

Securing iOS Data at Rest: The Keychain

Via [Tuts+ Code - Mobile Development](#) by Collin Stuart

Any app that saves the user's data has to take care of the security and privacy of that data. As we've seen with recent data breaches, there can be very serious consequences for failing to protect your users' stored data. In this tutorial, you'll learn some best practices for protecting your users' data.

In the [previous post](#), you learned how to protect files using the Data Protection API. File-based protection is a powerful feature for secure bulk data storage. But it might be overkill for a small amount of information to protect, such as a key or password. For these types of items, the keychain is the recommended solution.

Keychain Services

The keychain is a great place to store smaller amounts of information such as sensitive strings and IDs that persist even when the user deletes the app. An example might be a device or session token that your server returns to the app upon registration. Whether you call it a secret string or unique token, the keychain refers to all of these items as *passwords*.

There are a few popular third-party libraries for keychain services, such as [Strongbox](#) (Swift) and [SSKeychain](#) (Objective-C). Or, if you want complete control over your own code, you may wish to directly use the Keychain Services API, which is a C API.

I will briefly explain how the keychain works. You can think of the keychain as a typical database where you run queries on a table. The functions of the keychain API all require a `CFDictionary` object that contains attributes of the query.

Each entry in the keychain has a service name. The service name is an identifier: a *key* for whatever *value* you want to store or retrieve in the keychain. To allow a keychain item to be stored for a specific user only, you'll also often want to specify an account name.

Because each keychain function takes a similar dictionary with many of the same parameters to make a query, you can avoid duplicate code by making a helper function that returns this query dictionary.

```
import Security //... class func passwordQuery(service: String, account: String) -> Dictionary<String, Any> { let dictior
```

This code sets up the query Dictionary with your account and service names and tells the keychain that we will be storing a password.

Similarly to how you can set the protection level for individual files (as we discussed in the [previous post](#)), you can also set the protection levels for your keychain item using the [kSecAttrAccessible](#) key.

Adding a Password

The `SecItemAdd()` function adds data to the keychain. This function takes a `Data` object, which makes it versatile for storing many kinds of objects. Using the password query function we created above, let's store a string in the keychain. To do this, we just have to convert the `String` to `Data`.

```
@discardableResult class func setPassword(_ password: String, service: String, account: String) -> Bool { var status : OS
```

Deleting a Password

To prevent duplicate inserts, the code above first deletes the previous entry if there is one. Let's write that function now. This is accomplished using the `SecItemDelete()` function.

```
@discardableResult class func deletePassword(service: String, account: String) -> Bool { var status : OSStatus = -1 if
```

Retrieving a Password

Next, to retrieve an entry from the keychain, use the `SecItemCopyMatching()` function. It will return an `AnyObject` that matches your query.

```
class func password(service: String, account: String) -> String //return empty string if not found, could return an optional {
```

In this code, we set the `kSecReturnData` parameter to `kCFBooleanTrue`. `kSecReturnData` means the actual data of the item will be returned. A different option could be to return the attributes (`kSecReturnAttributes`) of the item. The key takes a `CFBoolean` type which holds the constants `kCFBooleanTrue` or `kCFBooleanFalse`. We are setting `kSecMatchLimit` to `kSecMatchLimitOne` so that only the first item found in the keychain will be returned, as opposed to an unlimited number of results.

Public and Private Keys

The keychain is also the recommended place to store public and private key objects, for example, if your app works with and needs to store EC or RSA `SecKey` objects.

The main difference is that instead of telling the keychain to store a password, we can tell it to store a key. In fact, we can get specific by setting the types of keys stored, such as whether it is public or private. All that needs to be done is to adapt the query helper function to work with the type of key you want.

Keys generally are identified using a reverse domain tag such as **com.mydomain.mykey** instead of service and account names (since public keys are openly shared between different companies or entities). We will take the service and account strings and convert them to a tag `Data` object. For example, the above code adapted to store an RSA Private `SecKey` would look like this:

```
class func keyQuery(service: String, account: String) -> Dictionary<String, Any> { let tagString = "com.mydomain." + serv
```

Application Passwords

Items secured with the `kSecAttrAccessibleWhenUnlocked` flag are only unlocked when the device is unlocked, but it relies on the user having a passcode or Touch ID set up in the first place.

The `applicationPassword` credential allows items in the keychain to be secured using an additional password. This way, if the user does not have a passcode or Touch ID set up, the items will still be secure, and it adds an extra layer of security if they do have a passcode set.

As an example scenario, after your app authenticates with your server, your server could return the password over HTTPS that is required to unlock the keychain item. This is the preferred way of supplying that additional password. Hardcoding a password in the binary is not recommended.

Another scenario might be to retrieve the additional password from a user-provided password in your app; however, this requires more work to secure properly (using [PBKDF2](#)). We will look at securing user-provided passwords in the next tutorial.

Another use of an application password is for storing a sensitive key—for example, one that you would not want to be exposed just because the user had not yet set up a passcode.

`applicationPassword` is only available on iOS 9 and above, so you will need a fallback that doesn't use `applicationPassword` if you are targeting lower iOS versions. To use the code, you will need to add the following into your bridging header:

```
#import <LocalAuthentication/LocalAuthentication.h> #import <Security/SecAccessControl.h>
```

The following code sets a password for the query `Dictionary`.

```
if #available(iOS 9.0, *) { //Use this in place of kSecAttrAccessible for the query var error: Unmanaged<CFError>? }
```


Notice that we set `kSecAttrAccessControl` on the Dictionary. This is used in place of `kSecAttrAccessible`, which was previously set in our `passwordQuery` method. If you try to use both, you'll get an `OSStatus -50` error.

User Authentication

Starting in iOS 8, you can store data in the keychain that can only be accessed after the user successfully authenticates on the device with Touch ID or a passcode. When it's time for the user to authenticate, Touch ID will take priority if it is set up, otherwise the passcode screen is presented. Saving to the keychain will not require the user to authenticate, but retrieving the data will.

You can set a keychain item to require user authentication by providing an access control object set to `.userPresence`. If no passcode is set up then any keychain requests with `.userPresence` will fail.

```
if #available(iOS 8.0, *) {    let accessControl = SecAccessControlCreateWithFlags(kCFAllocatorDefault, kSecAttrAccessible
```



This feature is good when you want to make sure that your app is being used by the right person. For example, it would be important for the user to authenticate before being able to log in to a banking app. This will protect users who have left their device unlocked, so that the banking cannot be accessed.


Also, if you do not have a server-side component to your app, you can use this feature to perform device-side authentication instead.

For the load query, you can provide a description of why the user needs to authenticate.

```
dictionary[kSecUseOperationPrompt as String] = "Authenticate to retrieve x"
```

When retrieving the data with `SecItemCopyMatching()`, the function will show the authentication UI and wait for the user to use Touch ID or enter the passcode. Since `SecItemCopyMatching()` will block until the user has finished authenticating, you will need to call the function from a background thread in order to allow the main UI thread to stay responsive.

```
DispatchQueue.global().async {    status = SecItemCopyMatching(dictionary as CFDictionary, &passwordData)    if status
```



Again, we are setting `kSecAttrAccessControl` on the query Dictionary. You will need to remove `kSecAttrAccessible`, which was previously set in our `passwordQuery` method. Using both at once will result in an `OSStatus -50` error.

Conclusion

In this article, you've had a tour of the Keychain Services API. Along with the Data Protection API that we saw in the [previous post](#), use of this library is part of the best practices for securing data.

However, if the user does not have a passcode or Touch ID on the device, there is no encryption for either framework. Because the Keychain Services and [Data Protection APIs](#) are commonly used by iOS apps, they are sometimes targeted by attackers, especially on jailbroken devices. If your app does not work with highly sensitive information, this may be an acceptable risk. While iOS is constantly updating the security of the frameworks, we are still at the mercy of the user updating the OS, using a strong passcode, and not jailbreaking their device.

The keychain is meant for smaller pieces of data, and you may have a larger amount of data to secure that is independent of the device authentication. While iOS updates add some great new features such as the application password, you may still need to support lower iOS versions and still have strong security. For some of these reasons, you may instead want to encrypt the data yourself.

The final article in this series covers encrypting the data yourself using AES encryption, and while it's a more advanced approach, this allows you to have full control over how and when your data is encrypted.

So stay tuned. And in the meantime, check out some of our other posts on iOS app development!

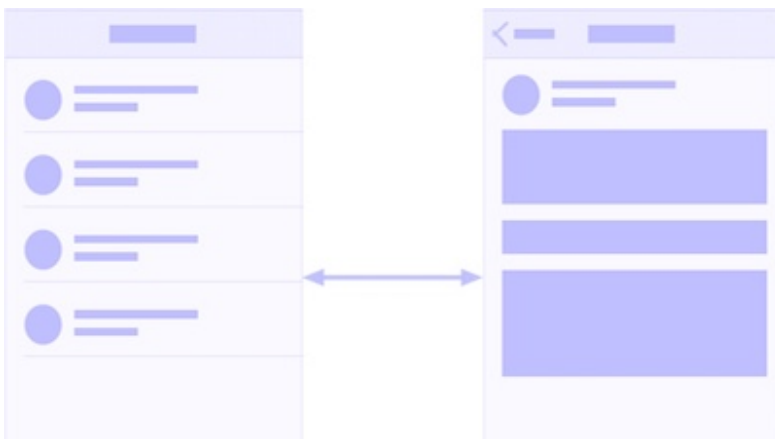


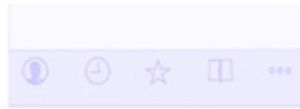
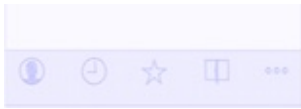


• iOS SDK
Securing Communications on iOS
Collin Stuart



• Mobile Development
Back-End as a Service for Mobile Apps
Bala Durage Sandamal Siripathi





- iOS SDK

The Right Way to Share State Between Swift View Controllers

Matteo Manferdini



- Swift

Swift From Scratch: Closures

Bart Jacobs

Original enclosures:



This file was saved from [Inoreader](#)