Assignment 1: Finvest Holdings

SYSC 4810: Introduction to Network and Software Security

by
Juanita Rodelo
101141857

Table of Contents

Problem 1: Access Control Mechanism

1.a) Access Control Model

I have chosen to use the Role-based Access Control (RBAC) method because it allows for different capabilities to be assigned to specific roles which provides us with the flexibility, control and security over access requirements. Given the requirements of Finvest Holdings, it is evident that the access control policy is determined by the role of the employee/client. For example, the systems states that regular clients have the capability to view their account balance, view their investment portfolio, and view the contact details of their financial advisor. In this case, the client is the role, and their permissions include viewing their account balance, investment portfolio and contact details of their financial advisor. There are other roles that have some of these capabilities like premium clients but the roles that do not have these permissions will not be able to access these resources, making it a secure option. The requirements explicitly define these roles which can be easily implemented with RBAC.
RBAC allows you to define roles that are assigned different access levels and operations.

Other access control options like the Discretionary Access Control (DAC) offer this flexibility but it does not offer secure data protection as any user can share their data with whoever they want, which can be a major security breach for a financial management company [1]. There is also the Mandatory Access Control (MAC) method that is highly secure, by following a zero-trust principle but is more difficult and complex to configure and requires constant maintenance from administrators, making it difficult to scaler [2]. This method is typically used by the government or military where security is number one priority and they require the highest levels of confidentially. Security is important for Finvest Holdings but so is performance and MAC systems require a high degree of administrative oversight which would be problematic and not suitable for this context.

There are other access control methods that provide similar flexibility like the Attribute-Based Access Control (ABAC) method which considers the role of the user, their attributes and the environment [1]. This would also be a good option to use but it is more complex and requires more effort setting up whereas RBAC is generally simpler and more straightforward. ABAC might be considered in situations where fine-grained access control is essential but for this system, I believe it is sufficient and best to choose RBAC. RBAC is also more scalable than ABAC since as the number of roles and resources increase, there is no need to maintain attribute-based policies for each user and resource combination [3]. For these reasons, RBAC provides the desired balance between security and performance that Finvest Holdings is looking for.

1.b) Access Control Representation

I have chosen to use a capabilities list to represent the RBAC method because it is resource-centric and focuses on what actions or operations a user is allowed to perform

on an object [4]. This provides fine-grained access control over roles and their actions on resources. Since the access control permissions were listed out for each type of user, this makes migrating this information into a capabilities list easy to implement. It provides the flexibility to add or remove capabilities with minimal overhead.

There are other options that would also work with this system such as an access control matrix that holds the capabilities of users in a matrix style data structure where rows correspond to subjects and columns correspond to objects and their entries specify the access rights [4]. While these offer a comprehensive view of the access rights, it can become large and hard to maintain as the number of roles and resources increase. Another method that is similar is an access control list (ACL) which is exactly the opposite of a capabilities list. It is a permission-centric list that is associates an object the permissions for each subject. While these are simple and easy to understand, it is more difficult to maintain when many resources exist.

A capability list allow for a fine-grained and dynamic access control and aligns with the least privilege principle by ensuring that entities only have the permissions necessary to perform their specific tasks [5]. This helps minimize potential vulnerabilities, security breaches and unauthorized access. For these reasons, I believe a capabilities list is the simplest, securest and easiest way to setup and maintain the access control mechanism required for Finvest Holdings.

1.c) Access Control Design

The following is a design of my capabilities list, it lists the permissions for each resource for each role. I chose to have the following permissions based on the requirements of the system: READ, WRITE, ACCESS. The READ permissions are associated with “viewing” a resource as the requirements state. The WRITE permissions are associating with modifying resources as stated in the requirements. The ACCESS permissions are associated with the “accessing” the system or having client account “access” as stated in the requirements. Since it was not made clear what access meant in this context, I thought it was best to define its own capability.

```
===== Capabilities List =====

Role: CLIENT
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ]
    - FA_CONTACT_DETAILS: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: PREMIUM_CLIENT
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ, WRITE]
    - FA_CONTACT_DETAILS: [READ]
    - IA_CONTACT_DETAILS: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: FINANCIAL_PLANNER
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ, WRITE]
    - MONEY_MARKET_INST: [READ]
    - PRIV_CONS_INST: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: FINANCIAL_ADVISOR
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ, WRITE]
    - PRIV_CONS_INST: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: INVESTMENT_ANALYST
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ, WRITE]
    - MONEY_MARKET_INST: [READ]
    - DERIVATIVES_TRADING: [READ]
    - INTEREST_INST: [READ]
    - PRIV_CONS_INST: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: TECHNICAL_SUPPORT
    - ACCOUNT_BALANCE: [READ]
    - INVESTMENT_PORTFOLIO: [READ]
    - CLIENT_ACCOUNT_ACCESS: [ACCESS]
    - SYSTEM_OFF_HOURS: [ACCESS]
    - SYSTEM_ON_HOURS: [ACCESS]

Role: TELLER
    - SYSTEM_ON_HOURS: [ACCESS]

Role: COMPLIANCE_OFFICER
    - INVESTMENT_PORTFOLIO: [READ]
    - SYSTEM_OFF_HOURS: [ACCESS]
```

1.d) Access Control Implementation

I chose to implement this access control method by using Enums to define Roles, Resources, and Permissions. The Role Enum has predefined number values that makes it easy to later store in a password file. For example, the regular Client role is associated with the number 1, Premium Client roles are associated with the number 2, Financial Planners are associated with the number 3 and so on. This provides a clear, readable, and extendible way to represent these objects. The Resource and Permissions Enums do not need a number associated with them, so I used the auto-assignment for these values to eliminate the need to manually assign values. This makes it simple to add new resources and permissions without maintaining unique values for them. In terms of security, Enums are strongly typed, which means that once they are defined a certain type, they cannot be changed later [6]. It is type safe which means that operations with mismatched types are not permitted which helps in catching errors at compile-time rather than runtime. They are an organized and secure way of defining and storing the definitions of these objects. The following is my implementation of these Enum:

```python
class Role(Enum):
    """Role defines all roles (employees and clients) in Finvest Holding"""
    CLIENT = 1
    PREMIUM_CLIENT = 2
    FINANCIAL_PLANNER = 3
    FINANCIAL_ADVISOR = 4
    INVESTMENT_ANALYST = 5
    TECHNICAL_SUPPORT = 6
    TELLER = 7
    COMPLIANCE_OFFICER = 8

class Resource(Enum):
    """Resources defines all resources that have access rights in Finvest Holdings"""
    ACCOUNT_BALANCE = auto()
    INVESTMENT_PORTFOLIO = auto()
    FA_CONTACT_DETAILS = auto()
    IA_CONTACT_DETAILS = auto()
    MONEY_MARKET_INST = auto()
    PRIV_CONS_INST = auto()
    DERIVATIVES_TRADING = auto()
    INTEREST_INST = auto()
    CLIENT_ACCOUNT_ACCESS = auto()
    SYSTEM_OFF_HOURS = auto()
    SYSTEM_ON_HOURS = auto()

class Permission(Enum):
    """Permissions define permissions to resources in Finvest Holdings"""
    READ = auto()
    WRITE = auto()
    ACCESS = auto()
```

Then, I chose to create an AccessControl class that holds the capabilities list and the interactions with it. I implemented the capabilities list with a dictionary that maps roles to resources and the associated permissions. I used a dictionary because it is an organized way to reflect the relationship between roles and their permissions. They provide a constant time complexity for the average case for lookups, meaning it can quickly retrieve permissions for resources for a given role. This is crucial when integrated in a real time system that users will use to view their information. They also allow for dynamic updates and modifications so you can easily add and remove entries without significant code changes. This flexibility is important in the case that the company wants to add a new role, resource or permissions to their system. The following screenshot shows how the capabilities list is implemented.

```python
class AccessControl:
    """AccessControl holds the capabilities list and functions that grab information from it"""

    def __init__(self):
        self.capabilities_list = {
            Role.CLIENT: {
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ],
                Resource.FA_CONTACT_DETAILS: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.PREMIUM_CLIENT: {
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ, Permission.WRITE],
                Resource.FA_CONTACT_DETAILS: [Permission.READ],
                Resource.IA_CONTACT_DETAILS: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.FINANCIAL_PLANNER: {
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ, Permission.WRITE],
                Resource.MONEY_MARKET_INST: [Permission.READ],
                Resource.PRIV_CONS_INST: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.FINANCIAL_ADVISOR: {
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ, Permission.WRITE],
                Resource.PRIV_CONS_INST: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.INVESTMENT_ANALYST: {
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ, Permission.WRITE],
                Resource.MONEY_MARKET_INST: [Permission.READ],
                Resource.DERIVATIVES_TRADING: [Permission.READ],
                Resource.INTEREST_INST: [Permission.READ],
                Resource.PRIV_CONS_INST: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.TECHNICAL_SUPPORT:{
                Resource.ACCOUNT_BALANCE: [Permission.READ],
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ],
                Resource.CLIENT_ACCOUNT_ACCESS: [Permission.ACCESS],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS],
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.TELLER: {
                Resource.SYSTEM_ON_HOURS: [Permission.ACCESS]
            },
            Role.COMPLIANCE_OFFICER: {
                Resource.INVESTMENT_PORTFOLIO: [Permission.READ],
                Resource.SYSTEM_OFF_HOURS: [Permission.ACCESS]
            }

        }
```

