Chapter 13

Using Keychain services to secure data

Gheorghe Chesler

Although the iPhone boasts the lowest rate of security issues for a major computing platform, many developers only take advantage of a handful of these features. In this chapter, we will introduce Keychain Services, Apple’s security framework for encrypting notes, passwords and SSL certificates at the system level.

At the core of modern encryption systems is the concept of public-key cryptography. Also known as “asymmetric” encryption, this mechanism allows the use of distinct keys for encryption vs decryption: you can use the public key only to encrypt and verify signatures, and you can use the private key only to decrypt content encrypted with the public key. The keys are generated using a very large random number to provide entropy, as seen in Figure 13-1.

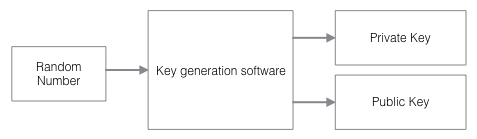


Figure 13-1: Generating the pair of private / public keys

This mechanism is used by the SSL encryption in the browser (the Transport Layer Security), S/MIME, and well known encryption platforms like GPG and PGP. The well known RSA cryptosystem uses both key distribution and secrecy (Diffie-Hellman key exchange) to secure data transfer.

If you ever generated a secure key on a linux box or even in a windows program like GnuPG or TrueCrypt, you noticed that generating the random number takes a bit of time, and that it is using some random data from the environment, like asking you to move the mouse around for a while. This ensures that the random number does not depend on any factors that can be replicated at any time by somebody else. One of the common flaws of some systems is that they rely on weak entropy to base their random number generation, and therefore the randomness is somewhat predictable and in some case reproducible.

When two parties want to communicate securely, they each send data that they encrypted with the public key of the other party. A well-known analogy is the one of Alice and Bob exchanging messages as shown in Figure 13-2.

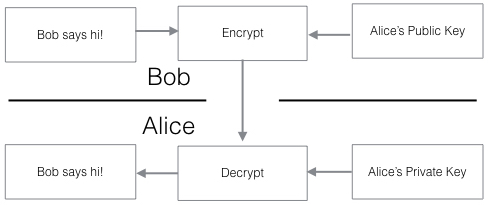


Figure 13-2: Bob and Alice exchanging messages

In the case of the Diffie-Hellman key exchange, each party uses its private key and the counterpart’s public key to generate a shared secret that can be then used to encrypt data as a symmetric cypher.

The beauty of the math behind the private/public key properties is that it allows generation of the same key starting with a combination of a public key from the other user and one’s own private key, so that no private key changes hands. We can see this depicted in Figure 13-3:

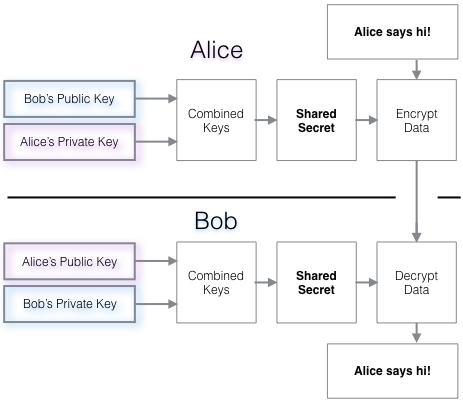


Figure 13-3: The Diffie-Hellman Key Exchange Mechanism

### Hardware Security on iOS devices

Since the person that can access the private key would be able to access the encrypted data, Apple does not provide the ability to access that key. On the current iOS devices there is a crypto chip named the “Secure Enclave”. This chip is completely isolated from the device CPU and its memory, and can be used only for the crypto operations. Each crypto chip is programmed with a unique 256 bit key for each device, and the firmware of the crypto chip is secured with a signing key known only to Apple.

That way Apple should not be able to decrypt your password, since they do not know the UID. It is not known whether Apple has the ability to extract the UID from a crypto chip. According to Apple, the “Secure Enclave” crypto chip maintains its security even if the device is jailbroken.

The crypto chip does not store any of your passwords, but instead provides a way to encrypt and decrypt any content and hide the encryption keys from the outside world.

When you enter a password in the device, the Apple Key Derivation function combines the password with the UID then applies a slow derivation function (PBKDF2-AES), and the iteration count is chosen so that it takes about 80ms on the particular device.

If one were to attempt to hack the device password, that attempt would have to run on the device itself, since only the device knows the UID. The number of combinations required to crack a large enough password would take a considerable amount of time: Apple advertises that time to over 5 years for a random 6 character password.

A device is considerably better secured if you configure it to erase itself after a number of invalid password retries. Note that retries with a thumbprint don’t count, but the device will stop accepting a thumbprint-based login after a few bad tries. The use of a complex password is also recommended.

### Securing the File Data

In addition to the hardware encryption features built into iOS devices, Apple uses a technology called Data Protection to further protect data stored in flash memory on the device.

Data Protection allows the device to respond to common events such as incoming phone calls, but also enables a high level of encryption for user data.

Key system apps, such as Messages, Mail, Calendar, Contacts, Photos, and Health data values use Data Protection by default, and third-party apps installed on iOS 7 or later receive this protection automatically.

Data Protection is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies built into each iOS device. Data Protection is controlled on a per-file basis by assigning each file to a class; accessibility is determined by whether the class keys have been unlocked.

Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the "per-file" key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.

The per-file key is wrapped with one of several class keys, depending on the circumstances under which the file should be accessible. Like all other wrappings, this is performed using NIST AES key wrapping, per RFC 3394. The wrapped per-file key is stored in the file’s metadata.

When a file is opened, its metadata is decrypted with the file system key, revealing the wrapped per-file key and a notation on which class protects it. The per-file key is unwrapped with the class key, then supplied to the hardware AES engine, which decrypts the file as it is read from flash memory.

The metadata of all files in the file system is encrypted with a random key, which is created when iOS is first installed or when a user wipes the device data. The file system key is stored in Effaceable Storage.

Since it’s stored on the device, this key is not used to maintain the confidentiality of data; instead, it’s designed to be quickly erased on demand (by the user, with the “Erase all content and settings” option, or by a user or administrator issuing a remote wipe command from a mobile device management (MDM) server, Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.

The content of a file is encrypted with a per-file key, which is wrapped with a class key and stored in a file’s metadata, which is in turn encrypted with the file system key. The class key is protected with the hardware UID and, for some classes, the user’s passcode. This hierarchy provides both flexibility and performance. For example, changing a file’s class only requires rewrapping its per-file key, and a change of passcode just rewraps the class key. Figure 13-4 visualises this process.

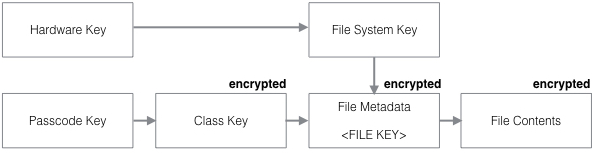


Figure 13-4: Overview of Apple key derivation and encryption (iOS Security Guide)

### The Apple Keychain

The Apple Keychain is Apple’s platform “vault” for your application’s sensitive data. Available on every device in the Apple landscape, the Apple Keychain allows applications to store in a secure way different bits of information, which can be retrieved next time the application starts. This allows for a certain level of security from attempts to access private bits of information by accessing the application as it is stored on the device.

Naturally, once secured data is accessed by the application, the data is still vulnerable to device memory snoop attacks, but at least the application and its stored data that you back up from your device does not store the very sensitive bits in a vulnerable format. This is not as much an issue, because accessing the device memory does require direct access to the device from a trusted computer.

You might have noticed that every time you connect it to another PC or Mac, your iOS device already alerts you whether you trust that device - this is considered a reasonable first line of defense. Furthermore, when backing up a device, sensitive data like the keychain does not get backed up.

Even when you back up the device in secure mode, you have the ability to save the entry in your keychain with the ThisDeviceOnly option that ensures that your data is saved only on the current device, and that way it does not get backed up.

One aspect of the Keychain implementation on iOS devices is that whatever you store in it, is locked to that device or its backup. On an OSX device (Macbook, Mac Pro, or iMac), the keychain is stored under the user’s ~/Library/Keychains/ path, and the files there can be copied to a different Mac and imported in its keychain, if you know the password to the keychains.

The Keychain from an iOS device can only be backed up when you back up the iOS device in encrypted mode, so it is fairly inaccessible to the normal user.

To save and access data in the keychain, Apple provides to any application the API called Apple Keychain Services.

### The Apple Keychain Services

The Apple Keychain Services is an API that allows applications to store key-value pairs in encrypted format on your iOS device. An app can save sensitive bits of data in the keychain, such as passwords, tokens, and keys. The keychain items can be shared between apps from the same developer.

The actual storage is a single SQLite database, accessed by a system daemon: this daemon queries for entries that correspond to the “keychain-access-group” of the app and the “application-identifier” entitlement.

### Components of a keychain item

Along with the access group, each keychain item contains administrative metadata (such as “created” and “last updated” timestamps).

It also contains SHA-1 hashes of the attributes used to query for the item (such as the account and server name) to allow lookup without decrypting each item. And finally, it contains the encryption data, which includes the following:

- Version number

- Access control list (ACL) data

- Value indicating which protection class the item is in

- Per-item key wrapped with the protection class key

- Dictionary of attributes describing the item (as passed to SecItemAdd), encoded as a binary plist and encrypted with the per-item key

For more detailed information, the iOS security guide is found here:

<https://www.apple.com/business/docs/iOS_Security_Guide.pdf>

### Implementing Keychain Services for storing passwords

For example, you can store a user name and password or other credentials that you use to access a resource, after you successfully authenticated to the resource. That way, when you reload your application, you can retrieve those credentials from the keychain, instead of the less secure local storage. If the application is restored from an unencrypted backup to a different device, it will not have the entry in the keychain to go back to, so your credentials are safe.

In the following example we will store a string that could be a password or a token. We use the SecItemAdd() function available in the Security framework to save data for a given key. Looking at the function signature in the Keychain Services Reference, we see the following:

func SecItemAdd(\_ attributes: CFDictionary!, \_ result: UnsafeMutablePointer<Unmanaged<AnyObject>?>) -> OSStatus

The attributes parameter is a dictionary that describes the data to be inserted, the security class of the item to be inserted, as well as a variety of optional parameters that describe the return type for the result.

To add multiple items to a keychain at once use the kSecUseItemList key with an array of items as its value. This is only supported for non-password items.

This is an example of the code needed to insert a password in the keychain (Listing 13-1):

Listing 13-1. Inserting a password in the keychain

let key = “password”

let value = “my Password”

if let data = value.dataUsingEncoding(NSUTF8StringEncoding) {

let query = [

(kSecClass as String) : kSecClassGenericPassword,

(kSecAttrAccount as String) : key,

(kSecValueData as String) : value

]

SecItemAdd(query as CFDictionaryRef, nil)

}

In our example we assume that they key did not exist before. In a practical implementation we would want to delete an existing key before inserting it with a given value. We can express this a bit cleaner by using a NSMutableDictionary object that allows us to pass the keys and values as lists (Listing 13-2):

Listing 13-2. Inserting keys and values as lists

let key = “password”

let value = “my Password”

if let data = value.dataUsingEncoding(NSUTF8StringEncoding) {

let query = NSMutableDictionary(

objects: [ kSecClassGenericPassword, key, value ],

forKeys: [ kSecClass, kSecAttrAccount, kSecValueData ]

) as CFDictionaryRef

SecItemAdd(query, nil)

}

We see in the examples above that the value string was converted to a NSData object, which is necessary since the value could be unicode.

### Retrieving data from Keychain Services

To read from the keychain services, we use the function SecItemCopyMatching() that returns one or more keychain items that match a search query, or copies attributes of specific keychain items.

The function signature is as follows:

func SecItemCopyMatching(query: CFDictionary!, \_ result:[UnsafeMutablePointer](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_UnsafeMutablePointer_Structure/index.html#//apple_ref/swift/struct/s:VSs20UnsafeMutablePointer)<[Unmanaged](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_Unmanaged_Structure/index.html#//apple_ref/swift/struct/s:VSs9Unmanaged)<[AnyObject](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_AnyObject_Protocol/index.html#//apple_ref/swift/intf/s:PSs9AnyObject)>?>) -> OSStatus{...}

The OSStatus return value is a result code. See Keychain Services Result Codes in the Security Framework Reference for more detail on these codes. This is buried in the documentation and you can get to it selecting the path: Security Framework Reference / Keychain Services Reference / Search Results Constants.

By default, this function returns only the first match found. To obtain more than one matching item at a time, specify the search key kSecMatchLimit with a value greater than 1. The result will be an object of type CFArrayRef containing up to that number of matching items.

To retrieve any data for the given key, we need to pass to *result* a reference to an object that will be populated by the function. Just like for the insert operation, the *query* is a dictionary that specifies the attribute name, its type, and a couple parameters that enable the data return and the number of items to retrieve.

Obviously the response is optional, as the key could not exist, and there is a chance that the operation will fail so we also need to check for the call return code that takes the value noErr when the operation succeeded(Listing 13-3).

