# Securing iOS Data at Rest: Protecting the User's Data

Via Tuts+ Code - Mobile Development by Collin Stuart

This is the first of three articles on securing user data at rest. In this post, we'll start off with the basics of protecting data on iOS so you can learn the current best practices for storing data securely with Swift.

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.

## Permissions

Before we get into storing your custom data, let's take a look at data that can be shared by system apps.

For many iOS versions, it has been required to request app permissions to use and store some of the user's private data that is external to the app, such as when saving and loading pictures to the photo library. Starting in iOS 10, any APIs that access the user's private data require you to declare that access ahead of time in your project's **info.plist** file.

There are many frameworks that can access data outside of your app, and each framework has a corresponding privacy key.

- Bluetooth Sharing: NSBluetoothPeripheralUsageDescription
- Calendar: NSCalendarsUsageDescription
- CallKit: NSVoIPUsageDescription
- Camera: NSCameraUsageDescription
- Contacts: NSContactsUsageDescription
- Health: NSHealthShareUsageDescription, NSHealthUpdateUsageDescription
- HomeKit: NSHomeKitUsageDescription
- Location: NSLocationAlwaysUsageDescription, NSLocationUsageDescription, NSLocationWhenInUseUsageDescription
- Media Library: NSAppleMusicUsageDescription
- Microphone: NSMicrophoneUsageDescription
- Motion: NSMotionUsageDescription
- Photos: NSPhotoLibraryUsageDescription
- Reminders: NSRemindersUsageDescription
- Speech Recognition: NSSpeechRecognitionUsageDescription
- SiriKit: NSSiriUsageDescription
- TV Provider: NSVideoSubscriberAccountUsageDescription

For example, here is an entry in **info.plist** to allow your app to load and store values to the calendar.

```
<key>NSCalendarsUsageDescription</key>  <string>View and add events to your calendar</string>
```

If a usage description is missing when the API tries to access the data, the app will simply crash.

## The Data Protection API

For any user data that's internal to the app, the first thing to think about is whether you need to store the information, and what data is essential to the app. Keep as much of that essential data in working memory instead of in file storage. This is especially important for any personally identifiable information.

But, if you must store data, it's a good idea to enable Apple's Data Protection.

Data Protection encrypts the contents of your app's container. It relies on the user having a passcode, and thus the security of the encryption is tied to the strength of the passcode. With Touch ID and the upgraded file system encryption introduced in iOS 10.3, the data protection system has had many improvements. You can enable data protection across your app by turning on **Data Protection** in the **Capabilities** section of your project file. This updates your provisioning profile and entitlements file to include the Data Protection capability. Data Protection offers four levels of protection, depicted by the FileProtectionType structure:

- none: no protection.
- complete: data is not accessible while the device is locked. This is the recommended setting for most applications.
- completeUnlessOpen: data is accessible when the device is unlocked, and continues to be accessible until the file is closed, even if the user locks the device. Files can also be created when the device is locked. This option is good for when you need to open a file to process and have the process continue even if the user puts the app into the background and locks the device. An example might be a job that uploads a file to a server.
- completeUntilFirstUserAuthentication: when the device is booted, files are not accessible until the user first unlocks the device. After that, files are available even when the device is locked again. The option is good for files that need to be accessed sometime later in the background when the device is locked, such as during a background fetch job.

complete is the default level. To help avoid crashes when your code tries to access data that is locked, you can register for notifications via UIApplicationProtectedDataDidBecomeAvailable and UIApplicationProtectedDataWillBecomeUnavailable to find out when the data is available.

```
NotificationCenter.default.addObserver(forName: .UIApplicationProtectedDataDidBecomeAvailable, object: nil, queue: Operatic
```

Additionally, you can also check the UIApplication.shared.isProtectedDataAvailable flag.

