Securing iOS Data at Rest: Encryption

Via Tuts+ Code - Mobile Development by Collin Stuart

In this post, we'll look at advanced uses of encryption for user data in iOS apps. We'll start with a high-level look at AES encryption, and then go on to look at some examples of how to implement AES encryption in Swift.

In the last post, you learned how to store data using the keychain, which is good for small pieces of information such as keys, passwords, and certificates.

-
  iOS SDK
  Securing iOS Data at Rest: The Keychain
  Collin Stuart

If you are storing a large amount of custom data that you want to be available only after the user or device authenticates, then it's better to encrypt the data using an encryption framework. For example, you may have an app that can archive private chat messages saved by the user or private photos taken by the user, or which can store the user's financial details. In these cases, you would probably want to use encryption.

There are two common flows in applications for encrypting and decrypting data from iOS apps. Either the user is presented with a password screen, or the application is authenticated with a server which returns a key to decrypt the data.

It's never a good idea to reinvent the wheel when it comes to encryption. Therefore, we are going to use the AES standard provided by the iOS Common Crypto library.

AES

AES is a standard that encrypts data given a key. The same key used to encrypt the data is used to decrypt the data. There are different key sizes, and AES256 (256 bits) is the preferred length to be used with sensitive data.

RNCryptor is a popular encryption wrapper for iOS that supports AES. RNCryptor is a great choice because it gets you up and running very quickly without having to worry about the underlying details. It is also open source so that security researchers can analyze and audit the code.

On the other hand, if your app deals with very sensitive information and you think your application will be targeted and cracked, you may want to write your own solution. The reason for this is that when many apps use the same code, it can make the hacker's job easier, allowing them to write a cracking app that finds common patterns in the code and applies patches to them.

Keep in mind, though, that writing your own solution only slows down an attacker and prevents automated attacks. The protection you are getting from your own implementation is that a hacker will need to spend time and dedication on cracking your app alone.

Whether you choose a third-party solution or choose to roll your own, it's important to be knowledgeable about how encryption systems work. That way, you can decide if a particular framework you want to use is really secure. Therefore, the rest of this tutorial will focus on writing your own custom solution. With the knowledge you'll learn from this tutorial, you'll be able to tell if you're using a particular framework securely.

We'll start with the creation of a secret key that will be used to encrypt your data.

Create a Key

A very common error in AES encryption is to use a user's password directly as the encryption key. What if the user decides to use a common or weak password? How do we force users to use a key that is random and strong enough (has enough entropy) for encryption and then have them remember it?

The solution is *key stretching*. Key stretching derives a key from a password by hashing it many times over with a salt. The salt is just a sequence of random data, and it is a common mistake to omit this salt—the salt gives the key its vitally important entropy, and without the salt, the same key would be derived if the same password was used by someone else.

Without the salt, a dictionary of words could be used to deduce common keys, which could then be used to attack user data. This is called a "dictionary attack". Tables with common keys that correspond to unsalted passwords are used for this purpose. They're called "rainbow tables".

Another pitfall when creating a salt is to use a random number generating function that was not designed for security. An example is the rand() function in C, which can be accessed from Swift. This output can end up being very predictable!

To create a secure salt, we will use the function SecRandomCopyBytes to create cryptographically secure random bytes—which is to say, numbers that are difficult to predict.

To use the code, you'll need to add the following into your bridging header:
#import <CommonCrypto/CommonCrypto.h>

Here is the start of the code that creates a salt. We will add to this code as we go along:

```
var salt = Data(count: 8)  salt.withUnsafeMutableBytes { (saltBytes: UnsafeMutablePointer<UInt8>) -> Void in      let saltStatu
```

Now we are ready to do key stretching. Fortunately, we already have a function at our disposal to do the actual stretching: the Password-Based Key Derivation Function (PBKDF2). PBKDF2 performs a function many times over to derive the key; increasing the number of iterations expands the time it would take to operate on a set of keys during a brute force attack. It is recommended to use PBKDF2 to generate your key.

```
var setupSuccess = true  var key = Data(repeating:0, count:kCCKeySizeAES256)  var salt = Data(count: 8)  salt.withUnsafeM
```

Server-Side Key

You may be wondering now about the cases where you don't want to require users to provide a password within your app. Perhaps they are already authenticating with a single sign-on scheme. In this case, have your server generate an AES 256-bit (32 byte) key using a secure generator. The key should be different for different users or devices. On authenticating with your server, you can pass the server a device or user ID over a secure connection, and it can send the corresponding key back.

This scheme has a major difference. If the key is coming from the server, the entity that controls that server has the capacity to be able to read the encrypted data if the device or data were ever obtained. There is also the potential for the key to be leaked or exposed at a later time.

On the other hand, if the key is derived from something only the user knows—the user's password—then only the user can decrypt that data. If you are protecting information such as private financial data, only the user should be able to unlock the data. If that information is known to the entity anyway, it may be acceptable to have the server unlock the content via a server-side key.

Modes and IVs

Now that we have a key, let's encrypt some data. There are different modes of encryption, but we'll be using the recommended mode: cipher block chaining (CBC). This operates on our data one block at a time.

A common pitfall with CBC is the fact that each next unencrypted block of data is XOR'd with the previous encrypted block to make the encryption stronger. The problem here is that the first block is never as unique as all the others. If a message to be encrypted were to start off the same as another message to be encrypted, the beginning encrypted output would be the same, and that would give an attacker a clue to figuring out what the message might be.

To get around this potential weakness, we'll start the data to be saved with what is called an initialization vector (IV): a block of random bytes. The IV will be XOR'd with the first block of user data, and since each block depends on all blocks processed up until that point, it will ensure that the entire message will be uniquely encrypted, even if it has the same data as another message. In other words, identical messages encrypted with the same key will not produce identical results. So while salts and IVs are considered public, they should not be sequential or reused.

We will use the same secure SecRandomCopyBytes function to create the IV.

```
var iv = Data.init(count: kCCBlockSizeAES128)  iv.withUnsafeMutableBytes { (ivBytes : UnsafeMutablePointer<UInt8>) in
```

Putting It All Together

To complete our example, we'll use the CCCrypt function with either kCCEncrypt or kCCDecrypt. Because we are using a block cipher, if the message doesn’t fit nicely into a multiple of the block size, we will need to tell the function to automatically add padding to the end.

As usual in encryption, it is best to follow established standards. In this case, the standard PKCS7 defines how to pad the data. We tell our encryption function to use this standard by supplying the KCCOptionPKCS7Padding option. Putting it all together, here is the full code to encrypt and decrypt a string.

```
class func encryptData(_ clearTextData : Data, withPassword password : String) -> Dictionary<String, Data>  {      var setupS
```

And here is the decryption code:

```
class func decryp(fromDictionary dictionary : Dictionary<String, Data>, withPassword password : String) -> Data  {      var setu
```

Finally, here is a test to ensure that data is decrypted correctly after encryption:

```
class func encryptionTest()  {      let clearTextData = "some clear text to encrypt".data(using:String.Encoding.utf8)!      let dicti
```

In our example, we package all the necessary information and return it as a Dictionary so that all the pieces can later be used to successfully decrypt the data. You only need to store the IV and salt, either in the keychain or on your server.

Conclusion

This completes the three-part series on securing data at rest. We have seen how to properly store passwords, sensitive pieces of information, and large amounts of user data. These techniques are the baseline for protecting stored user information in your app.

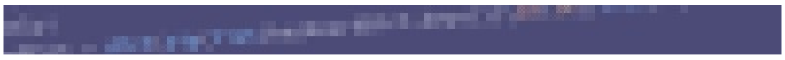
It is a huge risk when a user's device is lost or stolen, especially with recent exploits to gain access to a locked device. While many system vulnerabilities are patched with a software update, the device itself is only as secure as the user's passcode and version of iOS. Therefore it is up to the developer of each app to provide strong protection of sensitive data being stored.

All of the topics covered so far make use of Apple's frameworks. I will leave an idea with you to think about. What happens when Apple's encryption library gets attacked?

When one commonly used security architecture is compromised, all of the apps that rely on it are also compromised. Any of iOS's dynamically linked libraries, especially on jailbroken devices, can be patched and swapped with malicious ones. However, a static library that is bundled with the binary of your app is protected from this kind of attack because if you try and patch it, you end up changing the app binary. This will break the code signature of the app, preventing it from being launched. If you imported and used, for example, OpenSSL for your encryption, your app would not be vulnerable to a widespread Apple API attack. You can compile OpenSSL yourself and statically link it into your app.

So there is always more to learn, and the future of app security on iOS is always evolving. The iOS security architecture even now supports cryptographic devices and smart cards! In closing, you now know the best practices for securing data at rest, so it's up to you to follow them!

In the meantime, check out some of our other content about iOS app development and app security.



-
  Security
  How to Hack Your Own App
  Tanya Janca
-
  iOS
  Go Further With Swift: Animation, Networking, and Custom Controls
  Markus Mühlberger
-

Swift
Swift From Scratch: Delegation and Properties
Bart Jacobs

Original enclosures: