Keychain Services Programming Guide

Developer

Contents

Figures and Listings

Introduction

Keychain Services provides secure storage of passwords, keys, certificates, and notes for one or more users. A user can unlock a keychain with a single password, and any Keychain Services–aware application can then use that keychain to store and retrieve passwords. *Keychain Services Programming Guide* contains an overview of Keychain Services, discusses the functions and data structures that are most commonly used by developers, and provides examples of how to use Keychain Services in your own applications.

This document concentrates on the use of Keychain Services to store and retrieve passwords. You should read this document if your application needs to handle passwords for:

- Multiple users—for example, an email or scheduling server that has to authenticate many users
- Multiple servers—for example, a banking or insurance application, which might have to exchange information with more than one secure database server
- A user who needs to enter passwords—for example, a web browser, which can use a keychain to store the passwords a user needs for multiple secure web sites

You do not need any special knowledge of authentication schemes to use this document, but you should be familiar with the use and storage of passwords.

Organization of This Document

This document contains the following chapters:

Keychain Services Concepts (page 6) provides an overview of Keychain Services and explains what keychains are and how they are used.

OS X Keychain Services Tasks (page 29) contains sample code and detailed explanations of the most commonly-used Keychain Services functions.

Glossary (page 49) defines new terms introduced in this book.

See Also

The following documents provide references to Apple's keychain-related APIs.

- *Keychain Services Reference* documents all the functions and structures provided in the Keychain Services API. These include the functions and structures used in this document, plus others used primarily by keychain administrative applications such as the Keychain Access application.
- For more information about storing and retrieving certificates and keys, see *Certificate, Key, and Trust Services Reference*.

Keychain services and other OS X security APIs are built on the open source Common Data Security Architecture (CDSA) and its programming interface, Common Security Services Manager (CSSM). For more information about the CSSM API, see the following document:

- *Common Security: CDSA and CSSM, version 2 (with corrigenda)* from The Open Group (http://www.opengroup.org/security/cdsa.htm).

Keychain Services Concepts

Computer users typically have to manage multiple accounts that require logins with *user IDs* and *passwords*. Secure FTP servers, AppleShare servers, database servers, secure websites, instant messaging accounts, and many other services require authentication before they can be used. Users often respond to this situation by making up very simple, easily remembered passwords, by using the same password over and over, or by writing passwords down where they can be easily found. Any of these cases compromises security.

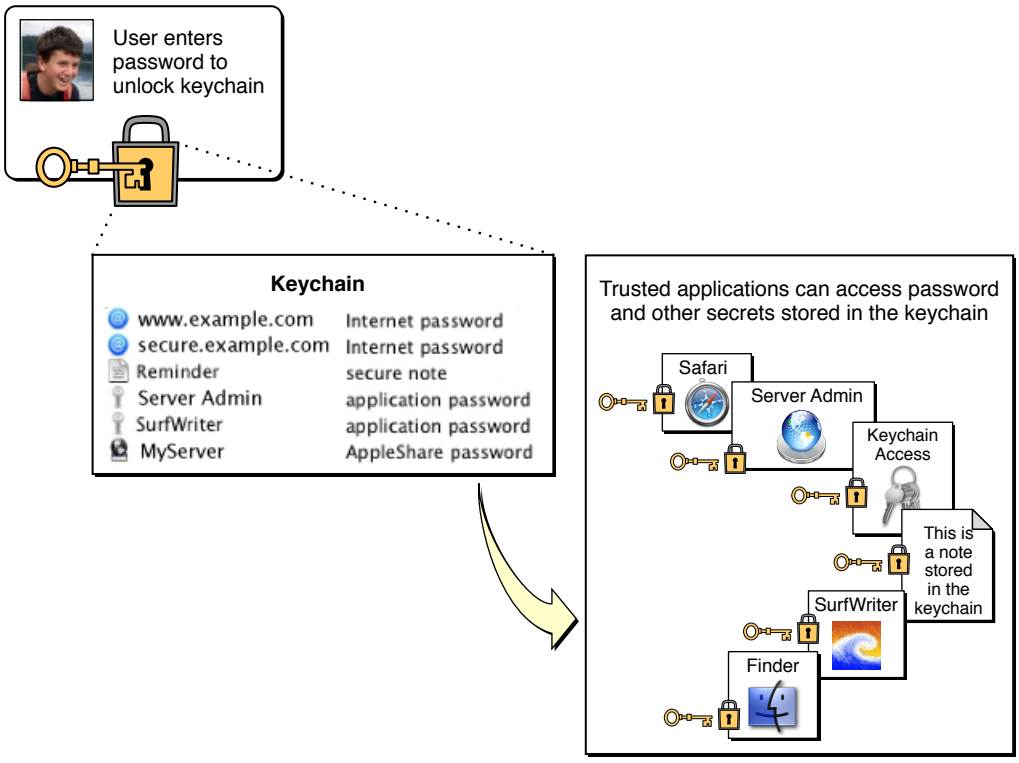
The Keychain Services API provides a solution to this problem. By making a single call to this API, an application can store login information on a keychain where the application can retrieve the information—also with a single call—when needed. A *keychain* is an encrypted container that holds passwords for multiple applications and secure services. Keychains are *secure storage* containers, which means that when the keychain is *locked*, no one can access its protected contents. In OS X, users can unlock a keychain—thus providing trusted applications access to the contents—by entering a single master password. In iOS, each application always has access to its own keychain items; the user is never asked to unlock the keychain. Whereas in OS X any application can access any keychain item provided the user gives permission, in iOS an application can access only its own keychain items.

> **Note:** On iPhone, Keychain rights depend on the provisioning profile used to sign your application. Be sure to consistently use the same provisioning profile across different versions of your application.

From the user's point of view, a keychain provides *transparent authentication*; that is, (after unlocking the keychain if in OS X) the user does not have to log in separately to any services whose passwords are stored in the keychain. In OS X, the user has only to enter one password once to access any number of applications, servers, websites, and so on. In iOS, the user need not even enter the keychain password. Figure 1-1 (page 7) shows the relationship between the user, the keychain, and the password-protected services. For iOS the first step (unlocking the keychain) is omitted. In iOS, an application can always access its own keychain items, but not items created by any other application.

Note: In addition to passwords, keychains can store cryptographic keys, certificates, and (in OS X) text strings (notes). Notes are generally entered by the user with the Keychain Access utility. Most applications that use Keychain Services need to store or retrieve passwords, and that is the subject of this document. If you need to store or retrieve keys or certificates, see *Certificate, Key, and Trust Services Reference*.

Figure 1-1 Accessing password-protected services using a keychain in OS X



By default, in OS X each login account has one keychain (for a new login on OS X v10.3, this keychain is named `login.keychain`); however, a user or application can create as many keychains as desired. The *login keychain* is automatically *unlocked* during login if it has the same password as the user's login account password. When first created, the login keychain is also the *default keychain*. The default keychain is used to store newly created keychain items unless a different keychain is specified in the function call; certain other Keychain Services functions also use the default keychain when no other keychain is specified. The user can use the Keychain Access utility to designate another keychain as the default; however, the login keychain doesn't change. In iOS, the situation is simpler: There is a single keychain accessible to applications. Although it stores the keychain items of all the applications on the system, an application can access only its own keychain items (with the possible exception of a keychain item for which the application that created it obtained a persistent reference).

Keychain Services and CDSA

In OS X, *Keychain Services* and other security APIs are built on the open source *Common Data Security Architecture (CDSA)* and its programming interface, *Common Security Services Manager (CSSM)*.

The OS X Keychain Services API provides functions to perform most of the operations needed by applications, including creating, deleting, and modifying keychains and keychain items, controlling access to keychain items, finding keychain items, and retrieving attributes and data from items. However, the underlying CSSM API provides more capabilities that might be of interest to specialty applications, such as applications designed to administer the security of a computer or network. For this reason, the Keychain Services API includes a number of functions that return or create CSSM structures so that, if you are familiar with the CSSM API, you can move freely back and forth between Keychain Services and CSSM.

For more information about the CSSM API, see *Common Security: CDSA and CSSM, version 2 (with corrigenda)* from The Open Group (http://www.opengroup.org/security/cdsa.htm).

In iOS, the Keychain Services API provides all the functions available to manipulate keychain items.

Structure of a Keychain

Each keychain can contain any number of keychain items. Each *keychain item* contains data plus a set of attributes. For a keychain item that needs protection, such as a password or private *key* (a string of bytes used to encrypt or decrypt data), the data is encrypted and protected by the keychain. For keychain items that do not need protection, such as certificates, the data is not encrypted.

The attributes associated with a keychain item depend on the class of the item; the item classes most used by applications (other than the Finder and the Keychain Access application in OS X) are *Internet passwords* and *generic passwords*. As you might expect, Internet passwords include attributes for such things as security domain, protocol type, and path. The passwords or other *secrets* stored as keychain items are encrypted. In OS X, encrypted items are inaccessible when the keychain is locked; if you try to access the item while the keychain is locked, Keychain Services displays a dialog prompting the user for the keychain password. The attributes are not encrypted, however, and can be read at any time, even when the keychain is locked. In iOS, your application always has access to its own keychain items.

In OS X, each protected keychain item (and the keychain itself) contains access control information in the form of an opaque data structure called an *access object*. The access object contains one or more *access control list* (ACL) entries for that item. Each ACL entry has a list of one or more *authorization tags* specifying operations that can be done with that item, such as decrypting or authenticating. In addition, each ACL entry has a list of *trusted applications* that can perform the specified operations without authorization from the user.

