Certificate, Key, and Trust Services Programming Guide

Security

2010-07-09

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

Contents

Listings

Introduction

Certificate, Key, and Trust Services Programmer's Guide contains an overview of Certificate, Key, and Trust services, discusses the functions and data structures that are most commonly used by developers, and provides examples of how to use Certificate, Key, and Trust Services in your own applications.

Certificate, Key, and Trust Services provides a C API for verifying certificates, evaluating trust, and generating asymmetric keys. You can use these services in your application to:

- Add a certificate to a keychain
- Find the certificate and private key associated with an identity
- Generate an asymmetric key pair and store the keys on a keychain
- Get a policy object for use in evaluating a certificate's trust
- Retrieve the anchor certificates stored by Mac OS X
- Set parameters to use in evaluating a certificate's trust
- Evaluate a certificate's trust
- Get detailed information about the results of a trust evaluation

In addition, the Certificate, Key, and Trust Services API includes a number of functions that make it easier to move between the Mac OS X security APIs and CSSM.

Certificate, Key, and Trust Services can be used in Carbon, Cocoa, and UNIX applications running in Mac OS X.

This document concentrates on the use of Certificate, Key, and Trust Services to evaluate trust of a certificate.

In order to read this document, you should be familiar with general concepts of computer security and with the use of the keychain to store certificates and keys. See "See Also" (page 8) for suggestions for further reading.

Organization of This Document

This document contains the following chapters:

- "Certificate, Key, and Trust Services Concepts" (page 9) discusses some of the concepts you need to understand in order to use the Certificate, Key, and Trust Services API.
- "Certificate, Key, and Trust Services Tasks for iOS" (page 11) contains iOS sample code and explanations for several common tasks associated with evaluating the trust of a certificate and recovering from a trust failure.

- "Certificate, Key, and Trust Services Tasks for Mac OS X" (page 25) contains Mac OS X sample code and explanations for several common tasks associated with evaluating the trust of a certificate and recovering from a trust failure.
- "Glossary" (page 35) defines new terms introduced in this book.

See Also

For more information on the APIs and concepts covered in this book, use the following resources:

- *Certificate, Key, and Trust Services Reference* in Security Documentation documents all the functions and structures provided in the Certificate, Key, and Trust Services API.
- For more information about storing and retrieving certificates and keys, see *Keychain Services Reference* in Security Documentation.
- Many security concepts, including keys and certificates, are discussed in more detail in *Security Overview* in Security Documentation.
- Certificate, Key, and Trust Services and other Mac 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 *Common Security: CDSA and CSSM, version 2 (with corrigenda)* from The Open Group (http://www.opengroup.org/security/cdsa.htm).

Certificate, Key, and Trust Services Concepts

Certificate, Key, and Trust Services is a collection of functions and data structures used to authenticate and authorize users and processes using keys and certificates. Because in Mac OS X and iOS, certificates and keys are stored on a keychain, many of the functions in this API must be used in conjunction with functions in the Keychain Services API.

This chapter discusses some of the concepts you need to understand in order to use the Certificate, Key, and Trust Services API. In addition, keys and certificates are defined and discussed in *Security Overview*.

Certificates, Keys, and Identities

A **digital certificate** is a collection of data used to verify the identity of the holder or sender of the certificate. For example, a certificate contains such information as:

- Certificate issuer
- Certificate holder
- Validity period (the certificate is not valid before or after this period)
- Public key of the owner of the certificate
- **Certificate extensions**, which contain additional information such as allowable uses for the private key associated with the certificate
- Digital signature from the certification authority to ensure that the certificate has not been altered and to indicate the identity of the issuer

Each certificate is verified through the use of another certificate, creating a chain of certificates that ends with the **root certificate**. The issuer of a certificate is called a **certification authority** (CA). The owner of the root certificate is the root certification authority. See *Security Overview* for more details about the structure and contents of a certificate.

Every public key is half of a public-private key pair. As implied by the names, the **public key** can be obtained by anyone, but the **private key** is kept secret by the owner of the key. Data encrypted with the private key can be decrypted only with the public key, and *vice versa*. In order to both encrypt and decrypt data, therefore, a given user must have both a public key (normally embedded in a certificate) and a private key. The combination of a certificate and its associated private key is known as an **identity**. Certificate, Key, and Trust Services includes functions to find the certificate or key associated with an identity and to find an identity when given search criteria. The search criteria include the permitted uses for the key.

In Mac OS X and iOS, keys and certificates are stored on a **keychain**, a database that provides secure (that is, encrypted) storage for private keys and other secrets as well as unencrypted storage for other security-related data. The Certificate, Key, and Trust Services functions that search for keys, certificates, and identities all use the keychain for this purpose. On a Mac OS X system, you can use the Keychain Access utility to see the contents of the keychain and to examine the contents of certificates.

Certificate, Key, and Trust Services and CDSA

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

In Mac OS X, Certificate, Key, and Trust 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 Certificate, Key, and Trust Services API provides functions to perform most of the operations needed by applications, including generating key pairs, retrieving the certificate or private key associated with an identity, retrieving root certificates from the system, validating certificates, and evaluating trust. 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 Certificate, Key, and Trust Services API includes a number of functions that return or create CSSM structures so that you can move freely back and forth between Certificate, Key, and Trust 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).

Policies and Trust

Certain attributes of a digital certificate (known as *certificate extensions*) are said to establish a **level of trust** for a digital certificate. The level of trust for a certificate is used to answer the question "Should I trust this certificate for this action?" A **trust policy** is a set of rules that specify how to evaluate a certificate to see if it is valid for a specific level of trust.

For example, in Mac OS X the AppleX509TP module implements a trust policy referred to as the S/MIME policy, which specifies how to verify email addresses in addition to basic validation of the certificate. When you set up a trust evaluation in the Certificate, Key, and Trust Services API, you specify which policy to use in evaluating trust. This is how you indicate the use for which you want to verify the certificate's validity. For example, if you specify the SSL policy, you are in effect asking whether the certificate can be trusted for use in establishing a secure connection over a network.

Some policies have options (see the AppleX509TP Trust Policies appendix in *Certificate, Key, and Trust Services Reference*). For example, the certificate revocation list policy includes options, which include flags. When the `CSSM_TP_ACTION_REQUIRE_CRL_PER_CERT` flag is set, a certificate is not valid unless every certificate in the certificate chain has been successfully verified using a certificate revocation list. Option structures for the AppleX509TP trust policies are defined in `cssmapple.h`. The Certificate, Key, and Trust Services API uses default option values for each policy.

Certificate, Key, and Trust Services Tasks for iOS

This chapter describes and illustrates the use of Certificate, Key, and Trust Services functions to import an identity, evaluate the trust of a certificate, determine the cause of a trust failure, and recover from a trust failure.

The sequence of operations illustrated in this chapter is:

1. Import an identity.
2. Obtain a certificate from the imported data.
3. Obtain a policy object for the policy used in evaluation of the certificate.
4. Validate the certificate and evaluate whether it can be trusted as specified by the policy.
5. Test for a recoverable trust error.
6. Determine whether the trust error is due to an expired certificate.
7. Change the evaluation criteria to ignore expired certificates.
8. Reevaluate the certificate.

