TASK 2

Angelo Antona

MOBILE SYSTEM SECURITY

CORRECTION OF SECURITY VULNERABILITIES IN MY PROJECT

In this report, I am attaching the changes I made to my project to improve its security. These changes are listed in two sections:

- New changes made for Task 2.
- Changes I had already made for Task 1.

NEW CHANGES MADE FOR TASK 2

The changes I made for the last task have made my app quite resilient in terms of security. However, I noticed a potential vulnerability during the password change process.

My previous code handled password changes as follows:

1. The user was asked to enter the *previous password* and the *new password*.
2. The notes were decrypted all at once using the previous encryption key.
3. The notes were re-encrypted all at once using the new encryption key.

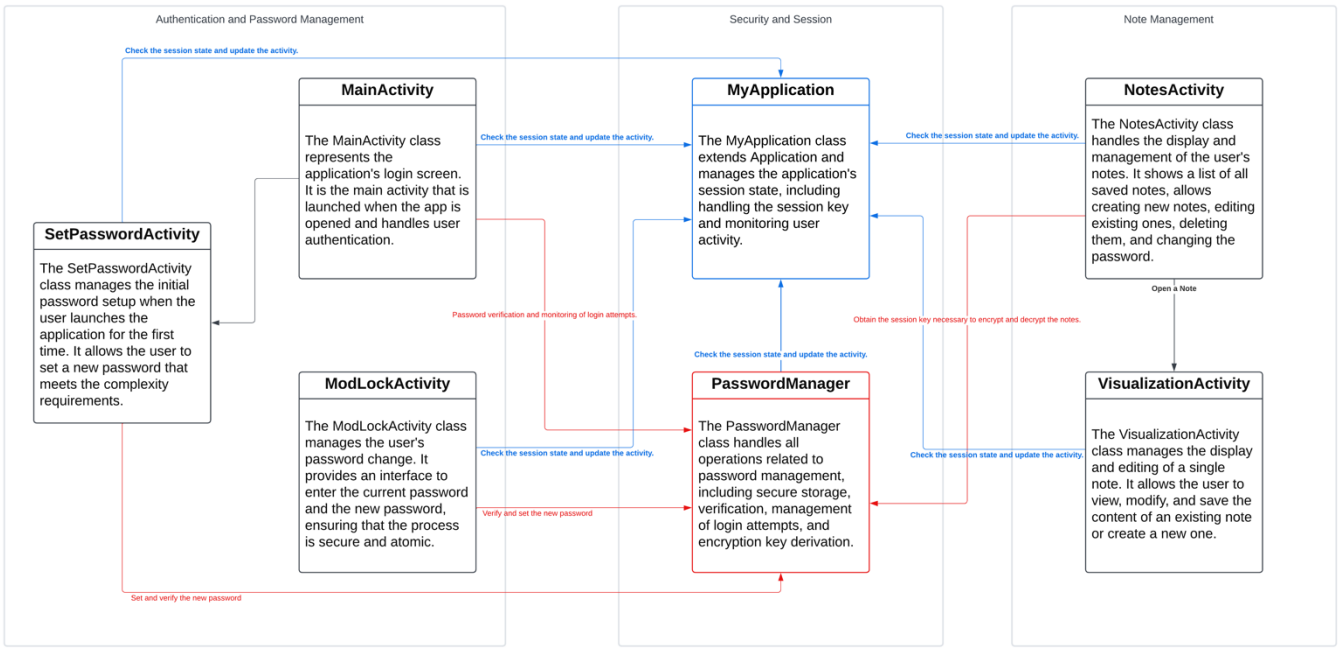
This procedure created the following vulnerability: if the application is interrupted between the decryption phase and the encryption phase, all the notes remain decrypted in memory.

To address this issue, I tried to make the password change process as atomic as possible by applying the encryption and decryption cycle individually to each note (encrypt and decrypt *note1*, if the process succeeds, move to *note2*, ...) instead of decrypting all notes in a single cycle and then re-encrypting them all together in a second cycle.

```
// Process each note individually
for (title in noteTitles) {
    val encodedContent = sharedPreferences.getString(title, null)
    if (encodedContent != null) {
        // Decrypt with the old session key
        val content = decryptNoteContent(encodedContent, oldSessionKey)
        if (content != null) {
            // Encrypt with the new session key
            val encryptedContent = encryptNoteContent(content, newSessionKey)
            if (encryptedContent != null) {
                val encodedNewContent = Base64.encodeToString(encryptedContent, Base64.DEFAULT)
                sharedPreferences.edit().putString(title, encodedNewContent).apply()
            } else {
                // In case of encryption error, restore the old password and salt
                passwordManager.restoreOldPassword(oldPasswordHash, oldPbkdf2Salt)
                return false
            }
        } else {
            // In case of decryption error, restore the old password and salt
            passwordManager.restoreOldPassword(oldPasswordHash, oldPbkdf2Salt)
            return false
        }
    }
}
```

Moreover, to robustly handle errors and edge cases during the password change process, I implemented the function **restoreOldPassword:** it restores the original password hash and PBKDF2 salt if an error occurs during the password change process. This prevents the application from being left in an inconsistent state where notes are partially re-encrypted.

The update Class Diagram is:



CHANGES I HAD ALREADY MADE FOR TASK 1

Password Security

Password Storage

The **`PasswordManager`** class stores the user's password in plaintext using **`EncryptedSharedPreferences`**.

```
fun setPassword(newPassword: String) {
    sharedPreferences.edit().putString(PASSWORD_KEY, newPassword).apply()
}

fun getPassword(): String {
    return sharedPreferences.getString(PASSWORD_KEY, DEFAULT_PASSWORD) ?: DEFAULT_PASSWORD
}

fun isPasswordCorrect(enteredPassword: String): Boolean {
    return enteredPassword == getPassword()
}
```

Additionally, password verification is done by directly comparing the entered password with the stored one, making it vulnerable to timing attacks:

```
fun isPasswordCorrect(enteredPassword: String): Boolean {
    return enteredPassword == getPassword()
}
```

Problems with that implementation:

- **Physical Access Attacks and Device Compromise** is possible because:
  - **Password stored in plaintext**: The user's password is stored in plaintext and is accessible via the Android Keystore. If an attacker gains physical access to the device or exploits a vulnerability in the Keystore, they can extract the encryption key and read the password.
  - **Note encryption key stored on the device**: Notes are encrypted with a locally stored key, which is vulnerable if the Keystore is compromised or if the attacker has root access to the device.
- **Timing-Based Attacks** are possible because password verification is performed with a simple equality comparison, which can take varying amounts of time depending on the number of correct characters, exposing the app to timing attacks that deduce the password character by character.

Solution: I modified the **`setPassword`** method to store the password hash instead of the plaintext password. Specifically, bcrypt is used to hash and salt the password securely:

```
import org.mindrot.jbcrypt.BCrypt

fun setPassword(newPassword: String) {
    val passwordHash = BCrypt.hashpw(newPassword, BCrypt.gensalt())
    sharedPreferences.edit().putString(PASSWORD_HASH_KEY, passwordHash).apply()
}

fun isPasswordCorrect(enteredPassword: String): Boolean {
    val storedHash = sharedPreferences.getString(PASSWORD_HASH_KEY, null)
    return if (storedHash != null) {
        BCrypt.checkpw(enteredPassword, storedHash)
    } else {
        false
    }
}
```

This method also performs the comparison in a secure and constant manner, resistant to timing attacks.

Password Initialization

In the previous version of the code, on the system's first startup, it requests a default password (0000) to access the application.

```
companion object {
    private const val PASSWORD_KEY = "user_password"
    private const val DEFAULT_PASSWORD = "0000"
}

fun getPassword(): String {
    return sharedPreferences.getString(PASSWORD_KEY, DEFAULT_PASSWORD) ?: DEFAULT_PASSWORD
}

fun isPasswordCorrect(enteredPassword: String): Boolean {
    return enteredPassword == getPassword()
}
```

Problem with this implementation: If the saved password is removed, the application falls back on a default password ("0000"), allowing an attacker to access data by clearing the password value in SharedPreferences.

Solution: I resolved this by eliminating the concept of a "default password" and requiring the user to create a new password on first startup:

```
companion object {
    private const val PASSWORD_HASH_KEY = "user_password_hash"
}

fun isPasswordSet(): Boolean {
    return sharedPreferences.contains(PASSWORD_HASH_KEY)
}

fun isPasswordCorrect(enteredPassword: String): Boolean {
    val storedHash = sharedPreferences.getString(PASSWORD_HASH_KEY, null)
    return if (storedHash != null) {
        BCrypt.checkpw(enteredPassword, storedHash)
    } else {
        false
    }
}
```

If **`isPasswordSet`** returns false, the user is redirected to **`SetPasswordActivity`** to create a new password, eliminating dependence on the default password:

```
if (!passwordManager.isPasswordSet()) {
    val intent = Intent(this, SetPasswordActivity::class.java)
    startActivity(intent)
    finish()
    return
}

loginButton.setOnClickListener {
    val enteredPassword = passwordEditText.text.toString()

    if (passwordManager.isPasswordCorrect(enteredPassword)) {
        val intent = Intent(this, NotesActivity::class.java)
        intent.putExtra("userPassword", enteredPassword)
        startActivity(intent)
        finish()
    } else {
        Toast.makeText(this, "Incorrect password", Toast.LENGTH_SHORT).show()
    }
}
```

Note: As we will see later, notes are encrypted using an encryption key derived directly from the user's chosen password. If an attacker deletes the current password by accessing the initial password setup screen, the old notes remain encrypted with the old encryption key and are therefore inaccessible. The only way to change the password without making the notes inaccessible is to use the password change function, as discussed in the following sections.

Absence of Key Stretching

No key stretching is applied to passwords.

Problem with this implementation: Key Stretching increases computational difficulty for brute-force attacks. If not implemented, it is easier to guess the password using that kind of techniques.

Solution: as illustrated in the previous point, bcrypt was used for hashing passwords, which includes built-in key stretching. It's important to ensure the cost factor is appropriately configured to balance security and performance. In the new code, I set the cost factor to 10 (~100 ms per hash):

```
private const val BCRYPT_COST = 10
```

and applied it during hashing:

```
val passwordHash = BCrypt.hashpw(newPassword, BCrypt.gensalt(BCRYPT_COST))
```

Lack of Password Complexity Requirements

The old application does not enforce any complexity requirements for passwords when the user changes their password through **`ModLockActivity`**.

Problem with this implementation: if the user set a weak password, it could be easier for an attacker to guess that password through a brute-force attack.

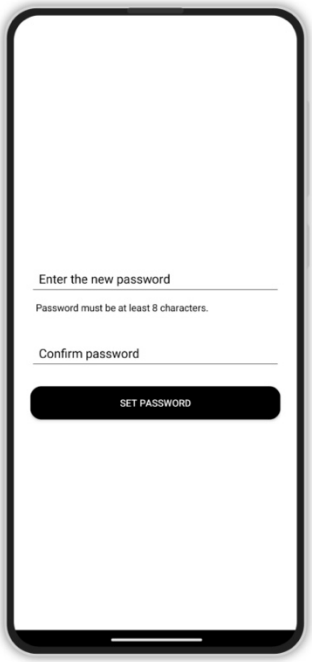
Solution: I implemented a function, **`checkPasswordComplexity`**, that verifies if the password meets complexity requirements, including:

- **Minimum length**: at least 8 characters
- **Required character types**:
  - At least one uppercase letter
  - At least one lowercase letter
  - At least one number
  - At least one special character This function returns a boolean (indicating password validity) and a message to provide feedback on unmet requirements.

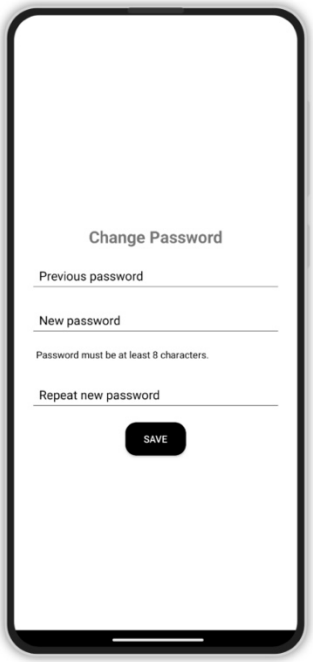
This function also provides to the user feedback on the strength of the password.

New password interface

To manage these new security features, a new password setup interface has been added for the first launch:



The password-change interface has also been modified to align with the rest of the features:



Note Security

Note Encryption Not Based on User's Password

In the old code, notes are stored with a key derived from **`MasterKeys`**, not from the user's password. using **`EncryptedSharedPreferences.`**

```
// Save note
private fun saveNote(newTitle: String, content: String) {
    val noteTitles = sharedPreferences.getStringSet(noteTitlesKey, mutableSetOf())!!.toMuta
bleSet()
    noteTitles.add(newTitle)
    sharedPreferences.edit().putStringSet(noteTitlesKey, noteTitles).apply()
    sharedPreferences.edit().putString(newTitle, content).apply()
}

// Get the note
val noteContent = sharedPreferences.getString(originalTitle, "")
bodyEditText.setText(noteContent)
```

Problem with this implementation: notes can be decrypted without the user's password if the device is compromised.

Solution: to tie note encryption to the user's password, a session key was derived using the password via PBKDF2. This key is used to encrypt and decrypt notes, ensuring they're accessible only when the correct password is entered.