I defined some methods that are used in other areas of the code to grab information from the capability list given different pieces of information. The following screenshot shows the implementation of these methods. The get_role_values() method returns the values associated with every role, it is ismply a list of numbers. The get_role_name() returns the role name associated with the number value. The print_role_capabilities() method prints all the permissions associated with the given role. The method can_access() determines whether the given role has the given capability on the given

resource. All these methods are used in other areas of the program. This is all stored in a file named problem1d.py.

```
def get_role_values(self):
    """Returns the values associated with the roles [1, 2, 3, 4, 5, 6, 7, 8]"""
    return [role.value for role in Role]

def get_role_name(self, role_number):
    """Returns the item name associated with the given role number """
    for role in Role:
        if role.value == role_number:
            return role
    return None

def print_role_capabilities(self, role):
    """Prints the capabilities that the given role has on all resources."""
    if role in self.capabilities_list:
        role_capabilities = self.capabilities_list[role]
        for resource, permissions in role_capabilities.items():
            permission_names = [permission.name for permission in permissions]
            print(f"Resource: {resource.name}, Permissions: {', '.join(permission_names)}")


def can_access(self, role, resource, capability):
    """Returns true if the given role has the given capability on the given resource. Returns False otherwise."""
    if role in self.capabilities_list and resource in self.capabilities_list[role]:
        return capability in self.capabilities_list[role][resource]

    return False
```

1.e) Access Control Testing

To test the access control method, I created a unit test file named test_problem1d.py that has a test method per method in the original problem1d.py file. The first test is called test_get_role_values() that tests that get_role_values() method correctly returns all the role values associated with the role names. Since there are eight defined roles the values that are returned should be [1, 2, 3, 4, 5, 6, 7, 8] so I simply check for this using an assertEqual().

```
def test_get_role_values(self):
    """Tests get_role_values()"""
    role_values = self.access_control.get_role_values()
    self.assertEqual(role_values, [1, 2, 3, 4, 5, 6, 7, 8])
```

The next test method is called test_get_role_name() that tests the original get_role_name() method that should return the role name associated with the given role value, which is a number from 1 to 8. Since the password file holds only the number associated with the role, this method was needed to grab the role name rather than just the number. I tested this by providing three sample role numbers that exist and I check that they return the correct role name. I also test an invalid role number and the method should return None.

```
def test_get_role_name(self):
    """Tests get_role_name() with different valid role numbers"""
    role_name = self.access_control.get_role_name(1)
    self.assertEqual(role_name, Role.CLIENT)

    role_name = self.access_control.get_role_name(2)
    self.assertEqual(role_name, Role.PREMIUM_CLIENT)

    role_name = self.access_control.get_role_name(7)
    self.assertEqual(role_name, Role.TELLER)

    role_name = self.access_control.get_role_name(99)
    self.assertIsNone(role_name)
```

The next test method is called test_print_role_capabilities() that tests the original print_role_capabilites() method. Since the original method prints the permissions for each resource of the given role, I redirect the output to the test so I can compare what is printed to what should be printed. I test this with the regular Client role. For the sake of keeping the test files concise I only included one role, but I tested this with each role to ensure that it printed the correct permissions for each resource.

```
def test_print_role_capabilities(self):
    """Tests print_role_capabilities()"""
    #redirect stdout to capture the print output
    captured_output = StringIO()
    sys.stdout = captured_output

    self.access_control.print_role_capabilities(Role.CLIENT)

    #reset redirect.
    sys.stdout = sys.__stdout__

    expected_output = "Resource: ACCOUNT_BALANCE, Permissions: READ\n" \
                      "Resource: INVESTMENT_PORTFOLIO, Permissions: READ\n" \
                      "Resource: FA_CONTACT_DETAILS, Permissions: READ\n" \
                      "Resource: SYSTEM_OFF_HOURS, Permissions: ACCESS\n" \
                      "Resource: SYSTEM_ON_HOURS, Permissions: ACCESS\n"
    self.assertEqual(captured_output.getvalue(), expected_output)
```

The next test method is called test_can_access() that tests the original can_access() method. For each role I provide a true condition and a false condition to check that the method correctly identifies when the given role does not have the given permission for the given resource. For the sake of keeping the test files not too long I kept these methods, but I tested each role for each type of permission for each resource to ensure that they are allowed the correct permissions for the resources.

```
def test_can_access(self):
    """Tests can_access() with true and false conditions for each role"""
    #true conditions
    result = self.access_control.can_access(Role.CLIENT, Resource.ACCOUNT_BALANCE, Permission.READ)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.PREMIUM_CLIENT, Resource.INVESTMENT_PORTFOLIO,Permission.WRITE)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.FINANCIAL_PLANNER, Resource.MONEY_MARKET_INST,Permission.READ)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.FINANCIAL_ADVISOR, Resource.PRIV_CONS_INST,Permission.READ)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.INVESTMENT_ANALYST, Resource.INTEREST_INST, Permission.READ)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.TECHNICAL_SUPPORT, Resource.CLIENT_ACCOUNT_ACCESS, Permission.ACCESS)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.TELLER, Resource.SYSTEM_ON_HOURS, Permission.ACCESS)
    self.assertTrue(result)

    result = self.access_control.can_access(Role.COMPLIANCE_OFFICER, Resource.INVESTMENT_PORTFOLIO, Permission.READ)
    self.assertTrue(result)

    #false conditions:
    result = self.access_control.can_access(Role.CLIENT, Resource.ACCOUNT_BALANCE, Permission.WRITE)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.PREMIUM_CLIENT, Resource.MONEY_MARKET_INST, Permission.READ)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.FINANCIAL_PLANNER, Resource.CLIENT_ACCOUNT_ACCESS,Permission.ACCESS)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.FINANCIAL_ADVISOR, Resource.PRIV_CONS_INST, Permission.WRITE)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.INVESTMENT_ANALYST, Resource.MONEY_MARKET_INST,Permission.WRITE)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.TECHNICAL_SUPPORT, Resource.INVESTMENT_PORTFOLIO,Permission.WRITE)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.TELLER, Resource.SYSTEM_OFF_HOURS, Permission.ACCESS)
    self.assertFalse(result)

    result = self.access_control.can_access(Role.COMPLIANCE_OFFICER, Resource.INVESTMENT_PORTFOLIO,Permission.WRITE)
    self.assertFalse(result)
```

Problem 2: Password File

2.a) Hash Function Selection

For the password file creation, I selected the Secure Hash Algorithm (SHA-256) hashing algorithm because it is an industry standard that is trusted and used widely for its security strength. It is part of the SHA 2 family has a 256-bit has output, providing many possible hash values, making it harder for attackers to find collisions. It uses a combination of complex mathematical and bitwise operations to generate the hash value. It follows the avalanche effect where one minor change to the input completely changes the hash value [7]. It is slower than the SHA-1 hashing algorithm, but SHA-1 has

been exposed for being vulnerable to brute force attacks. SHA-256 does not have any known vulnerabilities whereas many of the other hashing algorithms do [8].

For the salt, I chose to use the built in python secrets library that is used specifically for generating strong random numbers suitable for passwords and account authentication. It provides functions to generate tokens which can be used as salts for password files. I used the token_hex() method that returns an n-bit random text string, in hexadecimal. The default value is 16 bytes and I kept it this length because it is recommended to use a salt of at least this length. This ensures a sufficiently large space of possible salt values, making it very difficult for attackers to carry out an offline attack.

2.b) Password File Record Structure

I am using the following password file record structure:

username:role_number:plain_text_salt:hashed_password

I am keeping it simple with recording only the necessary information. The username is the unique ID that differentiates all the records in the file, so this is stored first. This facilitates the management of each individual account and is used in each operation such as logging in and displaying access rights. I implemented the access control method in a way where each role name is associated with a number so that is stored next after a colon. This number will be from 1-8 and is used to specify what access that user has with all the resources. By assigning a numeric value to each role, the system can efficiently represent and manage different user roles. The next item stored in the file is the plain text salt. This is stored in plaintext because it is going to be used when trying to hash the password that a user provides when trying to log in. This is required in protecting against rainbow attacks and ensures that the same password do not hash to the same password due to the uniqueness of the salt. The last item that is stored is the hashed password. This is also used in the login process to compare the concatenation of the salt with the password that the user provides when logging in with the one that is expected.

