TASK 1

Angelo Antona

MOBILE SYSTEM SECURITY

Index

INTRO

I divided the discussion of security bug hunting in the relevant code into the following sections:

- **Best Practices:** This section describes how the note application should be implemented to minimize the risk of unauthorized access to its content.
- **Correction of My Project:** After studying the concepts covered in class regarding mobile system security, I analyzed my code and found several security issues. In addition to analyzing them, I corrected them by creating a new version of the application, which can be found on GitHub (https://github.com/AngeloAntona/SecureNotes) and is described in detail in the respective chapter.
- **Analysis of Some Peers' Projects:** I then proceeded to analyze some of my peers' projects. Many errors were similar to those I made, while others were of a slightly different nature.

With the structure of the report established, we can now begin the discussion.

BEST PRACTICES

In the following section, we will define what a correct approach to the security of the note application should look like. This will serve as a basis for evaluating the security of the implementations we will analyze later.

Password Security

The password for the note application is a critical element and must be carefully protected. Below, we define techniques to minimize the risk of unauthorized acquisition of the password.

Secure Password Storage

The best way to store passwords is to use Hashing + Unique, Random Salt. Specifically:

- **Robust hashing algorithms** such as bcrypt, scrypt, or Argon2 are designed to be slow and resistant to brute-force and GPU-based attacks, providing strong security.
- **Unique, random salt for each password**: this salt should be sufficiently long (e.g., 16 bytes) and generated using a cryptographically secure random number generator.

```
Remember that hashing + salt for a password involves:
        hash = HashFunction(password + salt)
and then storing only the hash and associated salt, not the plaintext password.
```

Key Stretching

Key stretching involves increasing the computational complexity required to calculate the password hash.

```
Example with bcrypt:
      hash = bcrypt(password + salt, cost_factor)
```

This increases the time needed for each attack attempt, making brute-force attacks impractical.

Strong Password Policies

It's important to require the password to meet specific criteria:

- Minimum length of at least x characters.
- Presence of uppercase and lowercase letters, numbers, and special characters.
- …

Password Change

When changing a password, it's necessary to require the current password before allowing the change. Additionally, follow the same hashing and salting process for the new password.

Input Protection

Ensure entered passwords are not visible on-screen (masked input). This help to prevent **Eavesdropping**, which involves "intercepting" data related to password.

Note Security

Notes may contain confidential information, so it is important to encrypt them in a way that minimizes the risk of unauthorized access. We will explore how to achieve this below.

Note Encryption

It's essential to encrypt notes using an encryption key derived from the user-chosen password, ensuring that only users with the correct password can decrypt the notes. To achieve this, **use a secure Key Derivation Function (KDF) such as PBKDF2, bcrypt, or Argon2 with a separate salt.**

```
Example:
encryption_key = KDF(password, salt_encryption)
```

Algorithms like **AES-256** in **GCM mode** (*Galois/Counter Mode*) can ensure both confidentiality and data integrity (*allows detection of data modifications*).

Initialization Vectors (IVs) can also be used: the IV is a unique value utilized with the encryption algorithm to ensure that each time a note is encrypted (even with the same key and content), the ciphertext differs. This value should be stored alongside the encrypted data, as it will be needed for decryption. Typically, the IV is prepended or appended to the ciphertext.

```
Pseudocode example:
IV = generate_random_iv()
ciphertext = AES_GCM_encrypt(encryption_key, IV, note_content)
data_to_store = IV + ciphertext
```

Data at Rest Protection

Encrypted notes should be securely stored in files or databases with appropriate permissions to prevent unauthorized access at the file system level. Encryption modes that provide authentication, such as AES-GCM, should be used to detect unauthorized data alterations.

Access Control

It is also necessary to implement access control. The practices I consider essential are:

- **Mandatory Authentication:** access to notes or password change functions should not be permitted without proper authentication.
- **Secure Sessions:** implement session management mechanisms that expire after a period of inactivity. Sessions should be tied to the authenticated user and not easily predictable.
- **Limiting Access Attempts:** it's necessary to protect the system from brute-force attack attempts. Measures to implement this protection could include:
    - **Limit the number of failed attempts**: after a certain number of failed attempts (*e.g., 5*), temporarily lock the account or require further verification like CAPTCHA.
    - **Exponential Delay**: increase the waiting time between attempts to discourage automated attacks.

Other Secure Coding Practices

It's better to use well-tested and up-to-date libraries for all cryptographic operations, avoiding implementing cryptographic algorithms from scratch as they may be unreliable.

CORRECTION OF MY PROJECT

I will now proceed with the analysis of the errors in my project and the explanation of the strategies I used to correct these errors.

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.

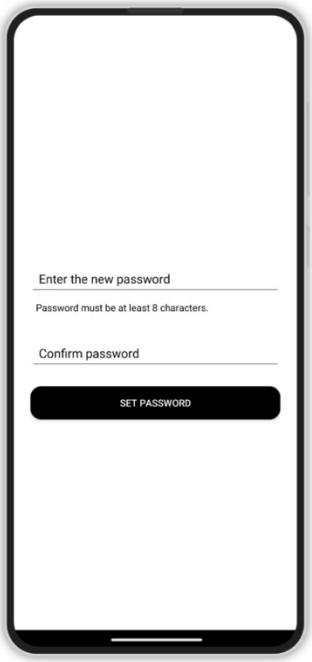
Solution: I implemented a function, `checkPasswordComplexity`, that verifies if the password meets complexity requirements, including:

- **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.

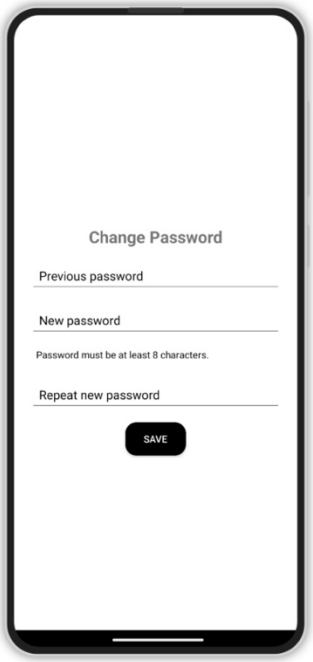
This function also provides to the user feedback on the strength of the password.

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())!!.toMutable
Set()
    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:

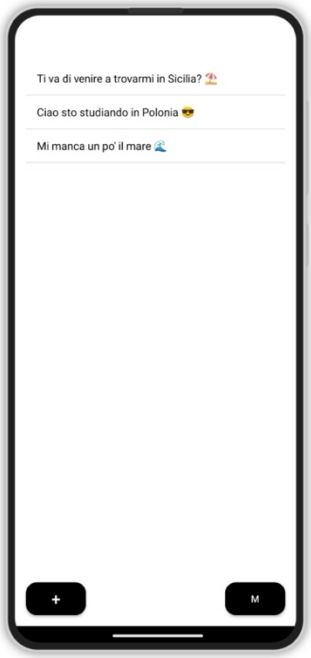
1. Prompt the user for the password each time a note is accessed, avoiding storing the encryption key in memory.
2. Prompt the user only at login or password change, deriving a session key via PBKDF2, temporarily stored in `MyApplication` for encryption and decryption operations. I chose the second option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

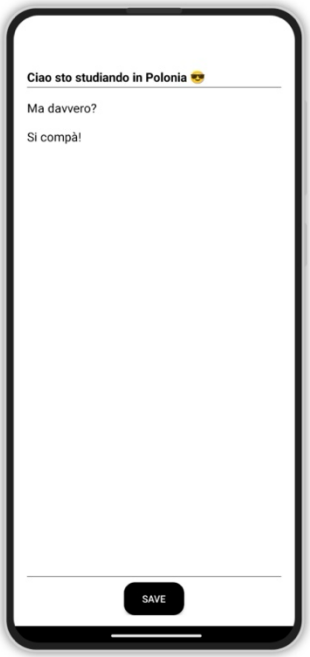
```
private fun deriveSessionKey(password: String) {
    val pbkdf2SaltString = sharedPreferences.getString(PBKDF2_SALT_KEY, null) ?: return
    val pbkdf2Salt = Base64.decode(pbkdf2SaltString, Base64.DEFAULT)
    val spec = PBEKeySpec(password.toCharArray(), pbkdf2Salt, PBKDF2_ITERATIONS, KEY_LENGTH)
    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.

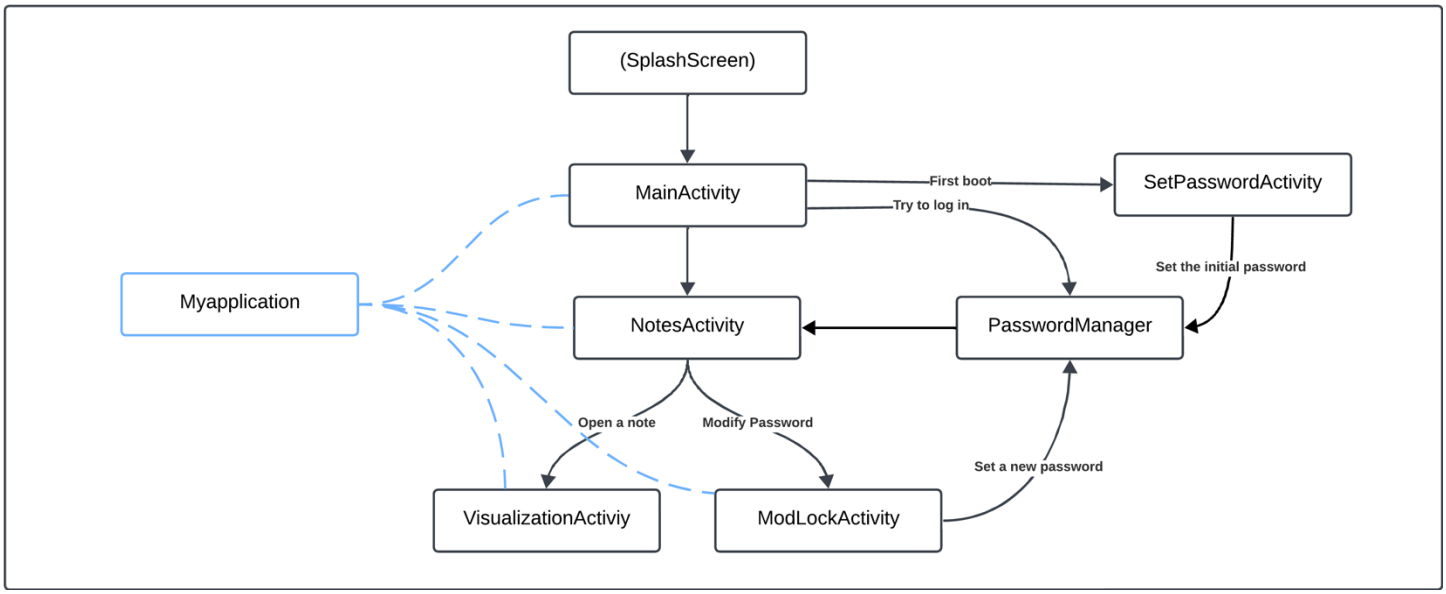
Problem with this implementation: Session Management Absence Attacks are possible because once logged in, the app remains accessible without requiring re-authentication. Anyone with physical access to the device can access notes if the app is open or running in the background.

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:



Analysis of Some Peers' Projects

Project 1.txt

Let's start with the analysis of the file "1.txt". I found in it almost the same errors I did in my implementation. I list them below.

Password Encrypted, not Hashed and Salted

The password is encrypted and stored but not hashed with a salt.

```
val encryptedPassword = EncryptionHelper.encryptData(password)
```

If an attacker gains access to the device and can extract the encryption key from the Keystore (*e.g., on a rooted device or via a vulnerability*), they can decrypt the stored passwords.

Key Stretching not implemented

Key stretching would be useful to increase the computational effort required to guess passwords, protecting against brute-force attacks. It is not implemented in this application.

Strong Password Policies not implemented

The application does not enforce any password complexity policies. Users might set weak passwords (e.g., "1234"), making them susceptible to guessing attacks.

Password Change does not require previous password

The changePassword function allows changing the password without verifying the current password.

```
// Change password without verifying current password
changePasswordButton.setOnClickListener {
	val newPassword = passwordInput.text.toString()
	changePassword(newPassword)
}
```

The password change function is accessible only after the user has logged in to the app, which might make this acceptable. However, if an attacker gains access to the app while it's already unlocked, they could change the password without authorization, locking out the legitimate user.

Note Encryption is almost ok

The encryption key is securely generated and stored in the Android Keystore and Initialization vectors are used. This implementation provides a good level of security, but it has the same problem that we saw in the paragraph dedicated to my implementation: if the phone is compromised, the attacker could get access to the notes without knowing the user's password.

Data Protection at Rest is ok

The app uses AES-256 GCM to encrypt both the password and the note content.
AES-256 GCM is a strong encryption algorithm that provides both confidentiality and integrity. Using GCM mode ensures that any unauthorized modifications to the encrypted data can be detected.

No session management

The app does not limit the number of failed login attempts. This allows attackers to perform brute-force attacks by trying multiple passwords without any restriction. An attacker can use a script or automated tool to try a large number of passwords until the correct one is found.
Additionally, there is no session management or timeout mechanism after authentication. Without session handling, the app remains in an authenticated state indefinitely, posing a risk if the device is lost or left unattended.

Not masked password input

The password input field may not be masked (not specified in the provided code), and the note content is displayed in plain text. If the password input is not masked, it can be seen by someone looking over the user's shoulder.

First-Run Logic Exploitation

The app allows setting a new password if savedPassword is null or empty. This is intended for the first run but can be manipulated. As it was for the old version of my implementation, by accessing the app's data storage and deleting or nullifying the stored encrypted password (PASSWORD_KEY), the app believes it's the first run. Upon launching the app, it will prompt to set a new password without requiring the old one. Since the secret encryption key remains the same (stored in the Keystore), the app can decrypt the existing notes after setting a new password.

Project 2.py

The code under analysis is somewhat simpler than the previous one and has a greater number of security vulnerabilities. We analyze them below.

Not Secure Password Storage

In the code, the password is stored in plain text within the `self.password` variable.

```
self.password = getpass.getpass(prompt="置新密; ")
```

If an attacker gains access to the memory or file system where the password is stored, they could read it directly without effort.
An immediate consequence is that the code does not implement any key stretching mechanism to increase the computational complexity of hashing the password, making the password more vulnerable to brute-force attacks.

Strong Password Policies not implemented

As the previous implementations, the code does not enforce any complexity requirements for passwords. Users could choose weak passwords, increasing the risk of compromise.

Not Secure Password Change

When changing the password, the code does not use hashing and salting for the new password, resulting in the same security issues as with initial storage.
A positive aspect is that the application requires the current password to change the password, which is a security plus.

```
def change_password(self):
 if self.check_password():
     # ...
```

Note Encryption Not Implemented

Notes are stored in plain text within the self.message variable.

```
self.message = new_message
```

If an attacker accesses the memory or file system, they can read the notes without restrictions.

Data Protection at Rest Not Implemented

No measures are applied to protect stored data. The data is vulnerable to unauthorized access and modification because authenticated encryption is not used to ensure data integrity.
It would be necessary to use encryption modes like AES-GCM, which provide both confidentiality and integrity.

Session Management Not Implemented

There is no session management or inactivity timeout, so if the user leaves the application open, another person could access it without authentication.

Access Attempt Limitation Not Implemented

There are no limits on the number of password entry attempts. An attacker can make unlimited attempts to guess the password (brute-force attack).

Use of Reliable Libraries

The code does not use any library for cryptographic operations. Manually implementing security functions is prone to errors.

Not secure password comparison

The application uses direct password comparison:

```
return entered_password == self.password
```

This type of comparison could be vulnerable to timing attacks, where an attacker measures the time taken to execute the comparison to deduce information about the password.
Direct password comparison could also be vulnerable if the language implementation does not handle encoding or special characters correctly.

Project 7.kt

In this code, the security shortcomings are similar to the previous codes. I will shortly list them below:

- **Weak Password Hashing Without Salt**: passwords are hashed using SHA-256 without incorporating a unique salt, making them vulnerable to rainbow table attacks.
- **Storing Plain Password Hash:** the password hash is stored in SharedPreferences without encryption, accessible if the device is compromised.
- **Notes Saved in Plain Text:** notes are saved in SharedPreferences without encryption, exposing sensitive data to unauthorized access.
- **Password Reset Without Verification:** the app allows password reset without requiring the current password, enabling unauthorized users to change it.
- **No Limitation on Failed Login Attempts:** there is no mechanism limiting the number of failed login attempts, making the app susceptible to brute-force attacks.
- **Absence of Session Timeout or Secure Session Management:** once logged in, the app remains authenticated indefinitely, increasing the risk of unauthorized access if the device is left unattended.
- **Lack of Password Complexity Requirements:** the app does not enforce rules on password complexity, allowing the use of weak passwords.
- **Potential Vulnerability to Timing Attacks:** direct comparison of password hashes may expose the app to timing-based attacks to guess the password.
- **Storing Sensitive Data Without Encryption:** sensitive information like passwords and notes are stored without adequate encryption, risking exposure if the device is compromised.
- **Inadequate Logging and Monitoring:** absence of logging security events like failed login attempts, making it difficult to detect and respond to attacks.

Project 9.kt

Also this time, security problems are similar to the previous cases:

- **Password stored in plain text:** the user's password is saved in SharedPreferences without any hashing or encryption.
- **Possibility to bypass authentication:** if a password is not set, the default password "Test1" is used. An attacker could modify SharedPreferences to reset the password.

- **Direct password comparison:** password verification occurs through direct comparison, vulnerable to timing attacks.
- **Absence of salting and key stretching:** salt or key stretching are not applied, increasing vulnerability to brute-force attacks.
- **Notes saved in plain text:** notes are saved in a simple text file without encryption.
- **No limitation on login attempts:** there is no limit on failed login attempts, allowing brute-force attacks.
- **Absence of password complexity requirements:** the app does not enforce rules on password complexity.
- **Insecure password change:** the new password is set without hashing or complexity verification.
- **Absence of session management:** there is no session management or timeout after authentication.