Chapter 2, "Certificate, Key, and Trust Services Concepts," (page 9) provides an introduction to the concepts and terminology of Certificate, Key, and Trust Services. For detailed information about all Certificate, Key, and Trust Services functions, see *Certificate, Key, and Trust Services Reference*.

Extracting and Evaluating an Identity From a *.P12 File

If you need a cryptographic identity (that is, a private key and its associated certificate) on an iOS-based device—for client-side authentication, for example—you can transfer it to the device securely as PKCS #12 data in a password-protected `*.p12` file. This section shows how to extract the identity and trust objects from the PKCS #12 data and how to evaluate the trust.

Listing 2-1 shows sample code for using the `SecPKCS12Import` function to extract identity and trust objects from a `*.p12` file and how to evaluate the trust. Listing 2-2 (page 13) shows how to get the certificate from the identity and display certificate information. Explanations for numbered lines of code follow each listing.

Be sure to add the Security framework to your Xcode project when compiling code with this snippet.

Listing 2-1 Extracting identity and trust objects from PKCS #12 Data

```
#import <UIKit/UIKit.h>
#import <Security/Security.h>
#import <CoreFoundation/CoreFoundation.h>
```

```
    NSString *thePath = [[NSBundle mainBundle]
                            pathForResource:@"MyIdentity" ofType:@"p12"];
    NSData *PKCS12Data = [[NSData alloc] initWithContentsOfFile:thePath];
    CFDataRef inPKCS12Data = (CFDataRef)PKCS12Data;                          // 1

     OSStatus status = noErr;
    SecIdentityRef myIdentity;
    SecTrustRef myTrust;
    status = extractIdentityAndTrust(
                                    inPKCS12Data,
                                    &myIdentity,
                                    &myTrust);                               // 2
    if status != 0 ...  //Do some error checking here

SecTrustResultType trustResult;

    if (status == noErr) {                                                   // 3
        status = SecTrustEvaluate(myTrust, &trustResult);
}


...                                                                          // 4
    if (trustResult == kSecTrustResultRecoverableTrustFailure) {
        ...;
    }

OSStatus extractIdentityAndTrust(CFDataRef inPKCS12Data,                     // 5
                                 SecIdentityRef *outIdentity,
                                 SecTrustRef *outTrust)
{
    OSStatus securityError = errSecSuccess;


    CFStringRef password = CFSTR("Password");
    const void *keys[] =   { kSecImportExportPassphrase };
    const void *values[] = { password };
    CFDictionaryRef optionsDictionary = CFDictionaryCreate(
                                                NULL, keys,
                                                values, 1,
                                                NULL, NULL);                 // 6


    CFArrayRef items = CFArrayCreate(NULL, 0, 0, NULL);
    securityError = SecPKCS12Import(inPKCS12Data,
                                    optionsDictionary,
                                    &items);                                 // 7


    //
    if (securityError == 0) {                                                // 8
        CFDictionaryRef myIdentityAndTrust = CFArrayGetValueAtIndex (items, 0);
        const void *tempIdentity = NULL;
        tempIdentity = CFDictionaryGetValue (myIdentityAndTrust,
                                             kSecImportItemIdentity);
        *outIdentity = (SecIdentityRef)tempIdentity;
        const void *tempTrust = NULL;
```

```
        tempTrust = CFDictionaryGetValue (myIdentityAndTrust,
kSecImportItemTrust);
        *outTrust = (SecTrustRef)tempTrust;
    }

    if (optionsDictionary)
        CFRelease(optionsDictionary);                                   // 9
     [PKCS12Data release];
```

Here's what the code does:

1. Finds the PKCS #12 file and gets the data. In this example, the file is included in the application bundle. However, you can transfer the file to your application over a network if you prefer.
2. Calls the function that gets the identity and trust from the PKCS #12 file (see step #5).
3. Evaluates the trust. In this case, the trust object, containing the policy and other information needed to determine whether the certificate is trusted, is included in the PKCS data. To evaluate the trust of an isolated certificate, see Listing 2-6 (page 16).
4. Handles the trust result. If the trust result is `kSecTrustResultInvalid`, `kSecTrustResultDeny`, `kSecTrustResultFatalTrustFailure`, you cannot proceed and should fail gracefully. If the trust result is `kSecTrustResultRecoverableTrustFailure`, you might be able to recover from the failure. See "Recovering From a Trust Failure" (page 17).
5. Implements the function called in step #2.
6. Sets up dictionary containing the password to pass to `SecPKCS12Import`. Notice that core foundation dictionaries—as used here—and the `NSDictionary` class are entirely equivalent. See Listing 2-9 (page 20) for an example using `NSDictionary` methods.
7. Extracts the certificate, key, and trust from the PKCS #12 data and puts them in an array.
8. Gets the first dictionary out of the array and gets the identity and trust out of the dictionary. The `SecPKCS12Import` function returns one dictionary for each item (identity or certificate) in the PKCS #12 data. In this sample, the identity being extracted is the first one in the array (item #0).
9. Disposes of the options dictionary and releases the PKCS12Data, which are no longer needed.

The following listing shows how to get the certificate from the identity and how to display information from the certificate. Be sure to add the Security framework to your Xcode project when compiling code with this snippet.

Listing 2-2 Displaying information from the certificate

```
// Get the certificate from the identity.
SecCertificateRef myReturnedCertificate = NULL;
status = SecIdentityCopyCertificate (myReturnedIdentity,
                                     &myReturnedCertificate);           // 1

CFStringRef certSummary = SecCertificateCopySubjectSummary
                                     (myReturnedCertificate);           // 2

NSString* summaryString = [[NSString alloc]
                     initWithString:(NSString*)certSummary];            // 3
```

```
    //Display the string
...
     [summaryString release];                                                    // 4
```

Here's what the code does:

1. Extracts the certificate from the identity.
2. Gets summary information from the certificate.
3. Converts the string to an `NSString` object so it can be displayed.
4. Releases the `NSString` object.

Getting and Using Persistent Keychain References

When you add an item to the keychain or find an item in the keychain, you can request a persistent reference. Because a persistent reference remains valid between invocations of your program and can be stored on disk, you can use one to make it easier to find a keychain item that you will need repeatedly. The following code sample shows how to obtain a persistent reference for the identity object obtained in Listing 2-1 (page 11).

Listing 2-3 Getting a persistent reference for an identity

```
CFDataRef persistentRefForIdentity(SecIdentityRef identity)
{
    OSStatus status;

    CFTypeRef  identity_handle = NULL;
    const void *keys[] =   { kSecReturnPersistentRef, kSecValueRef };
    const void *values[] = { kCFBooleanTrue,          identity };
    CFDictionaryRef dict = CFDictionaryCreate(NULL, keys, values,
                                              2, NULL, NULL);
    status = SecItemAdd(dict, &persistent_ref);

    if (dict)
        CFRelease(dict);

    return (CFDataRef)persistent_ref;
}
```

The following sample shows how to retrieve the identity object from the keychain using the persistent reference.

Listing 2-4 Getting an identity using a persistent reference

```
SecIdentityRef identityForPersistentRef(CFDataRef persistent_ref)
{
    CFTypeRef   identity_ref     = NULL;
    const void *keys[] =   { kSecReturnRef,  kSecValuePersistentRef };
    const void *values[] = { kCFBooleanTrue, persistent_ref };
    CFDictionaryRef dict = CFDictionaryCreate(NULL, keys, values,
```

```
                                                2, NULL, NULL);
    SecItemCopyMatching(dict, &identity_ref);

    if (dict)
        CFRelease(dict);

    return (SecIdentityRef)identity_ref;
}
```

Finding a Certificate In the Keychain

The following code sample shows how to find a certificate in the keychain using the name of the certificate to identify it. To find a keychain item using a persistent reference, see Listing 2-4 (page 14). To find a keychain item using an identifier string stored as a keychain item attribute, see "Encrypting and Decrypting Data" (page 18). An explanation for each numbered line of code follows the listing.

Listing 2-5 Finding a certificate In the Keychain

```
    CFTypeRef   certificateRef     = NULL;                                  // 1
    const char *certLabelString = "Romeo Montegue";
    CFStringRef certLabel = CFStringCreateWithCString(
                                NULL, certLabelString,
                                kCFStringEncodingUTF8);                     // 2

    const void *keys[] =   { kSecClass, kSecAttrLabel, kSecReturnRef };
    const void *values[] = { kSecClassCertificate, certLabel, kCFBooleanTrue };
    CFDictionaryRef dict = CFDictionaryCreate(NULL, keys,
                                              values, 3,
                                            NULL, NULL);                    // 3
    status = SecItemCopyMatching(dict, &certificateRef);                    // 4

    if (dict)
        CFRelease(dict);
```

Here's what the code does:

1. Defines a variable to hold the certificate object.
2. Creates a string containing the name of the certificate.
3. Creates a dictionary of attributes to be used in the certificate search. The `kSecReturnRef` key specifies that the function should return a reference to the keychain item when it's found.
4. Searches for the certificate in the keychain.

Obtaining a Policy Object and Evaluating Trust

Before you can evaluate the trust of a certificate, you must obtain a reference object for the certificate. You can obtain a certificate object by extracting it from an identity (see Listing 2-2 (page 13)), by creating one from DER certificate data using the `SecCertificateCreateWithData` function (see the following sample: Listing 2-6), or by finding the certificate on a keychain (Listing 2-5 (page 15)).

The criteria for evaluation of trust are set by trust policies. Listing 3-2 shows how you can obtain a policy object for use in an evaluation. There are two policies available in iOS for this purpose: Basic X509 and SSL (see AppleX509TP Trust Policies). You use the `SecPolicyCreateBasicX509` or `SecPolicyCreateSSL` function to obtain the policy object.

The following code sample shows how to obtain a policy object and use it to evaluate trust of a certificate. An explanation for each numbered line of code follows the listing.

Listing 2-6 Obtaining a policy reference object and evaluating trust

```
    NSString *thePath = [[NSBundle mainBundle]
                         pathForResource:@"Romeo Montegue" ofType:@"cer"];
    NSData *certData = [[NSData alloc]
                        initWithContentsOfFile:thePath];
    CFDataRef myCertData = (CFDataRef)certData;                              // 1

    SecCertificateRef myCert;
    myCert = SecCertificateCreateWithData(NULL, myCertData);                 // 2

    SecPolicyRef myPolicy = SecPolicyCreateBasicX509();                      // 3

    SecCertificateRef certArray[1] = { myCert };
    CFArrayRef myCerts = CFArrayCreate(
                                    NULL, (void *)certArray,
                                    1, NULL);
    SecTrustRef myTrust;
    OSStatus status = SecTrustCreateWithCertificates(
                                                myCerts,
                                                myPolicy,
                                                &myTrust);                   // 4

    SecTrustResultType trustResult;
    if (status == noErr) {
        status = SecTrustEvaluate(myTrust, &trustResult);                    // 5
    }
...                                                                          // 6
    if (trustResult == kSecTrustResultRecoverableTrustFailure) {
        ...;
    }
...
    if (myPolicy)
        CFRelease(myPolicy);                                                 // 7
```

Here's what the code does:

1. Finds the certificate file and gets the data. In this example, the file is included in the application bundle. However, you can transfer the certificate to your application over a network if you prefer. If the certificate is already in the keychain, see "Finding a Certificate In the Keychain" (page 15).
2. Creates a certificate reference from the certificate data.
3. Creates a policy to be used in evaluating trust.
4. Creates a trust object using the certificate and the policy. If you have intermediate certificates or an anchor certificate for the certificate chain, you can include those in the certificate array passed to the `SecTrustCreateWithCertificates` function. Doing so speeds up the trust evaluation.
5. Evaluates the trust.
6. Handles the trust result. If the trust result is `kSecTrustResultInvalid`, `kSecTrustResultDeny`, `kSecTrustResultFatalTrustFailure`, you cannot proceed and should fail gracefully. If the trust result is `kSecTrustResultRecoverableTrustFailure`, you might be able to recover from the failure. See "Recovering From a Trust Failure" (page 17).
7. Disposes of the policy object at the end of the routine, after it has been used to evaluate the trust.

Recovering From a Trust Failure

There are several possible results of a trust evaluation, depending on such factors as whether all the certificates in the chain were found, whether they are all valid, and what the user trust settings are for the certificates. It is up to your application to determine the course of action based on the result of the evaluation. For example, if the result is `kSecTrustResultConfirm`, you should display a dialog requesting that the user give permission to proceed.

The evaluation result `kSecTrustResultRecoverableTrustFailure` indicates that trust was denied, but that it is possible to change settings to get a different result. For example, if the certificate used to sign a document has expired, you can change the date used for the evaluation to see whether the certificate was valid when the document was signed. The code in Listing 3-4 illustrates how to change the evaluation date. Note that the `CFDateCreate` function takes an absolute time (the number of seconds since 1 January 2001); you can use the `CFGregorianDateGetAbsoluteTime` function to convert a calendar date and time into an absolute time. An explanation for each numbered line of code follows the listing.

Listing 2-7 Setting an evaluation date

```
    SecTrustResultType trustResult;
    status = SecTrustEvaluate(myTrust, &trustResult);                    // 1

//Get time used to verify trust
CFAbsoluteTime trustTime,currentTime,timeIncrement,newTime;
CFDateRef newDate;
if (trustResult == kSecTrustResultRecoverableTrustFailure) {             // 2
    trustTime = SecTrustGetVerifyTime(myTrust);                          // 3
    timeIncrement = 31536000;                                            // 4
    currentTime = CFAbsoluteTimeGetCurrent();                            // 5
    newTime = currentTime - timeIncrement;                               // 6
    if (trustTime - newTime){                                            // 7
```

```
            newDate = CFDateCreate(NULL, newTime);                          // 8
            SecTrustSetVerifyDate(myTrust, newDate);                        // 9
            status = SecTrustEvaluate(myTrust, &trustResult);               // 10
        }
    }
    if (trustResult != kSecTrustResultProceed) {                            // 11
...
}
```

Here's what the code does:

1. Evaluates the trust of the certificate. See "Obtaining a Policy Object and Evaluating Trust" (page 16).
2. Checks whether the result of the trust evaluation was a recoverable trust failure.
3. Gets the absolute time that was used to evaluate the trust. If the certificate expired before this time, then it is considered invalid.
4. Sets a time increment equal to the number of seconds in a year.
5. Gets the current (absolute) time.
6. Subtracts a year from the current time.
7. Checks whether the time used to evaluate trust was more recent than one year before the current time. If it was, then the trust is evaluated again using the new time; that is, the certificate is checked to see if it failed verification because it expired sometime in the past year.
8. Converts the new time to a `CFDateRef`. You can also use `NSDate` to manipulate the dates; `CFDateRef` and `NSDate` are toll-free bridged, meaning in a method where you see an `NSDate *` parameter, you can pass in a `CFDateRef`, and in a function where you see a `CFDateRef` parameter, you can pass in an instance of `NSDate` or of a concrete subclass of `NSDate`.
9. Sets the date used to verify trust to the new time (a year earlier).
10. Reevaluates the trust. If the reason the trust evaluation failed was because the certificate expired within a year of the current time, the evaluation should now succeed.
11. Checks whether the evaluation now succeeds. If not, you can try something else, such as asking the user to install an intermediate certificate, or you can tell the user that the certificate is not valid and fail gracefully.

Encrypting and Decrypting Data

The Certificate, Key, and Trust API includes functions for generating asymmetric key pairs and using them to encrypt and decrypt data. You might want to use this feature to encrypt data that you do not want to be accessible in backup data, for example. Or, you can use a private-public key pair shared between your iOS application and a desktop application to send encrypted data over a network. The code in Listing 2-8 shows how to generate a public-private key pair for use on the mobile device. Listing 2-9 (page 20) shows how to use a public key to encrypt data using Certificate, Key, and Trust functions, and Listing 2-10 (page 22) shows how to use a private key to decrypt data. Notice that these samples use Cocoa objects (such as `NSMutableDictionary`) rather than the core foundation objects (such as `CFMutableDictionaryRef`)

used in other samples in this chapter. The Cocoa objects and their Core Foundation counterparts are completely equivalent and are toll-free bridged; for example, in a method where you see an `NSMutableDictionary *` parameter, you can pass in a `CFMutableDictionaryRef`, and in a function where you see a `CFMutableDictionaryRef` parameter, you can pass in an instance of `NSMutableDictionary`. Explanations for numbered lines of code follow each listing.

Listing 2-8 Generating a key pair

```
static const UInt8 publicKeyIdentifier[] = "com.apple.sample.publickey\0";
static const UInt8 privateKeyIdentifier[] = "com.apple.sample.privatekey\0";
                                                                                // 1

- (void)generateKeyPairPlease
{
    OSStatus status = noErr;
    NSMutableDictionary *privateKeyAttr = [[NSMutableDictionary alloc] init];
    NSMutableDictionary *publicKeyAttr = [[NSMutableDictionary alloc] init];
    NSMutableDictionary *keyPairAttr = [[NSMutableDictionary alloc] init];
                                                                                // 2

    NSData * publicTag = [NSData dataWithBytes:publicKeyIdentifier
                        length:strlen((const char *)publicKeyIdentifier)];
    NSData * privateTag = [NSData dataWithBytes:privateKeyIdentifier
                       length:strlen((const char *)privateKeyIdentifier)];
                                                                                // 3

    SecKeyRef publicKey = NULL;
    SecKeyRef privateKey = NULL;                                                // 4

    [keyPairAttr setObject:(id)kSecAttrKeyTypeRSA
                                forKey:(id)kSecAttrKeyType];                    // 5
    [keyPairAttr setObject:[NSNumber numberWithInt:1024]
                        forKey:(id)kSecAttrKeySizeInBits];                      // 6

    [privateKeyAttr setObject:[NSNumber numberWithBool:YES]
                          forKey:(id)kSecAttrIsPermanent];                      // 7
    [privateKeyAttr setObject:privateTag
                       forKey:(id)kSecAttrApplicationTag];                      // 8

    [publicKeyAttr setObject:[NSNumber numberWithBool:YES]
                          forKey:(id)kSecAttrIsPermanent];                      // 9
    [publicKeyAttr setObject:publicTag
                       forKey:(id)kSecAttrApplicationTag];                      // 10

    [keyPairAttr setObject:privateKeyAttr
                          forKey:(id)kSecPrivateKeyAttrs];                      // 11
    [keyPairAttr setObject:publicKeyAttr
                           forKey:(id)kSecPublicKeyAttrs];                      // 12

    status = SecKeyGeneratePair((CFDictionaryRef)keyPairAttr,
                                  &publicKey, &privateKey);                     // 13
//    error handling...


    if(privateKeyAttr) [privateKeyAttr release];
    if(publicKeyAttr) [publicKeyAttr release];
```

```
    if(keyPairAttr) [keyPairAttr release];
    if(publicKey) CFRelease(publicKey);
    if(privateKey) CFRelease(privateKey);                                    // 14
}
```

Here's what the code does:

1. Defines unique strings to be added as attributes to the private and public key keychain items to make them easier to find later.
2. Allocates dictionaries to be used for attributes in the `SecKeyGeneratePair` function.
3. Creates `NSData` objects that contain the identifier strings defined in step 1.
4. Allocates `SecKeyRef` objects for the public and private keys.
5. Sets the key-type attribute for the key pair to RSA.
6. Sets the key-size attribute for the key pair to 1024 bits.
7. Sets an attribute specifying that the private key is to be stored permanently (that is, put into the keychain).
8. Adds the identifier string defined in steps 1 and 3 to the dictionary for the private key.
9. Sets an attribute specifying that the public key is to be stored permanently (that is, put into the keychain).
10. Adds the identifier string defined in steps 1 and 3 to the dictionary for the public key.
11. Adds the dictionary of private key attributes to the key-pair dictionary.
12. Adds the dictionary of public key attributes to the key-pair dictionary.
13. Generates the key pair.
14. Releases memory that is no longer needed.

You can send your public key to anyone, who can then use it to encrypt data. Assuming you keep your private key secure, then only you will be able to decrypt the data. The following code sample shows how to encrypt data using a public key. This can be a public key that you generated on the device (see the preceding code sample) or a public key that you extracted from a certificate that was sent to you or that is in your keychain. You can use the `SecTrustCopyPublicKey` function to extract a public key from a certificate. In the following code sample, the key is assumed to have been generated on the device and placed in the keychain. An explanation for each numbered line of code follows the listing.