One important thing to keep in mind when enabling data protection is that if you are using any background services such as background fetch, that code may need access to your data in the background when the device is locked. For those files, you will need to set a protection level of completeUntilFirstUserAuthentication. You can control the protection level of each file individually when creating files and directories using the FileManager class.

```
let ok = FileManager.default.createFile(atPath: somePath, contents: nil, attributes: [FileAttributeKey.protectionKey.rawValue: F
```

You can also set the protection level when you write to a file. The Data object has a method that can write its data to a file, and you can set the protection level when you call this method.

```
let data = Data.init()  let fileURL = try! FileManager.default.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: n
```

You can also set the protection level when setting up your Core Data model.

```
let storeURL = docURL?.appendingPathComponent("Model.sqlite")  let storeOptions: [AnyHashable: Any] = [NSPersistentStor
```

To change the protection level of an existing file, use the following:

```
do  {      try FileManager.default.setAttributes([FileAttributeKey.protectionKey : FileProtectionType.complete], ofItemAtPath: pa
```

## Data Integrity

Part of protecting your stored data includes checking its integrity. It's good practice not to blindly trust the data you are loading from storage; it may have been accidentally or maliciously altered. The NSSecureCoding protocol can be used to safely load and save your data objects from storage. It will make sure the objects you load contain the expected data. If you will be saving your own object, you can conform to the secure coding protocol inside your class.

```
class ArchiveExample : NSObject, NSSecureCoding  {      var stringExample : String?      ...
```

The class must be inherited from NSObject. Then, to turn on secure coding, override the supportsSecureCoding protocol method.

```
static var supportsSecureCoding : Bool  {      get      {          return true      }  }
```

If your custom object is deserialized with init?(coder aDecoder: NSCoder), the decodeObject(forKey:) method should be replaced with decodeObject(of:forKey:), which makes sure that the correct object types are unpacked from storage.

```
required init?(coder aDecoder: NSCoder)  {      stringExample = aDecoder.decodeObject(of: NSString.self, forKey: "string_exar
```

If you are using NSKeyedUnarchiver to load data from storage, make sure to set its requiresSecureCoding property.

```
class func loadFromSavedData() -> ArchiveExample?  {      var object : ArchiveExample? = nil      let path = NSSearchPathFo
```

Turning on secure coding for your save operations will prevent you from accidentally archiving an object that does not adhere to the secure coding protocol.

```
func save()  {      let path = NSSearchPathForDirectoriesInDomains(.documentDirectory, .userDomainMask, true)[0] as String
```

Beyond NSSecureCoding, it’s always good to implement your own data validation checks upon unpacking any archive or receiving any arbitrary input in general.

## Data Trails

As iOS continues to evolve, there are always new features that have the potential to leak stored data. Starting in iOS 9, you can have your content indexed in the Spotlight search, and on iOS 10 you can expose your content to Widgets such as the Today Widget that shows up on the lock screen. Use caution if you would like to expose your content with these new features. You might end up sharing more than you planned to!

iOS 10 also adds a new Handoff feature where your copied pasteboard data is automatically shared between devices. Again, be careful not to expose any sensitive data in the pasteboard to Handoff. You can do this by marking the sensitive content as localOnly. You can also set an expiry date and time for the data.

```
let stringToCopy = "copy me to pasteboard"  let pasteboard = UIPasteboard.general  if #available(iOS 10, *)  {      let tomorrow
```

Files that are saved to the device's storage can automatically get backed up, either in iTunes or in iCloud. Even though backups can be encrypted, it's a good idea to exclude any sensitive files that don't even need to leave the device. This can be done by setting the isExcludedFromBackup flag on the file.

```
let path: String = ...  var url = URL(fileURLWithPath: path)  do  {      var resourceValues = URLResourceValues()      //or if you
```

The animation that happens when putting an app into the background is achieved by iOS taking a screenshot of your app which it then uses for the animation. When you look at the list of open apps on the app switcher, this screenshot is used there too. The screenshot gets stored on the device.

