Finvest Holdings

Security Software & Analysis

SYSC 4810 Assignment

Nathan MacDiarmid

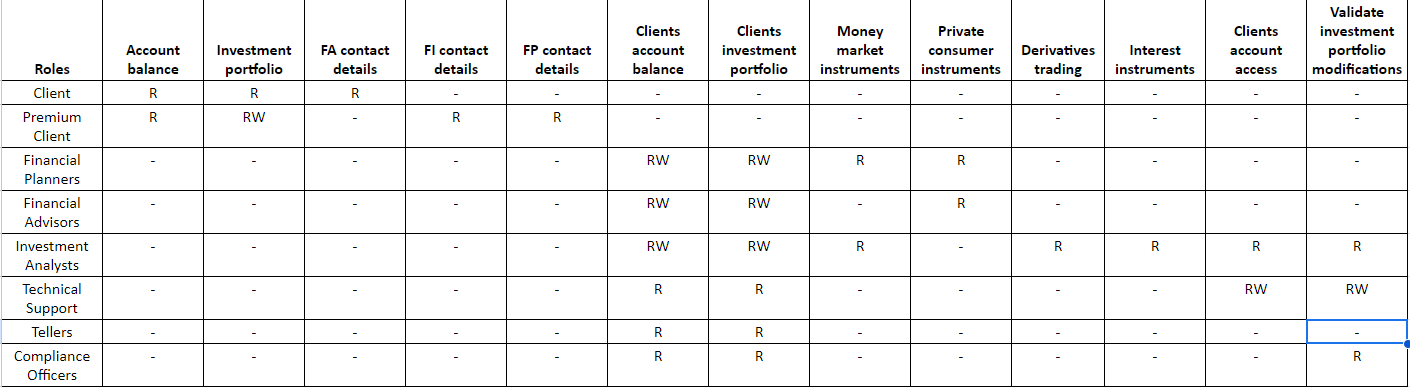
101098993

Passwords – IloveCats2!

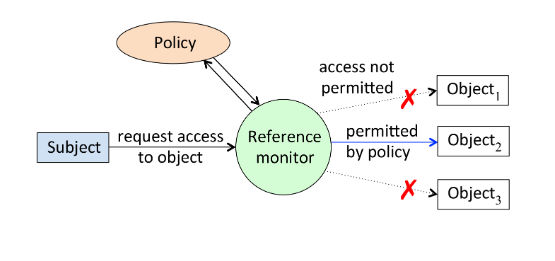
December 4th, 2023

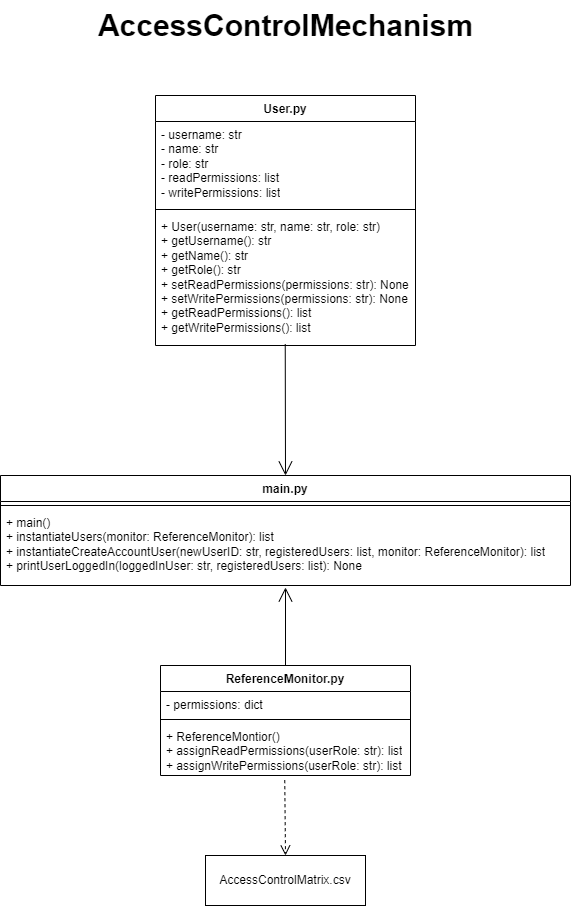
# Problem 1

1. 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 is similar to 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.
2. 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.
3. 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 the times they are allowed to access the program.

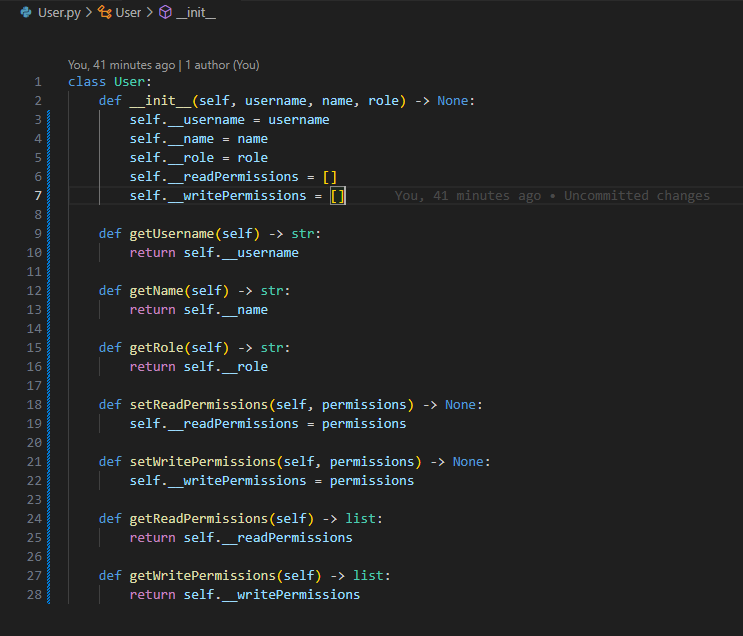


1. The access control mechanism was implemented in an object-oriented way such that it models the example of a reference model implementation that was given in class in slides Ch5\_2[3]. The example given in class and a UML diagram of the access control mechanism is depicted below.





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.



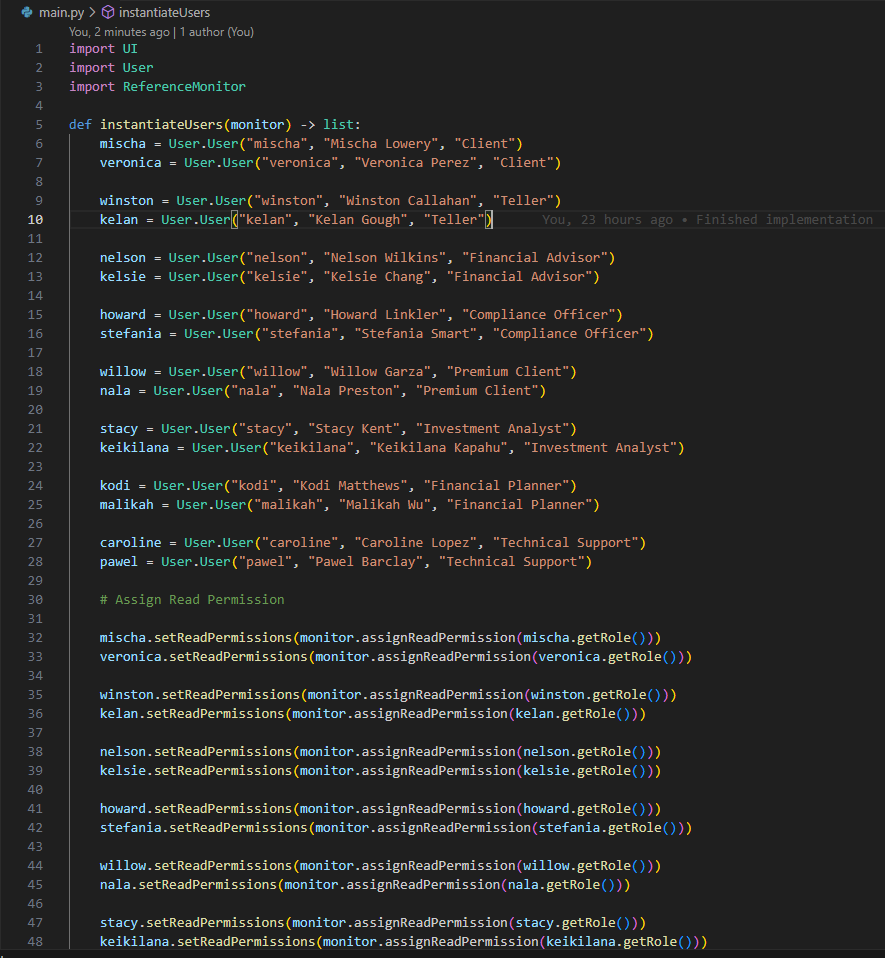
The Reference Monitor in the example controls what Subjects get access to what permissions based on a Policy. In my access control mechanism, the ReferenceMonitor reads from the AccessControlMatrix file and stores all the permissions in a private dictionary attribute called permissions. The ReferenceMonitor 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.





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.

1. Testing here

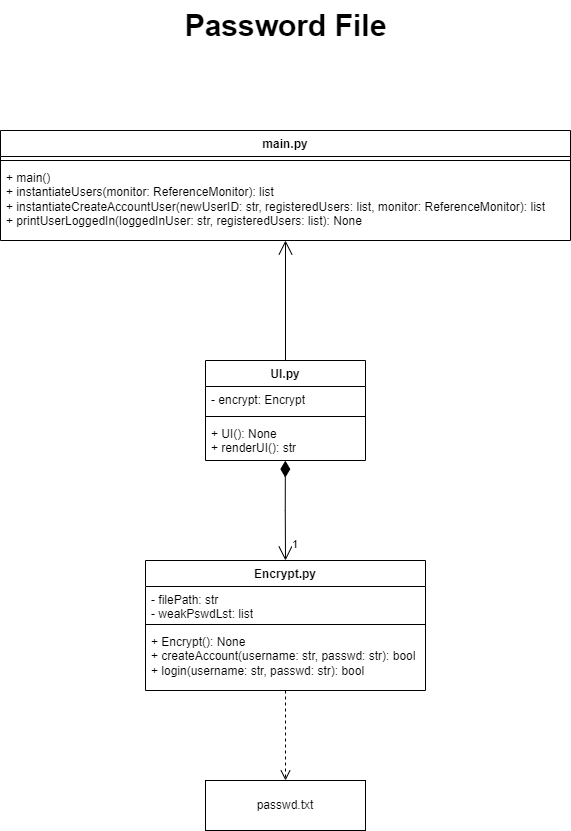
# Problem 2

1. Since I used Python, there is a library called hashlib [2] that can be imported from the Python API. 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 un-hashable 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.
2. 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 of the usernames are the names listed and all of their passwords are ‘IloveCats2!’. As you can see, all of 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.



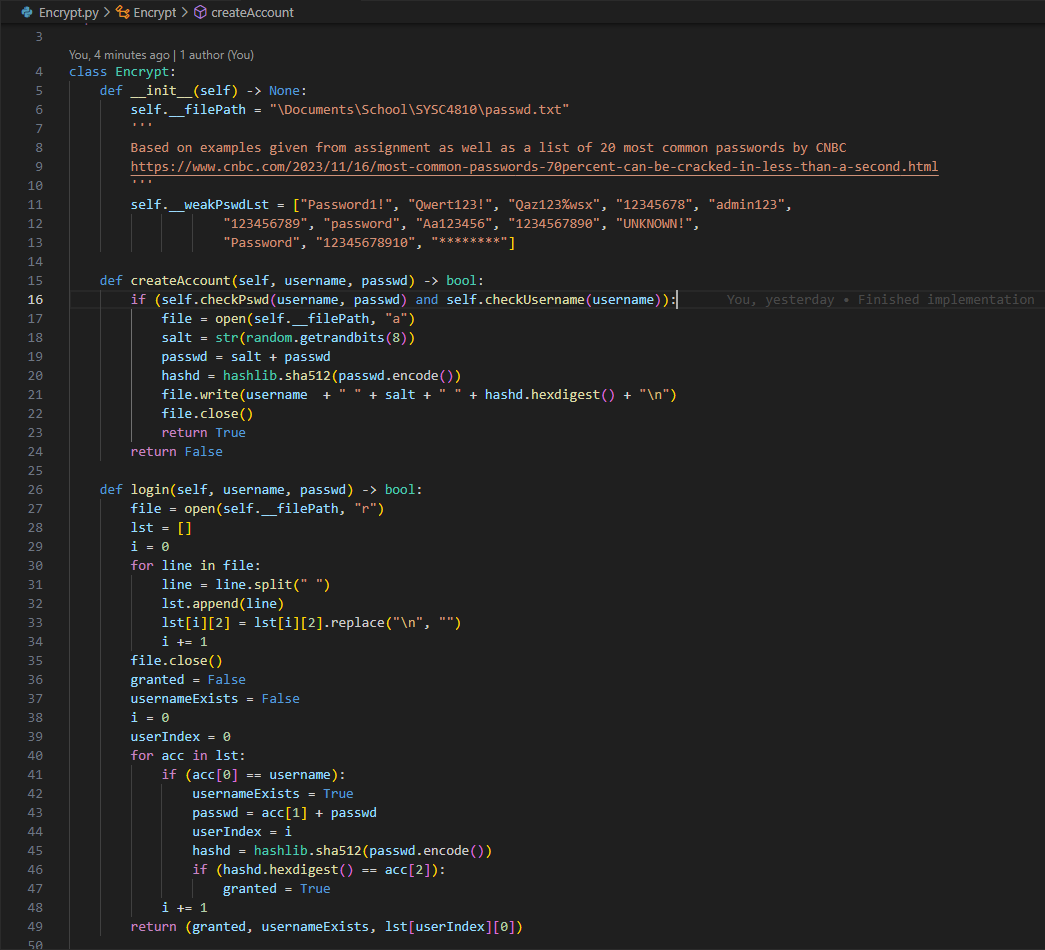
1. The password file is where a generic user interface began. Since Problem 4 contains instructions on designing a user interface to log Users in and this problem needed abilities to get and store passwords with other information, I decided to implement a basic UI so that I could pass my own information 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 UML diagram of password file portion of this assignment is depicted below.



The passwd.txt file is only accessible by an Encrypt class. This is similar to 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.

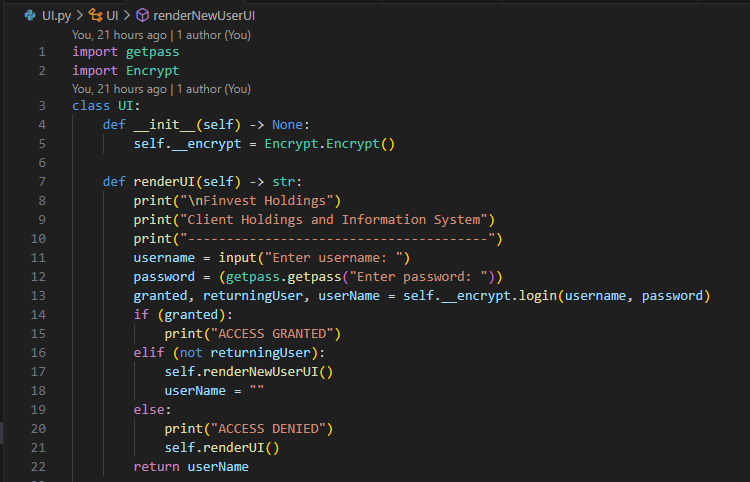
The Encrypt class also contains a createAccount() method as well as a login() method that allows for new information to be passed to the passwd.txt file as well as previously stored information to be retrieved. The createAccount() stores information in the passwd.txt file by taking in a username and password, generating a random salt value from 0 -> , 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, similar to 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.



1. Testing here

# Problem 3

1. 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 ‘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. The Python library getpass contains no known security flaws and does not store input anywhere.

A black screen with white text

Description automatically generated

1. The proactive password checker is implemented as part of the Encrypt class.

# References

1. <https://www.linkedin.com/pulse/comprehensive-overview-access-control-models-rbac-abac-jay-/>
2. <https://docs.python.org/3/library/hashlib.html>
3. <https://docs.python.org/3/library/getpass.html>
4. https://www.cnbc.com/2023/11/16/most-common-passwords-70percent-can-be-cracked-in-less-than-a-second.html