Listing 13-3. Checking for the call return codes

let key = “password”

var password : String!;

var response: Unmanaged<AnyObject>?

let query = NSMutableDictionary(

objects: [ kSecClassGenericPassword, key, kCFBooleanTrue, kSecMatchLimitOne ],

forKeys: [ kSecClass, kSecAttrAccount, kSecReturnData, kSecMatchLimit ]

) as CFDictionaryRef

let status = SecItemCopyMatching(query, &response)

if status == noErr && response != nil {

if let data = response!.takeRetainedValue() as? NSData {

password = NSString(data: data, encoding: NSUTF8StringEncoding)

}

}

In this example we read and assign a simple string to the password variable. As mentioned before, for data types other than passwords we can store multiple values.

### Invalidating Keychain Service records

To delete individual items, we use the SecItemDelete function. This function has the signature:

func SecItemDelete(*query*: CFDictionary!) -> OSStatus

The query is a dictionary containing an item class specification and optional attributes for controlling the search. As always, for details on the currently defined search attributes you should consult the Security Framework Reference, as the API might change with future versions of Swift or iOS.

Just like for the other calls the OSStatus return value is a result code.

Let’s see how deleting a password would look like(Listing 13-4):

Listing 13-4. Deleting a password from the keychain

let key = “password”

let query = NSMutableDictionary(

objects: [ kSecClassGenericPassword, key ],

forKeys: [ kSecClass, kSecAttrAccount ]

) as CFDictionaryRef

let status = SecItemDelete(query)

if status == noErr {

print(“password deleted”)

}

If we want to delete all passwords, we can use the same function call, this time with no value for the kSecAttrAccount. This will have the effect of deleting all items of the class kSecClassGenericPassword (Listing 13-5):

Listing 13-4. Deleting all passwords from the keychain

let query = NSMutableDictionary(

objects: [ kSecClassGenericPassword ],

forKeys: [ kSecClass ]

) as CFDictionaryRef

let status = SecItemDelete(query)

if status == noErr {

print(“all passwords deleted”)

}

Of course, this could be compressed as:

if SecItemDelete([(kSecClass as String) : kSecClassGenericPassword]) == noErr {

print("all passwords deleted")

}

### Setting up an application to test Keychain Services

We will use a one-page application just like we did for the FitBit application, and we will re-use some elements, particularly the UILogger.swift. To handle the Keychain API calls, we set up a class named Keychain,that will be described in detail later on.

To be able to use the Keychain services, we need to link the binary with the Security.framework library, just like in the screenshot below (Figure 13-5):

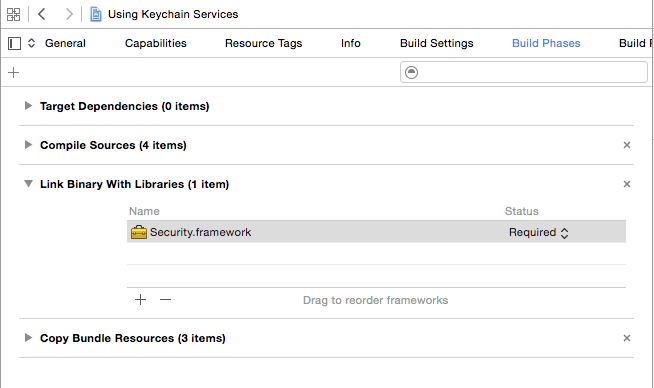


Figure 13-5: Linking the app with the Security Framework

#### The View Controller

The view controller code is very simple. We created a text input field where one can enter the string to be saved to the keychain. In this example we save to the keychain a field named “token”. To make things more interesting, if no value was provided, we populate it with the current date/time: this will help with debugging and verifying that the application reads correctly from the keychain after it was restarted (Listing 13-5).

import UIKit

class ViewController: UIViewController {

@IBOutlet var clearButton : UIButton!

@IBOutlet var clearKeychainButton : UIButton!

@IBOutlet var saveButton : UIButton!

@IBOutlet var readButton : UIButton!

@IBOutlet var textArea : UITextView!

@IBOutlet var textField : UITextField!

var logger: UILogger!

override func viewDidLoad() {

super.viewDidLoad()

// Do any additional setup after loading the view, typically from a nib.

logger = UILogger(out: textArea)

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

// Dispose of any resources that can be recreated.

}

@IBAction func saveToKeychain() {

var value = textField.text!;

if value.isEmpty {

let dateFormatter:NSDateFormatter = NSDateFormatter()

dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"

value = dateFormatter.stringFromDate(NSDate())

}

logger.logEvent("Save to keychain: \(value)")

Keychain.set("token", value: value)

}

@IBAction func readFromKeychain() {

// the response is an Optional<NSData> so it needs to be unwrapped

if let value = Keychain.get("token") {

logger.logEvent("Read from keychain: \(value)")

}

else {

logger.logEvent("No value found in the keychain")

}

}

@IBAction func clickClearButton() {

logger.clear()

}

@IBAction func clickClearKeychainButton() {

Keychain.clear()

logger.logEvent("The keychain data has been cleared")

}

}

Listing 13-5. The ViewController.swift file

We can see that we created a textArea for the inline logging of the activity, a textField that will be used to input the inserted value, as well as a few buttons. Figure 13-6 shows a detail view of the Main.storyboard that shows how the buttons and fields are wired.

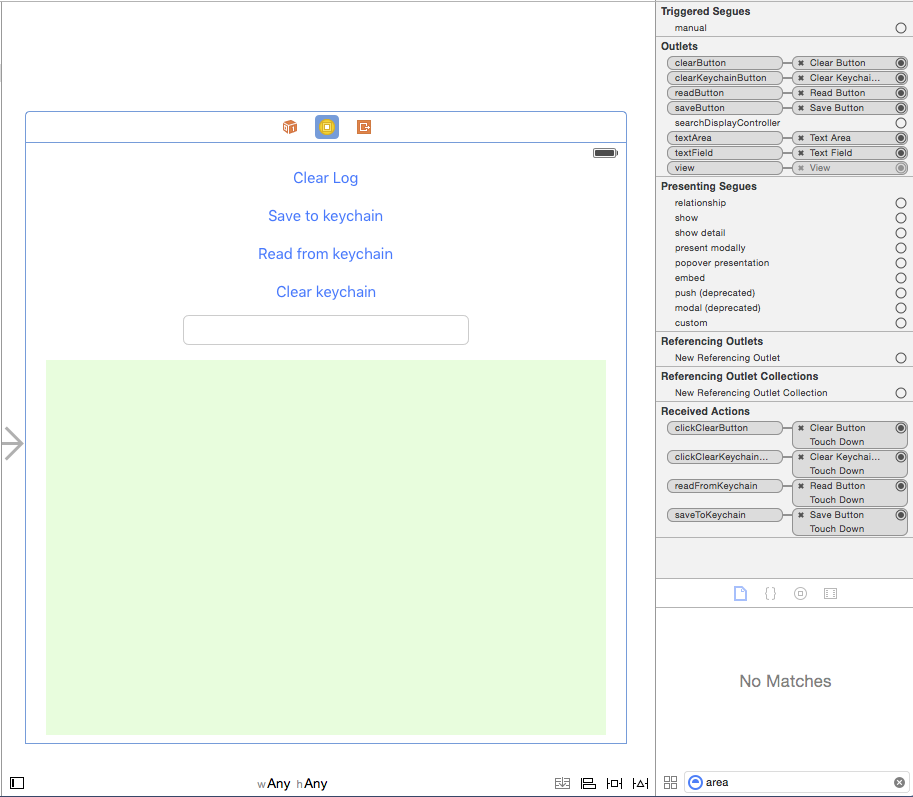


Figure 13-6: The application Storyboard

#### The Keychain class

We create a few private class methods to encapsulate the query constructors. This will be saved as the Keychain.swift file.

For the create/update query, we take a key name and the value to be inserted, as shown in Listing 13-6. We have to convert the values to String before assigning them to the dictionary – this used to work without the need of an explicit conversion in earlier versions of Swift:

private class func updateQuery(key: String, value: NSData) -> CFDictionaryRef {

return NSMutableDictionary(

objects: [ kSecClassGenericPassword, key, value ],

forKeys: [ String(kSecClass), String(kSecAttrAccount), String(kSecValueData) ]

)

}

Listing 13-6. The updateQuery() function

The deleteQuery looks very similar to the updateQuery but it takes only a key String (Listing 13-7):

private class func deleteQuery(key: String) -> CFDictionaryRef {

return NSMutableDictionary(

objects: [ kSecClassGenericPassword, key ],

forKeys: [ String(kSecClass), String(kSecAttrAccount) ]

)

}

Listing 13-7. The deleteQuery() function

The set function uses the two queries above to first delete an existing key, then add the key with the new value (Listing 13-8):

public class func set(key: String, value: String) -> Bool {

if let data = value.dataUsingEncoding(NSUTF8StringEncoding) {

SecItemDelete(deleteQuery(key))

return SecItemAdd(updateQuery(key, value: data), nil) == noErr

}

return false

}

Listing 13-8. The set() function

To read existing entries, we use the functions shown in Listing 13-9. The get() is rather simple, since it relies on the getData() to do the heavy lifting, and it only returns a NSString object. For the getData() we need to do a fancy dance to extract the status and the response:

public class func get(key: String) -> NSString? {

let query = searchQuery(key)

if let data = getData(query) {

return NSString(data: data, encoding: NSUTF8StringEncoding)

}

return nil

}

public class func getData(query: CFDictionaryRef) -> NSData? {

var response: AnyObject?

let status = withUnsafeMutablePointer(&response) {

SecItemCopyMatching(query, UnsafeMutablePointer($0))

}

return status == noErr && response != nil

? response as! NSData?

: nil

}

Listing 13-9. The get() and getData() functions in the Keychain.swift file

We observe that the response comes(eventually) as NSData and has to be converted back to a String. The getData() function can be used on its own to fetch a value given a well formatted query(not necessarily password type). It returns an NSData object that can be converted to any given data type we expect to read back from the keychain.

The code for the entire Keychain class is shown in the Figure 13-10 below. We leave as an exercise for the user to create a setData() function that would handle a variety of item types. Keep in mind that you have to use the same value for kSecClass when deleting the entries before inserting them.

import Foundation

import Security

public class Keychain {

public class func set(key: String, value: String) -> Bool {

if let data = value.dataUsingEncoding(NSUTF8StringEncoding) {

SecItemDelete(deleteQuery(key))

return SecItemAdd(updateQuery(key, value: data), nil) == noErr

}

return false

}

public class func get(key: String) -> NSString? {

let query = searchQuery(key)

if let data = getData(query) {

return NSString(data: data, encoding: NSUTF8StringEncoding)

}

return nil

}

public class func getData(query: CFDictionaryRef) -> NSData? {

var response: AnyObject?

let status = withUnsafeMutablePointer(&response) {

SecItemCopyMatching(query, UnsafeMutablePointer($0))

}

return status == noErr && response != nil

? response as! NSData?

: nil

}

public class func delete(key: String) -> Bool {

return SecItemDelete( deleteQuery(key) ) == noErr

}

public class func clear() -> Bool {

if SecItemDelete([(kSecClass as String) : kSecClassGenericPassword]) == noErr {

print("all passwords deleted")

}

return SecItemDelete( clearQuery() ) == noErr

}

private class func updateQuery(key: String, value: NSData) -> CFDictionaryRef {

return NSMutableDictionary(

objects: [ kSecClassGenericPassword, key, value ],

forKeys: [ String(kSecClass), String(kSecAttrAccount), String(kSecValueData) ]

)

}

private class func deleteQuery(key: String) -> CFDictionaryRef {

return NSMutableDictionary(

objects: [ kSecClassGenericPassword, key ],

forKeys: [ String(kSecClass), String(kSecAttrAccount) ]

)

}

private class func searchQuery(key: String) -> CFDictionaryRef {

return NSMutableDictionary(

objects: [ kSecClassGenericPassword, key, kCFBooleanTrue, kSecMatchLimitOne ],

forKeys: [ String(kSecClass), String(kSecAttrAccount), String(kSecReturnData), String(kSecMatchLimit) ]

)

}

private class func clearQuery() -> CFDictionaryRef {

return [ (kSecClass as String) : kSecClassGenericPassword ]

}

}

Listing 13-10. The Keychain.swift file

#### Running the demo application

There is no magic to this application: it simply saves some password-type values and retrieves them from the keychain, with the output shown in the screenshot below:

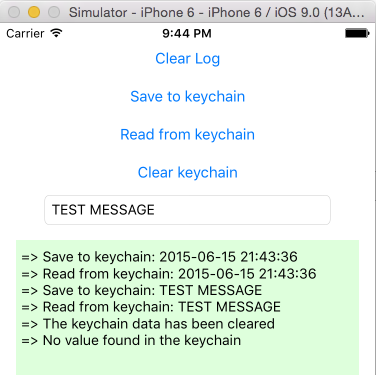


Figure 13-7: Application output

# Summary

In this chapter you learned some of the basics of key encryption, some of the particularities of the Apple’s implementation of Keychain services, and finally how enable your app to interact with the Keychain Services.