Listing 2-9 Encrypting data with a public key

```
- (void)encryptWithPublicKey
{
    OSStatus status = noErr;

    size_t cipherBufferSize;
    uint8_t *cipherBuffer;                                                   // 1

// [cipherBufferSize]
    const uint8_t nonce[] = "the quick brown fox jumps
                                        over the lazy dog\0";                // 2
```

```
    SecKeyRef publicKey = NULL;                                                 // 3

    NSData * publicTag = [NSData dataWithBytes:publicKeyIdentifier
             length:strlen((const char *)publicKeyIdentifier)];                 // 4

    NSMutableDictionary *queryPublicKey =
                            [[NSMutableDictionary alloc] init];                 // 5

    [queryPublicKey setObject:(id)kSecClassKey forKey:(id)kSecClass];
    [queryPublicKey setObject:publicTag forKey:(id)kSecAttrApplicationTag];
   [queryPublicKey setObject:(id)kSecAttrKeyTypeRSA forKey:(id)kSecAttrKeyType];
    [queryPublicKey setObject:[NSNumber numberWithBool:YES]
forKey:(id)kSecReturnRef];
                                                                                // 6

    status = SecItemCopyMatching
    ((CFDictionaryRef)queryPublicKey, (CFTypeRef *)&publicKey);                 // 7

//  Allocate a buffer

    cipherBufferSize = cipherBufferSize(publicKey);
    cipherBuffer = malloc(cipherBufferSize);

//  Error handling

    if (cipherBufferSize < sizeof(nonce)) {
        // Ordinarily, you would split the data up into blocks
        // equal to cipherBufferSize, with the last block being
        // shorter. For simplicity, this example assumes that
        // the data is short enough to fit.
        printf("Could not decrypt.  Packet too large.\n");
        return;
    }

    // Encrypt using the public.
    status = SecKeyEncrypt(    publicKey,
                               kSecPaddingPKCS1,
                               nonce,
                               (size_t) sizeof(nonce)/sizeof(nonce[0]),
                               cipherBuffer,
                               &cipherBufferSize
                               );                                               // 8

//  Error handling
//  Store or transmit the encrypted text

    if(publicKey) CFRelease(publicKey);
    if(queryPublicKey) [queryPublicKey release];                                // 9
    free(cipherBuffer);
}
```

Here's what the code does:

1. Allocates a buffer to hold the encrypted text.

2. Specifies the text to be encrypted.

3. Allocates a `SecKeyRef` object for the public key.
4. Creates an `NSData` object containing the unique string used to identify the public key in the keychain (see steps 1, 3, and 8 in Listing 2-8 (page 19)).
5. Allocates the dictionary to be used to find the public key in the keychain.
6. Specifies the key-value attribute pairs for the dictionary to be used to find the public key in the keychain. The attributes specify that the keychain item is an encryption key; that the keychain item has an attribute containing the unique string specified in step 4; that the item is an RSA key; and that a reference to the keychain item is to be returned.
7. Calls the `SecItemCopyMatching` function to find the key in the keychain.
8. Encrypts the data from step 2 using the key returned by the `SecItemCopyMatching` function in step 7 using PKCS1 padding.
9. Releases memory that is no longer needed.

The following code sample shows how to decrypt data. This sample uses the private key corresponding to the public key used to encrypt the data, and assumes you already have the cipher text created in the preceding example. It gets the private key from the keychain using the same technique as used in the preceding example to get the public key.

Listing 2-10 Decrypting with a private key

```
- (void)decryptWithPrivateKey
{
    OSStatus status = noErr;

    size_t plainBufferSize;;
    uint8_t *plainBuffer;

    SecKeyRef privateKey = NULL;

    NSData * privateTag = [NSData dataWithBytes:privateKeyIdentifier
                          length:strlen((const char *)privateKeyIdentifier)];

    NSMutableDictionary *queryPrivateKey = [[NSMutableDictionary alloc] init];

    // Set the private key query dictionary.
    [queryPrivateKey setObject:(id)kSecClassKey forKey:(id)kSecClass];
    [queryPrivateKey setObject:privateTag forKey:(id)kSecAttrApplicationTag];
   [queryPrivateKey setObject:(id)kSecAttrKeyTypeRSA forKey:(id)kSecAttrKeyType];
    [queryPrivateKey setObject:[NSNumber numberWithBool:YES]
forKey:(id)kSecReturnRef];
                                                                          // 1

    status = SecItemCopyMatching
  ((CFDictionaryRef)queryPrivateKey, (CFTypeRef *)&privateKey);           // 2

    if (plainBufferSize < cipherBufferSize) {
        // Ordinarily, you would split the data up into blocks
        // equal to plainBufferSize, with the last block being
        // shorter. For simplicity, this example assumes that
        // the data is short enough to fit.
```

```
        printf("Could not decrypt.  Packet too large.\n");
        return;
    }

//  Allocate the buffer
    plainBufferSize = SecKeyGetBlockSize(privateKey);
    plainBuffer = malloc(plainBufferSize)

//  Error handling

    status = SecKeyDecrypt(    privateKey,
                               kSecPaddingPKCS1,
                               cipherBuffer,
                               cipherBufferSize,
                               plainBuffer,
                               &plainBufferSize
                               );                                    // 3

//  Error handling
//  Store or display the decrypted text

    if(publicKey) CFRelease(publicKey);
    if(privateKey) CFRelease(privateKey);
    if(queryPublicKey) [queryPublicKey release];
    if(queryPrivateKey) [queryPrivateKey release];                   // 4
}
```

Here's what the code does:

1. Sets up the dictionary used to find the private key in the keychain.
2. Finds the private key in the keychain.
3. Decrypts the data.
4. Releases memory that is no longer needed.

Certificate, Key, and Trust Services Tasks for Mac OS X

This chapter describes and illustrates the use of Certificate, Key, and Trust Services functions to evaluate the trust of a certificate, determine the cause of a trust failure, and recover from a trust failure.

The sequence of operations illustrated in this chapter is:

1. Find a certificate in a keychain.
2. Obtain a policy object for the policy used in evaluation of the certificate.
3. Validate the certificate and evaluate whether it can be trusted as specified by the policy.
4. Test for a recoverable trust error.
5. Determine whether the trust error is due to an expired certificate.
6. Change the evaluation criteria to ignore expired certificates.
7. Reevaluate the certificate.

Chapter 2, "Certificate, Key, and Trust Services Concepts", (page 9) provides an introduction to the concepts and terminology of Certificate, Key, and Trust Services. For detailed information about all Certificate, Key, and Trust Services functions, see *Certificate, Key, and Trust Services Reference*.

Finding a Certificate on the Keychain

Before you can evaluate the trust of a certificate, you must obtain a reference object for the certificate. You can obtain a certificate object by using the Secure Transport API `SSLGetPeerCertificates` function, by creating one from certificate data using the `SecCertificateCreateFromData` function, or by finding the certificate on a keychain.

Listing 3-1 shows sample code for obtaining a certificate object by finding the certificate on a keychain. In this sample, the certificate is identified by the email address of the certificate owner. You can use other certificate attributes for this purpose, such as the label of the keychain item or its modification date. A detailed explanation for each numbered line of code follows the listing.

Listing 3-1 Finding a certificate on the keychain

```
#include <CoreFoundation/CoreFoundation.h>
#include <Security/Security.h>
#include <CoreServices/CoreServices.h>
OSStatus GetCertRef (SecKeychainAttributeList *attrList,
                                    SecKeychainItemRef *itemRef)
{
    OSStatus status;
```

```
    SecKeychainSearchRef searchReference = nil;

    status = SecKeychainSearchCreateFromAttributes (                    // 1
        NULL,                                                           // 2
        kSecCertificateItemClass,                                       // 3
        attrList,                                                       // 4
        &searchReference
    );

    status = SecKeychainSearchCopyNext (                                // 5
        searchReference,
        itemRef
        );
    if (searchReference)
        CFRelease(searchReference);                                     // 6

    return (status);
}
int main (int argc, const char * argv[]) {
    OSStatus status;
    SecKeychainItemRef itemRef = nil;
    SecKeychainAttributeList attrList;
    SecKeychainAttribute attrib;
    attrList.count = 1;                                                 // 7
    attrList.attr  = &attrib;
    attrib.tag = kSecAlias;                                             // 8
    attrib.data = "emailname@domain.com";                               // 9
    attrib.length = strlen(attrib.data);

    status = GetCertRef (&attrList, &itemRef);
    .
    .
    .
    if (itemRef)
        CFRelease(itemRef);                                             // 10
    return (status);
    }
```

Here's what the code does:

1. Sets up keychain item search criteria and gets a search reference object. This object must be disposed of when it's no longer needed.
2. Passes `NULL` to use the default keychain search list.
3. Specifies that the search is for a certificate.
4. Provides the list of attributes to match. The attributes are defined in the main routine (see steps 7 through 9).
5. Finds the certificate on the keychain and retrieves the keychain item reference object. This object must be disposed of when no longer needed. Note that a keychain item object for a certificate can be cast to a certificate object.
6. Disposes of the search reference object, which is no longer needed.
7. Specifies that there is only one attribute in the attribute list.

8. Specifies that the attribute is to be of type `kSecAlias`. In the case of a certificate, this indicates that the attribute is the email address of the certificate owner.

9. Specifies the email address to search for.

10. Disposes of the keychain item object at the end of the routine, after it has been used to evaluate the trust (Listing 3-3 (page 28)).

Obtaining a Policy Object

The criteria for evaluation of trust are set by trust policies. Trust policies can specify, for example, whether each certificate in the chain must be checked against a certificate revocation list, or that certificates' expiration dates should be ignored.

Listing 3-2 shows how you can obtain a policy object for use in an evaluation. To use this procedure, you must know the object identifier (OID) of the policy. OIDs of policies implemented by the AppleX509TP CDSA module are shown in Appendix A of *Certificate, Key, and Trust Services Reference*. A detailed explanation for each numbered line of code follows the listing.

Listing 3-2 Obtaining a policy reference object

```
OSStatus FindPolicy (const CSSM_OID *policyOID, SecPolicyRef *policyRef)
{
    OSStatus status1;
    OSStatus status2;
    SecPolicySearchRef searchRef;

    status1 = SecPolicySearchCreate (                                   // 1
        CSSM_CERT_X_509v3,                                              // 2
        policyOID,                                                      // 3
        NULL,
        &searchRef
        );

    status2 = SecPolicySearchCopyNext (                                 // 4
        searchRef,
        policyRef
        );

    if (searchRef)
        CFRelease(searchRef);                                           // 5
    return (status2);
}
int main (int argc, const char * argv[]) {
    OSStatus status;
    const CSSM_OID *myPolicyOID = &CSSMOID_APPLE_X509_BASIC;
    SecPolicyRef policyRef = nil;
    status = FindPolicy (myPolicyOID, &policyRef);
    .
    .
    .
    if (policyRef)
```

```
        CFRelease(policyRef);                                                   // 6
    return (status);
    }
```

Here's what the code does:

1. Sets up policy search criteria and gets a policy search reference object. This object must be disposed of when no longer needed.
2. Specifies the type of certificates used by the policy. Specify `CSSM_CERT_X_509v3` if you are uncertain of the certificate type.
3. Specifies the OID of the policy.
4. Finds the policy and retrieves the policy reference object. This object must be disposed of when no longer needed.
5. Disposes of the search reference object, which is no longer needed.
6. Disposes of the policy object at the end of the routine, after it has been used to evaluate the trust (Listing 3-3 (page 28)).

Evaluating Trust

Having obtained a certificate object (Listing 3-1 (page 25)) and a policy object (Listing 3-2 (page 27)), you can evaluate the trust of a certificate. If you know—or have a good guess for—the intermediate and root certificates needed to verify the certificate, you can include them in the array of certificates passed to the `SecTrustCreateWithCertificates` function. Whereas the intermediate and root certificates can be in any order, the leaf certificate—the one you want to evaluate—must be the first in the array. You can include certificates not needed for the evaluation with no ill effects. Any certificates in the certificate chain that you do not pass in to the function are sought in keychains on the system. Certificates and certificate chains are discussed in *Security Overview*.

Listing 3-3 illustrates how you use Certificate, Key, and Trust functions to evaluate trust of a certificate. A detailed explanation for each numbered line of code follows the listing.

Listing 3-3 Evaluating trust

```
OSStatus EvaluateCert (SecCertificateRef cert, CFTypeRef policyRef,
                           SecTrustResultType *result,
                           SecTrustRef *pTrustRef)
{
    OSStatus status1;
    OSStatus status2;

    SecCertificateRef evalCertArray[1] = { cert };                              // 1
    CFArrayRef cfCertRef = CFArrayCreate ((CFAllocatorRef) NULL,
                                          (void *)evalCertArray, 1,
                                          &kCFTypeArrayCallBacks);
    if (!cfCertRef)
        return memFullErr;                                                      // 2
```

```
    status1 = SecTrustCreateWithCertificates (                          // 3
        cfCertRef,                                                      // 4
        policyRef,                                                      // 5
        pTrustRef
        );
    if (status1)
        return status1;

    status2 = SecTrustEvaluate (                                        // 6
        *pTrustRef,
        result                                                          // 7
        );
    // Release the objects we allocated
    if (cfCertRef)
        CFRelease(cfCertRef);
    if (cfDate)
        CFRelease(cfDate);

    return (status2);
}
int main (int argc, const char * argv[]) {
    OSStatus status;
    OSStatus status1;
    OSStatus status2;

    SecKeychainItemRef itemRef = nil;
    SecKeychainAttributeList attrList;
    SecKeychainAttribute attrib;
    attrList.count = 1;
    attrList.attr  = &attrib;
    attrib.tag = kSecAlias;
    attrib.data = "emailname@domain.com";
    attrib.length = strlen(attrib.data);

    const CSSM_OID *myPolicyOID = &CSSMOID_APPLE_X509_BASIC;
    SecPolicyRef policyRef = nil;

    SecTrustRef trustRef = nil;
    SecTrustResultType result;

    CFArrayRef certChain;
    CSSM_TP_APPLE_EVIDENCE_INFO *statusChain = nil;

    status = GetCertRef (&attrList, &itemRef);                          // 8
    status1 = FindPolicy (myPolicyOID, &policyRef);                     // 9

    status2 = EvaluateCert (
                            (SecCertificateRef)itemRef,
                            (CFTypeRef) policyRef,
                             &result, &trustRef);                       // 10
    .
    .
    .
    if (itemRef)
        CFRelease(itemRef);
    if (policyRef)
        CFRelease(policyRef);
```

```
    if (trustRef)
        CFRelease(trustRef);                                    // 11
    return (status2);
    }
```

Here's what the code does:

1. Makes a CFArray of one element from the provided certificate. See Listing 3-1 (page 25) for code to find a certificate on a keychain.
2. Returns with an error if it can't allocate the array.
3. Sets up trust evaluation criteria and gets a trust management object. This object must be disposed of when no longer needed.
4. Provides an array of certificates, containing the certificate to be evaluated and possibly intermediate and root certificates that might be needed in the evaluation. In this sample, the array contains only one certificate.
5. Specifies the policy reference object of the policy to used in evaluating this certificate. See Listing 3-2 (page 27) for code to obtain a policy reference object.
6. Evaluates the certificate according to the specified policy.
7. Returns a result object that is used to obtain information about the result of the evaluation; see Listing 3-5 (page 31).
8. Gets a certificate reference object; see Listing 3-1 (page 25).
9. Gets a policy reference object; see Listing 3-2 (page 27).
10. Evaluates the certificate and obtains a trust reference object. This object must be disposed of when no longer needed. The keychain item object is cast to a certificate object, which is possible because the certificate is on the keychain.
11. Disposes of the trust reference object, after it has been used to recover from a trust failure (Listing 3-5 (page 31)).

Recovering From a Trust Failure

There are several possible results of a trust evaluation, depending on such factors as whether all the certificates in the chain were found, whether they are all valid, and what the user trust settings are for the certificates. It is up to your application to determine the course of action based on the result of the evaluation. For example, if the result is `kSecTrustResultConfirm`, you should display a dialog requesting that the user give permission to proceed.

The evaluation result `kSecTrustResultRecoverableTrustFailure` indicates that trust was denied, but that it is possible to change settings to get a different result. For example, if the certificate used to sign a document has expired, you can change the date used for the evaluation to see whether the certificate was valid when the document was signed. The code in Listing 3-4 illustrates how to change the evaluation date.

Note that the `CFDateCreate` function takes an absolute time (the number of seconds since 1 January 2001); you can use the `CFGregorianDateGetAbsoluteTime` function to convert a calendar date and time into an absolute time.

Listing 3-4 Setting an evaluation date

```
OSStatus status;

CFAbsoluteTime expDate;
CFDateRef cfDate = nil;
expDate = 157680000;  //seconds since 1 Jan 2001
cfDate = CFDateCreate (NULL, expDate);
status = SecTrustSetVerifyDate (*pTrustRef, cfDate);
```

Listing 3-5 illustrates recovery from a trust failure caused by an expired certificate. In this case, the evaluation criteria are changed to ignore expiration dates. A detailed explanation for each numbered line of code follows the listing.

Listing 3-5 Recovering from a trust failure

```
 int n;
 CSSM_TP_APPLE_CERT_STATUS AllStatusBits = 0;

status3 = EvaluateCert (
                    (SecCertificateRef)itemRef,
                    (CFTypeRef) policyRef,
                    &result, &trustRef);                                    // 1

if (status3 == noErr)
    {
    if (result ==  kSecTrustResultRecoverableTrustFailure)                  // 2
        {
            status3 = SecTrustGetResult (
                                trustRef,                                   // 3
                                &result,                                    // 4
                                &certChain,                                 // 5
                                &statusChain                                // 6
                                );
            if (!status3 && statusChain)
            {                                                               // 7
                for (n = 0; n <                                             // 8
                        CFArrayGetCount(certChain); n++)
                AllStatusBits =
                    AllStatusBits | statusChain[n].StatusBits;
            if (AllStatusBits & CSSM_CERT_STATUS_EXPIRED)
                {
                    CSSM_APPLE_TP_ACTION_DATA actionData;                   // 9
                    actionData.Version =
                            CSSM_APPLE_TP_ACTION_VERSION;                   // 10
                    actionData.ActionFlags =
                        CSSM_TP_ACTION_ALLOW_EXPIRED |
                        CSSM_TP_ACTION_ALLOW_EXPIRED_ROOT;                  // 11

                    CFDataRef myActionData =                                // 12
                        CFDataCreateWithBytesNoCopy
                            (NULL, (UInt8*) &actionData,
```

```
                        sizeof(actionData),
                        kCFAllocatorNull);                          // 13

                if (myActionData)
                    {
                        status2 = SecTrustSetParameters (
                                    trustRef,
                                    CSSM_TP_ACTION_DEFAULT,
                                    myActionData);                  // 14
                        status3 = SecTrustEvaluate (
                                    trustRef,
                                    &result
                                    );                              // 15
                        status3 = SecTrustGetResult (
                                    trustRef,
                                    &result,
                                    &certChain,
                                    &statusChain);                  // 16

                        CFRelease(myActionData);
                    }
                }
                }
            }
    }
```

Here's what the code does:

1. Evaluates trust for a specific certificate and policy (see Listing 3-3 (page 28)).
2. Checks the result of the evaluation. If there was a recoverable trust failure, the routine goes on to obtain more information.
3. Passes the trust management object obtained earlier.
4. Passes a pointer to the result object obtained earlier.
5. Returns the certificate chain used to verify the evaluated certificate.
6. Returns an array of structures, each of which contains information about the status of one certificate in the chain.
7. If the function suceeds, checks for the validity of `statusChain`. The `statusChain` pointer is left uninitialized if `(result != kSecTrustResultRecoverableTrustFailure)` or if `(status3 != noErr)`.
8. Checks to see if the status bits of any certificates in the chain indicate an expired certificate. If so, you can recover from this condition.

   Before proceeding, you should prompt the user to ask permission to use expired certificates. You can check through the status chain to determine which certificate has expired and give that information to the user. If the user approves of using expired certificates, continue with the rest of this sample. If not, quit here.
9. Allocates space on the stack for an action data structure.
10. Fills in the `Version` field of the action data structure.

11. Fills in the `ActionFlags` field of the action data structure to allow expired certificates and root certificates.
12. Creates a CFDataRef from the action data.
13. Passes `kCFAllocatorNull` as the last parameter (`bytesDeallocator`) so that the bytes the CFDataRef points to aren't freed automatically.
14. Uses the action data to set the parameters for the trust object so that the next time it is evaluated, it will allow expired certificates.
15. Reevaluates the trust.
16. Checks the results of the evaluation.

Glossary

access control list (ACL) A structure that specifies the action required (for example, display a confirmation dialog, ask for a password) to permit a specific operation. 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 on that item, such as encrypting or decrypting it. See also access object (page 35).

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) (page 35).

anchor certificate A digital certificate trusted to be valid, which can then be used to verify other certificates. Anchor certificates can include root certificate (page 37)s, cross-certified certificates (that is, certificates signed with more than one certificate chain (page 35)), and locally defined sources of trust.

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.

asymmetric keys A pair of related but dissimilar keys, one used for encrypting, and the other used for decrypting, a message or other data. See also public key cryptography (page 37).

attribute One data item (other than the secret (page 37)) for a keychain item. Examples are the name, type, date modified, and account number. 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 The process by which an entity such as a user or a server gets the right to perform a privileged operation (page 37). (Authorization can also refer to the right itself, as in "Bob has the authorization to run that program.") Authorization usually involves first authenticating the entity and then determining whether it has the appropriate permissions (page 37). Compare authentication (page 35).

certificate See digital certificate (page 36).

certificate chain A sequence of related digital certificate (page 36)s that are used to verify the validity of a digital certificate. Each certificate is digitally signed using the certificate of its certification authority (page 35). This process creates a chain of certificates ending in an anchor certificate (page 35).

certificate extension A data field in a digital certificate (page 36) containing information such as allowable uses for the certificate.

certification authority The issuer of a digital certificate (page 36). In order for the digital certificate to be trusted, the certification authority must be a trusted organization that authenticates an applicant before issuing a certificate.

CDSA Abbreviation for 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 (page 36). In addition, Mac OS X includes its own security APIs that call the CDSA API for you.

CSSM Abbreviation for Common Security Services Manager. A public application programming interface for CDSA (page 35). 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 A collection of data used to verify the identity of the holder or sender of the certificate. A digital certificate must conform to some standard in order for the recipient to be able to interpret it. Mac OS X supports the X.509 (page 38) standard for digital certificates. See also certificate chain (page 35).

digital signature A data structure associated with a document or other set of data that uniquely identifies the person or organization that is signing, or authorizing the contents of, the data and ensures the integrity of the signed data.

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.

identity A digital certificate together with an associated private key.

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 database used to store encrypted passwords, private keys, and other secrets. It is also used to store certificates and other non-secret information that is used in cryptography and authentication. The Keychain Manager and Keychain Services are public APIs that can be used to manipulate data in the keychain, and the Keychain Access utility is an application that can be used for the same purpose. See also keychain item (page 36).

Keychain Access application A utility that allows users to create, delete, and modify keychains and keychain items. In addition, adding or removing keychains in Keychain Access modifies the default keychain search list accordingly.

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

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

level of trust The confidence you can have in the validity of a certificate, based on the certificates in its certificate chain (page 35) and on the certificate extension (page 35)s the certificate contains. The

level of trust for a certificate is used together with the trust policy (page 38) to answer the question "Should I trust this certificate for this action?"

MIME Acronym for Multipurpose Internet Mail Extensions. A standard for transmitting formatted text, hypertext, graphics, and audio in electronic mail messages over the Internet.

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

permissions The type of access allowed to a file or directory (read, write, execute, traverse, and so forth). Which permissions are possible and which users or groups are granted specific permissions depend on the operating system. See also authorization (page 35).

policy See trust policy (page 38).

private key A cryptographic key (page 36) that must be kept secret.

privileged operation An operation that requires special rights or permissions; for example, changing a locked system preference.

public key A cryptographic key that can be shared or made public without compromising the cryptographic method. See also public key cryptography (page 37).

public key certificate See digital certificate (page 36).

public key cryptography A cryptographic method using asymmetric keys (page 35) in which one key is made public while the other (the *private key*) is kept secure. Data encrypted with one key must be decrypted with the other. If the public key is used to encrypt the data, only the holder of the private key can decrypt it; therefore the data is secure from unauthorized use. If the private key is used to encrypt the data, anyone with the public key can decrypt it. Because only the holder of the private key could have encrypted it, however, such data can be used for authentication (page 35). See also digital certificate (page 36); digital signature (page 36).

public key infrastructure (PKI) As defined by the X.509 (page 38) standard, a PKI is the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificate (page 36)s that are based on public key cryptography (page 37).

root certificate A certificate (page 35) that can be verified without recourse to another certificate. Rather than being signed by a further certification authority (CA), a root certificate is verified using the widely available public key of the CA that issued the root certificate.

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 attribute (page 35).

secret key A cryptographic key that cannot be made public without compromising the security of the cryptographic method. In *symmetric key cryptography*, the secret key is used both to encrypt and decrypt the data. In *asymmetric key cryptography*, the secret key is paired with a public key. Whichever one is used to encrypt the data, the other is used to decrypt it. See also public key (page 37); public key cryptography (page 37).

Secure Sockets Layer (SSL) A protocol that provides secure communication over a TCP/IP connection such as the Internet. It uses digital certificate (page 36)s for authentication (page 35) and digital signature (page 36)s to ensure message integrity, and can use public key cryptography (page 37) to ensure data privacy. An SSL service negotiates a secure session between two communicating endpoints. SSL is built into all major browsers and web servers.

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

Secure Transport The Mac OS X implementation of Secure Sockets Layer (SSL) (page 37) and Transport Layer Security (TLS) (page 38), used to create secure connections over TCP/IP connections such as the Internet. Secure Transport includes an API that is independent of the underlying transport protocol.

S-MIME Acronym for Secure Multipurpose Internet Mail Extensions. A specification that adds digital signature (page 36) authentication and encryption to electronic mail messages in MIME (page 37) format.

SSL See Secure Sockets Layer (SSL) (page 37).

Transport Layer Security (TLS) A protocol that provides secure communication over a TCP/IP connection such as the Internet. It uses digital certificate (page 36)s for authentication (page 35) and digital signature (page 36)s to ensure message integrity, and can use public key cryptography (page 37) to ensure data privacy. A TLS service negotiates a secure session between two communicating endpoints. TLS is built into recent versions of all major browsers and web servers. TLS is the successor to SSL (page 38). Although the TLS and SSL protocols are not interoperable, Secure Transport (page 37) can back down to SSL 3.0 if a TLS session cannot be negotiated.

trust See level of trust (page 36), trust policy (page 38).

trusted application An application that can read a keychain item's secret when the keychain is unlocked. See also access control list (ACL) (page 35).

trust policy A set of rules that specify the appropriate uses for a certificate that has a specific level of trust (page 36). For example, the trust policy for a browser might state that if a certificate has an SSL certificate extension (page 35) but the certificate has expired, the user should be prompted for permission before a secure session is opened with a web server.

X.509 A standard for digital certificates promulgated by the International Telecommunication Union (ITU). The X.509 ITU standard is widely used on the Internet and throughout the information technology industry for designing secure applications based on a public key infrastructure (PKI) (page 37).

Document Revision History

This table describes the changes to *Certificate, Key, and Trust Services Programming Guide*.

Date	Notes
2010-07-09	Made iOS name changes.
2009-10-16	Fixed minor bugs.
2008-11-19	Added information about, and code samples for, iOS.
2004-06-28	New document that explains how to use Certificate, Key, and Trust Services to evaluate trust for a certificate and how to recover from a trust error.

Document Revision History