It's a good idea to hide any views revealing sensitive data so that the data isn't captured in the screenshot. To do this, set up a notification when the application is going to the background and set the hidden property for the UI elements you want to exclude. They will be hidden before iOS captures the screen. Then when coming to the foreground, you can unhide the UI elements.

```
NotificationCenter.default.addObserver(self, selector: #selector(didEnterBackground), name: .UIApplicationDidEnterBackgroun
```

Remove your notifications when the view disappears.

```
NotificationCenter.default.removeObserver(self, name: .UIApplicationDidEnterBackground, object: nil)  NotificationCenter.defau
```

Your app also has a keyboard cache for text fields that have auto-correct enabled. Text that the user types, along with newly learned words, are stored in the cache so that it is possible to retrieve various words that the user has previously entered in your application. The only way to disable the keyboard cache is to turn off the auto-correct option.

```
textField.autocorrectionType = UITextAutocorrectionType.no
textField.autocorrectionType = UITextAutocorrectionType.no
```

You should mark password fields as secure text entry. Secure text fields don't display the password or use the keyboard cache.

```
textField.isSecureTextEntry = true
```

Debug logs are saved to a file and could be retrieved for production builds of your app. Even when you're coding and debugging your app, make sure not to log sensitive information such as passwords and keys to the console. You might forget to remove that information from the logs before submitting your code to the app store! While debugging, it's safer instead to use a breakpoint to view sensitive variables.

Network connections may also get cached to storage. More information about removing and disabling the network cache can be found in the article Securing Communications on iOS.

## Destroying Data

You may already know that when a file on a computer is deleted, often the file itself is not removed; only the reference for the file is removed. To actually remove the file, you can overwrite the file with random data before removing it.

The switch to solid state drives has made it hard to guarantee the data has been destroyed, and the best way to securely delete data is open to debate. However, this tutorial would not be complete without an example of how to wipe data from storage. Because of some other debates about the Swift optimizer, and because we hope to guarantee that each byte of the file is actually being overwritten, we are implementing this function in C.

The implementation below can go inside a .c file. You will need to add the function definition or the file that contains the function into your bridging header in order to use the function from Swift. You may then want to call this function right before places where you use FileManager's removeFile methods. Perhaps you may want to implement the best practices described in this and the upcoming tutorials on an app update. You could then wipe the previous unprotected data during migration.

```
#import <string.h>  #import <sys/stat.h>  #import <unistd.h>  #import <errno.h>  #import <fcntl.h>  #import <stdio.h>    #defin
```

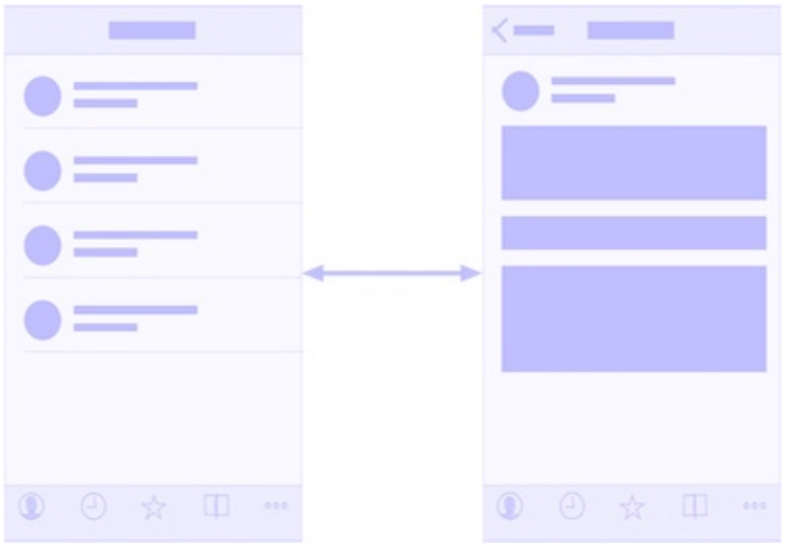
## Conclusion

In this article, you have learned about setting permissions for the data that your app has access to, as well as how to ensure basic file protection and integrity. We also looked at some ways that user data could be leaked accidentally from your app. Your users put their trust in you to protect their data. Following these best practices will help you repay that confidence.

While you're here, 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: