()
18        assert(checker.checkPswd("username", "") == False)
19        print('Passing Password too Short tests\n')
20
21    def testPasswordTooShort():
22        print('Running Password too Short tests')
23        checker = PasswordChecker.PasswordChecker()
24        assert(checker.checkPswd("username", "Pas1!") == False)
25        print('Passing Password too Short tests\n')
26
27    def testPasswordTooLong():
28        print('Running Password too Long tests')
29        checker = PasswordChecker.PasswordChecker()
30        assert(checker.checkPswd("username", "Passssssssssssss1!") == False)
31        print('Passing Password too Long tests\n')
32
```

Problem 4

a) The user interface that was described in Problem 2 c and Problem 3 a remains the same. The only thing that changes is after a User has been verified, it prints out the read and write permissions that a user has. This is delved into deeper in Problem 4 c. Below is what is printed out upon User validation.

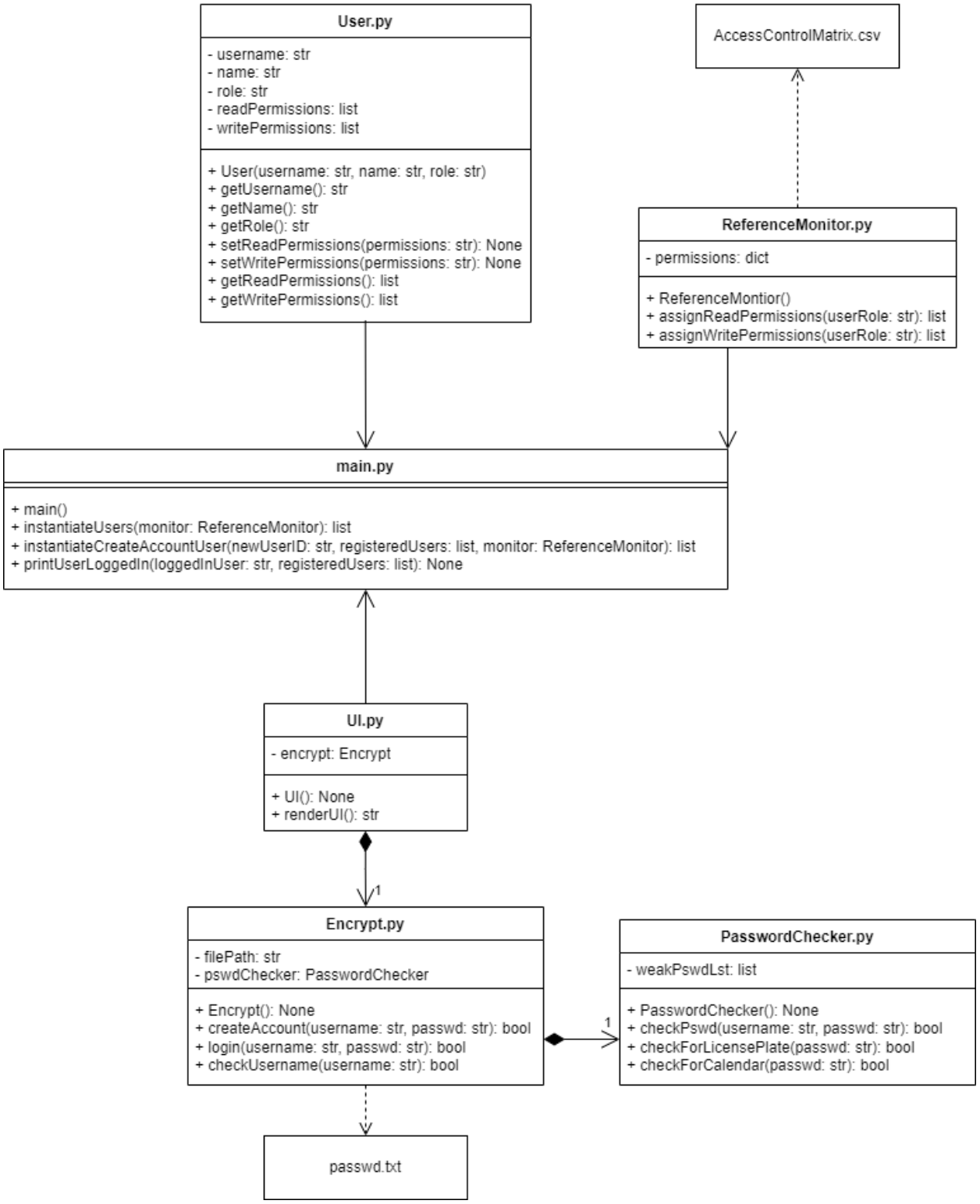


b) The password verification mechanism was developed as part of the Encrypt class. Since the Encrypt class is the only class that has access to the passwd.txt file, I left it that way, keeping only one entry point, diminishing any security vulnerabilities coming from multiple access points on the passwd.txt file. It is implemented in one method in Encrypt that is called by the UI class, passing the username and password pair. The method is called login(), which is very fitting for what is happening. Firstly, it opens the passwd.txt file and reads it line by line. On each line, it splits what is related to that account into three different elements that are stored in a local variable. That variable is then only referenced once in that iteration of the loop, checking that the username exists in the file. If the username exists, the plain text salt from that account is appended to the password that was passed from the User through the UI to the login method. That new password is then hashed and compared to the hash value that is stored in the passwd.txt file. If they match, the User is logged in, otherwise the 'ACCESS DENIED' message mentioned in Problem 3 a is displayed. The login method returns a three-digit tuple including a Boolean, true if the User was authenticated and false otherwise. It contains another Boolean indicating if the username exists. This is what's used for displaying the 'ACCESS DENIED' message instead of the 'CREATE ACCOUNT' message. Lastly, it contains the username that was passed in by the UI. If the username password doesn't match, it returns an empty string, indicating the User must create an account. The reason it returns the username that was passed and not the username that is contained in the file is to protect against injection attacks that attempt to get a valid username from the passwd.txt file.

```
20
21      def login(self, username, passwd) -> bool:
22          file = open(self.__filePath, "r")
23          usernameExists = False
24          for acc in file:
25              acc = acc.split(" ")
26              acc[2] = acc[2].replace("\n", "")
27              if (acc[0] == username):
28                  usernameExists = True
29                  passwd = acc[1] + passwd
30                  hashd = hashlib.sha512(passwd.encode())
31                  if (hashd.hexdigest() == acc[2]):
32                      return (True, usernameExists, username)
33          return (False, usernameExists, "")
34
```

c) To enforce the access control method, we need to show a depiction of the entire program. Throughout this report I have been showing different parts as they have been mentioned, but in this part, I will bring together all the parts to display the entire UML diagram of the system.

Finvest Holdings Security Program



To enforce the access control mechanism that was described in Problem 1, the bottom half of the UML diagram and the top half of the UML diagram must work together. The bottom half enforces all password related issues such as password verification and encryption. It also handles what the User sees via the UI. The top half of the UML diagram handles anything regarding the access control mechanism such as User accounts and permissions. Both halves are tightly coupled to one another, keeping similar responsibilities relatively accessible to either side relative to the other side. Furthermore, it keeps the responsibilities contained in the AccessControlMatrix and the passwords contained in the passwd.txt file far away from each other so that an attacker must go farther into the program to get both files. Additionally, if an attacker gets both files, it will be much harder to identify how they connect to each other via a single User.

As previously discussed in Problem 4 b, the login() method returns a three-digit tuple containing if the username and password match, if the username alone exists, and the username that was given by the User to the UI. The UI then returns the username or empty string that was passed back from the login() method in the Encrypt class to a variable stored in main.py called loggedInUser. This is when both sides of the UML connect. The loggedInUser variable is then passed to a method called instantiateCreateAccountUser() along with a reference to the ReferenceMonitor that was described in Problem 1 d. This method handles a User instance its permissions since permissions are stored within the User instance itself. If a User does not exist in the list of preloaded Users called registeredUsers, a new User is instantiated with the username returned by the UI and the 'Client' role permissions. This is because the 'Client' role has the lowest number of permissions in the AccessControlMatrix. This enforces the use of the access control mechanism onto the User, setting its read and writer permissions based on what is stored in the AccessControlMatrix.

```
117  def main():
118      monitor = ReferenceMonitor.ReferenceMonitor()
119      ui = UI.UI()
120
121      registeredUsers = instantiateUsers(monitor)
122
123      loggedInUser = ui.renderUI()
124
125      registeredUsers = instantiateCreateAccountUser(loggedInUser, registeredUsers, monitor)
126
127      printUserLoggedIn(loggedInUser, registeredUsers)
128
129  main()
```

```
33  def setUserPermissions(newUserID, registeredUsers, monitor) -> list:
34      userFound = False
35      for i in registeredUsers:
36          if (newUserID == i.getUsername()):
37              userFound = True
38              i.setReadPermissions(monitor.assignReadPermission(i.getRole()))
39              i.setWritePermissions(monitor.assignWritePermission(i.getRole()))
40      if (not userFound and newUserID != ""):
41          newUser = User.User(newUserID, newUserID, "Client")
42          registeredUsers.append(newUser)
43          newUser.setReadPermissions(monitor.assignReadPermission(newUser.getRole()))
44          newUser.setWritePermissions(monitor.assignWritePermission(newUser.getRole()))
45      return registeredUsers
```

Since every User has been instantiated either manually or by creating an account, I needed to display User information to that User once they have been logged in and authenticated. Up until this point, the User could not login and see what their specific read and write permissions are. The printUserLoggedIn() method in main.py prints out User information such as username, role, read, and write permissions. This is where the UI changes slightly for this implementation, resulting in the final version of the UI. The only information that is displayed differently based on role is that of a 'Teller'. If the role of a User is 'Teller', the system notifies them that they can only have access to the system from 9am to 5pm.

```
97  def printUserLoggedIn(loggedInUser, registeredUsers) -> None:
98      for user in registeredUsers:
99          if (loggedInUser == user.getUsername()):
100             print("Username (ID): " + user.getUsername())
101             print("Role: " + user.getRole())
102             if (user.getRole() == "Teller"):
103                 print("As a Teller, you can only access this account from 9am to 5pm")
104             print("\nRead Permissions:")
105             if (user.getReadPermissions() == []):
106                 print("No read permissions")
107             else:
108                 for i in user.getReadPermissions():
109                     print(i)
110             print("\nWrite Permissions:")
111             if (user.getWritePermissions() == []):
112                 print("No write permissions")
113             else:
114                 for j in user.getWritePermissions():
115                     print(j)
116
```

```
Finvest Holdings
Client Holdings and Information System
-------------------------------------
Enter username: mischa
Enter password:
ACCESS GRANTED
Username (ID): mischa
Role: Client

Read Permissions:
Account balance
Investment portfolio
FA contact details

Write Permissions:
No write permissions
```

d) To test the user login capabilities, I created unit tests that test if an existing User can login, if a User inputs the wrong password, and if a User doesn't exist. These tests can be found in the LoginTest.py file and can be run by running the file itself or by running UnitTests.py to have it run in a test suite.

LoginTests.py > ...

```python
testUser = "test"
testPassword = "ILoveCats2!"

def testLogin():
    print('Running Login tests')
    encrypt = Encrypt.Encrypt()
    encrypt.createAccount(testUser, testPassword)
    accessGranted, usernameExists, username = encrypt.login(testUser, testPassword)
    assert(accessGranted == True)
    assert(usernameExists == True)
    assert(username == testUser)
    print('Passing Login tests\n')

def testWrongPassword():
    print('Running Wrong Password tests')
    encrypt = Encrypt.Encrypt()
    accessGranted, usernameExists, username = encrypt.login('mischa', 'Ilvoecats2!')
    assert(accessGranted == False)
    assert(usernameExists == True)
    assert(username == "")
    print('Passing Wrong Password tests\n')