I considered two options:

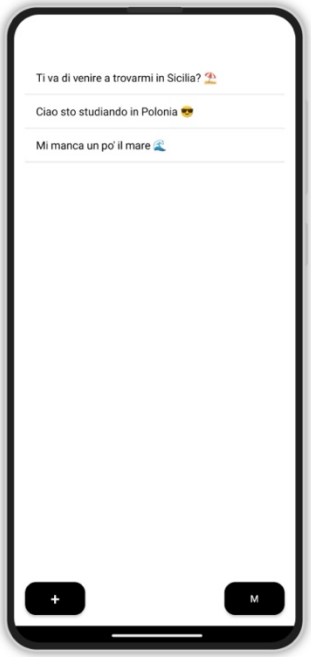
1. Prompt the user for the password each time a note is accessed, avoiding storing the encryption key in memory.
2. Prompt the user only at login or password change, deriving a session key via PBKDF2, temporarily stored in **`MyApplication`** for encryption and decryption operations. I chose the second option as the best trade-off between app usability and security.

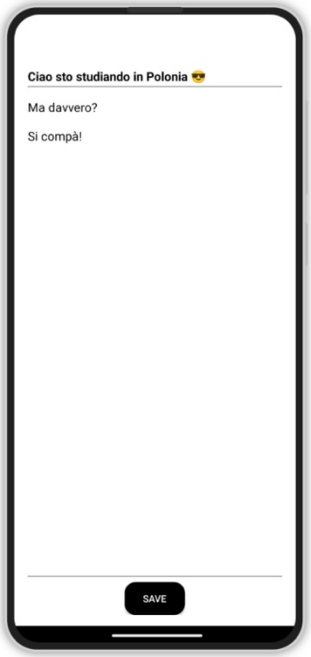
```
private fun deriveSessionKey(password: String) {
    val pbkdf2SaltString = sharedPreferences.getString(PBKDF2_SALT_KEY, null) ?: return
    val pbkdf2Salt = Base64.decode(pbkdf2SaltString, Base64.DEFAULT)
    val spec = PBEKeySpec(password.toCharArray(), pbkdf2Salt, PBKDF2_ITERATIONS, KEY_LENGTH
)
    val factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
    val key = factory.generateSecret(spec).encoded

    val app = context.applicationContext as MyApplication
    app.sessionKey = key
}
```

Note: If the password is changed, all notes are decrypted with the old key and re-encrypted with the new key derived from the new password.

Notes interface

The notes interface has not changed from the previous version.





Access Control

Lack of Access Attempt Limitation

In the old code, no limit is imposed on failed login attempts.

Problem with this implementation: an attacker can try infinite password combinations without penalty (brute-force attack).

Solution: I introduced a temporary lockout system after a certain number of failed attempts:

- **Failed Access Attempts Counter**: A counter is incremented on each failed attempt, saved in `SharedPreferences` under `ATTEMPT_COUNT_KEY`.
- **Temporary Lockout**: After a maximum number of failed attempts (defined as `MAX_ATTEMPTS`, set to 5), access is temporarily blocked. The time of the last failed attempt is recorded, and the lockout remains until `LOCKOUT_DURATION` (1 minute) has passed.
- **Lockout Status Check**: The `isLockedOut()` function checks if the user is locked out, considering both failed attempts and time since the last attempt. This function is made public for access from `MainActivity`.
- **Attempt Reset After Successful Access**: On the first successful login, failed attempts are reset with the `resetFailedAttempts()` function.

Lack of Secure Session Management

The old app does not implement session management mechanisms.

Problem with this implementation: Session Management Absence Attacks are possible because once logged in, the app remains accessible without requiring re-authentication. Anyone with physical access to the device can access notes if the app is open or running in the background.

Solution: I implemented a session timeout that requires re-authentication after inactivity, managed in `MyApplication` along with temporary key storage:

```
class MyApplication : Application() {
```

```
    var sessionKey: ByteArray? = null
    var lastActiveTime: Long = System.currentTimeMillis()

    companion object {
        private const val SESSION_TIMEOUT_DURATION = 5 * 60 * 1000 // 5 minutes
    }

    fun isSessionExpired(): Boolean {
        return (System.currentTimeMillis() - lastActiveTime) > SESSION_TIMEOUT_DURATION
    }

    fun updateLastActiveTime() {
        lastActiveTime = System.currentTimeMillis()
    }

    fun clearSession() {
        sessionKey = null
        lastActiveTime = 0L
    }
}
```

New Class Scheme

The new class diagram is as follows: