ECCS 3411
Computer Security
Password Manager Application

Kento Akazawa, Jada Benjamin, Grant Daly, Brian Watkins

April 28, 2024

Contents

Introduction

In today's digital landscape, where online security is paramount, the need for robust password management solutions has never been greater. For this project, we created password manager app in Swift. It offers users a secure and convenient way to store their passwords, along with associate titles, usernames, and websites. Leveraging advanced encryption and hashing techniques, this app ensures that sensitive user data remains protected against unauthorized access and potential breaches. This project can be found here.

Data

Access to stored passwords is controlled by a master password, ensuring confidentiality. Each password entry is assigned a unique ID, allowing for efficient management of multiple entries with the same information. Leveraging `CoreData`, passwords are stored securely within the application's database.

User Interface

When launching the app for the first time, it asks to set the master password as shown in Figure 1 below.

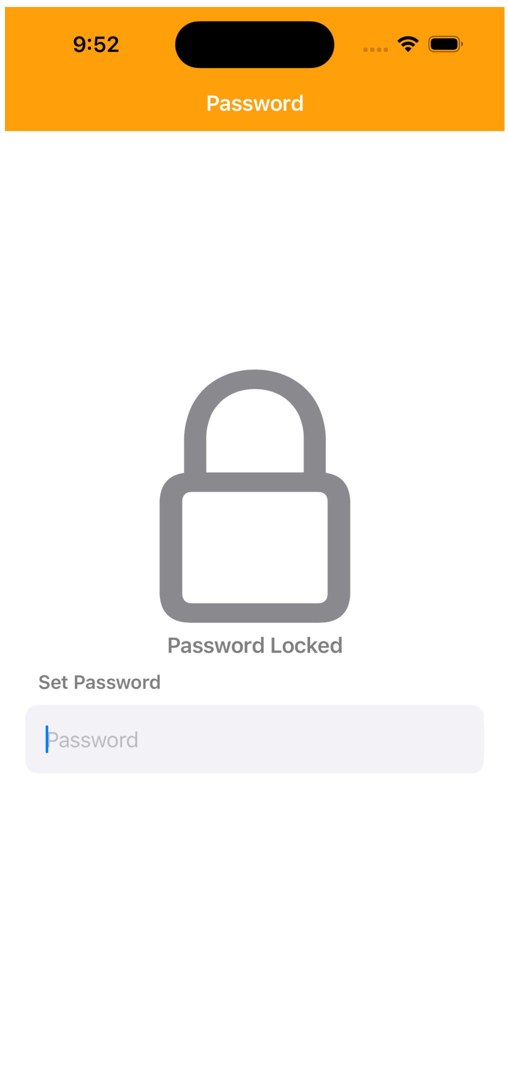


Figure 1: Setting Master Password

Enhancing the security of our application, we implemented a robust measure by hashing the master password using the SHA-256 algorithm before storage. This one-way hashing process ensures that even in the event of a breach, the original password remains secure. The hashing function, elaborated upon later in this report, provides an additional layer of protection against unauthorized access, as it is computationally infeasible to determine original password from hashed password.
When the app launches after the master password is set, it asks for the password as shown in Figure 2.

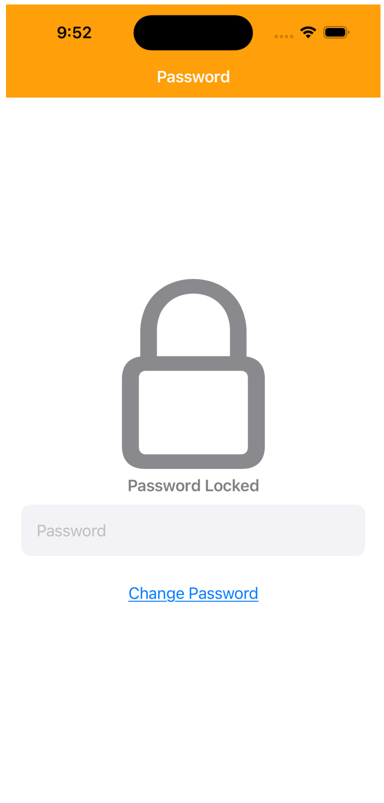


Figure 2: Entering Password

The entered password will be hashed and compared with saved hashed master password. If they do not match, it displays a message showing that entered password is wrong as shown in Figure 3.

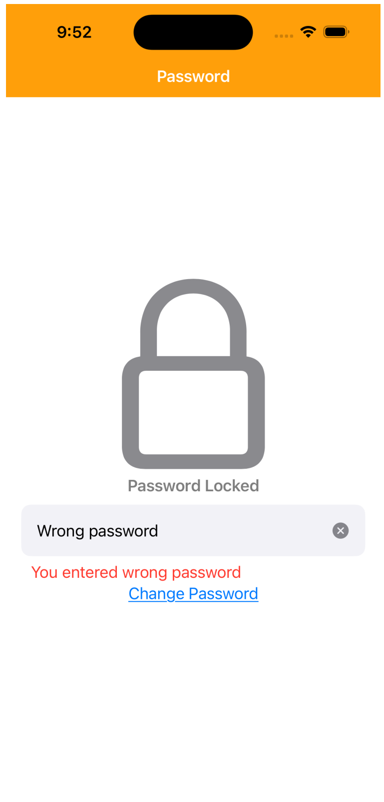


Figure 3: When wrong password is entered

The password can be changed by tapping `Change Password` link. This navigates to a screen shown in Figure 4.

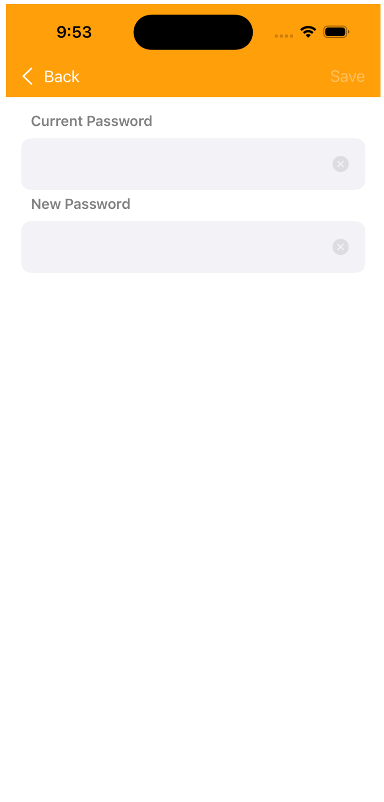


Figure 4: Changing master password

Once current password and new password are entered, save button on the top right will be enabled. If the current password matches with the saved password, new master password will be saved and navigate back to the initial screen (Figure 2).
If the entered password is correct in the initial screen (Figure 2), it grants access to the list of passwords saved and displays them as shown in Figure 5.

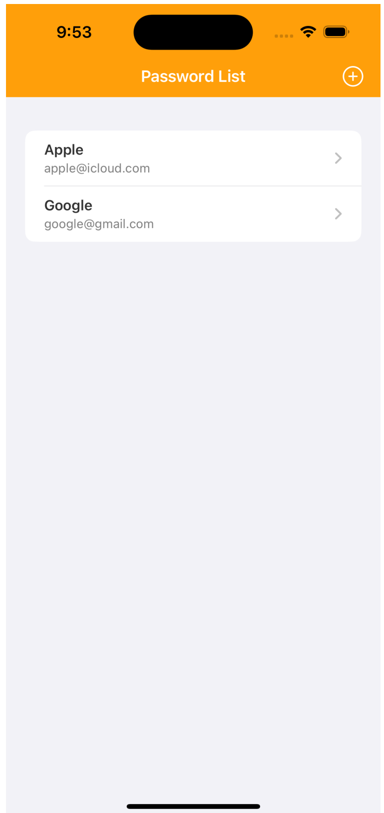


Figure 5: List of saved passwords

New password can be added by tapping on plus symbol on the top right. It will navigate to screen shown in Figure 6.

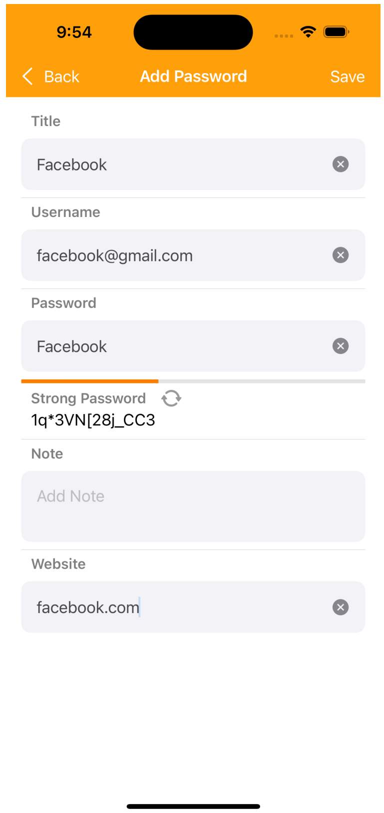


Figure 6: Adding new password

Each password stored in our system consists of a title, username, password, note, and website. We have implemented a feature that generates robust passwords automatically and provides users with a visual password strength meter to educate them on the strength of their chosen passwords. Additionally, if a user attempts to save a weak password, an alert is displayed to warn them about its vulnerability, as illustrated in Figure 7. This proactive approach encourages users to select stronger passwords, thereby enhancing the overall security of our application.

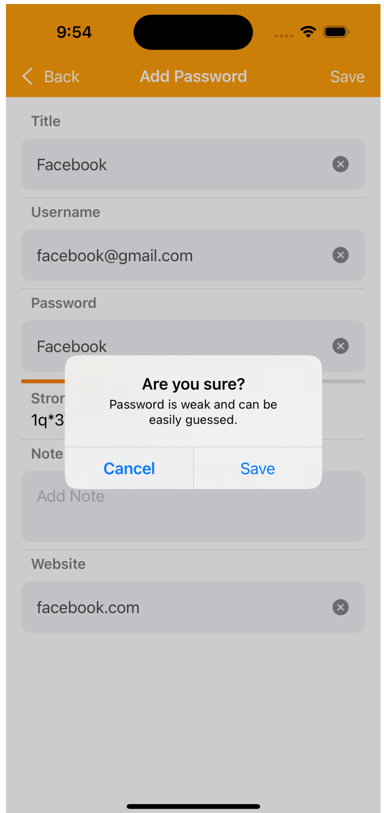
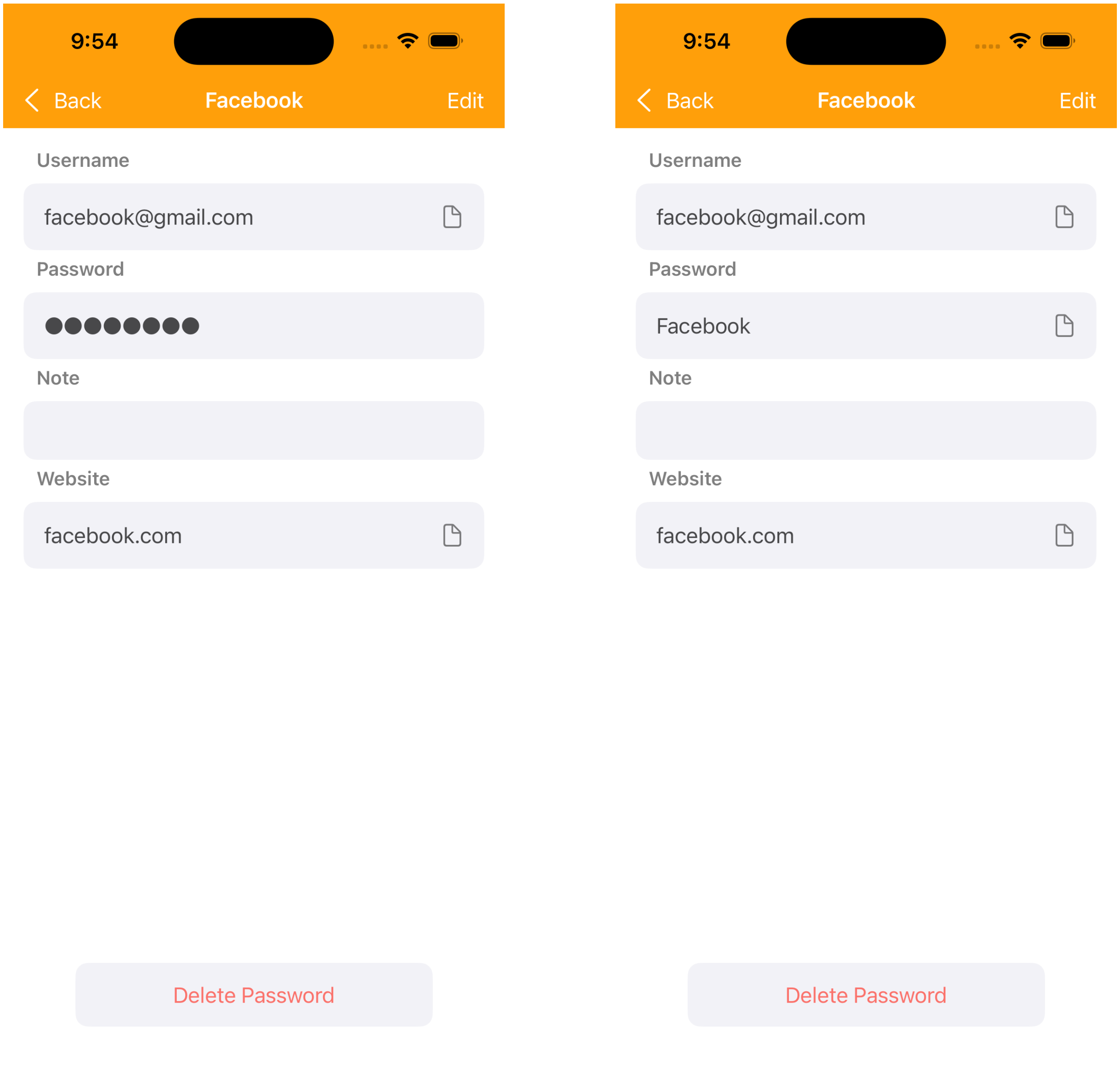