Keychain Access Controls

> **iOS:** The iOS gives an application access to only its own keychain items. The keychain access controls discussed in this section do not apply to iOS.

When an OS X application attempts to access a keychain item for a particular purpose (such as to sign a document), the system looks at each ACL entry for that item to determine whether the application should be allowed access. If there is no ACL entry for that operation, then access is denied and it is up to the calling application to try something else or to notify the user. If there are any ACL entries for the operation, then the system looks at the list of trusted applications of each ACL entry to see if the calling application is in the list. If it is—or if the ACL specifies that all applications are allowed access—then access is permitted without confirmation from the user (as long as the keychain is unlocked). If there is an ACL entry for the operation but the calling application is not in the list of trusted applications, then the system prompts the user for the keychain password before permitting the application to access the item.

The Keychain Services API provides functions to create, delete, read, and modify ACL entries. Because an ACL entry is always associated with an access object, when you modify an ACL entry, you are modifying the access object as well. Therefore, there is no need for a separate function to write a modified ACL entry back into the access object.

There can be any number of ACL entries in an access object. If two or more of the ACL entries are for the same operation, there is no way to predict the order in which they will be evaluated.

Although the Keychain Services API lets you create ACL entries and add them to access objects, it is limited in the ways you can configure the authorizations and lists of trusted applications. If you want to implement access controls that go beyond the complexity and structure supported by Keychain Services, you can use the CSSM API to create a set of CSSM data structures and then call the `SecAccessCreateFromOwnerAndACL` function. Although you can't use Keychain Services functions to extract information from or modify an access object made in this way, Keychain Services recognizes the access object as a CSSM data structure and uses it directly in its underlying calls to CSSM.

Trusted Applications

An ACL entry either permits all applications to perform the operations specified by its list of authorization tags or it contains a list of trusted applications. The trusted application list is actually a list of trusted application objects (objects with the opaque type `SecTrustedApplicationRef`). In addition to serving as a reference to the application, a trusted application object includes data that uniquely identifies the application, such as a cryptographic hash. The system can use this data to verify that the application has not been altered since the trusted application object was created. For example, when a trusted application requests access to an item

in the keychain, the system checks this data before granting access. Although you can extract this data from the trusted application object for storage or for transmittal to another location (such as over a network), this data is in a private format; there is no supported way to read or interpret it.

You can use the `SecTrustedApplicationCreateFromPath` function to create a trusted application object. The trusted application is the binary form of the application that's on the disk at the moment the trusted application object is created. If an application listed as a trusted application for a keychain item is modified in any way, the system does not recognize it as a trusted application. Instead, the user is prompted for confirmation when that application attempts to access the keychain item.

When a program attempts to access a keychain item for which it is not recognized as a trusted application, the system displays a confirmation dialog (Figure 1-2). The confirmation dialog has three buttons: Deny, Allow Once, and Always Allow. If the user clicks Always Allow, the system creates a trusted application object for the application and adds it to the access object for that keychain item.

Figure 1-2 Application access confirmation dialog

To make the launching of programs more efficient, the system prebinds executables to dynamically loaded libraries (DLLs). When a user updates a DLL, the system automatically changes the executables of all the programs that use that library, a process referred to as reprebinding. Reprebinding a trusted application therefore causes the application to no longer match the version represented in the application hash. In OS X v10.2 and earlier, the next time the application tries to use a protected keychain item, a confirmation dialog appears. When the user clicks Always Allow, the system adds it to the access object as a new trusted application. Starting with OS X v10.3, on the other hand, the system maintains a database that keeps track of applications that were reprebound so that in most cases no confirmation dialog appears.

In OS X v10.3 and later, in addition, the system keeps track of applications that were updated by Software Update. When the updated application attempts to access a protected keychain item, the system either recognizes that it is the same application or, if necessary, displays a confirmation dialog saying that the

application has been changed and asking the user whether to treat it like the older version (Figure 1-3). If the user clicks the Change All button, the system makes the change only to the current user's keychains, not to the keychains of other users on the system.

Figure 1-3 Changed software confirmation dialog

ACL Entries

The `SecAccessCreate` function creates an access object with three ACL entries. The first, referred to as *owner access*, determines who can modify the access object itself. By default, there are no trusted applications for owner access; the user is always prompted for permission if someone tries to change access controls. The second ACL entry is for operations considered safe, such as encrypting data. This ACL entry applies to all applications. The third ACL entry is for operations that should be restricted, such as decrypting, signing, deriving keys, and exporting keys. This ACL entry applies to the trusted applications listed as input to the function.

In addition to providing a list of trusted applications to `SecAccessCreate`, you specify a CFString that describes the keychain item. This is the name of the item that appears in dialogs (see Figure 1-2 (page 10) or Figure 1-3 (page 11), for example). This is not necessarily the same name as appears for the item in the Keychain Access utility.

You use other functions in Keychain Services to modify any of these default ACL entries or to add additional ACL entries to the access object (see *Keychain Services Reference*). These functions let you retrieve all the ACL entries for an access object, modify ACL entries, and create new ones. For each ACL entry, you can specify trusted applications, the item descriptor string, a list of authorization tags, and a prompt selector bit. If you set the prompt selector bit, the user is prompted for the keychain password each time a nontrusted application attempts to access the item, even if the keychain is already unlocked. Figure 1-4 shows the dialog; compare this figure with Figure 1-2 (page 10), which is the dialog that appears if the prompt selector bit is not set. If the user clicks Always Allow in response to this dialog, the application is added to the access object as a trusted application and the dialog does not appear again. This bit is clear by default—you must set it explicitly for any

ACL entry for which you want this extra protection. There is one exception to this rule: the Keychain Access application always requires a password to display the secret of a keychain item unless the Keychain Access application itself is included in the trusted application list.

Figure 1-4 Unlock keychain dialog to confirm access

As noted earlier, because an ACL entry is always associated with an access object, when you modify an ACL entry, you are modifying the access object as well. Therefore, there is no need for a separate function to write a modified ACL entry back into the access object. However, if you modify an access object, you must write the new version of the access object to the keychain item before the keychain item can use it.

Keeping Your OS X Keychain Data Secure

In OS X, to provide security for the passwords and other valuable secrets stored in your keychain, you should adopt at least the following measures:

- Set your keychain to lock itself when not in use: in the Keychain Access utility, choose Edit > Change Settings for Keychain, and check both Lock checkboxes.
- Use a different password for your keychain than your login password: In Keychain Access utility, choose Edit > Change Password to change your keychain's password. Click the lock icon in the Change Password dialog to get the password assistant, which tells you how secure your password is and can suggest passwords. Be sure to pick one you can remember—don't write it down anywhere.

iPhone Keychain Backups

In iOS, an application always has access to its own keychain items and does not have access to any other application's items. The system generates its own password for the keychain, and stores the key on the device in such a way that it is not accessible to any application. When a user backs up iPhone data, the keychain data

is backed up but the secrets in the keychain remain encrypted in the backup. The keychain password is not included in the backup. Therefore, passwords and other secrets stored in the keychain on the iPhone cannot be used by someone who gains access to an iPhone backup. For this reason, it is important to use the keychain on iPhone to store passwords and other data (such as cookies) that can be used to log into secure web sites.

Keychain Services Ease of Use

Although the structure of the keychain provides a great deal of power and flexibility, it also introduces a level of complexity that most application writers would rather not have to deal with. Fortunately, you don't. The Keychain Services API provides a handful of high-level functions that handle all of the keychain operations most applications will ever need to perform.

To create a keychain item and add it to a keychain in OS X, for example, you call one of two functions, depending on whether you want to add an Internet password or some other type of password: `SecKeychainAddInternetPassword` or `SecKeychainAddGenericPassword`. In your function call, you pass only those attributes for which there is no obvious default value. For example, you must specify the name of the service and the user's account name, but you do not have to specify the creation date and creator, because the function can figure those out by itself. You also pass the data (usually a password) that you want to store in the keychain. You do not even have to specify a keychain; if you pass `NULL` for the keychain reference, the function uses the default keychain. If the keychain is locked, the function automatically displays a dialog prompting the user to unlock the keychain. The function also creates the access object for you, listing the calling application as the only trusted application.

The iOS Keychain Services API uses a different paradigm (see the following section, iOS Keychain Services Search Dictionaries). This API has a single function (`SecItemAdd`) for adding an item to a keychain.

Similarly, when your Mac app needs access to a user's password, you call either `SecKeychainFindInternetPassword` or `SecKeychainFindGenericPassword`, passing some attributes (such as the name of the service and the user's account name) so that the function can find the password that you need. If you pass `NULL` for the keychain to search, the function searches the keychains in the *default keychain search list*, which is the same as the keychain list in the Keychain Access utility. The function prompts the user to unlock the keychain if necessary. In iOS, you call the `SecItemCopyMatching` function to find a keychain item owned by your application. In this case there's only one keychain and the user is never prompted to unlock it.

OS X Keychain Services Tasks (page 29) contains detailed information on how to add basic Keychain Services features to your application.

iOS Keychain Services Search Dictionaries

In iOS, Keychain Services uses a key-value dictionary to specify the attributes of the keychain item that you want search for or create. For general discussions of key-value pairs and dictionaries, see *Collections Programming Topics* and *Collections Programming Topics for Core Foundation*.

A typical search dictionary consists of:

- The class key-value pair, which specifies the class of items (for example, Internet passwords or cryptographic keys) for which to search.
- One or more key-value pairs that specify the attribute data (such as label or creation date) to be matched.
- One or more search key-value pairs, which specify values that further refine the search, such as issuing certificates or email addresses to match.
- A return-type key-value pair, specifying the type of results you want (for example, a dictionary or a persistent reference).

Which attributes can be specified depends on the class of the item for which you wish to search. For example, if you specify a value of `kSecClassGenericPassword` for the `kSecClass` key, then you can specify values for creation date and modification date, but not for subject or issuer (which are used with certificates).

For example, if you wanted to perform a case-insensitive search for the password for an Apple Store account with the account name of "ImaUser", you could use the following dictionary with the `SecItemCopyMatching` function:

Type of key	Key	Value
Item class	`kSecClass`	`kSecClassGenericPassword`
Attribute	`kSecAttrAccount`	`"ImaUser"`
Attribute	`kSecAttrService`	`"Apple Store"`
Search attribute	`kSecMatchCaseInsensitive`	`kCFBooleanTrue`
Return type	`kSecReturnData`	`kCFBooleanTrue`

The `kSecReturnData` key causes the function to return the keychain item's data—in this case, the password. If instead you want a dictionary of attribute keys and values (so you can determine, for example, the creation date of the item), you use the `kSecReturnAttributes` return-type key with a value of `kCFBooleanTrue`.

OS X Keychain Services Advanced Features

> **iOS:** The iOS provides only a single keychain and does not require interaction with the user to unlock the keychain or create new keychains. The Keychain Services advanced features discussed in this section do not apply to iOS.

OS X Keychain Services allows you to create keychains, manipulate elements within a keychain, and manage collections of keychains. In most cases, a keychain-aware application does not have to do any keychain management and only has to call a few functions to store or retrieve passwords. By default, Keychain Services automatically interacts with the user to create or unlock a keychain when necessary. For example, if you are trying to add a password to a keychain but no keychain exists, Keychain Services prompts the user as shown in Figure 1-5. When the user clicks Reset To Defaults, Keychain Services creates a keychain with the name `login.keychain` and the same password as the user's login account.

Figure 1-5 Prompting the user to create a keychain

If the keychain is locked when you try to save or retrieve a password, Keychain Services prompts the user to unlock the keychain, as shown in Figure 1-6. Once the user enters the correct password, your application can access the keychain. Keychain Services also displays dialogs to confirm that the user wants the application to access the keychain (see Figure 3-2 (page 47)) and for other reasons.

Figure 1-6 The Unlock Keychain dialog box

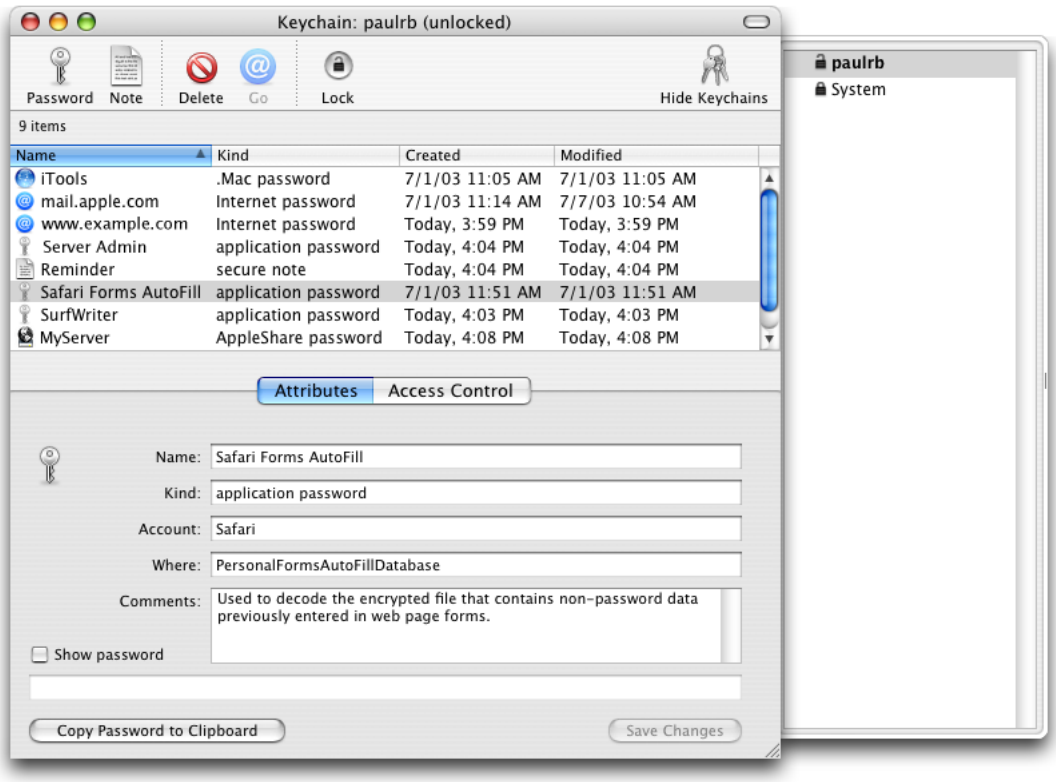
In addition to storing and retrieving passwords, there are a few operations that some applications might need to perform, including:

- Disabling or enabling Keychain Services functions that display a user interface; for example, a server might want to suppress the Unlock Keychain dialog box and unlock the keychain itself instead.
- Unlocking a locked keychain when the user is unable to do so, as for an unattended server.
- Adding trusted applications to the access object of a keychain item if, for example, a server application wants to let an administration application have access to its passwords.

OS X Keychain Services includes many other functions that can be used to manipulate keychains, keychain items, attributes, access objects, and ACLs. You can even register a callback function so that your application is called when a keychain event (such as unlocking the keychain) occurs. However, these functions are generally needed only by programs designed specifically to administer keychains. OS X includes a keychain administration

program, called *Keychain Access* (Figure 1-7). With this utility, a user can lock or unlock keychains, create new keychains, change the default keychain, add and delete keychain items, change the values of the attributes of keychain items, and see or change the data stored in a keychain item.

Figure 1-7 The Keychain Access application



To duplicate the capabilities of this program in OS X, you would need to be familiar with the open source Common Data Security Architecture (CDSA) and its API, the Common Security Services Manager (CSSM), in addition to all of the functions provided by Keychain Services. For more information about the CSSM API, see *Common Security: CDSA and CSSM, version 2 (with corrigenda)* from The Open Group (http://www.opengroup.org/security/cdsa.htm).

iOS Keychain Services Tasks

This chapter describes and illustrates the use of basic Keychain Services functions that are available in both iOS and OS X.

The functions described in this chapter enable you to:

- Add an item to a keychain
- Find an item in a keychain
- Get the attributes and data in a keychain item
- Change the attributes and data in a keychain item

Keychain Services Concepts (page 6) provides an introduction to the concepts and terminology of Keychain Services. For detailed information about all Keychain Services functions, see *Keychain Services Reference*.

> **Note:** On iPhone, Keychain rights depend on the provisioning profile used to sign your application. Be sure to consistently use the same provisioning profile across different versions of your application.

Adding Keychain Services to Your Application

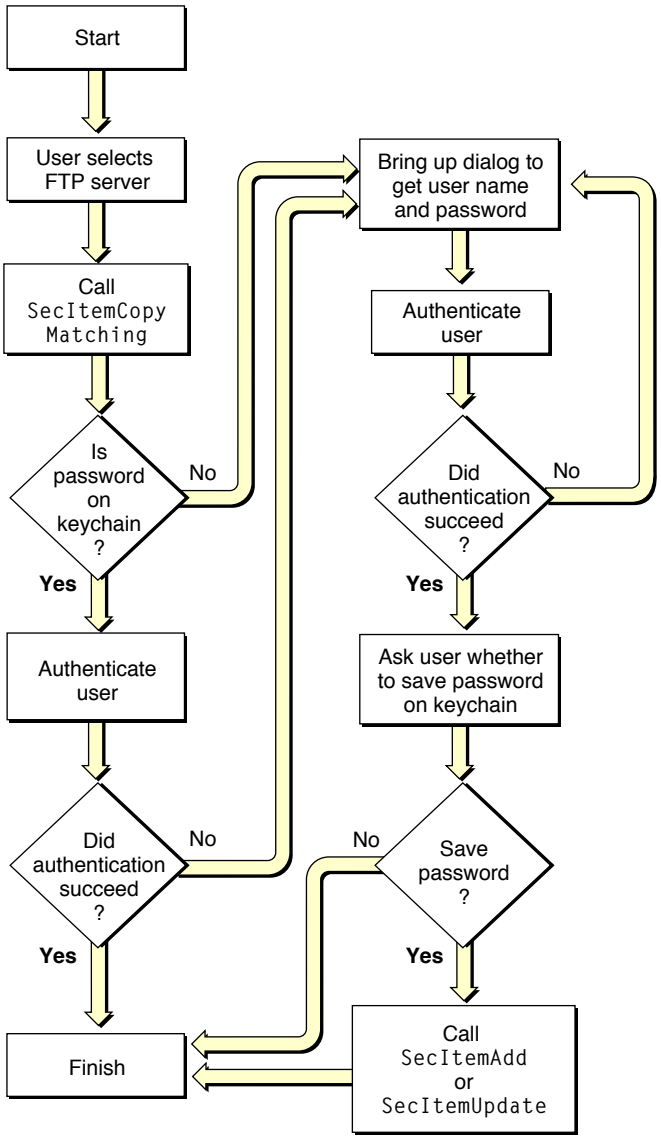
Most iOS applications need to use Keychain Services only to add a new password to a keychain, modify an existing keychain item, or retrieve a password when needed. Keychain Services provides the following functions for accomplishing these tasks:

- `SecItemAdd` to add an item to a keychain
- `SecItemUpdate` to modify an existing keychain item
- `SecItemCopyMatching` to find a keychain item and extract information from it

You use Internet passwords for accessing servers and websites over the Internet, and generic passwords for any other password-protected service (such as a database or scheduling application). In iOS Keychain Services, certificates, keys, and identities are stored and retrieved in exactly the same way as passwords, except that they have different attributes.