def testUserNotExist():
    print('Running Username Doesnt Exist tests')
    encrypt = Encrypt.Encrypt()
    accessGranted, usernameExists, username = encrypt.login('nathan', 'Ilvoecats2!')
    assert(accessGranted == False)
    assert(usernameExists == False)
    assert(username == "")
    print('Passing Username Doesnt Exist tests\n')
```

To test the access control enforcement, I created unit tests that ensure proper permissions are set to a User. I also made tests that cover if a User is new as well if a User doesn't exist. These, like all other tests in the unit tests that I developed cover all edge cases in the code. These test cases can be found in the AccesscontrolEnforcementTests.py file and can be run by running the file itself, or by running it in a test suite by running UnitTests.py.

AccessControlEnforcementTests.py > ...

```
You, 44 minutes ago | 1 author (You)
import ReferenceMonitor
import User
import Encrypt

testUser = "test"
testPassword = "IloveCats2!"
      You, 44 minutes ago • Finished tests and ensured access control enforce…
def testEnsureUserHasPermissions():
    print('Running Ensure User Has Permissions tests')
    encrypt = Encrypt.Encrypt()
    monitor = ReferenceMonitor.ReferenceMonitor()
    user = User.User('mischa', testUser, 'Client')
    accessGranted, usernameExists, username = encrypt.login('mischa', testPassword)
    assert(user.getUsername() == username)
    if (accessGranted and usernameExists):
        user.setReadPermissions(monitor.assignReadPermission(user.getRole()))
        user.setWritePermissions(monitor.assignWritePermission(user.getRole()))
    assert(user.getReadPermissions() == ['Account balance', 'Investment portfolio', 'FA contact details'])
    assert(user.getWritePermissions() == [])
    print('Passing Ensure User Has Permissions tests\n')

def testEnsureNewUserHasPermissions():
    print('Running Ensure New User Has Permissions tests')
    encrypt = Encrypt.Encrypt()
    monitor = ReferenceMonitor.ReferenceMonitor()
    user = User.User(testUser, testUser, 'Client')
    encrypt.createAccount(testUser, testPassword)
    accessGranted, usernameExists, username = encrypt.login(testUser, testPassword)
    assert(user.getUsername() == username)
    if (accessGranted and usernameExists):
        user.setReadPermissions(monitor.assignReadPermission(user.getRole()))
        user.setWritePermissions(monitor.assignWritePermission(user.getRole()))
```

Problem 5

This summary contains a detailed explanation of the Finvest Holdings Security System that has been developed. It delves into the requirements that were laid out in the contractual obligations including a password checker during sign and encrypted password file storage. It also contains details regarding the client's requirements and constraints as well as instructions on how to run the program itself.

The security system was designed using a role-based access control (RBAC) mechanism to ensure that user permissions are enforces and no users can escalate their permissions manually. This also prevents attackers from escalating permissions on a potential burner account, validating the security of the software. To implement this RBAC model, an access control matrix was established containing all the read and write permissions associated with a given role. These permissions are then attached to a user and displayed upon successful login. To ensure security, only a reference monitor can read from the access control matrix. It then passes those permissions to appropriate users if requested.

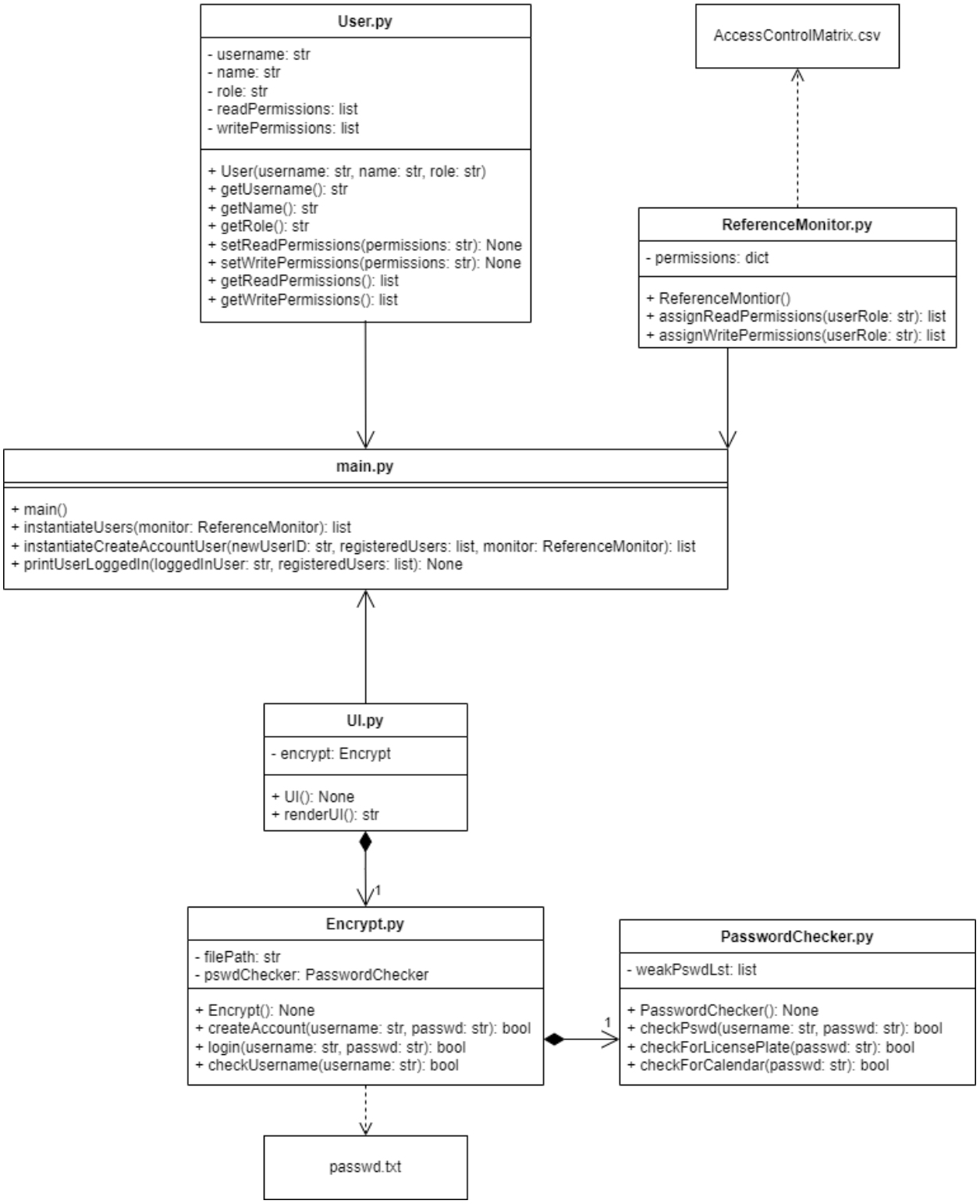
The security system contains a password file that contains all the encrypted passwords that are linked to a given username. It is structured such that each file line is a user, and each line contains a username, salt, and encrypted password. The salt is stored as it is used to reference the encrypted password by checking if the salt concatenated to the password result in the same encryption. This is how a user is authenticated. To ensure security, only a specific file called Encrypt will be able to access the password file.

The security system was designed with the ability to enroll users into the system. It does this by taking a username and password pair from the user interface (UI) and verifying that the username is not already attached to an existing user in the password file. If not, the password is then encrypted with a salt and saved in the password file with the username and the salt it was encrypted with. This new user also has the 'Client' role permissions as it is the lowest permissions that can be given to a user. During this enrolment process, the password needs to meet the specific requirements as laid out in the contractual obligations and will deny the user enrolment if the password does not meet those requirements. If a user is declined enrolment, the UI will display the password requirements that need to be met to be enrolled.

The security system also needs to have the ability for existing users to log in. The UI does this by requesting a username and password pair from the user and passes it into the system. It checks if the username and password pair exist in the password file and displays 'ACCESS GRANTED' if the pair was successful and 'ACCESS DENIED' otherwise. If the username is not in the password file, the user is prompted to create an account during the enrolment process.

The system is laid out such that there are two distinct parts of the program. A part that accesses the access control matrix and gives permissions to specific users, as well as a part that handles account storing and encryption within the password file. Both the access control matrix and password file have one entry point and cannot be entered any other way. This is to ensure that each file is kept as separate as possible and is very difficult to intertwine. The complete UML diagram is displayed below.

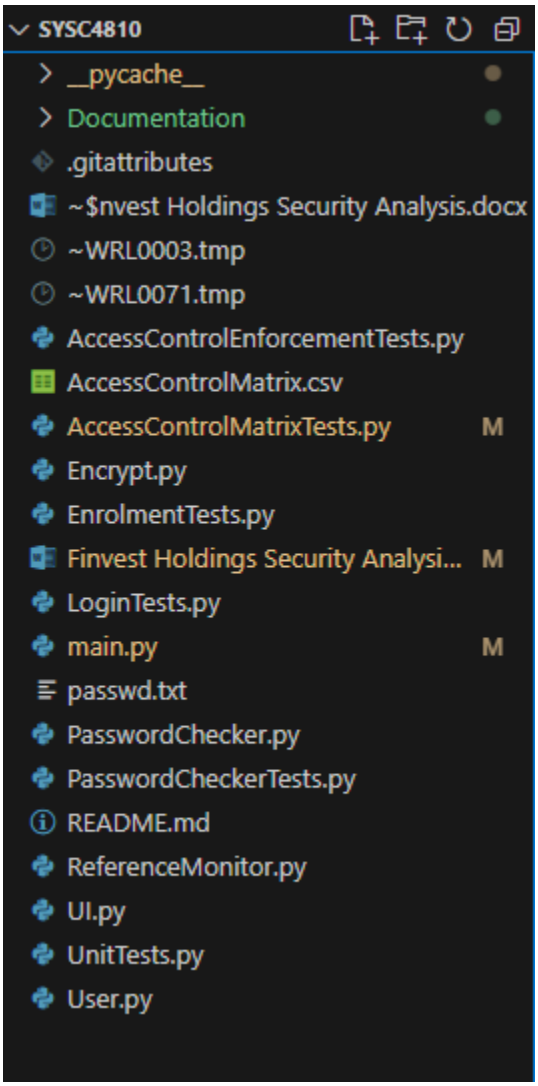
Finvest Holdings Security Program



The security system also contains extensive unit testing that ensures all the systems features described above are running correctly and security is kept top of mind. These unit tests can be found in the same folder as the program files themselves.

This system shall be run on the most recent version of Windows or MacOS to ensure security coverage within the machine itself. It shall also be constrained to internal use only except for the enroll and login UI being displayed to existing and potential new users to register as ‘Clients’.

To run this system, ensure that all files are in the same directory. A list of all the files is listed below.



Once all folders are in the same directory, run main.py. This will begin the UI. There are many accounts that are already loaded into the password file as examples. A list of the accounts can be found in the main.py file and all their passwords are ‘IloveCats2!’. As an example, to login, use the username ‘mischa’ and the password ‘IloveCats2!’. This will login you in as Mischa Lowery, a test user. It will then display ‘ACCESS GRANTED’ as well as the list of read and write permissions that Mischa has.

If you want to create an account, simply type in an incorrect username and password combination, this will bring up the enrolment page. You can then choose any username and password combination that you would like only if the username is unique, and the password meets the requirements. After the enrolment is complete, rerun the main.py file and you can login as your created user with the ‘Client’ permissions added to your account.

There are also unit tests that can be run to ensure that the program is running correctly. You can run all the tests at once by running UnitTests.py or run each of them individually by running any file with the naming format XYZTests.py. Both options will display a test suite output in the terminal, depicting what test cases have passed and what have failed.

References

1. Python (November 23rd, 2023) Hashlib Library (3.12) [Python Library API] https://docs.python.org/3/library/hashlib.html
2. Python (November 23rd, 2023) Getpass Library (3.12) [Python Library API] https://docs.python.org/3/library/getpass.html
3. Charmaine Jacob CJ. “Most overused passwords in the world – make sure yours isn’t on the list”, CNBC, November 15th, 2023.