Figure 7: Alert for attempting to save weak password

When users access the password list screeen (Figure 5), they are presented with the titles and usernames of their stored passwords. Upon selecting a specific entry, users are directed to the password detail screen (Figure 8a), where the password is initially concealed to protect against shoulder surfing. However, users can reveal the password by tapping on it, as depicted in Figure 8b. Each password is encrypted before storage and decrypted when launching password detail screen (Figure 8a). This approach balances security with usability, allowing users quick access to their passwords while maintaining confidentiality.



(a)

(b)

Figure 8: Password detail view

Stored information can be modified by tapping on `Edit` button on top right of the screen (Figure 8). This will navigate to screen show in Figure 9.

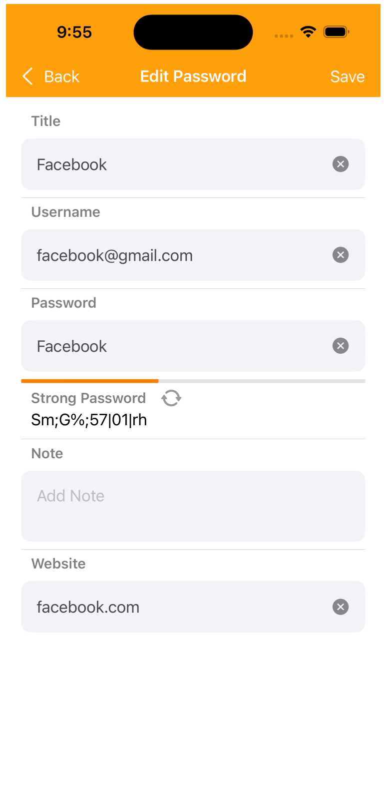


Figure 9: Editing password

This interface mirrors the layout shown in Figure 6, with the added convenience that all fields are prepopulated with previously stored data.

Hash

The master password is securely hashed using the SHA-256 algorithm prior to storage. The following code snippet illustrates this implementation:

```swift
private let saltLength = 16

// set the master password
func setMasterPassword(_ pw: String) {
  // generate salt and save to UserDefault
  let salt = getSalt()
  UserDefaults.standard.set(salt, forKey: saltKey)
  // add salt, hash and save to UserDefault
  UserDefaults.standard.set(hashPassword(pw, salt: salt),
                            forKey: masterPwKey)
}

// generates random data with @saltLength as bite size
private func getSalt() -> Data {
  let salt = Data(count: saltLength)
  var mutableSalt = salt
  _ = mutableSalt.withUnsafeMutableBytes { mutableBytes in
    SecRandomCopyBytes(kSecRandomDefault, saltLength,
                       mutableBytes.baseAddress!)
  }
  return mutableSalt
}

// hash @pw using SHA-256
private func hashPassword(_ pw: String, salt: Data) -> Data {
  // convert string to data
  guard let pwData = pw.data(using: .utf8) else {
    fatalError("Failed to convert password to data")
  }
  // add salt
  let data = pwData + salt
  // hash the password using SHA-256
  return Data(SHA256.hash(data: data))
}
```

Firstly, the `setMasterPassword(_:)` function initiates the process by generating a random salt of 16 bytes in length using the `getSalt()` method. This salt is then securely stored in the user defaults for future reference. Subsequently, the master password, accompanied by the generated salt, undergoes a hashing procedure via the `hashPassword(_:salt:)` function. This function converts the password string into data and concatenates it with the salt before applying the SHA-256 hashing algorithm from `CryptoKit` framework to generate a hashed representation of the password. The resultant hashed password is then stored securely in the user defaults under a designated key.

The `getSalt()` function is responsible for generating random data of specified length, serving as the salt for password hashing. It utilizes `SecRandomCopyBytes` from Apple's Security framework to ensure the creation of cryptographically secure random bytes.

The function that checks if entered password is correct is shown below.

```
// checks if master password saved in UserDefaults matches @pw
func doesMasterPasswordMatch(_ pw: String) -> Bool {
  // get master password from UserDefault
  guard let masterPw = UserDefaults.standard.data(forKey: masterPwKey) else {
    return false
  }
  // get salt from UserDefault
  guard let salt = UserDefaults.standard.data(forKey: saltKey) else {
    return false
  }
  // since master password is saved after being hashed
  // compare @pw after adding same salt and hashing
  return hashPassword(pw, salt: salt) == masterPw
}
```

Firstly, the function retrieves both the hashed master password and its corresponding salt from UserDefaults. If either of these values is absent, indicating that no master password has been previously set, the function immediately returns false, signaling a mismatch. Next, the function hashes the provided password using the same salt retrieved from UserDefaults and compares the resultant hash with the stored master password. This comparison ensures that the provided password, when processed with the same salt and hashing algorithm used during the initial password setup, matches the stored master password byte-for-byte.

Encryption

All passwords are encrypted using the Advanced Encryption Standard (AES) with Galois Counter Mode (GCM), a highly secure encryption method endorsed by prestigious organizations worldwide. AES has become the encryption standard of choice, endorsed by the US Government and numerous prestigious organizations worldwide [1]. Hackers can only crack an AES-encrypted password by employing a brute-force attack, attempting various password combinations until they find the correct one. AES-GCM operates with a symmetric key, meaning the same key is used for both encryption and decryption processes. This symmetric key system ensures efficiency and security, as it simplifies the encryption and decryption processes while maintaining robust protection against unauthorized access.

```
private var key = SymmetricKey(size: .bits256)

init() {
  // if the symmetric key has already been generated, assign it to @key
  // otherwise, create new key and save it to UserDefaults
  if let keyData = UserDefaults.standard.data(forKey: symmetricKey) {
    key = SymmetricKey(data: keyData)
  } else {
    // new symmetric key is created when object of this class is created
    // convert symmetric key to Data and store in UserDefaults
    let keyData = key.withUnsafeBytes { Data($0) }
    UserDefaults.standard.set(keyData, forKey: symmetricKey)
  }
}
```

It begins by defining constant `symmetricKey` to hold the identifier for storing and retrieving the key from user defaults and initializing symmetric key using `SymmetricKey(size:.bits256)` which creates a symmetric key with 256 bits.
Within the `init()` method, it attempts to retrieve the symmetric key stored in the user defaults under the identifier `symmetricKey`. If the key data is found, it is used to initialize `key` variable. Otherwise, key has not been generated

yet, so convert key generated in line 1 into a `Data` object and stored in the user defaults under the `symmetricKey` identifier for future use. When adding a new password or changing existing password, this `key` is used to encrypt.

```
// add @pw to the database after encrypting its password
func addPassword(_ pw: Password, context: NSManagedObjectContext) {
  // encrypt password
  guard let password = encrypt(pw.password) else { return }
  // save to database
  DataController().addPassword(title: pw.title,
                               username: pw.username,
                               password: password,
                               note: pw.note,
                               website: pw.website,
                               context: context)
}

// edit @pw in the database after encrypting its password
func editPassword(_ pw: Password, to passwords: Passwords,
                    context: NSManagedObjectContext) {
  // encrypt password
  guard let password = encrypt(pw.password) else { return }
  // save to database
  DataController().editPassword(passwords,
                                title: pw.title,
                                username: pw.username,
                                password: password,
                                note: pw.note,
                                website: pw.website,
                                context: context)
}
```

When adding or editing the password, password is encrypted before storage. This process ensures that sensitive user information remains protected against unauthorized access. The code for encryption is shown below.

```
// encrypts @pw and return encrypted password as Data
private func encrypt(_ pw: String) -> Data? {
  // convert string to data before encryption
  guard let pwData = pw.data(using: .utf8) else {
    return nil
  }
  // add salt
  let data = pwData + getSalt()
  do {
    let sealedBox = try AES.GCM.seal(data, using: key)
    return sealedBox.combined
  } catch {
    print("Encryption failed: \(error.localizedDescription)")
    return nil
  }
}
```

The encryption process begins by converting the plaintext password into a `Data` object using UTF-8 encoding. This step ensures that the password is in a format suitable for cryptographic operations. It then incorporates a salt to enhance the security of the encryption process.

Within a do-catch block, the encryption attempts to use the `AES.GCM.seal` method, which seals the plaintext data using the provided encryption key, `key`. If successful, the encrypted data is encapsulated within a sealed box. Finally, the combined ciphertext and authentication tag are extracted from the sealed box and returned as Data.

```swift
// decrypts @encryptedData and return the original password as string
func getPassword(_ encryptedData: Data) -> String {
  return decryptPassword(encryptedData) ?? ""
}

// decrypts @encryptedData and return password as string
private func decryptPassword(_ encryptedData: Data) -> String? {
  do {
    // decrypts using AES-GCM algorithm
    let sealedBox = try AES.GCM.SealedBox(combined: encryptedData)
    let decryptedData = try AES.GCM.open(sealedBox, using: key)
    // extract salt from decrypted data
    let pwData = decryptedData.dropLast(saltLength)
    // converts data to string using UTF8
    return String(data: pwData, encoding: .utf8)
  } catch {
    print("Decryption failed: \(error.localizedDescription)")
    return nil
  }
}
```

When displaying the password, the program needs to decrypt the saved password. The function, `decryptPassword` uses same symmetric key to decrypt. The encrypted data is first encapsulated within a sealed box using the AES-GCM algorithm. This sealed box is then passed to the `AES.GCM.open` method along with the symmetric key. If successful, the decrypted data is obtained, salt is extracted and it is converted back to a string using UTF-8 encoding.

Conclusion

The password manager app developed in Swift ensures robust security for managing passwords. It employs advanced encryption and hashing techniques, including SHA-256 for hashing master password and AES-GCM algorithm for encrypting and decrypting passwords. By prioritizing both security and convenience, the app addresses the crucial need for safeguarding sensitive data.

References

[1] TeamPassword. (2023) What is password encryption and how does it work? [Online]. Available: https://teampassword.com/blog/what-is-password-encryption-and-how-much-is-enough