2.c) Password File Record Implementation

My password management file is called problem2d.py and is contained within a class called PasswordManagement. It takes an optional parameter ‘password_file’, which defaults to “passwd.txt” since this is the typical password file name. The first method is called hash_password() and it takes in a plain text password and salt as input and hashes the combination using the SHA-256 encryption algorithm. I chose to store the hashed password in hexadecimal to avoid confusion with the hashed password, salt and the colons that separate each component in the password file.

```python
import hashlib
import secrets

class PasswordManagement:

    def __init__(self, password_file = "passwd.txt"):
        self.password_file = password_file


    def hash_password(self, password, salt):
        """Hashes the given plaintext password and salt using SHA-256 encryption algorithm.
        Returns the hashed password."""
        sha256 = hashlib.sha256()
        sha256.update(password.encode() + salt.encode())
        hashed_psswd = sha256.hexdigest()
        return hashed_psswd
```

The second method is called generate_salt() and this simply returns the salt generated by the secrets library method token_hex(). I created its own method for this to keep each method only having one main functionality.

```python
def generate_salt(self):
    """Generates a random secure salt and returns it in hexadecimal format"""
    return secrets.token_hex()
```

The third method is called save_password_record() and this takes parameters for a username, role, salt and hashed password. It appends a new password record to the specified password file with the format I explained in part b of this question.

```python
def save_password_record(self, username, role, salt, hashed_password):
    """Saves the given password record parameters in password file."""
    try:
        with open(self.password_file, 'a') as file:
            record = f"{username}:{role}:{salt}:{hashed_password}\n"
            file.write(record)
    except FileNotFoundError:
        print("Password file not found. ")
    except IOError as e:
        print(f"Error writing to file {e}")
```

The next method is called get_password_record() which reads the password file and searches for the given username. If found it returns the pull password record for that username and if it does not find it, it returns None.

```python
def get_password_record(self, username):
    """Returns the full password record for the given usernamevthat is stored in the password file.
    Returns null if username does not exist in file."""
    try:
        with open(self.password_file, 'r') as file:
            for line in file:
                usernameInFile = (line.split(":"))[0].strip()
                if username == usernameInFile:
                    return line
    except FileNotFoundError:
        print("Password file not found. ")
    return None
```

The next method is called get_role_num() which retrieves the role number associated with the given username from the password file. If there is an error extracting the role number, it handles it gracefully and returns None. Proper error handling is important in this context to ensure that the system is always in an expected state, reducing vulnerabilities.

```
def get_role_num(self, username):
    """Returns the role number stored in the password file associated with the given username.
    If there was a problem extracting the role number, this returns None. """
    full_record = self.get_password_record(username)
    role_num = None
    try:
        role_num = int(full_record.split(":")[1].strip())
    except (IndexError, ValueError):
        print("Error extracting the role number from the password file. ")
    return role_num
```

The next method is called get_salt() which retrieves the salt associated with the given username from the password file. If there is an error with this extraction, the error is handled and the method will return None.

```
def get_salt(self, username):
    """Returns the salt stored in the password file associated with the given username. If there was an error
    extracting the salt, this returns None. """
    full_record = self.get_password_record(username)
    salt = None
    try:
        salt = full_record.split(":")[2].strip()
    except (IndexError, ValueError):
        print("Error extracting salt.")
```

The next method is called get_hashed_password(), which retrieves the hashed password associated with the given username from the password file. If there is an error extracting the hased password, the method safely returns None.

```
def get_hashed_password(self, username):
    """Returns the hashed password stored in the password file associated with the given username. If there was a
    problem extracting the hashed password, this returns None. """
    full_record = self.get_password_record(username)
    hashed_password = None
    try:
        hashed_password = full_record.split(":")[3].strip()
    except (IndexError, ValueError):
        print("Error extracting hashed password.")

    return hashed_password

def get_password_file(self):
    return self.password_file
```

The last method in this class is called get_password_file() and this simply returns the password file path. Overall, these methods are used in the enrollment and login processes. This class provides functionality for hashing passwords, generating salts,

saving password records into a file, and retrieving different components of the password record based on a username.

2.d) Password File Testing

To test the password management methods, I created a unit test file named test_problem2d.py that has a test method per method in the original problem2d.py file. The first thing that I did was create a test password file to store records that I will test to ensure that the real password file is working as expected. I created a test password file so I could perform these tests on something that wasn't the actual password file to ensure that nothing gets deleted or exposed that shouldn't. This test password file gets cleared at the end of the tests so that it can all be done from scratch every time the tests are run. The first test method is called test_hash_password() which tests the original hash_password() method. First, I test that the hash is not just the concatenation of the plaintext password and the plaintext salt. Then, I hash the same password with the same salt and check that they do product the same hashed password, which is used in the login process.

```
def test_hash_password(self):
    """Tests hash_password()"""

    #Test that the hash is not just a concatenation of the password and the salt
    hashed_password1 = self.password_manager.hash_password("password123", "salt123")
    self.assertNotEqual("password123" + "salt123", hashed_password1)

    #Test that the same password and hash will produce the same hashed password
    hashed_password2 = self.password_manager.hash_password("password123", "salt123")
    self.assertEqual(hashed_password1, hashed_password2)
```

The next test method is called test_generate_salt() that tests the original generate_salt() method. To ensure that a unique salt is being created each time, I simply test that two salts are not the same after they are created using the generate_salt().

```
def test_generate_salt(self):
    """Tests generate_salt() by generating different salts and checking that none of them are equal. """

    salt1 = self.password_manager.generate_salt()
    salt2 = self.password_manager.generate_salt()
    self.assertNotEqual(salt1, salt2)

    salt3 = self.password_manager.generate_salt()
    salt4 = self.password_manager.generate_salt()
    self.assertNotEqual(salt3, salt4)
```

The next test method is called test_save_password_record() which tests the original save_password_record() method. I test this by creating a test username, role, salt and hashed password and call on the original method to save this record in the test password file. I then open and read the test password file to ensure that the expected

record is there. I manually tested this functionality multiple times but in the unit test file I only included one test for this method.

```
def test_save_password_record(self):
    """Tests save_password_record() by saving a tst user name, role, salt and hashed password in a test password file
    then check the file to see if it is there. """
    username = "test_username"
    role = 1
    salt = "test_salt"
    hashed_password = "test_hashed_password"
    self.password_manager.save_password_record(username, role, salt, hashed_password)
    expected_record = f"{username}:{role}:{salt}:{hashed_password}\n"
    try:
        with open('test_passwd.txt', 'r') as file:
            for line in file:
                usernameInFile = (line.split(":"))[0].strip()
                if username == usernameInFile:
                    self.assertEqual(line, expected_record)
                    break
    except FileNotFoundError:
        print("In test_save_password_record: Password file not found. ")
```

The next test method is called test_get_password_record() which tests the original get_password_record() method. This test is similar to the previous but instead of calling the save_password_record() method to write the given record in the test password file, I write to the test password file in the test method and check that get_password_record() properly retrieves this record.

```
def test_get_password_record(self):
    """Tests get_password_record() by saving a test user name, role, salt and hashed password in a test password file
    then check the file to see if it is there."""
    username = "test_username2"
    role = 2
    salt = "test_salt2"
    hashed_password = "test_hashed_password2"

    try:
        with open('test_passwd.txt', 'a') as file:
            record = f"{username}:{role}:{salt}:{hashed_password}\n"
            file.write(record)
    except FileNotFoundError:
        print("In test_get_password_record: Password file not found. ")
    except IOError as e:
        print(f"In test_get_password_record: Error writing to file {e}")

    expected_record2 = f"{username}:{role}:{salt}:{hashed_password}\n"
    self.assertEqual(self.password_manager.get_password_record(username), expected_record2)
```

The next three methods test_get_role_num(), test_get_salt(), test_get_hashed_password() all work similarily to the test_get_password_record() but now they retrieve specific entries in the password record. The test_get_role_num() tests that the get_role_num() properly retrieves only the role number in the record. I test this by calling on that method and comparing that it returns a role number that I expect and also that it does not return one that I'm not expecting. I do the same thing for the test_get_salt() and test_get_hashed_password().

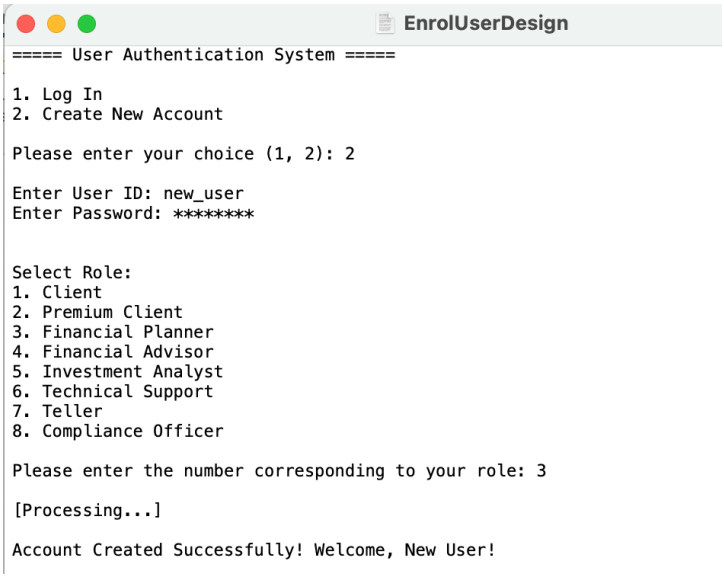
```
def test_get_role_num(self):
    """Tests get_role_num() by creating a record in the test password file and comparing the role number from that record
    with what is expected"""
    #Test getting a role that exists in the password file:
    username = "test_username3"
    role = 3
    salt = "test_salt3"
    hashed_password = "test_hashed_password3"

    try:
        with open('test_passwd.txt', 'a') as file:
            record = f"{username}:{role}:{salt}:{hashed_password}\n"
            file.write(record)
    except FileNotFoundError:
        print("In test_get_role_num: Password file not found. ")
    except IOError as e:
        print(f"In test_get_role_num: Error writing to file {e}")

    self.assertEqual(self.password_manager.get_role_num(username), 3)
    #Test getting a role that does not exists in the password file:
    self.assertNotEqual(self.password_manager.get_role_num(username), 10)

def test_get_salt(self):
    """Tests get_salt() by creating a record in the test password file and comparing the salt from that record
    with what is expected"""
    #Test getting a salt that exists in the password file:
    username = "test_username4"
    role = 4
    salt = "test_salt4"
    hashed_password = "test_hashed_password4"

    try:
        with open('test_passwd.txt', 'a') as file:
            record = f"{username}:{role}:{salt}:{hashed_password}\n"
            file.write(record)
    except FileNotFoundError:
        print("In test_get_salt: Password file not found. ")
    except IOError as e:
        print(f"In test_get_salt: Error writing to file {e}")

    self.assertEqual(self.password_manager.get_salt(username), "test_salt4")
    #Test getting a salt that does not exists in the password file:
    self.assertNotEqual(self.password_manager.get_salt(username), "invalid_salt")
def test_get_hashed_password(self):
    """Tests get_hashed_password() by creating a record in the test password file and comparing the hashed password from that record
    with what is expected"""
    #Test getting a hashed password that exists in the password file:
    username = "test_username5"
    role = 5
    salt = "test_salt5"
    hashed_password = "test_hashed_password5"

    try:
        with open('test_passwd.txt', 'a') as file:
            record = f"{username}:{role}:{salt}:{hashed_password}\n"
            file.write(record)
    except FileNotFoundError:
        print("In test_get_hashed_password: Password file not found. ")
    except IOError as e:
        print(f"In test_get_hashed_password: Error writing to file {e}")

    self.assertEqual(self.password_manager.get_hashed_password(username), "test_hashed_password5")

    #Test getting a hashed password that does not exists in the password file:
    self.assertNotEqual(self.password_manager.get_salt(username), "invalid_hashed_password")
```

Problem 3: Enrolment Mechanism

3.a) User Interface Design

For the user interface portion of the enrolment process of the system, I will keep it simple, following the design outlined in the instructions document. This will all be showed through a console, and I will display the company name followed by a prompt

that asks the user whether they want to log in or enroll. In this section I will only implement the enrollment process, but I will implement the login capability in the next part of this assignment. I will prompt the user to input 1 if they want to create a new account and 2 if they want to login. When the user inputs their password I will hide the password from the console to keep it secure. I will make sure to handle invalid inputs properly to ensure the system does not break if the user accidentally inputs another input that is not expected. It will look something like the following if they input a password that is valid:



It will look something like this if the user inputs an invalid password:

```
===== User Authentication System =====

1. Log In
2. Create New Account

Please enter your choice (1, 2): 2

Enter User ID: new_user
Enter Password: ********

Invalid password. Password must include at least one upper-case letter. Password must include at
least one special character: {!@#$%?*}.
Try again
Enter new password then press Enter:
*****
```

3.b) Password Checker Design

For the password checker mechanism, I will create a method that validates a given password and returns true if the password checks all the conditions outlined in the instructions document. It would be beneficial to the user if they understood why a password that they provided is not accepted, so I will also return the message that is associated with the condition(s) that it failed. For some of the formatting conditions such as the password not being able to be in a common date format or a telephone number, I plan to use the regular expression pattern matcher to search for these types of

sequences in the password. The following is pseudo code for the password checker mechanism:

PasswordCheckerDesign

```
def validate_password(passwordToCheck, username):
        error_message = ''
        success = true

        # Check 1: Password should not match the username.
        if passwordToCheck.lower() == username.lower():
            errors.append("Password should not match the username.")
            success = false

        # Check 2: Password length must be between 8 and 12 characters.
        if not 8 <= len(passwordToCheck) <= 12:
            errors.append("Password must be between 8 to 12 characters long.")
            success = false

        # Check 3-6: Password must have at least one uppercase letter, one lowercase letter, one
        # numerical digit, and one special character.
        if not any(char.isupper() for char in passwordToCheck):
            errors.append("Password must include at least one upper-case letter.")
            success = false
        if not any(char.islower() for char in passwordToCheck):
            errors.append("Password must include at least one lower-case letter.")
            success = false
        if not any(char.isdigit() for char in passwordToCheck):
            errors.append("Password must include at least one numerical digit.")
            success = false
        specialChars = {'!', '@', '#', '$', '%', '?', '*'}
        if not any(char in specialChars for char in passwordToCheck):
            errors.append("Password must include at least one special character: {!@#$%?*}.")
            success = false

        # Check 7-9: Password must not have a calendar date, license plate, or telephone number
        # format.
        if passwordToCheck has a date format:
            errors.append("Password should not match the format of common dates.")
            success = false
        if passwordToCheck has a license plate format:
            errors.append("Password should not match the format of license plate numbers.")
            success = false
        if passwordToCheck has a phone number format:
            errors.append("Password should not match the format of common phone numbers.")
            success = false

        # Check 10: Password must not be listed in the weak passwords list.
        if passwordToCheck is in weak password file:
            errors.append("Password is too common, choose a new one.")
            success = false


        return success, errors
```

3.c) Enrolment Mechanism Implementation

To implement the enrolment mechanism, I created a class called Enrol that receives a password file since it interacts with the PasswordManagement() class. It creates an AccessControl() and PasswordManagement() object when it is created since the enrollment process uses methods of each of these classes. I will go into detail of how this class interacts with these components and how the enrolment process is implemented. The first method in this class is called validate_password() which takes a username and a password to check. The checks that are included cover length constraints, uppercase and lowercase letters, numerical digits, special characters, and restrictions on formats (matching username, common data format, license plate numbers, telephone numbers, etc). it also checks that the password is not present in a list of weak passwords. This weak password list was created by researching most common passwords and asking multiple people what they think the most common passwords are [9]. It returns false and a message if the password fails any password policy conditions. It returns true and a message indicating a valid password if the given password passes all the checks. It implements the pseudo code I designed in the previous question.

```
class Enrol:

    def __init__(self, password_file = "passwd.txt"):
        self.password_file = password_file
        self.AccessControl = AccessControl()
        self.PasswordManagement = PasswordManagement(self.password_file)

    def validate_password(self, username, passwordToCheck):
        """Returns True and an success message if the given password meets the password policy requirements.
        Returns False along with a message that says which requirement(s) the password does not meet. """
        m = ""
        returnVal = True

        # password must not match the username
        if passwordToCheck.lower() == username.lower():
            returnVal = False
            m += "Password should not match the username. "

        # must be 8-12 chars
        if not 8 <= len(passwordToCheck)<= 12:
            returnVal = False
            m += "Password must be between 8 to 12 characters long. "

        # must have at least one uppercase letter:
        if not any(char.isupper() for char in passwordToCheck):
            returnVal = False
            m += "Password must include at least one upper-case letter. "

        # must have at least one lowercase letter
        if not any(char.islower() for char in passwordToCheck):
            returnVal = False
            m += "Password must include at least one lower-case letter. "

        # must have at least one numerical digit
        if not any(char.isdigit() for char in passwordToCheck):
            returnVal = False
            m += "Password must include at least one numerical digit. "

        # must have at least one special character
        specialChars = {'!', '@', '#', '$', '%', '?', '*'}
        if not any(char in specialChars for char in passwordToCheck):
            returnVal = False
            m += "Password must include at least one special character: {!@#$%?*}. "

        # must not have a calendar date format ((MM/DD/YYYY or YYYY-MM-DD)) - is this fine?
        dateFormatsRegex = re.compile(r'\b(?:\d{1,2}/\d{1,2}/\d{4}|\d{4}-\d{1,2}-\d{1,2})\b')
        if dateFormatsRegex.search(passwordToCheck):
            returnVal = False
            m += "Password should not match the format of common dates. "

        # must not have a license plate format
        #the following regex looks for a sequence of characters that have either letters or digits
        licensePlateRegex = re.compile(r'\b[A-Z0-9]{2,8}\b') #Ontarion license plates are between 2 - 8 characters
        if licensePlateRegex.match(passwordToCheck):
            returnVal = False
            m += "Password should not match the format of license plate numbers. "

    # must not have a telephone number format
    phoneNumberRegex = re.compile(r'\b\d{3}[-.\s]?\d{3}[-.\s]?\d{4}\b')
    if phoneNumberRegex.search(passwordToCheck):
        returnVal = False
        m += "Password should not match the format of common phone numbers. "

    # must not be listed in weak passwords list
    with open('weakPasswords.txt', 'r') as file:
        weakPasswordList = [line.strip() for line in file]
        if passwordToCheck in weakPasswordList:
            returnVal = False
            m += "Password is too common, choose a new one. "

    if(returnVal):
        m = 'Password Created!'

    return returnVal, m
```

The next method I implemented is called username_exists() and it takes a username and returns true if that username does exist in the password file and false otherwise. This is going to be used in the enrolment process as a user should not be able to enrol an

account with a username that already exists since an account's username is their unique identifier. If the username already exists, then their password should not be checked. I ensure to handle a file related errors by implemented the functionality in a try-except block and output a descriptive error.

```
def username_exists(self, username):
    """Returns True if the given username exists in the password file. Returns False otherwise. """
    try:
        with open(self.password_file, 'r') as file:
            for line in file:
                usernameInFile = (line.split(":"))[0].strip()
                if username == usernameInFile:
                    return True
    except FileNotFoundError:
        print("Password file not found. ")

    return False
```

The next method I implemented is called prompt_user_role() which prompts a user for their role in the company when they are creating a new account. Since I created an Enum to represent the roles of this company, each role is already associated with a number, so these numbers are displayed to the user to select their respective role. The method returns the role that the user selected. I ensure to check that the role number a user inputs is a valid one, or else they are prompted to select another one.

```
def prompt_user_role(self):
    """Prompts user for their role in the company. Returns the number associated with the role they select. """
    invalidRole = True
    while(invalidRole):
        userRole = input("What is your role?\nEnter 1 if you are a Client.\nEnter 2 if you are a Premium Client.\nEnter 3 if you are" +
            "a Financial Planner.\nEnter 4 if you are a Financial Advisor.\nEnter 5 if you are an Investment Analyst.\n" +
            "Enter 6 if you are a Technical Supprt.\nEnter 7 if you are a Teller.\nEnter 8 if you are a Compliance Officer.\n")
        try:
            intUserRoleVal = int(userRole)
            if intUserRoleVal in self.AccessControl.get_role_values():
                invalidRole = False
            else:
                print("\nInvalid role number. Try again. ")
        except ValueError:
            print("Please enter a number. Try again.\n")

    return intUserRoleVal
```

The last method is the method that ties all the above methods together, it is called prompt_enrol() and it prompts the user for a username and if it exists it prompts them for a password and calls the password checker method to determine if it a valid one. If it is then it generates a salt, hashes the valid password, and stores it in the password file.

```python
def prompt_enrol(self):
    """Prompts user for a username, checks that it doesn't already exist in the password file and prompts
    then for a password if it is a unique username. """
    userNameTaken = True
    while(userNameTaken):
        newUserName = input("Enter new username:\n")
        userNameTaken = self.username_exists(newUserName)
        if userNameTaken:
            print("Username already exists. Use a different one. ")

    #ask user for password and check that it is valid
    passwordWorks = False
    while(not passwordWorks):
        userPsswd = getpass.getpass("Enter new password then press Enter:\n")
        passwordWorks, m = self.validate_password(newUserName, userPsswd)
        if(not passwordWorks):
            print("Invalid password.", m, "\nTry again")

    #ask user what role they want to have
    userRole = self.prompt_user_role()

    #hash password and store record in file
    gen_salt = self.PasswordManagement.generate_salt()
    newHashedPsswd = self.PasswordManagement.hash_password(userPsswd, gen_salt)
    self.PasswordManagement.save_password_record(newUserName, userRole, gen_salt, newHashedPsswd)
```

3.d) Enrolment Mechanism Testing

To test the enrolment mechanism and proactive password checker, I created a unit test file named test_problem3d.py that has a test method per method in the original problem3d.py file. The first thing I did was write a test record in the test password file to test the functionality of trying to enrol a user with a username that already exists. Before the tests close, I make sure to delete the contents of the test password file so that I can continue to test this from scratch every time the tests are run.

```python
class TestEnrol(unittest.TestCase):

    def setUp(self):
        self.enrol = Enrol("test_passwd.txt")
        username = "admin"
        role = 6
        salt = 112233
        hashed_password = "Hel12!0#"
        try:
            with open("test_passwd.txt", 'a') as file:
                record = f"{username}:{role}:{salt}:{hashed_password}\n"
                file.write(record)
        except FileNotFoundError:
            print("Password file not found. ")
        except IOError as e:
            print(f"Error writing to file {e}")

    def tearDown(self):
        #delete contents of test_passwd.txt file after running tests
        try:
            with open("test_passwd.txt", 'w'):  # 'w' mode truncates the file or creates a new empty file
                pass  # Nothing needs to be written; the file is effectively emptied
        except FileNotFoundError:
            print(f"File 'test_passwd.txt' not found.")
```

The first test method I created is called test_validate_password() which tests the original validate_password() method. This test method passes example passwords to the validate_password() method, some valid and some valid and ensures that the correct return value and message is returned. For each password policy I pass a password that specifically violates that condition. I also test the method on a password that should pass all the checks.

```
def test_validate_password(self):
    """Tests validate_password() by passing the actual method example passwords that should violate
    each condition and test passwords that should pass all conditions"""

    #Checks that the password isn't equal to username
    result, message = self.enrol.validate_password("john_doe", "john_doe")
    self.assertFalse(result)
    self.assertTrue("Password should not match the username. " in message)

    #Checks that the password is too short
    result, message = self.enrol.validate_password("john_doe", "short")
    self.assertFalse(result)
    self.assertTrue("Password must be between 8 to 12 characters long. " in message)

    #Checks that the password is too long
    result, message = self.enrol.validate_password("john_doe", "PasswordThatIsWayTooLongToPassThisCheck")
    self.assertFalse(result)
    self.assertTrue("Password must be between 8 to 12 characters long. " in message)

    #Checks that the password must have at least one upper case letter
    result, message = self.enrol.validate_password("john_doe", "nouppercases")
    self.assertFalse(result)
    self.assertTrue("Password must include at least one upper-case letter. " in message)

    #Checks that the password must have at least one lower case letter
    result, message = self.enrol.validate_password("john_doe", "NOLOWERCASES")
    self.assertFalse(result)
    self.assertTrue("Password must include at least one lower-case letter. " in message)

    #Checks that the password must have at least one digit
    result, message = self.enrol.validate_password("john_doe", "PsswdNoDigit")
    self.assertFalse(result)
    self.assertTrue("Password must include at least one numerical digit. " in message)

    #Checks that the password must have at least one special character
    result, message = self.enrol.validate_password("john_doe", "N0Chars083")
    self.assertFalse(result)
    self.assertTrue("Password must include at least one special character: {!@#$%?*}. " in message)

    #Checks that the password is not in a date format
    result, message = self.enrol.validate_password("john_doe", "12/04/2023")
    self.assertFalse(result)
    self.assertTrue("Password should not match the format of common dates. " in message)
```

```
#Checks that the password is not in a license plate format
result, message = self.enrol.validate_password("john_doe", "AB123CD")
self.assertFalse(result)
self.assertTrue("Password should not match the format of license plate numbers. " in message)

#Checks that the password is not in a telephone number format
result, message = self.enrol.validate_password("john_doe", "123-456-7890")
self.assertFalse(result)
self.assertTrue("Password should not match the format of common phone numbers. " in message)

#Checks that the password is not listed in the weak passwords list
result, message = self.enrol.validate_password("john_doe", "passw0rd")
self.assertFalse(result)
self.assertTrue("Password is too common, choose a new one. " in message)

#Checks that a valid example password passes all the above checks
result, message = self.enrol.validate_password("john_doe", "Hel12!0#")
self.assertTrue(result)
self.assertTrue("Password Created!" in message)
```

The next test method is called test_username_exists() which tests the original username_exists() method. This test passes a password that should exist and tests that the original method returns true. It also passes usernames that do not exist and checks that the original method returns false.

```
def test_username_exists(self):
    """Tests username_exists() returns the correct boolean when checking is a username exists. """
    #Checks that an existing username exists
    self.assertTrue(self.enrol.username_exists("admin"))
    #Checks that a non existing username exists
    self.assertFalse(self.enrol.username_exists("Admin"))
    #Checks that a non existing username exists
    self.assertFalse(self.enrol.username_exists("DoesntExist"))
```

The last test method that I implemented is called test_prompt_user_role() which tests the original prompt_user_role() method by using creating a mock input. This is done by using the Python unittest.mock library that has a “with patch” statement that is used to temporarily modify the behaviour of a function. In the context of this method, it is used to mock the input function. The test_inputs list contains the values that will be returned every time the patched function is called, allowing me to simulate the different user inputs. The first test case inputs a valid role number and tests that the original method returns the correct role number. The next test case tests the method by passing in an invalid role number but still passes a digit. The third test case passes a string which is also invalid, so it tests that there was a error raised in the function that is being called.

```python
@patch("builtins.input", test_inputs=["1", "13", "invalid"])
def test_prompt_user_role(self, mock_input):
    # Test valid role input
    with patch("builtins.input", return_value="1"):
        result = self.enrol.prompt_user_role()
        self.assertEqual(result, 1)

    # Test invalid role input, still a number
    with patch("builtins.input", return_value="13"):
        self.assertRaises(ValueError)

    # Test invalid role input, not a number
    with patch("builtins.input", return_value="invalid"):
        self.assertRaises(ValueError)
```

Problem 4: Login Mechanism

4.a) User Interface Design

The design of the login user interface is the similar as the enrollment process, but now I am focusing on what happens when a user enters the input to log in. When a user attempts to login the system should prompt them for a username. If the username does not even exist, the system should relay this message. If the username does exist, the system should prompt them for a password. If the password is correct, the system should display the role of the user and the permissions it has for every resource. If the password is incorrect, the system should give a user a capped number of attempts before locking them out. It is recommended to choose a value of 10 to start, but it depends on the organization's risk level [10]. Since Finvest Holdings is a financial institution that has access to numerous financial instruments, access management and security is one of main properties. I will choose this number to be 5 because of these reasons. The user interface should look something like this for a successful login for a user that has a technical support role:

```
===== User Authentication System =====

1. Log In
2. Create New Account

Please enter your choice (1, 2): 1

Enter User ID: admin
Enter Password: ********

ACCESS GRANTED

UserID: admin with the role:  TECHNICAL_SUPPORT has the following permissions:
Resource: ACCOUNT_BALANCE, Permissions: READ
Resource: INVESTMENT_PORTFOLIO, Permissions: READ
Resource: CLIENT_ACCOUNT_ACCESS, Permissions: ACCESS
Resource: SYSTEM_OFF_HOURS, Permissions: ACCESS
Resource: SYSTEM_ON_HOURS, Permissions: ACCESS
```

The user interface should look something like this for a failed login, exceeding the capped number of attempts a user has before the session is terminated:

```
===== User Authentication System =====

1. Log In
2. Create New Account

Please enter your choice (1, 2): 1

Enter User ID: admin
Enter Password: ********

Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Maximum attempts reached. Ending session.
```

4.b) Password Verification Mechanism Implementation

To implement the password verification and access control mechanisms, I created a Login class that handles these functionalities. The first thing I did was create a method called login_user which is called when a user selects the option to login rather than creating a new account. Within this method, I first prompt the user for a username and if the username they provide does not exist I display a message that relays this and prompt them to enter another username. This is on a loop until the user provides a valid username or until they exceed the max number of attempts, which is 5. Then after a user provides a valid username, I grab the salt and the hashed password in the record in the password file that is associated with that username. I do this by calling on the get_salt() and get_hashed_password() methods in the PasswordManagement class. I then prompt the user for a password and after they enter a password, I hash this password by calling the hash_password() from the PasswordManagement class and compare this new hash with the one stored in the password file. If they are equal, I display a message to the user indicating that they have been granted access to the system. I also display all the permissions this user has with every resource. I do this by calling a method called display_permissions() which will be discussed in further detail in the next question. This functionality is very closely tied to the password file management done in Problem 2 and the functions called in this class are explained in detail under Problem 2.

```
def login_user(self):
    """Logs in user"""
    userNameNotExist = True

    #loop until user enters a valid username
    while(userNameNotExist):
        username = input("Enter username: ")
        passwordRecord = self.PasswordManagement.get_password_record(username)

        if passwordRecord is None:
            print("That username does not exist, try again. ")
        else:
            userNameNotExist = False
            incorrectPassword = True
            saltInFile = self.PasswordManagement.get_salt(username)
            hashedPsswdInFile = self.PasswordManagement.get_hashed_password(username)

            numOfAttempts = 0
            MAX_ATTEMPTS = 5

            #loop until user enters correct password or until they reach max attempts
            while incorrectPassword and numOfAttempts < MAX_ATTEMPTS:

                userPasswd = getpass.getpass("Enter password: ")
                newHash = self.PasswordManagement.hash_password(userPasswd, saltInFile)

                if(hashedPsswdInFile == newHash):
                    print("\nACCESS GRANTED\n")
                    self.display_permissions(username)
                    incorrectPassword = False
                else:
                    numOfAttempts += 1
                    print("Incorrect Password. Try again!\n")

            if numOfAttempts == MAX_ATTEMPTS:
                print("Maximum attempts reached. Ending session.")
```

4.c) Enforcing Access Control Mechanism Implementation

After a user successfully authenticates into the system, their userID and information should be displayed to them. The information associated with a user's role is stored in the AccessControl class I created from problem 1. I implemented a display_permissions() method in the Login class that is invoked in the login_user() method after a user inputs a valid username and password. This method prints the permissions a given username has by calling on the get_role_num() method in the PasswordManagement class that takes a username and returns their role. This role number is then used to grab the role name by invoking the get_role_name() from the AccessControl class. I then print the user id and role name then invoke the print_role_capabilities() that is in the AccessControl method. This method prints the permissions the given role has on all the resources. This is what the implementation of this method looks like:

```
def display_permissions(self, username):
    """Prints permissions for the given username"""
    roleNum = self.PasswordManagement.get_role_num(username)
    role = self.AccessControl.get_role_name(roleNum)
    print("UserID:", username, "with the role: ", role.name ,"has the following permissions: ")
    self.AccessControl.print_role_capabilities(role)
```

4.d) Login Mechanism Testing

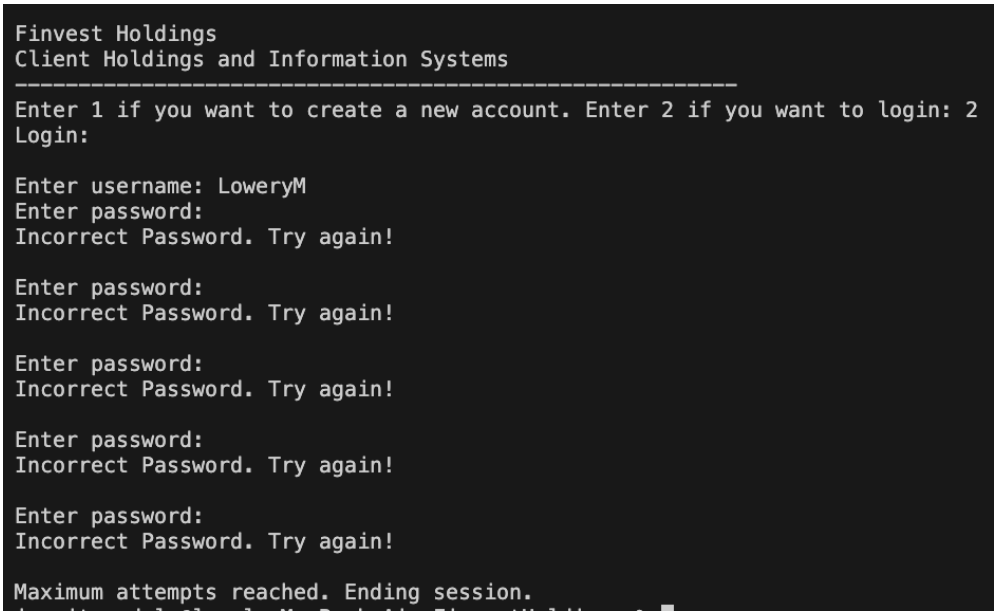
To test the user login and access control mechanisms, I performed a combination of unit and manual testing. For the unit tests portion, I created a file called test_problem4d.py which creates a test password file and writes a test user in it with a CLIENT role. I ensure to clear the contents of this file before the test file is closed. This file has only one test method called test_display_permissions() that tests the original display_permissions() method. Since this method prints the premissions of a user, I redirect the output to the test so that I can compare it with what is expected for the test user.

```
class TestLogin(unittest.TestCase):

    def setUp(self):
        self.login = Login("test_passwd.txt")
        username = "test_client_username"
        role = 1
        salt = 112233
        hashed_password = "Hel12!0#"
        try:
            with open("test_passwd.txt", 'a') as file:
                record = f"{username}:{role}:{salt}:{hashed_password}\n"
                file.write(record)
        except FileNotFoundError:
            print("Password file not found. ")
        except IOError as e:
            print(f"Error writing to file {e}")

    def tearDown(self):
        #delete contents of test_passwd.txt file after running tests
        try:
            with open("test_passwd.txt", 'w'):  # 'w' mode truncates the file or creates a new empty file
                pass  # Nothing needs to be written; the file is effectively emptied
        except FileNotFoundError:
            print(f"File 'test_passwd.txt' not found.")


    def test_display_permissions(self):
        #redirect stdout to capture the print output
        captured_output = StringIO()
        sys.stdout = captured_output

        self.login.display_permissions("test_client_username")

        #reset redirect.
        sys.stdout = sys.__stdout__

        expected_output = "UserID: test_client_username with the role:  CLIENT has the following permissions: \n" \
                          "Resource: ACCOUNT_BALANCE, Permissions: READ\n" \
                          "Resource: INVESTMENT_PORTFOLIO, Permissions: READ\n" \
                          "Resource: FA_CONTACT_DETAILS, Permissions: READ\n" \
                          "Resource: SYSTEM_OFF_HOURS, Permissions: ACCESS\n" \
                          "Resource: SYSTEM_ON_HOURS, Permissions: ACCESS\n"
        self.assertEqual(captured_output.getvalue(), expected_output)
```

For the login_user() method I performed extensive manual testing to test that existing users are able to login and non-existing users cannot. I first created accounts for each of the following users from the instructions document:

Name	**Role**	**Username**	**Password**
Mischa Lowery	Regular Client	LoweryM	S3cureP@ss
Veronica Perez	Regular Client	PerezV	P@sswd56!
Winston Callahan	Teller	CallahanW	Str0ngP@ss
Kelan Gough	Teller	GoughK	Secure123!
Nelson Wilkins	Financial Advisor	WilkinsN	R@ainb0w123
Kelsie Chang	Financial Advisor	ChangK	G00dP@ss!
Howard Linkler	Compliance Officer	LinklerH	H3110W01d!
Stefania Smart	Compliance Officer	SmartS	P@ssw0rd!123
Willow Garza	Premium Client	GarzaW	SafP@sswd0rd
Nala Preston	Premium Client	PrestonN	Pa$$w0rd_!
Stacy Kent	Investment Analyst	KentS	MyP@ss123
Keikilana Kapahu	Investment Analyst	KapahuK	Abc123!def
Kodi Matthews	Financial Planner	MatthewsK	MyPass!321
Malikah Wu	Financial Planner	WuM	Un1quePwd!11
Caroline Lopez	Technical Support	LopezC	TestPwd42!
Pawel Barclay	Technical Support	BarclayP	Summer2024!

After I created these accounts, I attempted to log back into all of these accounts by first providing incorrect passwords to ensure that I would not be granted access and correct passwords to ensure that this also worked. This is an example output for an unsuccessful and successful login for the first user: Mischa Lowery



```
Finvest Holdings
Client Holdings and Information Systems
--------------------------------------------------------
Enter 1 if you want to create a new account. Enter 2 if you want to login: 2
Login:

Enter username: LoweryM
Enter password:

ACCESS GRANTED

UserID: LoweryM with the role:  CLIENT has the following permissions:
Resource: ACCOUNT_BALANCE, Permissions: READ
Resource: INVESTMENT_PORTFOLIO, Permissions: READ
Resource: FA_CONTACT_DETAILS, Permissions: READ
Resource: SYSTEM_OFF_HOURS, Permissions: ACCESS
Resource: SYSTEM_ON_HOURS, Permissions: ACCESS
```

Problem 5: Summary and Demo

5. a) Summary

In summary, I chose to use a RBAC method to implement the access control mechanism for this system. I defined a set of roles representing different user types in Finvest Holdings such as CLIENT, PREMIUM_CLIENT, FINANCIAL_PLANNER etc. I Identified resources that users can access including ACCOUNT_BALANCE, INVESTMENT_PORTFOLIO, FA_CONTACT_DETAILS, etc. I specified permissions such as READ, WRITE, ACCESS and associated them with each resource, indicating the actions allowed for each role. This was done by utilizing Enumerations to represent roles, resources and permissions in a structured manner. I implemented a capabilities list as a nested dictionary, mapping roles and resources and their associated permissions.

For the password management mechanism, I use the SHA-256 hashing algorithm to hash the password by concatenating the password and salt in hexadecimal format. I used the Python secrets module to generate random unique salt in hexadecimal format. I save the username, role, salt and hashed password separated by colons in the password file. This design follows best practices by using a secure hashing algorithm and incorporating random and secure salts for password hashing. I ensure to handle potential errors that could result with interacting with a file gracefully.

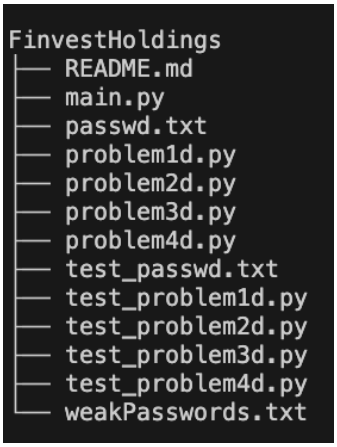
For the enrollment mechanism I prompt the user for a new username until a unique one is provided and then ask the user to enter a password and validate it to ensure it meets the specified policy requirements. This policy includes length constraints, uppercase and lowercase letters, numerical digits, special characters, and restrictions on formats (no matching usernames, no common date formats, etc). It also checks that the password is not present in a list of weak passwords. After these checks, I prompt the user to select a role from a list of pre-defined options. Their passwords are then hashed using a generated salt and the users' record is stored in a password file. This system ensures that

usernames are unique and passwords adhere to security policies. There is exception handling included for potential file-related errors and invalid inputs.

For the login mechanism, I prompt the user for a username until a valid, existing one is entered. Upon entering a valid username, the system requests a password. This password is hashed using the stored salt associated with the provided username and it is hashed and compared to the stored hashed password. If the password is correct, access is granted, and the associated permissions are displayed. If the password is incorrect, the user has 5 attempts to enter the correct password before the session is terminated. This login system incorporates role-based access control, displaying permissions upon successful login. The password entry is secure by using the getpass() method to hide input. A maximum number of login attempts is defined to enhance security. Throughout this process, the system provides informative messages to guide users.

5.b) How to Run

My prototype is contained in a folder called Finvest Holdings and this folder has the following files:



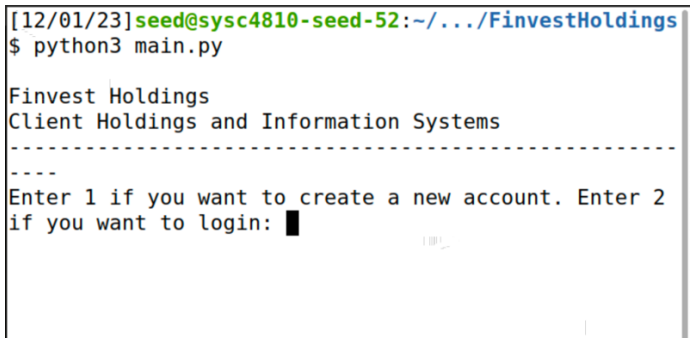
To run this program, enter the version of python that you have followed by main.py:

```
python3 main.py
```

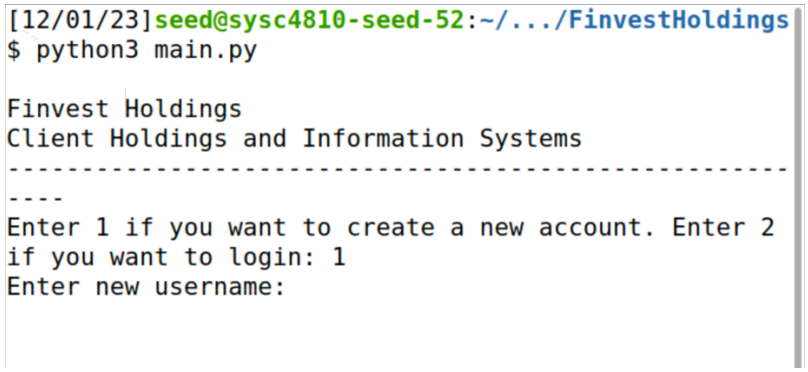
5.c) Regular Demo

The following is an example use case that is going to register the client Mischa Lowery. Before we begin, this is what the current passwd.txt file looks like:

After running the main.py program this is what is displayed:



I will first create this account so I will input the number 1:



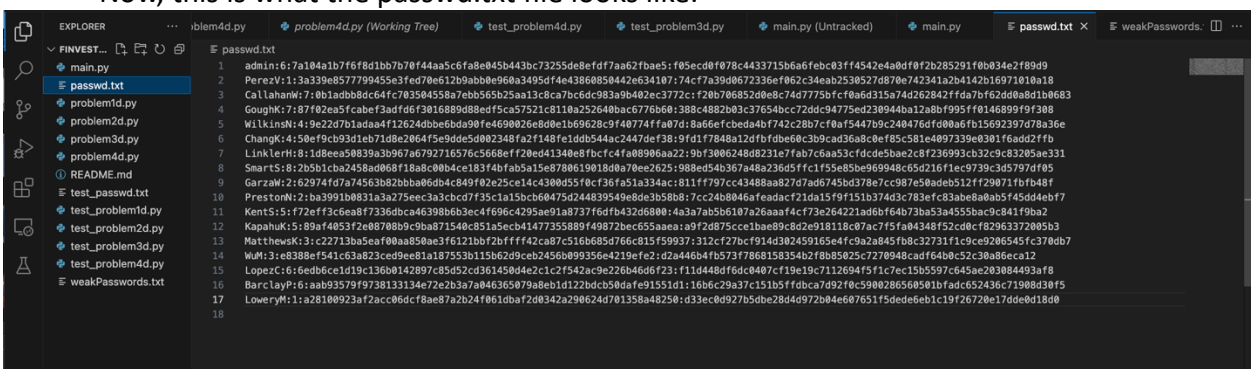
I will enter the username: LoweryM and password: S3cureP@ss .After doing this, the system will prompt me for a role:

```
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$ python3 main.py

Finvest Holdings
Client Holdings and Information Systems
------------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 1
Enter new username:
LoweryM
Enter new password then press Enter:

What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
```

I will select the number 1 then program terminates:

```
Finvest Holdings
Client Holdings and Information Systems
------------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 1
Enter new username:
LoweryM
Enter new password then press Enter:

What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
1
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$
```

Now, this is what the passwd.txt file looks like:



Now I am going to demo what the login process looks like for the same user: Mischa Lowery. The first thing to do is re-run the program to display the initial prompts. Then I enter 2 this time to login to an account:

```
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$ python3 main.py

Finvest Holdings
Client Holdings and Information Systems
------------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 2
Login:

Enter username:
```

Then I enter the username for Mischa Lowery, which is LoweryM and I enter the same password I used to create this account. After doing this the system should display the account details and the permissions Mischa has as a client, which should include READ permissions for the ACCOUNT_BALANCE resource, READ permissions for the INVESTMENT_PORTFOLIO resource, READ permissions for the FA_CONTACT_DETAILS (which is the financial advisor contact details), ACCESS permissions for the SYSTEM_OFF_HOURS and SYSTEM_ON_HOURS resources which represents accessing the system during on hours (between 9:00AM and 5:00PM) and off hours (outside of 9:00AM to 5:00PM).

```
$ python3 main.py

Finvest Holdings
Client Holdings and Information Systems
-----------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 2
Login:

Enter username: LoweryM
Enter password:

ACCESS GRANTED

UserID: LoweryM with the role:  CLIENT has the follow
ing permissions:
Resource: ACCOUNT_BALANCE, Permissions: READ
Resource: INVESTMENT_PORTFOLIO, Permissions: READ
Resource: FA_CONTACT_DETAILS, Permissions: READ
Resource: SYSTEM_OFF_HOURS, Permissions: ACCESS
Resource: SYSTEM_ON_HOURS, Permissions: ACCESS
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
```

5.d) Error Handling Demo

The following demo shows what happens when you provide invalid inputs during every step in enrolling and logging into the system. When the first prompt pops up asking to select a number to either create a new account or login to one, this is what is displayed if a user inputs another option:

```
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$ python3 main.py

Finvest Holdings
Client Holdings and Information Systems
-----------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 5
Invalid Input
Enter 1 if you want to create a new account. Enter 2
if you want to login: asdas
Invalid Input
Enter 1 if you want to create a new account. Enter 2
if you want to login: ##.
Invalid Input
Enter 1 if you want to create a new account. Enter 2
if you want to login:
```

I will attempt to create a new account for the username test_user and will provide various invalid passwords.

```
Enter new username:
test_user
Enter new password then press Enter:

Invalid password. Password must be between 8 to 12 ch
aracters long. Password must include at least one num
erical digit.
Try again
Enter new password then press Enter:

Invalid password. Password must be between 8 to 12 ch
aracters long.
Try again
Enter new password then press Enter:

Invalid password. Password must be between 8 to 12 ch
aracters long. Password must include at least one upp
er-case letter. Password must include at least one nu
merical digit. Password must include at least one spe
cial character: {!@#$%?*}. Password is too common, ch
oose a new one.
Try again
Enter new password then press Enter:
```

I will now enter a valid password to progress with the demo. The password I will use is: tE3tP@ss!
Now the prompt for selecting a role number is displayed and this is what occurs when you enter an invalid role number:

```
Enter new password then press Enter:

What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
9

Invalid role number. Try again.
What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
```

This is what happens when you enter something that isn't a number:

```
Invalid role number. Try again.
What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
invalid_role
Please enter a number. Try again.

What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
```

I will select a valid role number to continue to the logging in demo. I will select 7, a TELLER role.

```
What is your role?
Enter 1 if you are a Client.
Enter 2 if you are a Premium Client.
Enter 3 if you area Financial Planner.
Enter 4 if you are a Financial Advisor.
Enter 5 if you are an Investment Analyst.
Enter 6 if you are a Technical Supprt.
Enter 7 if you are a Teller.
Enter 8 if you are a Compliance Officer.
7
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$
```

Now this account is saved in the system, and I will go through the error handling that occurs when entering incorrect/invalid inputs in the logging in process. After entering the option to login (by entering 2), it asks the user to enter a username. This is what happens when you enter a username that does not exist:

```
Finvest Holdings
Client Holdings and Information Systems
-----------------------------------------------------
----
Enter 1 if you want to create a new account. Enter 2
if you want to login: 2
Login:

Enter username: invalid_user
That username does not exist, try again.
Enter username:
```

After entering the correct user, the next prompt is to enter the password. When you input the wrong password, this message is displayed. If you enter an incorrect password more than 5 times, the system will end the session and you need to start from the beginning. This is what that looks like:

```
Login:

Enter username: invalid_user
That username does not exist, try again.
Enter username: test_user
Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Enter password:
Incorrect Password. Try again!

Maximum attempts reached. Ending session.
[12/01/23]seed@sysc4810-seed-52:~/.../FinvestHoldings
$
```

References

[1] Dmytro.tkach@apriorit.com, “Mandatory (MAC) vs Discretionary Access Control (DAC) differences,” Ekran System, https://www.ekransystem.com/en/blog/mac-vs-dac#:~:text=What%20is%20discretionary%20access%20control,users%20or%20groups%20of%20users (accessed Dec. 1, 2023).

[2] Dmytro.tkach@apriorit.com, “Zero trust architecture: What it is, benefits and key principles,” Ekran System, https://www.ekransystem.com/en/blog/zero-trust-security-model (accessed Dec. 1, 2023).

[3] Okta, “RBAC vs. ABAC: Definitions & when to use,” Okta, https://www.okta.com/identity-101/role-based-access-control-vs-attribute-based-access-control/#:~:text=The%20main%20difference%20between%20RBAC,%2C%20action%20types%2C%20and%20more (accessed Dec. 1, 2023).

[4] “Difference between access control list and capability list,” GeeksforGeeks, https://www.geeksforgeeks.org/difference-between-access-control-list-and-capability-list/ (accessed Dec. 1, 2023).

[5] Imi, “Access control matrix and capability list,” Identity Management Institute®, https://identitymanagementinstitute.org/access-control-matrix-and-capability-list/ (accessed Dec. 1, 2023).

[6] “What is a type-safe enum in java?,” Online Tutorials, Courses, and eBooks Library, https://www.tutorialspoint.com/what-is-a-type-safe-enum-in-java#:~:text=The%20enums%20are%20type%2Dsafe,a%20class%20or%20an%20interface (accessed Dec. 1, 2023).

[7] P. Callaghan, “Why you should use SHA-256 in evidence authentication,” Pagefreezer Blog, https://blog.pagefreezer.com/sha-256-benefits-evidence-authentication (accessed Dec. 1, 2023).

[8] Admin, “What is the best hashing algorithm?,” Code Signing Store, https://codesigningstore.com/what-is-the-best-hashing-algorithm#:~:text=To%20protect%20passwords%2C%20experts%20suggest,your%20best%20hashing%20algorithm%20choices (accessed Dec. 1, 2023).

[9] A. Cruz, “The top 100 worst passwords of 2018, WE hope yours is not on the list.,” PUREVPN, https://www.purevpn.com/blog/worst-password-list/ (accessed Dec. 1, 2023).

[10] Vinaypamnani-Msft, “Account lockout threshold - windows security,” Windows Security | Microsoft Learn, https://learn.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/account-lockout-threshold (accessed Dec. 1, 2023).