Figure 2-1 shows a flowchart of how an application might use these functions to gain access to an Internet FTP server.

Figure 2-1 Accessing an Internet server using iPhone Keychain Services



The user of the application starts by selecting a File Transfer Protocol (FTP) server. The application calls `SecItemCopyMatching`, passing it a dictionary containing attributes that identify the keychain item. If the password is on the keychain, the function returns the password to the application, which sends it to the FTP server to authenticate the user. If the authentication succeeds, the routine is finished. If the authentication fails, the application displays a dialog to request the user name and password.

If the password is not on the keychain, then `SecItemCopyMatching` returns the `errSecItemNotFound` result code. In this case as well, the application displays a dialog to request the user name and password. (This dialog should also include a Cancel button, but that choice was omitted from the figure to keep the flowchart from becoming overly complex.)

Having obtained the password from the user, the application proceeds to authenticate the user to the FTP server. When the authentication has succeeded, the application can assume that the information entered by the user was valid. The application then displays another dialog asking the user whether to save the password on the keychain. If the user selects No, then the routine is finished. If the user selects Yes, then the application calls the `SecItemAdd` function (if this is a new keychain item) or the `SecItemUpdate` function (to update an existing keychain item) before ending the routine.

Listing 2-1 shows how a typical application might use Keychain Services functions to get and set passwords for generic items. You can get and set keychain item attributes (such as user name or service name) using these same functions.

Listing 2-1 Getting and setting passwords in iOS Keychain Services

```
#import <Foundation/Foundation.h>
#import <Security/Security.h>

//Define an Objective-C wrapper class to hold Keychain Services code.
@interface KeychainWrapper : NSObject {
    NSMutableDictionary        *keychainData;
    NSMutableDictionary        *genericPasswordQuery;
}

@property (nonatomic, strong) NSMutableDictionary *keychainData;
@property (nonatomic, strong) NSMutableDictionary *genericPasswordQuery;

- (void)mySetObject:(id)inObject forKey:(id)key;
- (id)myObjectForKey:(id)key;
- (void)resetKeychainItem;

@end

/* ********************************************************************** */
//Unique string used to identify the keychain item:
```

```
static const UInt8 kKeychainItemIdentifier[]    = "com.apple.dts.KeychainUI\0";

@interface KeychainWrapper (PrivateMethods)



//The following two methods translate dictionaries between the format used by
// the view controller (NSString *) and the Keychain Services API:
- (NSMutableDictionary *)secItemFormatToDictionary:(NSDictionary
*)dictionaryToConvert;
- (NSMutableDictionary *)dictionaryToSecItemFormat:(NSDictionary
*)dictionaryToConvert;
// Method used to write data to the keychain:
- (void)writeToKeychain;

@end

@implementation KeychainWrapper

//Synthesize the getter and setter:
@synthesize keychainData, genericPasswordQuery;

- (id)init
{
    if ((self = [super init])) {

        OSStatus keychainErr = noErr;
        // Set up the keychain search dictionary:
        genericPasswordQuery = [[NSMutableDictionary alloc] init];
        // This keychain item is a generic password.
        [genericPasswordQuery setObject:(__bridge id)kSecClassGenericPassword
                                forKey:(__bridge id)kSecClass];
        // The kSecAttrGeneric attribute is used to store a unique string that is
 used
        // to easily identify and find this keychain item. The string is first
        // converted to an NSData object:
        NSData *keychainItemID = [NSData dataWithBytes:kKeychainItemIdentifier
```

```
                                    length:strlen((const char
*)kKeychainItemIdentifier)];
        [genericPasswordQuery setObject:keychainItemID forKey:(__bridge
id)kSecAttrGeneric];
        // Return the attributes of the first match only:
        [genericPasswordQuery setObject:(__bridge id)kSecMatchLimitOne
forKey:(__bridge id)kSecMatchLimit];
        // Return the attributes of the keychain item (the password is
        //  acquired in the secItemFormatToDictionary: method):
        [genericPasswordQuery setObject:(__bridge id)kCFBooleanTrue
                                 forKey:(__bridge id)kSecReturnAttributes];

        //Initialize the dictionary used to hold return data from the keychain:
        CFMutableDictionaryRef outDictionary = nil;
        // If the keychain item exists, return the attributes of the item:
        keychainErr = SecItemCopyMatching((__bridge
CFDictionaryRef)genericPasswordQuery,
                                          (CFTypeRef *)&outDictionary);
        if (keychainErr == noErr) {
            // Convert the data dictionary into the format used by the view
controller:
           self.keychainData = [self secItemFormatToDictionary:(__bridge_transfer
 NSMutableDictionary *)outDictionary];
        } else if (keychainErr == errSecItemNotFound) {
            // Put default values into the keychain if no matching
            // keychain item is found:
            [self resetKeychainItem];
            if (outDictionary) CFRelease(outDictionary);
        } else {
            // Any other error is unexpected.
            NSAssert(NO, @"Serious error.\n");
            if (outDictionary) CFRelease(outDictionary);
        }
    }
    return self;
}
```

```
// Implement the mySetObject:forKey method, which writes attributes to the keychain:
- (void)mySetObject:(id)inObject forKey:(id)key
{
    if (inObject == nil) return;
    id currentObject = [keychainData objectForKey:key];
    if (![currentObject isEqual:inObject])
    {
        [keychainData setObject:inObject forKey:key];
        [self writeToKeychain];
    }
}

// Implement the myObjectForKey: method, which reads an attribute value from a
dictionary:
- (id)myObjectForKey:(id)key
{
    return [keychainData objectForKey:key];
}

// Reset the values in the keychain item, or create a new item if it
// doesn't already exist:

- (void)resetKeychainItem
{
    if (!keychainData) //Allocate the keychainData dictionary if it doesn't exist
 yet.
    {
        self.keychainData = [[NSMutableDictionary alloc] init];
    }
    else if (keychainData)
    {
        // Format the data in the keychainData dictionary into the format needed
for a query
        //  and put it into tmpDictionary:
        NSMutableDictionary *tmpDictionary =
```

```
                            [self dictionaryToSecItemFormat:keychainData];
        // Delete the keychain item in preparation for resetting the values:
       OSStatus errorcode = SecItemDelete((__bridge CFDictionaryRef)tmpDictionary);
        NSAssert(errorcode == noErr, @"Problem deleting current keychain item."
);
    }

    // Default generic data for Keychain Item:
    [keychainData setObject:@"Item label" forKey:(__bridge id)kSecAttrLabel];
    [keychainData setObject:@"Item description" forKey:(__bridge
id)kSecAttrDescription];
    [keychainData setObject:@"Account" forKey:(__bridge id)kSecAttrAccount];
    [keychainData setObject:@"Service" forKey:(__bridge id)kSecAttrService];
    [keychainData setObject:@"Your comment here." forKey:(__bridge
id)kSecAttrComment];
    [keychainData setObject:@"password" forKey:(__bridge id)kSecValueData];
}

// Implement the dictionaryToSecItemFormat: method, which takes the attributes
that
// you want to add to the keychain item and sets up a dictionary in the format
// needed by Keychain Services:
- (NSMutableDictionary *)dictionaryToSecItemFormat:(NSDictionary
*)dictionaryToConvert
{
    // This method must be called with a properly populated dictionary
    // containing all the right key/value pairs for a keychain item search.

    // Create the return dictionary:
    NSMutableDictionary *returnDictionary =
                    [NSMutableDictionary
dictionaryWithDictionary:dictionaryToConvert];

    // Add the keychain item class and the generic attribute:
    NSData *keychainItemID = [NSData dataWithBytes:kKeychainItemIdentifier
                            length:strlen((const char *)kKeychainItemIdentifier)];
```

```
    [returnDictionary setObject:keychainItemID forKey:(__bridge id)kSecAttrGeneric];
    [returnDictionary setObject:(__bridge id)kSecClassGenericPassword
forKey:(__bridge id)kSecClass];

    // Convert the password NSString to NSData to fit the API paradigm:
    NSString *passwordString = [dictionaryToConvert objectForKey:(__bridge
id)kSecValueData];
    [returnDictionary setObject:[passwordString
dataUsingEncoding:NSUTF8StringEncoding]
                                                        forKey:(__bridge
id)kSecValueData];
    return returnDictionary;
}

// Implement the secItemFormatToDictionary: method, which takes the attribute
dictionary
//  obtained from the keychain item, acquires the password from the keychain, and
//  adds it to the attribute dictionary:
- (NSMutableDictionary *)secItemFormatToDictionary:(NSDictionary
*)dictionaryToConvert
{
    // This method must be called with a properly populated dictionary
    // containing all the right key/value pairs for the keychain item.

    // Create a return dictionary populated with the attributes:
    NSMutableDictionary *returnDictionary = [NSMutableDictionary

dictionaryWithDictionary:dictionaryToConvert];

    // To acquire the password data from the keychain item,
   // first add the search key and class attribute required to obtain the password:
    [returnDictionary setObject:(__bridge id)kCFBooleanTrue forKey:(__bridge
id)kSecReturnData];
    [returnDictionary setObject:(__bridge id)kSecClassGenericPassword
forKey:(__bridge id)kSecClass];
    // Then call Keychain Services to get the password:
    CFDataRef passwordData = NULL;
    OSStatus keychainError = noErr; //
```

```
    keychainError = SecItemCopyMatching((__bridge CFDictionaryRef)returnDictionary,
                                        (CFTypeRef *)&passwordData);
    if (keychainError == noErr)
    {
        // Remove the kSecReturnData key; we don't need it anymore:
        [returnDictionary removeObjectForKey:(__bridge id)kSecReturnData];

       // Convert the password to an NSString and add it to the return dictionary:
        NSString *password = [[NSString alloc] initWithBytes:[(__bridge_transfer
NSData *)passwordData bytes]
                  length:[(__bridge NSData *)passwordData length]
encoding:NSUTF8StringEncoding];
        [returnDictionary setObject:password forKey:(__bridge id)kSecValueData];
    }
    // Don't do anything if nothing is found.
    else if (keychainError == errSecItemNotFound) {
            NSAssert(NO, @"Nothing was found in the keychain.\n");
            if (passwordData) CFRelease(passwordData);
    }
    // Any other error is unexpected.
    else
    {
        NSAssert(NO, @"Serious error.\n");
        if (passwordData) CFRelease(passwordData);
    }

    return returnDictionary;
}

// Implement the writeToKeychain method, which is called by the mySetObject routine,
//   which in turn is called by the UI when there is new data for the keychain.
This
//   method modifies an existing keychain item, or--if the item does not already
//   exist--creates a new keychain item with the new attribute value plus
//  default values for the other attributes.
```

```
- (void)writeToKeychain
{
    CFDictionaryRef attributes = nil;
    NSMutableDictionary *updateItem = nil;

    // If the keychain item already exists, modify it:
    if (SecItemCopyMatching((__bridge CFDictionaryRef)genericPasswordQuery,
                            (CFTypeRef *)&attributes) == noErr)
    {
        // First, get the attributes returned from the keychain and add them to
the
        // dictionary that controls the update:
       updateItem = [NSMutableDictionary dictionaryWithDictionary:(__bridge_transfer
 NSDictionary *)attributes];

        // Second, get the class value from the generic password query dictionary
 and
        // add it to the updateItem dictionary:
        [updateItem setObject:[genericPasswordQuery objectForKey:(__bridge
id)kSecClass]
                                                            forKey:(__bridge
id)kSecClass];

        // Finally, set up the dictionary that contains new values for the
attributes:
        NSMutableDictionary *tempCheck = [self
dictionaryToSecItemFormat:keychainData];
        //Remove the class--it's not a keychain attribute:
        [tempCheck removeObjectForKey:(__bridge id)kSecClass];

        // You can update only a single keychain item at a time.
        OSStatus errorcode = SecItemUpdate(
            (__bridge CFDictionaryRef)updateItem,
            (__bridge CFDictionaryRef)tempCheck);
        NSAssert(errorcode == noErr, @"Couldn't update the Keychain Item." );
    }
    else
```

```
    {
        // No previous item found; add the new item.
       // The new value was added to the keychainData dictionary in the mySetObject
 routine,
      // and the other values were added to the keychainData dictionary previously.
        // No pointer to the newly-added items is needed, so pass NULL for the
second parameter:
        OSStatus errorcode = SecItemAdd(
          (__bridge CFDictionaryRef)[self dictionaryToSecItemFormat:keychainData],
           NULL);
        NSAssert(errorcode == noErr, @"Couldn't add the Keychain Item." );
        if (attributes) CFRelease(attributes);
    }
}


@end
```

This example follows the same general sequence that was shown in Figure 2-1 (page 19). In this sample, the generic attribute is used to create a unique string that can be used to easily identify the keychain item. If you wish, you can use standard attributes such as the service name and user name for the same purpose.

For More Information

For an additional iPhone Keychain example, see the *GenericKeychain* sample code project.

OS X Keychain Services Tasks

This chapter describes and illustrates the use of basic Keychain Services functions in OS X. For the use of Keychain Services in iOS, see iOS Keychain Services Tasks (page 18).

The functions described in this chapter enable you to:

- Add a password to a keychain
- Find a password in a keychain
- Get the attributes and data in a keychain item
- Change the attributes and data in a keychain item
- Unlock a keychain
- Lock a keychain
- Suppress the automatic display of the create keychain or open keychain dialogs
- Add trusted applications to a keychain item's access control list

Keychain Services Concepts (page 6) provides an introduction to the concepts and terminology of Keychain Services. For detailed information about all Keychain Services functions, see *Keychain Services Reference*.

Adding Simple Keychain Services to Your Application

Most applications need to use Keychain Services only to add a new password to a keychain or retrieve a password when needed. Keychain Services provides the following pairs of functions for accomplishing these tasks:

- `SecKeychainAddInternetPassword` and `SecKeychainFindInternetPassword` (for Internet passwords)
- `SecKeychainAddGenericPassword` and `SecKeychainFindGenericPassword` (for generic passwords

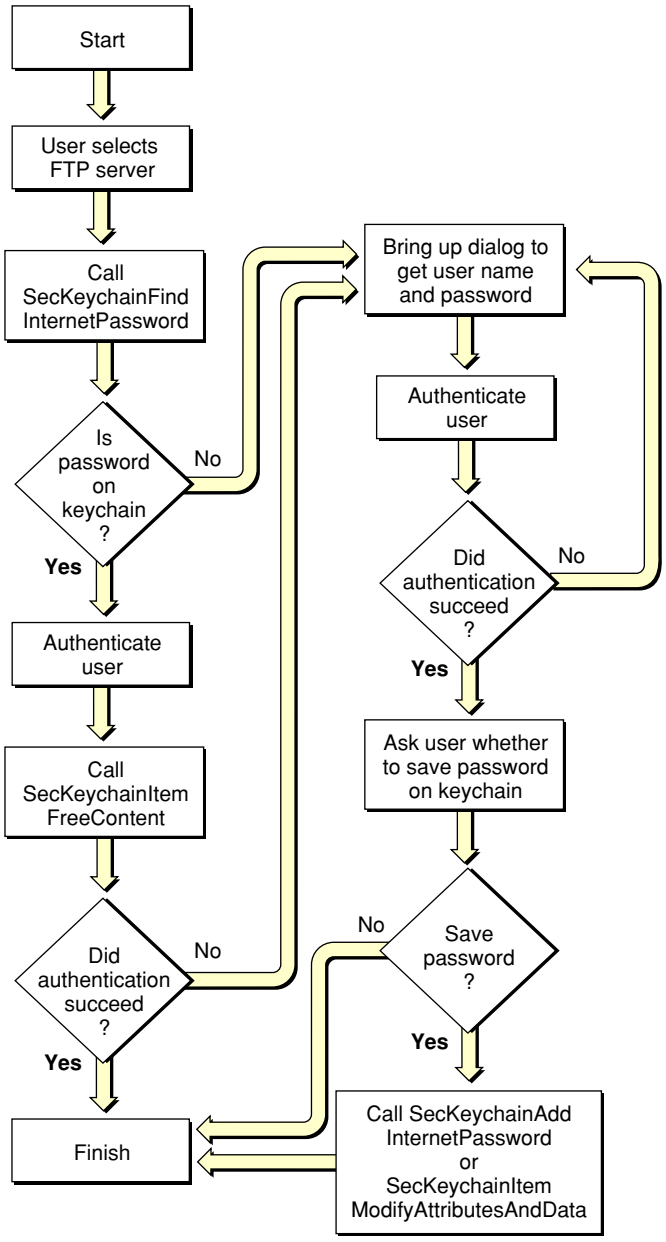
You use Internet passwords for accessing servers and websites over the Internet, and generic passwords for any other password-protected service (such as a database or scheduling application). AppleShare passwords (that is, keychain items with a class code of `kSecAppleSharePasswordItemClass`) are stored as generic passwords.

> **Note:** AppleShare passwords created with the *Keychain Manager* function `KCAddAppleSharePassword` are stored as Internet password items. You can use the `SecKeychainAddInternetPassword` function to store an AppleShare password, but it will appear in the Keychain Access utility as an Internet password. To create a keychain item with a class code of `kSecAppleSharePasswordItemClass`, you must use the lower-level API described in *Keychain Services Reference*.

The "find" functions retrieve information (attributes or secure data) from an item in the keychain. The "add" functions add an item to a keychain. These functions call other Keychain Services functions to accomplish their tasks. Because Keychain Services allocates the buffers in which the item data and attributes are returned, you must call `SecKeychainItemFreeContent` to free these buffers after using one of the find functions to retrieve attributes or secure data from a keychain item. (If Keychain Services does not find the item or fails to return any data for some other reason, it does not allocate any buffers and you should not call the `SecKeychainItemFreeContent` function.)

Figure 3-1 shows a flowchart of how an application might use these functions to gain access to an Internet FTP server.

Figure 3-1 Accessing an Internet server using OS X Keychain Services



The user of the application starts by selecting a File Transfer Protocol (FTP) server. The application calls `SecKeychainFindInternetPassword`, passing it attributes that identify the service and the user to seek. If the password is on the keychain, the function returns the password to the application, which sends it to the FTP server to authenticate the user. The application then calls `SecKeychainItemFreeContent` to free the

data buffer allocated for the password (note that you should not call this function if no data is returned). If the authentication succeeds, the routine is finished. If the authentication fails, the application displays a dialog to request the user name and password.

If the password is not on the keychain, then `SecKeychainFindInternetPassword` returns the `errSecItemNotFound` result code. In this case as well, the application displays a dialog to request the user name and password. (This dialog should also include a Cancel button, but that choice was omitted from the figure to keep the flowchart from becoming overly complex.)

Having obtained the password from the user, the application proceeds to authenticate the user to the FTP server. When the authentication has succeeded, the application can assume that the information entered by the user was valid. The application then displays another dialog asking the user whether to save the password on the keychain. If the user selects No, then the routine is finished. If the user selects Yes, then the application calls the `SecKeychainAddInternetPassword` function (if this is a new keychain item), or the `SecKeychainItemModifyAttributesAndData` function (to update an existing keychain item) before ending the routine.

If there is no keychain, the `SecKeychainFindInternetPassword` or `SecKeychainAddInternetPassword` function displays a dialog allowing the user to "reset to defaults" (see Figure 1-5 (page 15)), which creates a new keychain named `login.keychain` with the user's login account password. If the keychain is locked, the function displays a dialog requesting the user to enter a password to unlock the keychain (Figure 1-6 (page 16)). The user can cancel the operation at this time as well.

Listing 3-1 shows how a typical application might use Keychain Services functions to get and set passwords for generic items. You can get and set keychain item attributes (such as user name or service name) using these same functions; see Listing 3-2 (page 37) for an example.

Listing 3-1 Getting and setting passwords in OS X Keychain Services

```
#include <CoreFoundation/CoreFoundation.h>
#include <Security/Security.h>
#include <CoreServices/CoreServices.h>


//Call SecKeychainAddGenericPassword to add a new password to the keychain:
OSStatus StorePasswordKeychain (void* password,UInt32 passwordLength)
{
    OSStatus status;
```

```
    status = SecKeychainAddGenericPassword (
                NULL,           // default keychain
                10,             // length of service name
                "SurfWriter",   // service name
                10,             // length of account name
                "MyUserAcct",   // account name
                passwordLength, // length of password
                password,       // pointer to password data
                NULL            // the item reference
    );
    return (status);
}

//Call SecKeychainFindGenericPassword to get a password from the keychain:
OSStatus GetPasswordKeychain (void **passwordData,UInt32 *passwordLength,
                                                SecKeychainItemRef *itemRef)
{
    OSStatus status1 ;


    status1 = SecKeychainFindGenericPassword (
                NULL,           // default keychain
                10,             // length of service name
                "SurfWriter",   // service name
                10,             // length of account name
                "MyUserAcct",   // account name
                passwordLength,  // length of password
                passwordData,   // pointer to password data
                itemRef         // the item reference
    );
    return (status1);
}

//Call SecKeychainItemModifyAttributesAndData to change the password for
// an item already in the keychain:
```

```
OSStatus ChangePasswordKeychain (SecKeychainItemRef itemRef)
{
    OSStatus status;
    void * password = "myNewP4sSw0rD";

    size_t passwordLength = strlen(password);
    assert(passwordLength <= 0xffffffff);

    status = SecKeychainItemModifyAttributesAndData (
                itemRef,         // the item reference
                NULL,            // no change to attributes
                (UInt32)passwordLength,  // length of password
                password         // pointer to password data
    );
    return (status);
}


/* ****************************************************************************** */

int tryIt(void)
{
    OSStatus status;

    void * myPassword = "myP4sSw0rD";

    size_t myPasswordLength = strlen(myPassword);
    assert(myPasswordLength <= 0xffffffff);

    void *passwordData = NULL; // will be allocated and filled in by
                               //SecKeychainFindGenericPassword
    SecKeychainItemRef itemRef = NULL;

    UInt32 passwordLength = 0;
```

```
// Call SecKeychainFindGenericPassword
status = GetPasswordKeychain (&passwordData,&passwordLength,&itemRef);

if (status == noErr)       //If call was successful, authenticate user
                                //and continue.
{
    //Free the data allocated by SecKeychainFindGenericPassword:
    status = SecKeychainItemFreeContent (
             NULL,           //No attribute data to release
             passwordData    //Release data buffer allocated by
             //SecKeychainFindGenericPassword
    );
} else if (status == errSecItemNotFound) { //Is password on keychain?
    /*
       If password is not on keychain, display dialog to prompt user
       for name and password.

       Authenticate user.

       If unsuccessful, prompt user again for name and password.
       If successful, ask user whether to store new password on keychain.

       If no, return.
       If yes, store password by calling SecKeychainAddGenericPassword.
    */
    status = StorePasswordKeychain (myPassword,(UInt32)myPasswordLength);

    return (status);
}

/*
   If password is on keychain, authenticate user.

   If authentication succeeds, return.
```

```
        If authentication fails, prompt user for new user name and password and
        authenticate again.

        If unsuccessful, prompt again.
        If successful, ask whether to update keychain with new information.

        If no, return.
        If yes, store new information by calling
SecKeychainItemModifyAttributesAndData.
    */
    status = status ?: ChangePasswordKeychain (itemRef);
    if (itemRef) CFRelease(itemRef);
    return (status);

 }
```

This example follows the same general sequence that was shown in Figure 3-1 (page 31); however, unlike the figure, the example illustrates the use of generic passwords rather than Internet passwords.

Important: You should not cache passwords, because the user can change them using Keychain Access or another program and the data may no longer be valid. In addition, the long-term storage of passwords by applications negates the value of the keychain.

Although the example in Listing 3-1 (page 32) is written in procedural C, you can call these same functions using Objective C. Listing 3-2 (page 37) shows how you might create an Internet password item if you wanted some custom attribute values. Note that attribute strings should all be encoded in UTF-8 format.

Advanced Topics

Most applications need only the Keychain Services functions described in Adding Simple Keychain Services to Your Application (page 29). However, in certain circumstances you might want to use some of the other functions provided by Keychain Services. For example, you might want to display a custom label for your application in the Keychain Access utility. Or, if you are writing a server, you might want to disable the automatic display of dialogs and instead take care of unlocking the keychain within your application.

Creating a Custom Keychain Item

This section discusses how to create a keychain item if you want lower-level control than is provided by `SecKeychainAddInternetPassword` or `SecKeychainAddGenericPassword`. For example, you might want to have the Keychain Access application display a custom label for your keychain item or you might want to specify more than one trusted application that can access the item. Specifically, this section illustrates the use of the `SecKeychainItemCreateFromContent` function to create a keychain item and the `SecAccessCreate` function to set up an access list. For more details about these and other low-level functions, see *Keychain Services Reference*.

When you use `SecKeychainAddInternetPassword` or `SecKeychainAddGenericPassword`, the function creates a label for the keychain item automatically. For an Internet password, it uses the URL, minus the scheme name, colon, and leading slashes. For example, the label (displayed in the Name field in Keychain Access) for a new Internet password for the URL http://www.apple.com becomes www.apple.com. For a generic password, the function uses the service attribute for the label. In both cases, the Name and Where fields in Keychain access are identical. If you want a unique name for the keychain item that is different from the URL or service name, you can specify the label (`kSecLabelItemAttr`) attribute when you call `SecKeychainItemCreateFromContent`.

The `SecKeychainAddInternetPassword` and `SecKeychainAddGenericPassword` functions create an initial access list for you. This default access list includes only one trusted application (that is, one application that can access the keychain item whenever the keychain is unlocked), namely the application that created the keychain item. The `SecKeychainItemCreateFromContent` function accepts an access list as input. However, if you pass `NULL`, this function creates an access list consisting of the single application calling the function. Therefore, you must call the `SecAccessCreate` function before calling the `SecKeychainItemCreateFromContent` function if you want an access list with more than one trusted application. Alternatively, you can alter the access list of an existing keychain item; see Modifying the Access List of an Existing Keychain Item (page 40).

Listing 3-2 (page 37) illustrates the creation of an Internet keychain item with a custom label and an access list that includes two trusted applications. This listing also illustrates the use of the Keychain Services API from an Objective-C application.

Listing 3-2 Creating a keychain item with custom attributes

```
#import <Foundation/Foundation.h>

#include <Security/Security.h>

SecAccessRef createAccess(NSString *accessLabel)
{
```

```
    OSStatus err;
    SecAccessRef access=nil;
    NSArray *trustedApplications=nil;

    //Make an exception list of trusted applications; that is,
    // applications that are allowed to access the item without
    // requiring user confirmation:
    SecTrustedApplicationRef myself, someOther;

    //Create trusted application references; see SecTrustedApplications.h:
    err = SecTrustedApplicationCreateFromPath(NULL, &myself);
    err = err ?: SecTrustedApplicationCreateFromPath("/Applications/Mail.app",
                                                     &someOther);

    if (err == noErr) {
       trustedApplications = [NSArray arrayWithObjects:(__bridge_transfer id)myself,
                                                (__bridge_transfer id)someOther,
 nil];
    }

    //Create an access object:
    err = err ?: SecAccessCreate((__bridge CFStringRef)accessLabel,
                            (__bridge CFArrayRef)trustedApplications, &access);
    if (err) return nil;

    return access;
}


OSStatus addInternetPassword(NSString *password, NSString *account,
                    NSString *server, NSString *itemLabel, NSString *path,
                    SecProtocolType protocol, int port)
{
    OSStatus err;
    SecKeychainItemRef item = nil;
```

```
    const char *pathUTF8 = [path UTF8String];
    const char *serverUTF8 = [server UTF8String];
    const char *accountUTF8 = [account UTF8String];
    const char *passwordUTF8 = [password UTF8String];
    const char *itemLabelUTF8 = [itemLabel UTF8String];

    //Create initial access control settings for the item:
    SecAccessRef access = createAccess(itemLabel);

    //Following is the lower-level equivalent to the
    // SecKeychainAddInternetPassword function:

    assert(strlen(itemLabelUTF8) <= 0xffffffff);
    assert(strlen(accountUTF8) <= 0xffffffff);
    assert(strlen(serverUTF8) <= 0xffffffff);
    assert(strlen(pathUTF8) <= 0xffffffff);

    //Set up the attribute vector (each attribute consists
    // of {tag, length, pointer}):
    SecKeychainAttribute attrs[] = {
        { kSecLabelItemAttr, (UInt32)strlen(itemLabelUTF8), (char *)itemLabelUTF8
 },
        { kSecAccountItemAttr, (UInt32)strlen(accountUTF8), (char *)accountUTF8
},
        { kSecServerItemAttr, (UInt32)strlen(serverUTF8), (char *)serverUTF8 },
        { kSecPortItemAttr, sizeof(int), (int *)&port },
        { kSecProtocolItemAttr, sizeof(SecProtocolType),
                                        (SecProtocolType *)&protocol },
        { kSecPathItemAttr, (UInt32)strlen(pathUTF8), (char *)pathUTF8 }
    };
    SecKeychainAttributeList attributes = { sizeof(attrs) / sizeof(attrs[0]),
                                            attrs };



    assert(strlen(passwordUTF8) <= 0xffffffff);
```

```
    err = SecKeychainItemCreateFromContent(
        kSecInternetPasswordItemClass,
        &attributes,
        (UInt32)strlen(passwordUTF8),
        passwordUTF8,
        NULL, // use the default keychain
        access,
        &item);

    if (access) CFRelease(access);
    if (item) CFRelease(item);

    return err;
}


int tryAdd(void)
{
         //Add an example password to the keychain:
    return addInternetPassword(@"sample password", @"sample account",
            @"samplehost.apple.com", @"sampleName", @"cgi-bin/bogus/testpath",
                                                kSecProtocolTypeHTTP, 8080);
}
```

Modifying the Access List of an Existing Keychain Item

By default, a keychain item's access list contains only the application that created the keychain item. Listing 3-2 (page 37) illustrates the creation of an Internet keychain item with an access list that includes two trusted applications. Listing 3-3 illustrates how to modify an existing keychain item. The listing finds a specific keychain item, extracts the access object, including the list of trusted applications, authorization tags, and other information (see ACL Entries (page 11)), adds a trusted application to the list, reconstitutes the access object, and writes the new access object back to the keychain item. Although this sample illustrates modifying the list of trusted applications, you can use a similar sequence to modify any of the information in the access object—for example, to add or remove an authorization tag.

Listing 3-3 starts by calling the `SecKeychainSearchCreateFromAttributes` function to create a search reference. In this example, the code looks for a keychain item by its label (as seen in the Keychain Access application); you can search for other attributes, however, such as modification date. Using this search reference, the sample calls `SecKeychainSearchCopyNext` to find the keychain item.

The listing next calls `SecKeychainItemCopyAccess` to retrieve the keychain item's access object. As discussed in Keychain Access Controls (page 9), the access object includes one or more ACL entries. The listing calls the `SecAccessCopySelectedACLList` function and passes it an authorization tag value of `CSSM_ACL_AUTHORIZATION_DECRYPT`. This authorization tag is one of the tags normally associated with the access list used for sensitive operations and is the list displayed by the Keychain Access application. To retrieve all the ACL entries for an access object, use the `SecAccessCopyACLList` function instead.

The `SecAccessCopySelectedACLList` function returns a CFArrayRef object containing all of the ACL entries that meet the selection criteria. In this case, the code is assuming that there should be only one ACL entry that employs the `CSSM_ACL_AUTHORIZATION_DECRYPT` tag. The listing calls the `CFArrayGetValues` function to create a C array of `SecACLRef` objects from the CFArrayRef and then calls the `SecACLCopySimpleContents` function, passing it the first (and presumably only) item in the array. The `SecACLCopySimpleContents` function also retrieves a CFArrayRef containing the list of trusted applications, the keychain item description string, and the prompt selector flag. These values are needed in order to reconstitute the ACL entry after adding a trusted application to the list.

Next, the listing uses the `CFArrayGetValues` function again, this time to extract the array of trusted applications from the CFArrayRef. The listing calls the `SecTrustedApplicationCreateFromPath` function to create a new trusted application object, appends the new application to the list, and calls `CFArrayCreate` to create a new CFArrayRef.

Before adding a new ACL entry to the access object, the listing calls the `SecACLGetAuthorizations` function to get the list of authorization tags from the old access object. Then, having extracted all the information from the old ACL entry, the code calls the `SecACLRemove` function to remove the old ACL entry from the access object. The listing then calls the `SecACLCreateFromSimpleContents` function to create a new ACL entry for the access object. This function automatically adds the entry to the access object; there is no separate call for that purpose. The new ACL entry has only the default authorization list, however, so the listing calls the `SecACLSetAuthorizations` function, passing in the authorization list extracted from the old ACL entry.

The access object is now complete, with the new ACL entry containing all of the trusted applications in the old entry plus the new one added here. Only two steps remain: First, the listing calls the `SecKeychainItemSetAccess` function to replace the access object in the keychain item with the new access object; then the listing calls the `CFRelease` function for each core foundation object that is no longer needed, in order to release the memory.

Note that this sample routine causes the user to be prompted twice for permission: once when the old ACL entry is deleted from the access object, and once when the new access object is written to the keychain item.

Listing 3-3 Modifying a keychain item access list

```
#include <CoreFoundation/CoreFoundation.h>
#include <Security/Security.h>
#include <CoreServices/CoreServices.h>


//Get an ACL out of a CFArray:
SecACLRef GetACL (CFIndex numACLs, CFArrayRef ACLList,
                CFArrayRef *applicationList, CFStringRef *description,
                SecKeychainPromptSelector *promptSelector)
{
    OSStatus status;
    //Because we limited our search to ACLs used for decryption, we
    // expect only one ACL for this item. Therefore, we extract the
    // application list from the first ACL in the array.
    const SecACLRef acl = (SecACLRef) CFArrayGetValueAtIndex(ACLList, 0);
    status = SecACLCopyContents (
        acl,                    // the ACL from which to extract
                                //  the list of trusted apps
        applicationList,        // the list of trusted apps
        description,            // the description string
        promptSelector          // the value of the prompt selector flag
                                        );

    if (status == noErr) {
        return acl;
    } else {
        return NULL;
    }
}


int modifyTheACL(void)
{
    OSStatus status;
```

```
SecKeychainSearchRef searchReference = NULL;
SecKeychainItemRef itemRef = NULL;

SecAccessRef itemAccess = NULL;
SecACLRef oldACL = NULL, newACL = NULL;

CFIndex arrayCount;
CFRange arrayRange;
SecTrustedApplicationRef trustedAppArray[10];
SecKeychainPromptSelector promptSelector;
CFStringRef description = NULL;
CFArrayRef newTrustedAppArray = NULL;

const char *path = "/Applications/Mail.app";       //path to
                                    // trusted app to add to ACL
SecTrustedApplicationRef trustedApp = NULL;

/* Construct a search dictionary to find the desired item. */
const void *keys[] = {
 kSecAttrLabel,
    kSecReturnRef, /* return the item. */
    NULL
};
const void *values[] = {
    /* kSecAttrLabel => */ CFSTR("www.TestItem.com"),
    /* kSecReturnRef => */ kCFBooleanTrue,
    NULL
};

CFDictionaryRef searchDict = CFDictionaryCreate(kCFAllocatorDefault,
    keys,
    values,
    sizeof(keys) / sizeof(keys[0]),
    &kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
```

```
 CFArrayRef aclList = NULL;
 CFIndex numACLs = 0;
 CFArrayRef applicationList;
 CFTypeRef authorizationTag =
    kSecACLAuthorizationDecrypt;  // the authorization tag
                                  //  to search for.

 // Find the keychain item and obtain a keychain item reference object.
// This returns a SecKeychainItemRef, which we must release when
//  we're finished using it.
status = SecItemCopyMatching(searchDict, (CFTypeRef *)&itemRef);

if (status == noErr)
    {
    // Obtain the access reference object for the keychain item.
    // This returns a SecAccessRef, which we must release when
    //  we're finished using it.
    status = SecKeychainItemCopyAccess (itemRef, &itemAccess);
    // Obtain an array of ACL entry objects for the access object.
    // Limit the search to ACL entries with the specified
    //  authorization tag.
    aclList = SecAccessCopyMatchingACLList(itemAccess,
                                authorizationTag);
    numACLs = CFArrayGetCount (aclList);
    // Extract the ACL entry object from the array of ACL entries,
    //  along with the ACL entry's list of trusted applications,
    //  its description, and its prompt selector flag setting.
    // This returns a SecACLRef and a CFArrayRef, which we must
    //  release we're finished using them.
    oldACL = GetACL (numACLs, aclList, &applicationList,
                               &description, &promptSelector);
    if (oldACL) { CFRetain(oldACL); }
    arrayCount = CFArrayGetCount (applicationList);
```

```
    //  The application list is a CFArray.  Extract the list of
    //  applications from the CFArray.
    arrayRange.location = (CFIndex) 0;
    arrayRange.length = arrayCount;
    CFArrayGetValues (applicationList, arrayRange,
                                        (void *) trustedAppArray);
    // Create a new trusted application reference object for
    //  the application to be added to the list.
   status = status ?: SecTrustedApplicationCreateFromPath (path, &trustedApp);
    if (status == noErr)   // the function fails if the application is
                           // not found.
    {
        // Append the new application to the array and create a
        //  new CFArray.
        trustedAppArray[arrayCount] = trustedApp;
        newTrustedAppArray = CFArrayCreate (NULL,
                           (void *)trustedAppArray, arrayCount+1,
                                               &kCFTypeArrayCallBacks);
        // Get the authorizations from the old ACL.
        CFArrayRef authorizations = SecACLCopyAuthorizations(oldACL);

        // Delete the old ACL from the access object. The user is
        // prompted for permission to alter the keychain item.
        status = status ?: SecACLRemove (oldACL);

        // Create a new ACL with the same attributes as the old
        // one, except use the new CFArray of trusted applications.
        status = status ?: SecACLCreateWithSimpleContents (itemAccess,
                    newTrustedAppArray, description, promptSelector,
                                                        &newACL);
        // Set the authorizations for the new ACL to be the same as
        //  those for the old ACL.
       status = status ?: SecACLUpdateAuthorizations (newACL, authorizations);
```

```
            // Replace the access object in the keychain item with the
            //  new access object. The user is prompted for permission
            //  to alter the keychain item.
            status = status ?: SecKeychainItemSetAccess (itemRef, itemAccess);

            CFRelease(authorizations);
        }
        else {
            // Handle the error if the application was not found.
            // ...
        }

    // Release the objects we allocated or retrieved
    if (searchReference)
        CFRelease(searchReference);     //SecKeychainSearchRef
    if (itemRef)
        CFRelease(itemRef);             //SecKeychainItemRef
    if (itemAccess)
        CFRelease(itemAccess);          //SecAccessRef
    if (oldACL)
        CFRelease(oldACL);              //SecACLRef
    if (newACL)
        CFRelease(newACL);              //SecACLRef
    if (description)
        CFRelease(description);         //CFStringRef
    if (newTrustedAppArray)
        CFRelease(newTrustedAppArray);  //CFArrayRef
    if (trustedApp)
        CFRelease(trustedApp);          //SecTrustedApplicationRef
    if (aclList)
        CFRelease(aclList);             //CFArrayRef
    if (applicationList)
        CFRelease(applicationList);     //CFArrayRef
    }
```

```
    return (status);

}
```

Servers and the Keychain

Several Keychain Services functions automatically display a dialog requesting that the user unlock the keychain or create a new keychain when necessary. Keychain Services also displays dialogs to confirm that the user wants the application to access the keychain (Figure 3-2) and for other reasons. If you are writing a server application that must run unattended, you might want to disable the automatic display of dialogs and programatically unlock or create keychains.

Figure 3-2 Confirm access dialog

Use the `SecKeychainSetUserInteractionAllowed` function to enable or disable the automatic display of dialogs. When you call this function with a value of `FALSE` for the `state` parameter, any Keychain Services functions that ordinarily open dialogs instead return immediately with the result `errSecInteractionRequired`. You can use the `SecKeychainGetStatus` function to find out whether the keychain exists and is unlocked. You can then use the `SecKeychainUnlock` function to unlock the keychain, or the `SecKeychainCreate` function to create a new keychain, as necessary.

Important: After you have disabled the automatic display of dialogs, no Keychain Services function called by any application displays dialogs until someone calls the `SecKeychainSetUserInteractionAllowed` function to reenable dialogs, or until the system is restarted. Therefore, if there might be other applications operating on the same computer, you should be careful to reenable this feature once you are finished opening or creating a keychain.

Adding, Removing, and Working With Keys and Certificates

In addition to passwords, a keychain can also hold encryption keys and certificates. When working with keys and certificates, you should use the following functions:

- `SecItemAdd` to add an item to a keychain
- `SecItemUpdate` to modify an existing keychain item
- `SecItemCopyMatching` to find a keychain item and extract information from it

For examples of these functions, see iOS Keychain Services Tasks (page 18). You can also learn more about these functions in *Certificate, Key, and Trust Services Programming Guide*.

Glossary

access control list (ACL) A structure containing information describing what must happen (display a confirmation dialog, ask for a password, and so forth) in order to permit a specific operation to occur. An ACL may also contain a list of applications that are always trusted to perform that operation. Each keychain item has one or more associated ACLs, and each ACL applies to a single operation that can be done with that item, such as encrypting or decrypting it. See also access object.

access object An opaque data structure containing one or more access control lists. Each keychain item has one access object.

ACL See access control list (ACL).

application programming interface (API) The set of routines, data structures, constants, and other programming elements that allow developers to use some part of the system software.

attribute One data item, such as the name, type, date modified, account number, and so on, for a keychain item, other than the secret. The attributes associated with a keychain item depend on the class of the item.

authentication The act of verifying identity with something the user provides. For example, a user can provide information such as a name and password, a physical item such as a smart card, or a physical feature such as a fingerprint or retinal scan.

authorization tag A data field in an access control list (ACL) that specifies an operation that can be done with that keychain item, such as decrypting or authenticating

certificate See digital certificate.

CDSA (Common Data Security Architecture) An open software standard for a security infrastructure that provides a wide array of security services, including fine-grained access permissions, authentication of users, encryption, and secure data storage. CDSA has a standard application programming interface, called CSSM (Common Security Services Manager). In addition, OS X includes its own security APIs that call the CDSA API for you.

CSSM (Common Security Services Manager) A public application programming interface for CDSA (Common Data Security Architecture). CSSM also defines an interface for plug-ins that implement security services for a particular operating system and hardware environment.

default keychain The keychain accessed by certain Keychain Services functions when no other keychain is specified in the function call. For example, newly-created keychain items are stored in the default keychain unless a different keychain is specified in the function call. A default keychain is created for each new login account, but the user can use the Keychain Access utility to designate another keychain as the default.

default keychain search list The list of keychains searched by certain Keychain Services functions when no other keychain or list of keychains is

specified in the function call. The default keychain search list contains the same keychains as the keychain list displayed in the Keychain Access utility.

digital certificate Digitally signed data that can be used as a component of digital authentication systems. Because certificates might be referred to more than once, Keychain Services allows applications to store them in a keychain.

encrypt To secure data so that it cannot be read by unauthorized entities, in such a way that its original state can be restored later (decrypted). In most cryptographic systems, encryption and decryption are performed by manipulating the data with a string of bytes called a key.

generic password A password other than an Internet password.

Internet password A password for an Internet server, such as a Web or FTP server. Internet password items on the keychain include attributes such as the security domain and IP address.

key A string of bytes used by an encryption algorithm to encrypt or decrypt data.

keychain A container that holds encrypted secrets for multiple applications and secure services. See also keychain item.

Keychain Access A utility that allows users to create, delete, and modify keychains and keychain items.

keychain item A secret that is encrypted and protected by the keychain, plus its associated attributes and access object. Each keychain item has a class that determines what attributes it has; for example Internet password items include an IP address attribute. The password or other secret stored as a keychain item is encrypted and is inaccessible when the keychain is locked. When the keychain is unlocked, the secret can be read by the trusted applications listed in the item's access object and by the user (with the Keychain Access utility). The attributes are not currently encrypted.

Keychain Manager A legacy API used to create, delete, and modify keychains and keychain items prior to OS X v10.2. Keychain Services is preferred for use with OS X v10.2 and later.

Keychain Services The API used to create, delete, and modify keychains and keychain items starting with OS X v10.2.

locked A keychain state in which no key is available in memory to decrypt the passwords and other secrets protected by the keychain. When an application attempts to retrieve a secret from a locked keychain, the user is prompted for a password. The login keychain is unlocked automatically at login if its password matches that of the user's login account.There is no way to extract secrets from a locked keychain without providing the keychain's password. All of a users keychains lock automatically when the user logs out.

login keychain A keychain automatically created for a new login account. The login keychain is automatically unlocked at login if its password matches that of the user's login account. For OS X v10.2, the login keychain had the same name as the login account. For OS X v10.3, the login keychain is named `login.keychain`. OS X v10.3 also recognizes login keychains created under OS X v10.2.

password Data, usually a character string, used to authenticate a user for a service or application.

secret The encrypted data in a keychain item, such as a password. Only a trusted application can read the secret of a keychain item. Compare with attribute.

secure storage Encrypted storage of data that requires a user or process to authenticate itself before the data is decrypted.

system keychain A keychain that belongs to the system as a whole, rather than to a particular user. System keychains are usually used by system daemons and services, but can also be accessed by applications on behalf of users. One system keychain, named `System.keychain`, is created by the system and added by default to every user's keychain list.

transparent authentication Authentication without intervention by the user. After unlocking the keychain, the user does not have to log in separately to any services whose passwords are stored in the keychain.

trusted application An application that can read a keychain item's secret when the keychain is unlocked. See also access control list (ACL).

unlocked A state in which the keychain has a key in memory that can be used to encrypt or decrypt items in that particular keychain. A keychain is considered unlocked after its password has been entered (by the user or programmatically). An unlocked keychain can provide access to its secrets, subject to ACL checks, until it is locked again.

user ID Data, usually a character string, used to identify a user for a service or application.

Document Revision History

This table describes the changes to *Keychain Services Programming Guide*.

Date	Notes
2014-02-11	Updated code listings for ARC. Corrected function name in Figure 3-1 (page 31).
2012-06-11	Updated code listings for ARC. Corrected a figure. Corrected function name in Figure 3-1 (page 31).
2009-10-19	Fixed typos in code listing.
2009-03-12	Corrected a typo.
2008-05-23	Added information on Keychain APIs for iOS .
2007-01-08	Minor changes and corrections to code samples. Changed name from "Enabling Secure Storage With Keychain Services"
2004-06-28	Added section on Keychain Access Controls (page 9). Added code sample for modifying access controls; see Modifying the Access List of an Existing Keychain Item (page 40).
2003-10-03	First released version of this document.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010