Security Overview

Security

2010-07-13

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to Security Overview

Computer security has been much in the news in recent years, whether it's the latest computer virus spread via email, questions about the safety of secrets on the computers of our national weapons laboratories, or stories about hackers stealing thousands of credit card numbers from an online vendor. Whatever the source of attack, the problem is the same: how to protect information and software from being accessed by unauthorized people. In response to this need, Apple has built a number of security features into the Mac OS X and iOS operating systems and provides a variety of APIs that developers can use to make their applications more secure.

This document describes the security features of Mac OS X and iOS, explains concepts that you must understand in order to use the security APIs, and describes the security APIs. This document is intended for anyone who is interested in computer security, but especially for developers new to the security APIs in Mac OS X and iOS. No programming knowledge is assumed, though it will be helpful if you have some familiarity with Apple computers or mobile devices. If you are a software developer, this book will help you understand the security architecture of Mac OS X and iOS and will help you determine which of the available security features and APIs will be of most use to you. It will also direct you to further documentation and sample code, so you can get started writing secure code more quickly.

This book does not discuss how to write secure software. While the security APIs can help you write programs that are resistant to unauthorized access, malicious attacks often exploit vulnerabilities caused by avoidable coding errors. For more information on writing secure code, see *Secure Coding Guide* and the books listed in the section "Books on Computer Security" (page 9).

Organization of This Document

This document describes the security architecture of Mac OS X and iOS, explains some concepts common to computer security on all platforms, describes some features specific to security in Mac OS X and iOS, and describes the APIs that are useful in computer security. Where appropriate, it points out the differences in APIs and security features between Mac OS X and iOS. This document contains the following chapters:

- "Security Architecture" (page 11) describes and diagrams the operating system components that contribute to the security of data both on an individual device and across networks. It shows where each of the major security APIs fits into the architecture of the operating system.
- "Security Concepts" (page 23) introduces and explains concepts and technologies important to keeping data secure and to preventing unauthorized access of files over a network. The section "Mac OS X" (page 57) describes the differences in access permissions between Mac OS X and other UNIX systems, and the section "Network File Systems" (page 59) describes the extent to which various networking protocols, including Apple Filing Protocol (AFP), implement Mac OS X access permissions.
- "Security Services" (page 65) describes all of the Mac OS X and iOS APIs that you can use to create secure applications or to ensure security over a network. It also has brief descriptions of some user-level security features, such as the Keychain Access application and FileVault.
- "Glossary" (page 81) defines security-related terms used in this document.

See Also

For a general background on Mac OS X and iOS security, use the following resources:

- To get the latest updates on Apple's security services and for pointers to other Apple security resources, go to the ADC technology page for security at http://developer.apple.com/security/.
- Reference and conceptual documentation, technical notes, Q&As, and sample code for security APIs are available from Reference Library > Security on the ADC Mac OS X website or [Topics > Security] on the iPhone Reference Library site.
- For an introduction to Mac OS X system architecture and system technologies, see *Mac OS X Technology Overview*.
- For an introduction to iOS system architecture and system technologies, see *iOS Technology Overview*.
- To see which security protocols and algorithms are supported by Apple's Mac OS X security implementation, see the documentation provided with the Open Source security code, which you can download at http://developer.apple.com/opensource/security/, and the Security Release Notes in the latest Xcode Tools from Apple.

Security API Documentation

For documentation on the security APIs, see the following documents:

- For information on Mac OS X Authorization Services, see *Authorization Services C Reference* and *Authorization Services Programming Guide*. Authorization services APIs are not available on iOS.
- Technical Note TN2095, *Authorization for Everyone*, also discusses the use of Authorization Services.
- To learn how to store and retrieve secrets and certificates using the keychain, see *Keychain Services Programming Guide* and *Keychain Services Reference*.
- To learn how to read and validate certificates, see *Certificate, Key, and Trust Services Reference*.
- See *Security Interface Framework Reference* in Reference Library > Security for an objective-C interface to Authorization Services and for a variety of security-related user interface elements (Mac OS X only).
- For information about the Secure Transport API, see *Secure Transport Reference* (Mac OS X only).
- If you are want to set up a secure data stream, see *CFNetwork Programming Guide*.
- To add passwords to QuickTime movies, see "Movie Toolbox Access Keys".

Standards and Protocol References

For information on standards, protocols, and algorithms used by Apple, see the following sources:

- The authentication model for HTTP is described in RFC 2617, *HTTP Authentication: Basic and Digest Access Authentication*, which you can find at http://www.ietf.org/rfc/rfc2617.txt.
- For information on the SSL protocol for secure networking, see http://wp.netscape.com/eng/ssl3/. For the TLS protocol, see http://www.ietf.org/html.charters/tls-charter.html.

- CDSA, implemented as part of the Mac OS X security architecture, is an Open Source standard by the Open Group (http://www.opengroup.org/security/cdsa.htm). For an introduction to CDSA, see *CDSA Explained*, second edition, from the Open Group. The CDSA/CSSM technical standard is *Common Security: CDSA and CSSM*, version 2 (with corrigenda), also from the Open Group.
- Documentation of the AES encryption algorithm used for FileVault is available on the National Institute of Standards and Technology (NIST) website at http://csrc.nist.gov/CryptoToolkit/aes/rijndael/.
- For information on Kerberos authentication, see http://web.mit.edu/kerberos/. For information on MIT's Kerberos for Macintosh, see http://web.mit.edu/macdev/Development/MITKerberos/MITKerberosLib/Common/Documentation/KerberosFramework.html
- See *Mac OS X Server Open Directory Administration* available at http://www.apple.com/server/documentation/ for details on the services that support Kerberos and on how to implement a Kerberos KDC on your Mac OS X server.
- The PC/SC Workgroup has established a standard for accessing smart cards and writing card reader drivers. Their website is at http://www.pcscworkgroup.com/.
- Apple's smart card support is based on the Movement for the Use of Smart Cards in a Linux Environment (MUSCLE) Open Source implementation of the PC/SC standard. The MUSCLE home page is http://www.linuxnet.com/.

Books on Computer Security

You may find the following books useful in learning more about security, cryptography, and networking.

- Garfinkel, Simson, Gene Spafford, and Alan Schwartz. *Practical Unix & Internet Security*. 3d ed. O'Reilly & Associates, Inc. 2003.
- Brands, S. *Rethinking PKI and Digital Certificates: Building in Privacy*. The MIT Press. 2000.
- Gray, John Shapley. *Interprocess Communications in UNIX*. 2d ed. Prentice Hall Professional. 1997.
- McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley. 1996.
- Schneier, Bruce. *Applied Cryptography*. 2d ed. Wiley. 1996.
- Stevens, Richard W. *UNIX Network Programming: Interprocess Communications*. Vol. 2, 2d ed. Prentice Hall Professional. 1998.
- Stevens, Richard W. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Vol. 1. Prentice Hall Professional. 1997.
- Viega, John, and Gary McGraw. *Building Secure Software*. Addison-Wesley. 2002.

Security Architecture

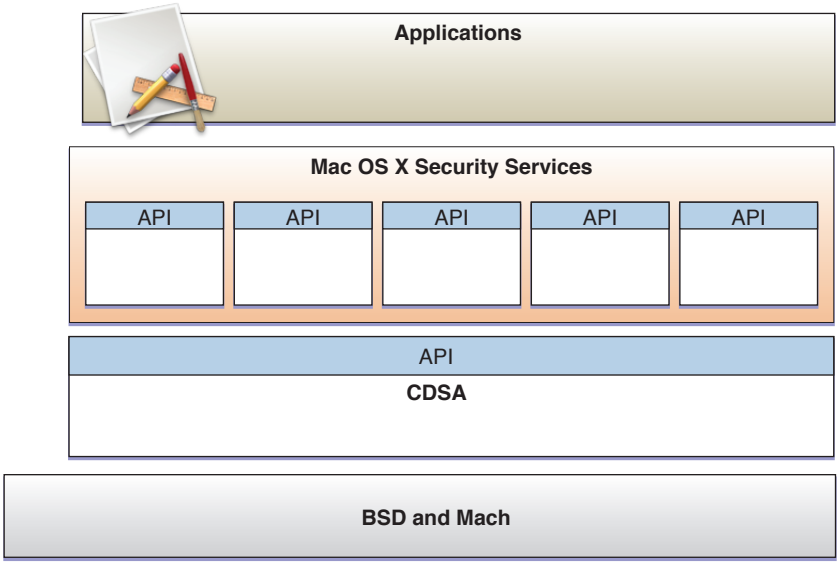
This chapter describes the components of security services on Mac OS X (in the following section) and iOS (see "iOS" (page 20)). For each platform, the available security APIs are briefly described as well. For more extensive descriptions of the security APIs in each platform, see "Security Services" (page 65).

Mac OS X

Mac OS X security services are built on two open-source standards: BSD (Berkeley Software Distribution) and CDSA (Common Data Security Architecture). BSD is a form of the UNIX operating system and provides fundamental services, such as the basis for the Mac OS X file system, including file access permissions. CDSA provides a much wider array of security services, including finer-grained access permissions, authentication of users' identities, encryption, and secure data storage. Although CDSA has its own standard application programming interface (API), it is complex and does not follow standard Macintosh programming conventions. Therefore, Mac OS X includes its own security APIs that call the CDSA API for you.

The Mac OS X security architecture is layered, with BSD on the bottom, CDSA in the middle, Mac OS X security APIs above that, and applications that call the security services at the top. Figure 1-1 illustrates this architecture.

Figure 1-1 Mac OS X security architecture overview



BSD and Mach

The Mac OS X kernel—the heart of the operating system—is built from BSD and Mach. Among other things, **BSD** provides basic file system and networking services and implements a user and group identification scheme. BSD enforces access restrictions to files and system resources based on the user and group IDs. **Mach** provides memory management, thread control, hardware abstraction, and interprocess communication. Mach enforces access by controlling which tasks can send a message to a given Mach port, where a Mach port represents a task or some other Mach resource (see "Mach Port Rights" (page 45) for a discussion of Mach ports). BSD and Mach are both critical to enforcing local security; if you can break the security provided by one, you can break the other as well.

BSD security policies and Mach access permissions constitute an essential part of security in Mac OS X, but fall far short of all the capabilities that are needed. For example, although BSD can restrict access to a file to a particular user ID, it provides no facility for checking passwords or otherwise verifying that the person or process using that ID is who they claim to be. Similarly, once Mach has given the rights to one task to control another, it cannot restrict the data or features the controlling task can access.

It's important to note that BSD and Mach access permissions are enforced by the operating system and therefore affect every process running in Mac OS X. In contrast, application-defined security policies must be enforced by those applications. The Mac OS X security APIs discussed in this document are available for that purpose.

For more information about the roles BSD and Mach play in Mac OS X, see *Mac OS X Technology Overview*. For a more detailed description of Mach and BSD access permissions, see "Mach Port Rights" (page 45) and "BSD" (page 46).

CDSA

CDSA is an Open Source security architecture adopted as a technical standard by the Open Group (http://www.opengroup.org/security/cdsa.htm). Apple has developed its own Open Source implementation of CDSA, available as part of Darwin at http://developer.apple.com/darwin/projects/security/. The core of CDSA is **CSSM** (Common Security Services Manager), a set of Open Source code modules that implement a public application programming interface called the CSSM API. CSSM provides APIs for cryptographic services (such as creation of cryptographic keys, encryption and decryption of data), certificate services (such as creation of digital certificates, reading and evaluation of digital certificates), secure storage of data, and other security services (see "Apple CDSA Plug-ins" (page 14) for a more complete list).

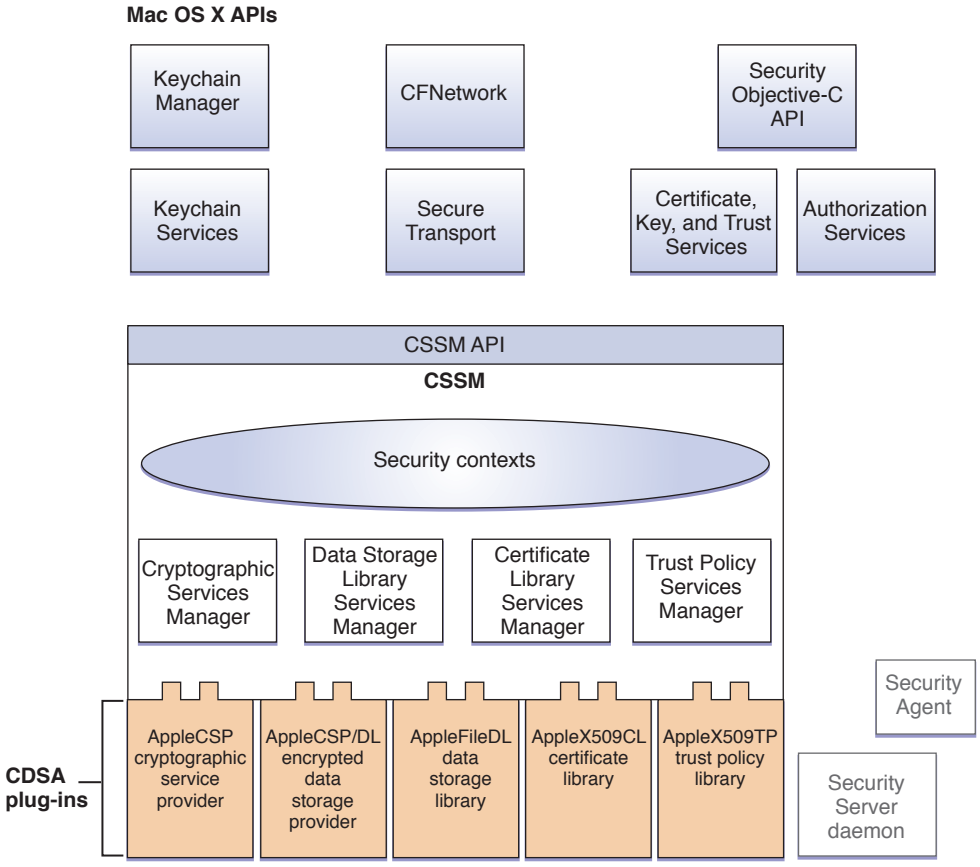
CSSM also defines an interface for plug-ins that implement security services for a particular operating system and hardware environment. The implementation on a given platform can optionally supply a middleware layer that provides an operating-system-specific API for applications. Whether such a layer is present or not, applications can call the CSSM API directly.

Mac OS X implements nearly all the standard features of CSSM, plus a set of middleware security services to provide a Mac OS X-standard interface for application programmers. In addition, to enhance the security of the most sensitive operations, the Mac OS X implementation runs a Security Server daemon as a separate process. The Security Server daemon launches another process, the Security Agent, which serves as the user interface for Security Server.

Figure 1-2 illustrates the Mac OS X implementation of CDSA. The CDSA standard defines a four-layer architecture, with the top layer being the applications that use the CDSA security features. Figure 1-2 shows the first three layers: the CDSA plug-ins, CSSM, and the Mac OS X security APIs, which constitute the middleware

layer referred to in the specification. The Mac OS X Authorization Services API, the Security Server daemon, and the Security Agent shown in the figure are technically outside of CDSA, but they are shown here for completeness because they constitute an integral part of the Mac OS X security architecture.

Figure 1-2 Mac OS X implementation of CDSA



Security contexts (see Figure 1-2) are data structures used by CSSM to assist applications in managing the many parameters used in security operations. The CSSM managers implement the standard CSSM API. (A fifth manager defined in the CDSA standard, called the Authorization Computation Services Manager, is not implemented in Mac OS X. Instead of using a CSSM API, Mac OS X Authorization Services calls the Security Server daemon directly.)

The CDSA plug-ins shown in Figure 1-2 are those provided as part of Mac OS X. The CDSA specification allows any number of plug-ins. As long as a plug-in follows the rules for interfacing with the CSSM managers, it can implement any portion of the CDSA feature set, including a combination of features associated with two or more of the CSSM managers. (See "AppleCSP/DL Module" (page 14) for an example of a multiservice CDSA plug-in.) The CDSA specification even allows for the expansion of CDSA by the addition of elective module managers and associated plug-ins. Plug-ins can call each other as well as being called by the CSSM managers—in fact, it is common for them to do so.

For an introduction to CDSA, see *CDSA Explained*, second edition, from the Open Group. The CDSA/CSSM technical standard is *Common Security: CDSA and CSSM*, version 2 (with corrigenda), also from the Open Group.

As long as you use the Mac OS X security APIs, you don't have to worry about the details of the Mac OS X CDSA implementation. However, because a call to the CSSM API allows you to specify the plug-in module to which you want to direct your request, if you want to call CSSM directly you should have some understanding of the Mac OS X CDSA plug-ins.

Apple CDSA Plug-ins

The Mac OS X implementation of CDSA includes five CDSA plug-ins (see Figure 1-2 (page 13)):

- AppleCSP cryptographic service provider
- AppleFileDL data storage library
- AppleCSP/DL encrypted data storage provider
- AppleX509CL certificate library
- AppleX509TP trust policy library

This section briefly describes the purpose and function of each of these modules. (See "Glossary" (page 81) for explanations of any unfamiliar terms.)

AppleCSP Module

All secure communications and authentication protocols are based on keys and encryption. The Apple cryptographic service provider (AppleCSP) is a basic plug-in module used by several of the security services for creating cryptographic keys and encrypting or decrypting data. Digital signatures also use the AppleCSP module to create message digests used to create and verify the signature. A CSP can use any number of algorithms.

AppleFileDL Module

A data storage library (DL) module provides persistent **secure storage**; that is, storage of encrypted data on disk or another medium that persists when the power is turned off. The CDSA standard allows a DL module to use any sort of database or other data store. Keeping things simple, the AppleFileDL module stores its data in files in the Mac OS X file system. It provides lower-level services used by the AppleCSP/DL plug-in for storing secrets on the **keychain**, Apple's database used to store encrypted passwords, private keys, and other secrets.

AppleCSP/DL Module

The AppleCSP/DL plug-in is a multifunction module that combines cryptographic service and data storage functions to implement the Apple keychain, used for storage of passwords, keys, and other secrets. The AppleCSP/DL module calls the AppleFileDL module to perform file I/O, and the Security Server daemon to encrypt and decrypt secrets.

AppleX509CL Module

A certificate library (CL) module performs operations on digital certificates. Digital certificates are used to establish or confirm the identity of an entity such as a website or the sender of a digitally signed message. They do so by using a digital signature to ensure that only the identified entity could have provided the

certificate (see "Digital Certificates" (page 31)). A CL module performs such functions as creating new certificates (in memory), creating certificate revocation lists (that indicate which certificates are no longer valid), verifying the digital signature contained in a certificate, and extracting information from a certificate. CL modules do not store persistent copies of certificates. Rather, a DL module is used for that purpose.

The AppleX509CL plug-in performs these functions for certificates that conform to the X.509 standard 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** (the set of hardware, software, people, policies, and procedures needed to create and manage digital certificates that are based on public key cryptography). See "Asymmetric Keys" (page 26) for more information on public keys.

AppleX509TP Module

A digital certificate has a level of trust associated with it, based on attributes of the certificate. A trust policy is a set of rules that specify the actions that can be taken given a specific level of trust. In other words, the purpose of establishing a level of trust for a certificate is to answer the question "Should I trust this certificate for this action?"

The issuer of a digital certificate adds a digital signature to the certificate to ensure that the certificate has not been altered and to verify the identity of the issuer. In general, a digital signature is verified through the use of another certificate. Consequently, each certificate is typically part of a chain of certificates that ends with a **root certificate**, which can be verified without recourse to another certificate (see "Digital Certificates" (page 31)).

Note: The set of root certificates stored and maintained by Mac OS X is in a system keychain at /System/Library/Keychains/X509Anchors. You can browse these keychains in the Keychain Access utility (by using the File > Add Keychain menu item to add them to your list of keychains) to see what they contain and how the certificate chains are constructed.

A trust policy (TP) plug-in performs two main functions: it assembles the chain of certificates needed to verify a given certificate, and it determines the level of trust that can be accorded the certificate.

The AppleX509TP module performs these functions on X.509 certificates, using trust policies established by Apple.

Security Server Daemon

The Mac OS X security implementation includes a daemon, called the Security Server, and a separate process, called the Security Agent, that is used by the Security Server for a user interface. This section briefly describes their roles.

Security Server

The **Security Server** is a daemon running in Mac OS X that implements several security protocols, such as encryption, decryption, and authorization computation. The use of the Security Server to perform actions with cryptographic keys enables the security implementation to maintain the keys in a separate address space from the client application, keeping them more secure. The Security Server also provides an interface

in which developers use references to keys rather than using the keys directly. With this architecture, if Apple introduces a new technology, such as security smart cards, existing applications that use the Mac OS X security APIs automatically work with it.

Because Authorization Services performs functions not provided by the CDSA specification, the Authorization Services API interfaces directly with the Security Server rather than going through CDSA. The Security Server hosts authorization plug-ins that it can load as needed. This feature, like the use of key references, makes it possible to add new technologies without requiring applications to modify their code. In fact, because the Security Server handles its own communication with the user (through the Security Agent), application developers don't have to worry about writing new user interface code to support new authentication technologies, such as fingerprint readers or iris scanners.

The Security Server has no public API. Instead, Authorization Services communicates directly with the Security Server and Keychain Services communicates with it indirectly (through the AppleCSP/DL plug-in module). In turn, the Security Server calls the AppleCSP plug-in module to perform cryptography. The Security Server uses the Security Agent to communicate with the user through dialogs and other user interface elements.

Security Agent

The **Security Agent** is a separate process that deals with all of the security-related user interface for the Security Server. For example, when the Security Server requires the user to authenticate, the Security Agent opens a dialog requesting a user name and password. If Apple introduced an authentication smart card reader for Macintosh computers, a new Security Server plug-in and modifications to the Security Agent would enable the Security Agent to prompt the user to insert their card instead of entering their user name and password. The application developer would not have to do a thing to get this new behavior.

The Security Agent runs with restricted permissions so that the user must be physically present, using the graphical user interface, in order to be authenticated. The graphical user interface elements can't be used through a command-line interface such as the Terminal application or a secure shell (ssh) remote session. This restriction makes it much more difficult for a malicious user to breach an application's security.

CSSM Services

Although the Mac OS X security APIs provide all the capabilities you are ever likely to need for developing secure applications, nearly all the standard CSSM APIs are also available for your use. This section briefly describes the functions provided by each CSSM service. For details, see *Common Security: CDSA and CSSM*, version 2 (with corrigenda), from the Open Group (http://www.opengroup.org/security/cdsa.htm).

Cryptographic Services

Cryptographic Services in CSSM provides functions to perform the following tasks:

- Encrypting and decrypting text and data
- Creating and verifying digital signatures
- Creating a cryptographic hash (used for message digests and other purposes)
- Generating symmetric and asymmetric pairs of cryptographic keys
- Generating pseudorandom numbers
- Controlling access to the CSP for creation of keys

To see exactly which security protocols and algorithms are supported by Apple's CSP implementation, see the documentation provided with the Open Source security code, which you can download at http://developer.apple.com/darwin/projects/security/, and the Security Release Notes in the latest Xcode Tools from Apple.

Data Store Services

CSSM Data Store Services provides an API for storing and retrieving data that is independent of the type of storage used. If there is more than one DL module installed, the caller can query Data Store Services to learn the capabilities of each and select which one to use in a particular call. The Apple implementation of Data Store Services supports any standard CDSA DL plug-in module. The AppleFileDL Data Storage Library and AppleCSP/DL Encrypted Data Storage module both implement functions called by Data Store Services.

Certificate Services

Certificate Services as specified by CDSA performs the following functions:

- Verifies the signatures on certificates and certificate revocation lists
- Creates certificates and certificate revocation lists
- Signs certificates and certificate revocation lists
- Extracts values of fields from certificates and certificate revocation lists
- Searches certificate revocation lists for specified certificates

Apple's implementation of Certificate Services supports all of the CL API functions in the CDSA/CSSM specification.

Trust Policy Services

The Mac OS X implementation of CSSM Trust Policy Services provides functions to verify certificates, to determine what attributes they contain and therefore the level of trust they can be given, and to construct a chain of related certificates. It does not implement other trust policy functions in the CSSM standard. Documentation for the CSSM trust policy functions supported by Apple's TP implementation can be found with the Open Source security code, which you can download at http://developer.apple.com/darwin/projects/security/.

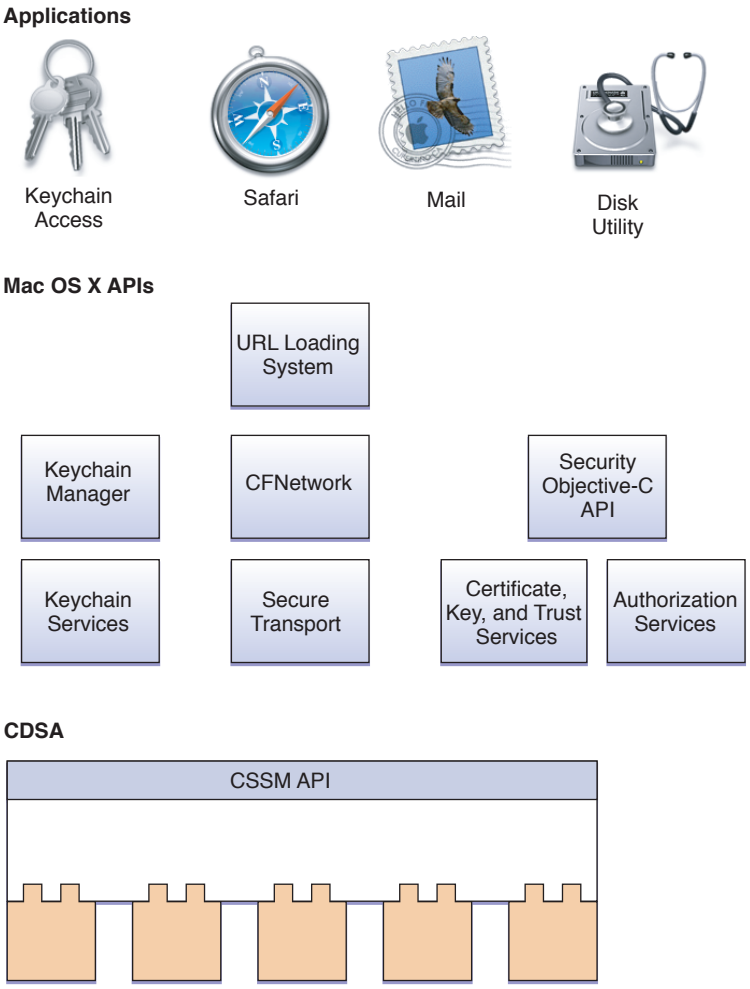
Authorization Computation Services

Apple's implementation of CSSM does not include the Authorization Computation Services defined in the CDSA standard. Instead, the Authorization Services API calls the Security Server daemon directly (see Figure 1-2 (page 13)).

Mac OS X Security APIs

In the context of CDSA, the Mac OS X security APIs constitute a middleware layer between the CSSM APIs and applications that use CDSA (Figure 1-3). This section discusses how each of the Mac OS X security APIs fits into the overall security architecture. See "Security Services" (page 65) for a more detailed description of each API.

Figure 1-3 Mac OS X security APIs



Keychains

The keychain is used to store passwords, keys, certificates, and other secrets. Its implementation therefore requires both cryptographic functions (to encrypt and decrypt secrets) and data storage functions (to store the secrets and related data in files). To achieve these aims, Keychain Services calls the CSSM API, which calls the AppleCSP/DL CDSA plug-in module. The AppleCSP/DL module in turn calls the AppleFileDL module to perform file I/O, and the Security Server daemon to encrypt and decrypt secrets.

Keychain Manager is maintained only for compatibility with older versions of Mac OS X (v10.1 and earlier). For newer versions of Mac OS X (starting with Mac OS X v10.2), Keychain Manager calls Keychain Services rather than calling CSSM directly. Because Keychain Services is more efficient and offers more features than Keychain Manager, any new code should be written to use Keychain Services. For more information on the differences between these two APIs, see "Keychain Manager and Keychain Services" (page 69).

Secure Transport

Secure Transport is used to implement the **Secure Sockets Layer** (**SSL**) and **Transport Layer Security** (**TLS**) protocols, which provide secure communications over a TCP/IP connection such as the Internet. These protocols use encryption and certificate exchange to prevent eavesdropping and message tampering. Secure Transport calls CSSM to manage certificates and keys and to encrypt and decrypt data.

CFNetwork

CFNetwork is an API that provides network services. It provides a higher-level interface than Secure Transport that can be used by applications to create and maintain secure data streams and to add authentication information to a message. CFNetwork calls Secure Transport to set up a secure connection.

URL Loading System

URL Loading System is another high-level transport API. In contrast to CFNetwork, it is used to access the contents of URLs rather than to maintain a data stream. Because URL Loading System works with secure https:// URLs, it can be used for secure transport of data. URL Loading System is built on top of CFNetwork.

Certificate, Key, and Trust Services

The **Certificate, Key, and Trust Services** API includes functions to:

- Create, manage, and read certificates
- Add certificates to a keychain
- Create encryption keys
- Manage trust policies

To carry out all these services, the API calls a variety of CSSM functions, ultimately using the services of all of the CSSM managers discussed in "CSSM Services" (page 16).

Authorization Services

Authorization Services gives applications control over access to specific operations within the application. For example, a directory application that can be started by any user can use Authorization Services to restrict access for modifying directory items to administrators. In contrast, BSD provides access permission only to an entire file or application. Authentication Services calls the Security Server daemon directly. In turn, the Security Server calls specific CDSA plug-ins as needed.

Security Objective-C API

The **Security Objective-C API** is provided for the convenience of Cocoa programmers. It provides a set of Objective-C methods that are wrappers for the Authentication Services functions. In addition, it provides classes that display security-related UI elements, such as the contents of certificates, interfaces to create keychains and change keychain settings, and others. Depending on the class in use, this API calls Authorization Services; Certificate, Key, and Trust Services; or Keychain Services.

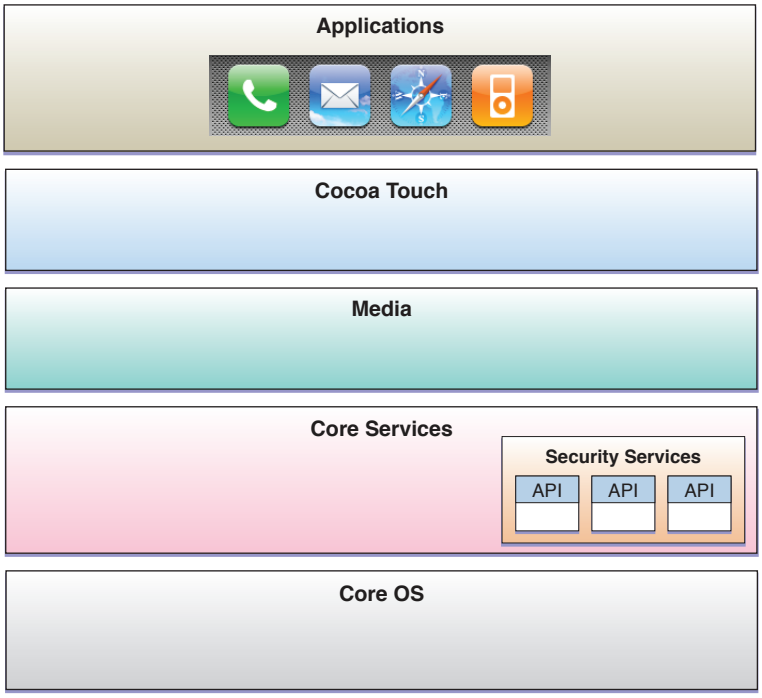
Movie Toolbox Access Keys

Movie Toolbox Access Keys is a QuickTime API that can be used to add password protection to QuickTime data. QuickTime implements its own mechanism for storing and verifying passwords. QuickTime does not call Authorization Services or any CSSM functions.

iOS

Figure 1-4 illustrates the architecture of the iOS and shows the location of the security services. The iOS security APIs are located in the Core Services layer of the operating system and are based on services in the Core OS (kernel) layer of the operating system. Applications on the iPhone call the security services APIs directly rather than going through the Cocoa Touch or Media layers. Networking applications can also access secure networking functions through the CFNetwork API, which is also located in the Core Services layer.

Figure 1-4 iOS security architecture overview



Security Server Daemon

The iOS security implementation includes a daemon called the **Security Server** that implements several security protocols, such as access to keychain items and root certificate trust management.

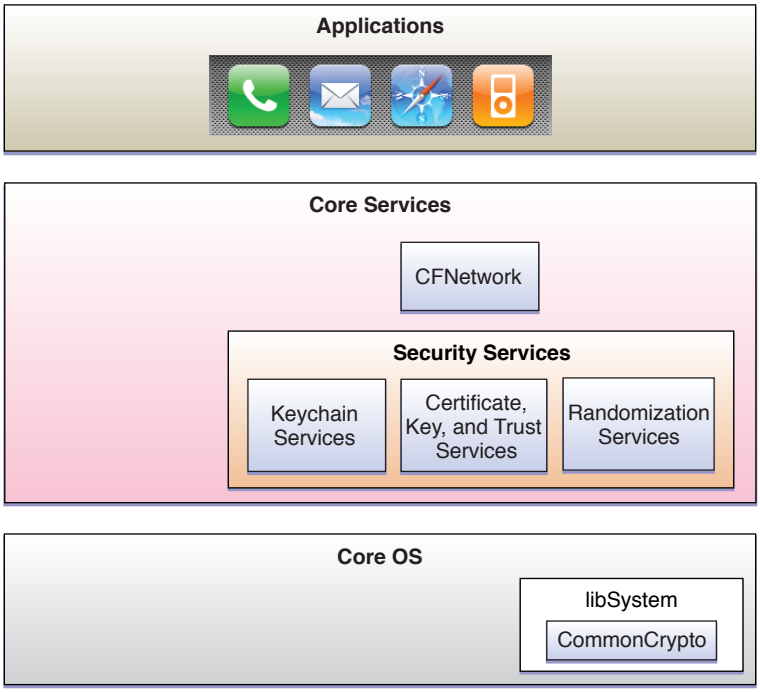
The Security Server has no public API. Instead, applications use the Keychain Services API and the Certificate, Key, and Trust services API, which in turn communicate with the Security Server. Because, unlike the Mac OS X security services, the iOS security services do not provide an authentication interface, there is no need for the Security Server to have a user interface.

iOS Security APIs

The iOS security APIs are based on services in the Core Services layer, including the Common Crypto library in the libSystem dynamic library (Figure 1-5). This section discusses how each of the iOS security APIs fits into the overall security architecture. See "Security Services" (page 65) for a more detailed description of each API. For more information about the Common Crypto library, see the manual page for CC_crypto(3cc) in *iOS Manual Pages*.

OpenSSL Note: Although Mac OS X includes a low-level command-line interface to the OpenSSL open-source cryptography toolkit, this interface is not available on the iOS. For iOS development, use the CFNetwork API for secure networking and the Certificate, Key, and Trust Services API for cryptographic services.

Figure 1-5 iOS security APIs



Keychain

The **keychain** is used to store passwords, keys, certificates, and other secrets. Its implementation therefore requires both cryptographic functions (to encrypt and decrypt secrets) and data storage functions (to store the secrets and related data in files). To achieve these aims, Keychain Services calls the Common Crypto dynamic library.

CFNetwork

CFNetwork is a high-level API that can be used by applications to create and maintain secure data streams and to add authentication information to a message. CFNetwork calls underlying security services to set up a secure connection.

Certificate, Key, and Trust Services

The **Certificate, Key, and Trust Services** API includes functions to:

- Create, manage, and read certificates
- Add certificates to a keychain
- Create encryption keys
- Encrypt and decrypt data
- Sign data and verify signatures
- Manage trust policies

To carry out all these services, the API calls the Common Crypto dynamic library and other Core OS–level services.

Randomization Services

Randomization Services provides cryptographically secure **pseudorandom numbers**. Pseudorandom numbers are generated by a computer algorithm (and are therefore not truly random), but the algorithm is not discernible from the sequence. To generate these numbers, Randomization Services calls a random-number generator in the Core OS layer.

Security Concepts

This chapter introduces several concepts necessary to understanding the Mac OS X and iOS security APIs. It does not explain any of these concepts in depth, provide programming algorithms, or give the mathematical foundations of cryptography. If you are already familiar with security concepts, you may skip this chapter. If you are already familiar with security concepts in general but would like to know more about how permissions are implemented in Mac OS X, see "Permissions" (page 44).

For references to more detailed sources of information on security concepts and on how to write secure applications, see "See Also" (page 8).

Aspects of Security

The fundamental purpose of security is to control who has access to valuable property, whether physical or intellectual. This is the reason we have locks on the doors of our houses, why the military encrypts classified information, and why Mac OS X and iOS enable users to require a password or PIN every time someone logs on to their computer, iPhone, or iPod Touch.

Security features on a personal computer can be classified into two general groups: those designed to protect programs and data on the computer from unauthorized access by users on the system ("local security"); and those designed to protect the system, programs, and data from unauthorized access over a network or other transport medium, such as removable disks ("remote transport security").

When considering local security, you must be aware of whether access is being controlled by the operating system or by the application itself.

Local Security

Local security is important when a computer is being shared, such as in libraries or schools; or when an unauthorized person might get access to the computer, such as a computer kept in an open cubicle in a large office. Security features useful in such environments include the password protection offered by the Finder, encryption of data provided by FileVault, BSD access permissions, and access permissions added to applications through use of Authorization Services.

Remote Transport Security

Remote transport security is important to all users, and especially to users whose computers are connected to a LAN or to the Internet. Web browsers, for example, use secure transport protocols ("Protocols for Secure Communication" (page 62)) to protect data from interception while in transit, digital signatures ("Digital Signatures" (page 29)) to ensure data integrity, and digital certificates ("Digital Certificates" (page 31)) to verify the identity of people or servers trying to get access to data. Many of the security APIs provided by

Mac OS X and iOS are useful in this regard, including the secure networking APIs (Secure Transport, CFNetwork, and URL Loading System), Keychain Services (used to store certificates, passwords, and encryption keys), and Certificate, Key, and Trust Services.

System-Restricted or Self-Restricted Access

It is important to understand that certain forms of access permission are enforced by the operating system, whereas others are enforced by individual applications. BSD permissions ("BSD" (page 46)) control who can execute a program or open a file, and are built into the operating system. On the other hand, if you want finer-grained control over access, such as restricting certain operations to a subset of users, you must enforce these restrictions yourself. Authorization Services provides functions you can use to implement such restrictions, and you can make the restrictions optional so that they operate only when your application is being used in an environment where they are necessary. For example, you might want to restrict access to some application preferences to administrators on a shared computer but not require a password when the computer is not shared. See Authorization for Everyone, Technical Note TN2095, for techniques and sample code for implementing self-restricted access permissions.

Authentication, Identification, and Authorization

Authentication is the process by which a person or other entity (such as a server) proves that it is who (or what) it says it is. Authentication is achieved through presenting something that you know, something that you have, some unique identifying feature, or some combination of these. A common example is the way you authenticate yourself in order to use a teller machine: you insert your ATM card (something you have) and enter your personal identification number (PIN, something you know). Unique identifying features include such things as fingerprints, retina patterns, and voice prints.

It is always desirable to authenticate the person or server with which you are dealing before transferring something valuable, such as information or money. Authentication, however, is time consuming and can inconvenience users. For example, once having shown your photo ID card to enter the Apple Worldwide Developer's Conference, you would not want to get it out again every time you walked into one of the conference rooms. To make situations like this more convenient and efficient, many systems use some method of **identification**, which verifies that the person or entity is the same one you communicated with last time. The means of identification can be through the use of a ticket or token issued when authentication is done. For example, the conference badge you are given to wear during the Developer's Conference identifies you as a legitimate attendee who was authenticated when you first came in.

In general, authentication or identification is not sufficient to gain access to information or code. For that, the entity requesting access must have **authorization**. Authorization requires first a determination that the authenticated entity has the appropriate **permissions**—that is, the right to the specific type of access (such as read, write, or execute) requested—and then the actual granting of that access. For example, the mere possession of a conference badge does not grant you the right to enter a restricted area, such as the speakers' preparation room. You must have permission to enter this area (indicated, in this case, by the color of your badge), and you must be granted access by the guard at the door.

Authentication and identification methods available in Mac OS X are described in "Authentication and Identification Methods" (page 35). Permissions in BSD and Mac OS X are described in "Permissions" (page 44). Authorization is discussed in more detail in "Authorization" (page 61). Note that iOS relies on the device's PIN and the sandboxing ("Sandboxing and the Mandatory Acccess Control Framework" (page 47)) of applications to provide security—no authorization or authentication interface is provided for iOS.

Encryption and Decryption

Most of the security APIs in Mac OS X and iOS rely to some degree on encryption of text or data. For example, encryption is used in the creation of certificates and digital signatures, in secure storage of secrets in the keychain, and in secure transport of information. For the purposes of this book, **encryption** is defined as the transformation of data into a form in which it cannot be made sense of without the use of some **key**. Such transformed data is referred to as **ciphertext**. Use of a key to reverse this process and return the data to its original (or **plaintext**) form is called **decryption**.

Encryption can be anything from a simple process of substituting one character for another—in which case the key is the substitution rule—to a complex mathematical algorithm. For purposes of security, the more difficult it is to decrypt the ciphertext, the better. On the other hand, if the algorithm is too complex, takes too long to do, or requires keys that are too large to store easily, it becomes impractical for use in a personal computer. Therefore, some balance must be reached between **strength** of the encryption (that is, how difficult it is for someone to discover the algorithm and the key) and ease of use.

For practical purposes, the encryption need only be strong enough to protect the data for the amount of time the data might be useful to a person with malicious intent. For example, if you need to keep your bid on a contract secret only until after the contract has been awarded, an encryption method that can be broken in a few weeks will suffice. If you are protecting your credit card number, you probably want an encryption method that cannot be broken for many years.

There are two main types of encryption in use in computer security, referred to as *symmetric key encryption* and *asymmetric key encryption*. A closely related process to encryption, in which the data is transformed using a key and a mathematical algorithm that cannot be reversed, is called **cryptographic hashing**. The remainder of this section discusses encryption keys, key exchange mechanisms (including the Diffie-Hellman key exchange used in some Mac OS X secure transport protocols), and cryptographic hash functions.

Symmetric Keys

Symmetric key cryptography (also called private key cryptography or secret key cryptography) is the classic use of keys that most people are familiar with: the same key is used to encrypt and decrypt the data. The classic, and most easily breakable, version of this is the Caesar cipher (named for Julius Caesar), in which each letter in a message is replaced by a letter that is a fixed number of positions away in the alphabet (for example, “a” is replaced by “c”, “b” is replaced by “d”, and so forth). In this case, the key used to encrypt and decrypt the message is simply the number of positions in the alphabet to shift the letters. Modern symmetric key algorithms are much more sophisticated and much harder to break. However, they share the property of using the same key for encryption and decryption.

There are many different algorithms used for symmetric key cryptography, offering anything from minimal to nearly unbreakable security. Some of these algorithms offer strong security, easy implementation in code, and rapid encryption and decryption. Such algorithms are very useful for such purposes as encrypting files stored on a computer to protect them in case an unauthorized individual uses the computer. They are somewhat less useful for sending messages from one computer to another, because both ends of the communication channel must possess the key and must keep it secure. Distribution and secure storage of such keys can be difficult and can open security vulnerabilities.

In 1968, the USS *Pueblo*, a U.S. Navy intelligence ship, was captured by the North Koreans. At the time, every Navy ship carried symmetric keys for a variety of code machines at a variety of security levels. Each key was changed daily. Because there was no way to know how many of these keys had not been destroyed by the Pueblo’s crew and therefore were in the possession of North Korea, the Navy had to assume that all keys

being carried by the Pueblo had been compromised. Every ship and shore station in the Pacific theater (that is, several thousand installations, including ships at sea) had to replace all of their keys by physically carrying code books and punched cards to each installation.

The Pueblo incident was an extreme case. However, it has something in common with the problem of providing secure communication for commerce over the Internet. In both cases, codes are used for sending secure messages, not between two locations, but between a server (the Internet server or the Navy's communications center) and a large number of communicants (individual web users or ships and shore stations). The more end users that are involved in the secure communications, the greater the problems of distribution and protection of the secret symmetric keys.

Although secure techniques for exchanging or creating symmetric keys can overcome this problem to some extent (see, for example, "Diffie-Hellman Key Exchange" (page 28)), a more practical solution for use in computer communications came about with the invention of practical algorithms for asymmetric key cryptography.

Asymmetric Keys

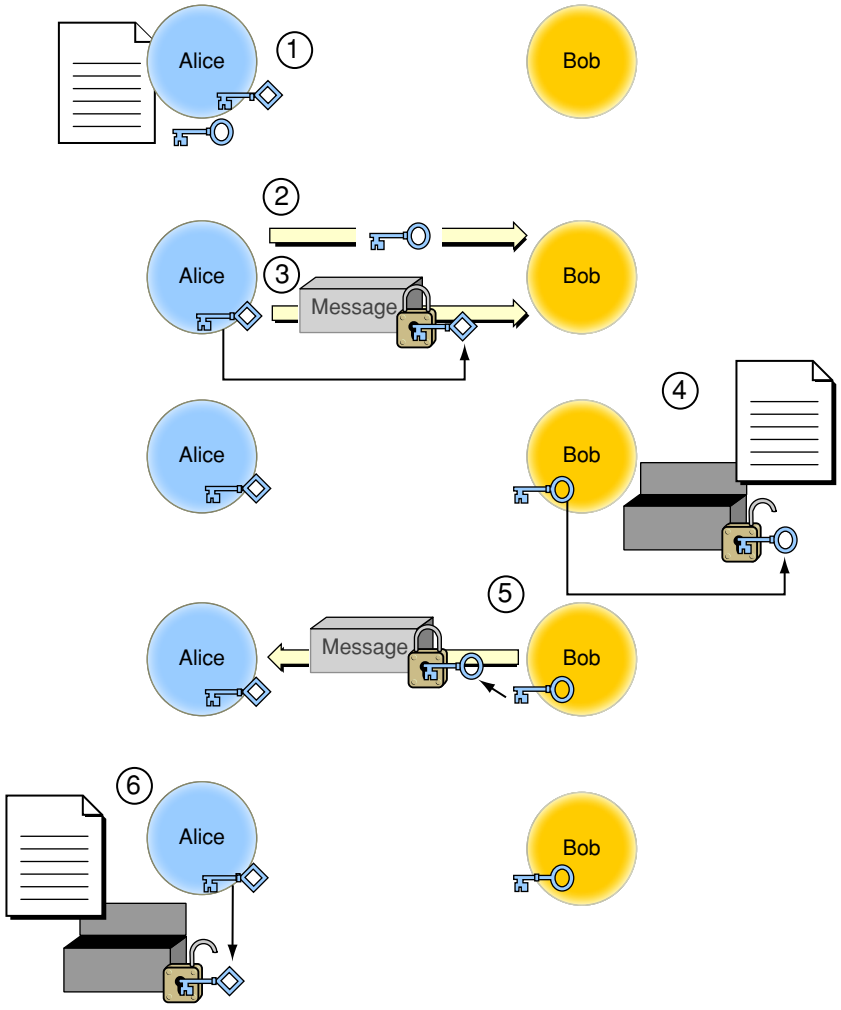
In **asymmetric key cryptography**, different keys are used for encrypting and decrypting a message. The asymmetric key algorithms that are most useful are those in which neither key can be deduced from the other. In that case, one key can be made public while the other is kept secure. There are some distinct advantages to this public-key–private-key arrangement, often referred to as **public key cryptography**: the necessity of distributing secret keys to large numbers of users is eliminated, and the algorithm can be used for authentication as well as for cryptography.

The first public key algorithm to become widely available was described by Ron Rivest, Adi Shamir, and Len Adleman in 1977, and is known as **RSA encryption** from their initials. Although other public key algorithms have been created since, RSA is still the most commonly used. The mathematics of the method are beyond the scope of this document, and are available on the Internet and in many books on cryptography. The algorithm is based on mathematical manipulation of two large prime numbers and their product. Its strength is believed to be related to the difficulty of factoring a very large number. With the current and foreseeable speed of modern digital computers, the selection of long-enough prime numbers in the generation of the RSA keys should make this algorithm secure indefinitely. However, this belief has not been proved mathematically, and either a fast factorization algorithm or an entirely different way of breaking RSA encryption might be possible. Also, if practical **quantum computers** are developed, factoring large numbers will no longer be an intractable problem.

Other public key algorithms, based on different mathematics of equivalent complexity to RSA, include ElGamal encryption and elliptic curve encryption. Their use is similar to RSA encryption (though the mathematics behind them differs), and they will not be discussed further in this document.

To see how public key algorithms address the problem of key distribution, assume that Alice wants to receive a secure communication from Bob. The procedure is illustrated in Figure 2-1.

Figure 2-1 Asymmetric key encryption



The secure message exchange illustrated in Figure 2-1 (page 27) has the following steps:

1. Alice uses one of the public key algorithms to generate a pair of encryption keys: a private key, which she keeps secret, and a public key. She also prepares a message to send to Bob.
2. Alice sends the public key to Bob, unencrypted. Because her private key cannot be deduced from the public key, doing so does not compromise her private key in any way.
3. Alice can now easily prove her identity to Bob (a process known as *authentication*). To do so, she encrypts her message (or any portion of the message) using her private key and sends it to Bob.

4. Bob decrypts the message with Alice's public key. This proves the message must have come from Alice, as only she has the private key used to encrypt it.

5. Bob encrypts his message using Alice's public key and sends it to Alice. The message is secure, because even if it is intercepted, no one but Alice has the private key needed to decrypt it.

6. Alice decrypts the message with her private key.

Since encryption and authentication are subjects of great interest in national security and protecting corporate secrets, some extremely smart people are engaged both in creating secure systems and in trying to break them. Therefore, it should come as no surprise that actual secure communication and authentication procedures are considerably more complex than the one just described. For example, the authentication method of encrypting the message with your private key can be got around by a **man-in-the-middle attack**, where someone with malicious intent (usually referred to as Eve in books on cryptography) intercepts Alice's original message and replaces it with their own, so that Bob is using not Alice's public key, but Eve's. Eve then intercepts each of Alice's messages, decrypts it with Alice's public key, alters it (if she wishes), and reencrypts it with her own private key. When Bob receives the message, he decrypts it with Eve's public key, thinking that the key came from Alice.

Although this is a subject much too broad and technical to be covered in detail in this document, see "Digital Certificates" (page 31), "Digital Signatures" (page 29), and "Authentication and Identification Methods" (page 35) for a few of the approaches used to address these security problems.

Diffie-Hellman Key Exchange

The **Diffie-Hellman key exchange** protocol is a way for two ends of a communication session to generate symmetric private keys through the exchange of public keys. The two sides agree beforehand on the exact algorithm to use and certain parameters, such as the size of the keys. Then each side selects a random number as a private key and uses that number to generate a public key, according to the algorithm. The security of this algorithm depends in part on it being extremely difficult to derive or guess the private key from this public key.

The two sides exchange public keys and then each generates a session key using their own private key and the other side's public key. The mathematics of the algorithm is such that, even though neither side knows the other side's private key, both sides' session keys are identical. A third party intercepting the public keys but lacking knowledge of either private key cannot generate a session key. Therefore, data encrypted with the session key is secure while in transit.

Although Diffie-Hellman key exchange provides strong protection against compromise of intercepted data, it provides no mechanism for ensuring that the entity on the other end of the connection is who you think it is. That is, this protocol is vulnerable to a man-in-the-middle attack. Therefore, it is sometimes used together with some other authentication method to ensure the integrity of the data.

Diffie-Hellman key exchange is supported by Apple Filing Protocol (AFP) version 3.1 and later and by Apple's Secure Transport API. Because RSA encryption tends to be slower than symmetric key methods, Diffie-Hellman (and other systems where public keys are used to generate symmetric private keys) can be useful when a lot of encrypted data must be exchanged.

Cryptographic Hash Functions

A **cryptographic hash function** takes any amount of data and applies an algorithm that transforms it into a fixed-size output value. For a cryptographic hash function to be useful, it has to be extremely difficult or impossible to reconstruct the original data from the hash value, and it must be extremely unlikely that the same output value could result from any other input data.

Sometimes it is more important to verify the integrity of data than to keep it secret. For example, if Alice sent a message to Bob instructing him to shred some records (legally, of course), it would be important to Bob to verify that the list of documents was accurate before proceeding with the shredding. Since the shredding is legal, however, there is no need to encrypt the message, a computationally expensive and time-consuming process. Instead, Alice could compute a hash of the message (called a **message digest**) and encrypt the digest with her private key. When Bob receives the message, he decrypts the message digest with Alice's public key (thus verifying that the message is from Alice), and computes his own message digest from the message text. If the two digests match, then Bob knows the message has not been corrupted or tampered with. For more information on ensuring data integrity, see "Digital Signatures" (page 29)

Commonly used hash functions include MD5, from RSA Data Security, which hashes any amount of input data into a 128-bit output value, and SHA-1, developed and published by the U.S. Government, which produces a 160-bit hash value from any data up to 264 bits in length.

Encrypting Messages

Either symmetric or asymmetric key methods can be used to encrypt messages. Whereas the iOS Certificate, Key, and Trust Services API includes functions to encrypt and decrypt data, Mac OS X does not provide a high-level encryption API. In Mac OS X, you can call CSSM Cryptographic Services Manager functions to encrypt data. See "CDSA" (page 12) and "CSSM Services" (page 16) for more information about CSSM. For examples of code using CSSM for common encryption tasks, see the *CryptoSample* sample code.

Apple's Mail application (and other email applications) can extract a public key from the signing certificate of any signed email and use it to encrypt messages sent to the owner of that key. See "Digital Signatures" (page 29) for more information about digital signatures and Help for the Mail application for details on sending encrypted email.

If you use the Secure Transport or CFNetwork APIs to set up a secure communication session, all data sent over that communication link is encrypted. See "Secure Communication" (page 62) for more information.

Digital Signatures

Digital signatures are a way to ensure the integrity of a message or other data using public key cryptography. Like traditional signatures written with ink on paper, they can be used to authenticate the identity of the signer of the data. However, digital signatures go beyond traditional signatures in that they can also ensure that the data itself has not been altered. This is like signing a check in such a way that if someone changes the amount of the sum written on the check, an "Invalid" stamp becomes visible on the face of the check.

To create a digital signature, the signer generates a message digest of the data and then uses a private key to encrypt the digest. The signature includes the encrypted digest and information about the signer's digital certificate. The certificate is used to verify the signature; it includes the public key needed to decrypt the

digest and the algorithm used to create the digest. To verify that the signed document has not been altered, the recipient uses the algorithm to create their own message digest and uses the public key to decrypt the digest in the signature. If the two digests are identical, then the message cannot have been altered and must have been sent by the owner of the public key.

To ensure that the person who provided the signature is not only the same person who provided the data but is also who they say they are, the certificate is also signed—in this case by the certification authority who issued the certificate. Digital certificates are described in "Digital Certificates" (page 31). Starting with Mac OS X v10.5, developers are encouraged to sign their applications. On execution, each application's signature is checked for validity. Digital signatures are required on all applications for iOS. See "Code Signing" (page 68) for more information on how code signing is used by Mac OS X and iOS.

Figure 2-2 illustrates the creation of a digital signature.

Figure 2-2 Creating a digital signature

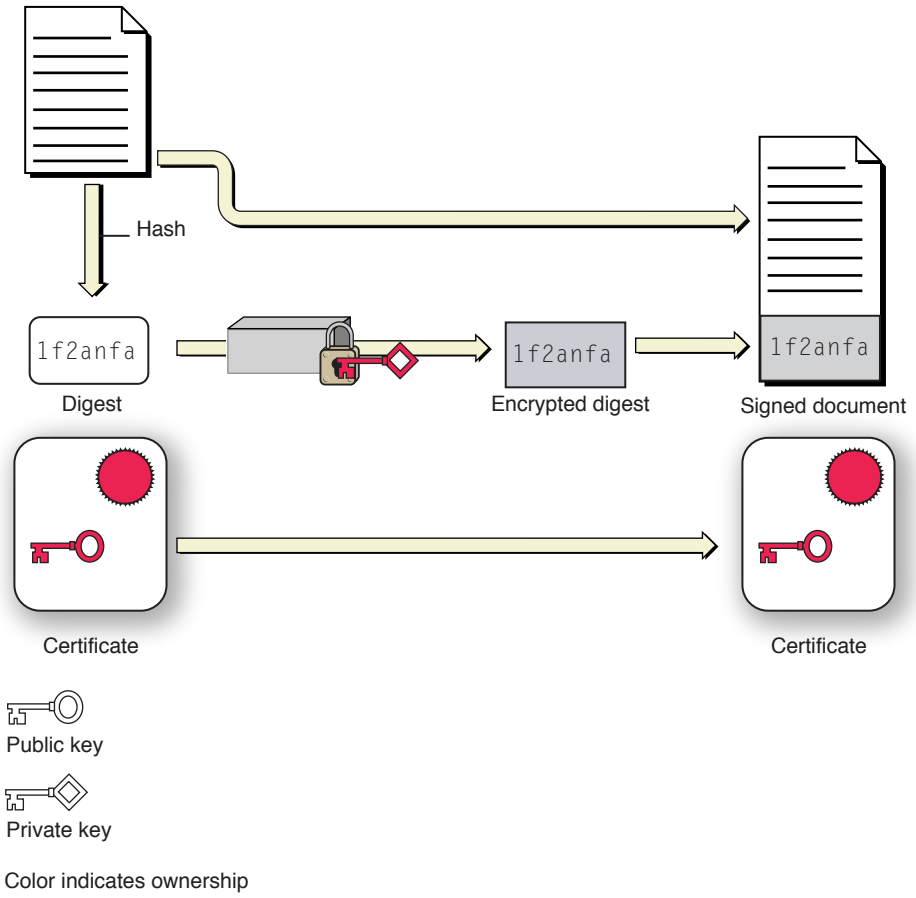
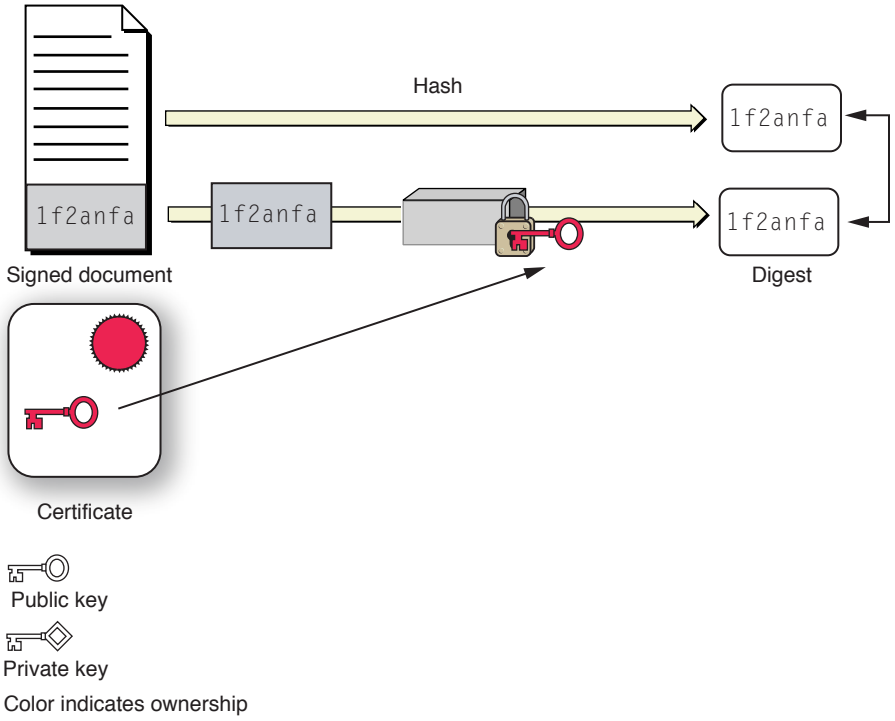


Figure 2-3 illustrates the verification of a digital signature. The recipient gets the signer's public key from the signer's certificate and uses that to decrypt the digest. Then, using the algorithm indicated in the certificate, the recipient creates a new digest of the data and compares the new digest to the decrypted copy of the one delivered in the signature. If they match, then the received data must be identical to the original data created by the signer.

Figure 2-3 Verifying a digital signature



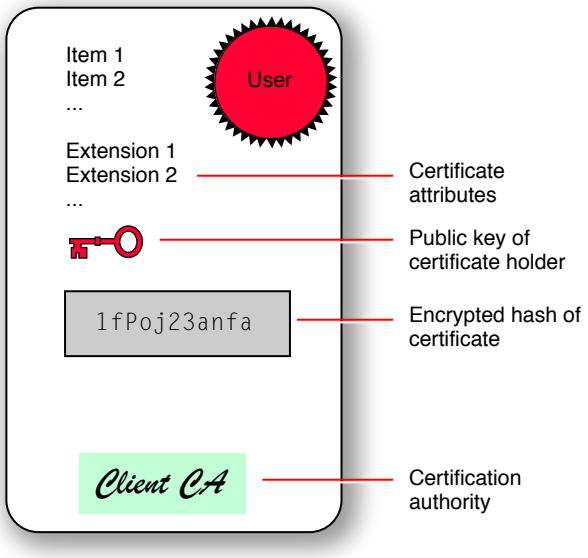
Digital Certificates

A **digital certificate** is a collection of data used to verify the identity of the holder or sender of the certificate. For example, an X.509 certificate contains such information as:

- Version
- Serial number
- Certificate issuer
- Certificate holder
- Validity period (the certificate is not valid before or after this period)
- Attributes, known as **certificate extensions**, that contain additional information such as allowable uses for this certificate
- Digital signature from the certification authority to ensure that the certificate has not been altered and to indicate the identity of the issuer
- Public key of the owner of the certificate
- Message digest algorithm used to create the signature

The careful reader will have noticed that a digital signature indicates the certificate of the signer, and a certificate contains a digital signature, which indicates another certificate. In general, 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**. Figure 2-4 illustrates the anatomy of a digital certificate.

Figure 2-4 Anatomy of a digital certificate



The root certificate is self-signed, meaning the signature of the root certificate was created by the root certification authority themselves. Figure 2-5 and Figure 2-6 illustrate how a chain of certificates is created and used. Figure 2-5 shows how the root certification authority creates its own certificate and then creates a certificate for a secondary certification authority.

Figure 2-5 Creating the certificates for the root CA and a secondary CA

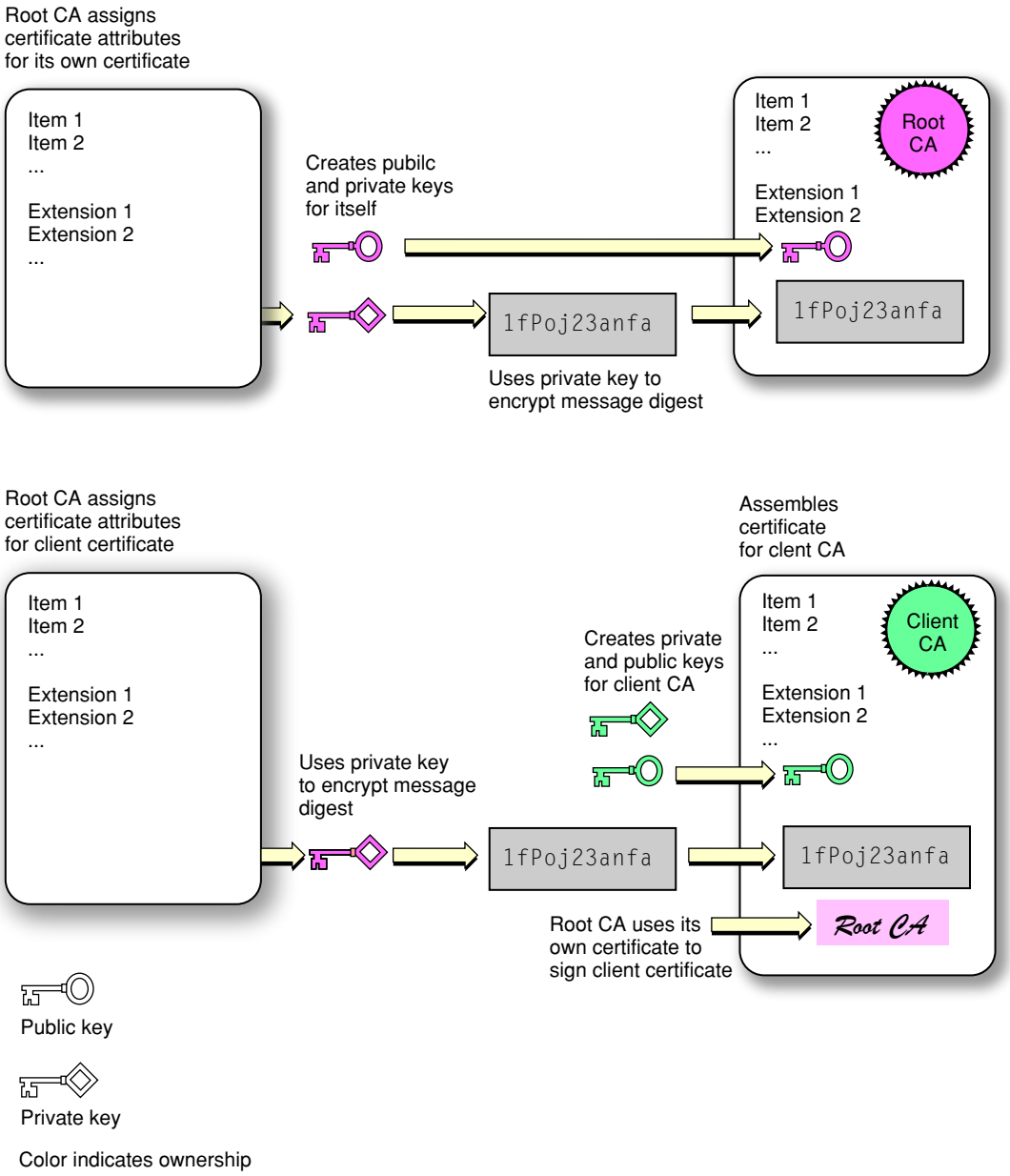
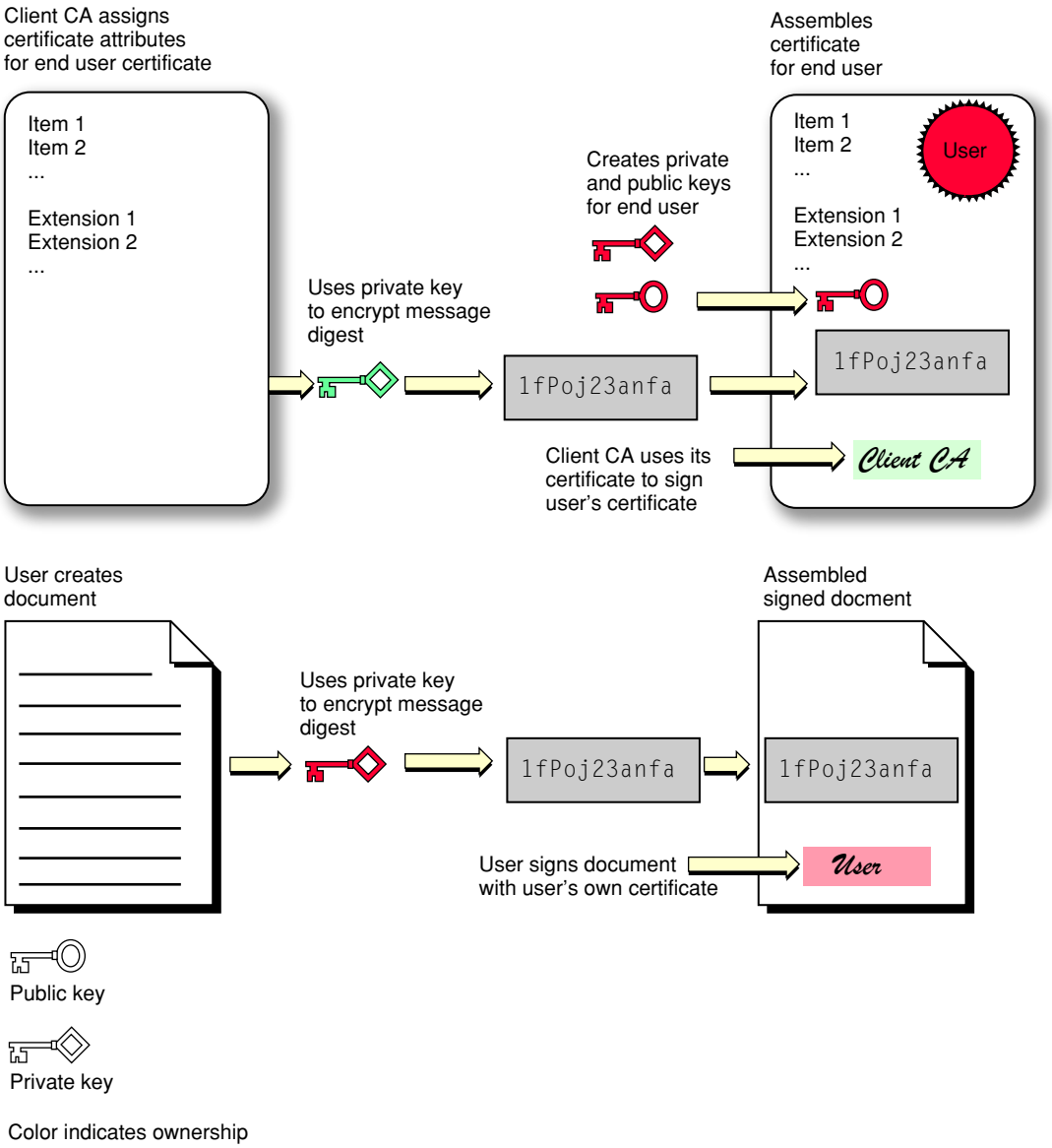


Figure 2-6 shows how the secondary certification authority creates a certificate for an end user and how the end user uses it to sign a document.

Figure 2-6 Creating the certificate for an end user and signing a document with it



In Figure 2-6, the creator of the document has signed the document. The signature indicates the certificate of the document's creator (labeled "User" in the figure). The document's creator signs the document with a private key, and the signing certificate contains the corresponding public key, which can be used to decrypt the message digest to verify the signature (see "Digital Signatures" (page 29). This certificate—together with the private and public keys—was provided by a certification authority (CA). In order to verify the validity of the user's certificate, the certificate is signed using the certificate of the CA. The certificate of the CA includes the public key needed to decrypt the message digest of the user's certificate. Continuing the certificate chain, the certificate of the CA is signed using the certificate of the authority who issued that certificate. The chain can go on through any number of intermediate certificates, but in Figure 2-5 (page 33) the issuer of the CA's certificate is the root certification authority. Note that the certificate of the root CA, unlike the others, is "self signed." That is, it does not refer to a further certification authority but is signed using the root CA's own private key.

When a CA creates a certificate, it uses its private key to encrypt the certificate's message digest. The signature of every certificate the CA issues refers to its own signing certificate. The CA's public key is in this certificate, and the application verifying the signature must extract this key to verify the certificate of the CA. So it continues, on down the certificate chain, to the certificate of the root CA. When a root CA issues a certificate, it, too, signs the certificate. However, this signing certificate was not issued by another CA; the chain stops here. Rather, the root CA issues its own signing certificate, as shown in Figure 2-5 (page 33).

The certificate of the root CA can be verified by creating a digest and comparing it with one widely available. Typically, the root certificate and root CA's public key are already stored in the application or on the computer that needs to verify the signature.

It's possible to end a certificate chain with a trusted certificate that is not a root certificate. For example, a certificate can be certified as trusted by the user, or can be cross certified—that is, signed with more than one certificate chain. The general term for a certificate trusted to certify other certificates—including root certificates and others—is **anchor certificate**. Because most anchor certificates are root certificates, the two terms are often used interchangeably.

The confidence you can have in a given certificate depends on the confidence you have in the anchor certificate; for example, the trust you have in the certificate authorities and in their procedures for ensuring that subsequent certificate recipients in the certificate chain are fully authenticated. For this reason, it is always a good idea to examine the certificate that comes with a digital signature, even when the signature appears to be valid. In Mac OS X and iOS, all certificates you receive are stored in your keychain. In Mac OS X, you can use the Keychain Access utility to view them.

Certain attributes of a digital certificate (known as **certificate extensions**) are said to establish a **level of trust** for a digital certificate. A **trust policy** is a set of rules that specify the appropriate uses for a certificate that has a specific level of trust. In other words, the level of trust for a certificate is used to answer the question "Should I trust this certificate for this action?"

For example, the AppleX509TP module ("AppleX509TP Module" (page 15)) enforces a trust policy referred to as the S/MIME policy, which specifies that in order to be trusted to verify a digitally signed email, a certificate must contain an email address that matches the address of the sender of the email.

Authentication and Identification Methods

This section describes some of the authentication and identification methods used in software in general and in Mac OS X in particular, but gives no implementation details or programming samples.

If you use Authorization Services, your application can take advantage of any authentication methods supported by Mac OS X, even if new methods are added after you write your code. See "Authorization Services" (page 66) for more information on Mac OS X Authorization Services.

Mac OS X

Mac OS X servers and clients can authenticate users by:

- Using the Kerberos Key Distribution Center (KDC), which is built into Mac OS X Server (see "Kerberos" (page 37))
- Using a password stored securely in the Open Directory Password Server database

- Using shadow hash authentication, in which cryptographic hashes for NT and LAN Manager authentication are stored in a local file that only the root user can access
- Using an encrypted password stored directly in the user's account
- Using a non-Apple LDAP (Lightweight Directory Access Protocol) server

Shared Secret

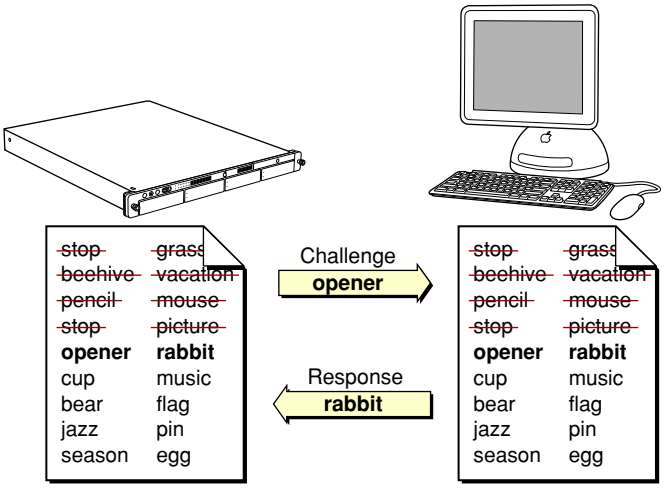
Many authentication methods are based on shared secrets, such as passwords. The security of **shared secret authentication** methods depends on the ability of both parties to keep the secret safe. If the secret is ever intercepted or is simple enough to be easily guessed, the method is not secure at all. The physicist Richard Feynman had a reputation as a safecracker when he worked at Los Alamos. However, in most instances he merely guessed the safe's combination by using such numbers as the safe owner's birthday or address. Similarly, if you select a password based on your middle name and then use the same password on all your accounts, none of the accounts will be secure.

On the other hand, carefully picked and guarded secrets can be extremely secure. Two examples of highly secure shared secret methods are one-time pads and time-based passwords.

One-Time Pads

One-time pad authentication requires that both parties have an identical list of pairs of numbers, words, or symbols. The most secure lists are randomly generated. When one party (Alice) requests an interchange with the other (Bob), Bob sends Alice a challenge in the form of one of the items selected from the list. Alice must respond with the corresponding paired item (Figure 2-7). Once a challenge has been used, it is crossed off the list and never used again.

Figure 2-7 Authentication using one-time pads



The one-time nature of one-time pad authentication makes it impossible for someone to guess the appropriate response to any one particular challenge by a brute force attack—that is, by responding to the challenge repeatedly with different answers until hitting the right answer. Similarly, it is impossible to guess what the next challenge should be. Assuming that the lists are truly randomly generated, the only way to break such a system is to know some portion of the contents of the list of pairs.

For this reason, once the one-time pad has been shared securely, it can be used over insecure communication channels. If someone snoops the communication, they can obtain that challenge-response pair. However, that information is of no use to them, since that particular challenge is never issued again.

The first problem with challenge-response pairs is that generating truly random lists is difficult with computers. Pseudorandom lists are much easier to obtain, and can (in principle at least) be duplicated, given the right algorithm and a good guess at the seed value. The second problem is that this method is useful only if every pair of correspondents has a unique set of lists. Otherwise, it is impossible to determine which holder of the list is authenticating. Finally, as with any shared secret method, the lists must be kept secret when they are shared and stored securely. If one of the lists is obtained by a third party, the method is compromised and the security breach might be undetectable.

Time-Based Authentication

Time-based authentication is a shared secret method in which the secret is changed periodically in a way known only to the two parties involved. In one variant, both parties begin with the same small amount of seed numbers (as few as two or three) and use a mathematical function that calculates a new number from them. Every time interval (for example, once a minute), a new number is calculated from the previous two or three numbers and the oldest number is discarded.

As long as both parties keep their clocks synchronized and start with the same seed numbers, they can calculate the current authentication number. In order to guard against a third party intercepting enough numbers to calculate the series or authenticating themselves before the number expires, the numbers should be transmitted over a secure communication channel. To further increase security (for example, if the number-generating ID card is lost or stolen), the generated number is combined with a password or PIN known only to the two parties.

As with other shared secret methods, time-based authentication depends on the physical security of the shared secret, and each individual wishing to use the system must have a unique generated number and PIN.

Kerberos

In Greek mythology, Kerberos was the three-headed dog that guarded the gates of Hades. In computer security, Kerberos is an industry-standard protocol created by the Massachusetts Institute of Technology (MIT) to provide authentication over a network. **Kerberos** is a symmetric-key, server-based protocol and is used widely in Macintosh, Windows, and UNIX networks. Kerberos has been integrated into Mac OS X since Mac OS X v10.1. Kerberos is highly secure, and unlike some other shared secret, private-key methods, it can be used for one-to-many and many-to-many communications as well as one-to-one. Kerberos achieves this ability by storing all users' passwords in a central location, the directory server. Mac OS X works with all common directory servers, including **LDAP** (Lightweight Directory Access Protocol) servers and Microsoft Windows Active Directory servers. Mac OS X Server v10.3 and later includes an open-source LDAP server. Mac OS X Server v10.2 and later can host Kerberos authentication services (called a Kerberos **Key Distribution Center** (**KDC**)). Furthermore, a Mac OS X Server v10.3 or later installation that is configured to include a shared LDAP server automatically includes a Kerberos KDC. Mac OS X Server v10.3 and later uses Kerberos v5. Starting with Mac OS X v10.5, Kerberos client and server implementations are both included in the operating system, so any user's computer can be configured as a KDC.

Although users' passwords cannot be intercepted during authentication (because they are never sent over the network), it is very important to keep the machine containing the directory server in a secure location. All passwords and private encryption keys are stored in the directory server and are therefore vulnerable to attack if a malicious person gains access to the server.

Starting with Mac OS X v10.5, a user with a .Mac account can use Kerberos over the internet to access and control a computer remotely, a service known as Back To My Mac. This service uses public key cryptography to authenticate the two computers, which then follow standard Kerberos protocols, with one computer acting as the KDC and the other as the Kerberos client. The protocol that defines the use of public key cryptography for initial authentication in Kerberos is known as **PKINIT**. You use the open-source Generic Security Service Application Program Interface (**GSS-API**) to adapt your application to use Kerberos.

Kerberos tickets are blocks of data used to identify a user who has been previously authenticated. Because Kerberos uses tickets (which are issued for a specific user, service, and period of time), it is possible to access additional kerberized services without requiring the user to reauthenticate (by reentering their password). This feature is called **single signon**. A **kerberized service** is one that has been configured to take Kerberos tickets.

For more information on Kerberos in Mac OS X, see http://developer.apple.com/opensource/kerberosintro.html. For general information on Kerberos, see http://web.mit.edu/kerberos/. For information on MIT's Kerberos for Macintosh, see http://web.mit.edu/macdev/Development/MITKerberos/MITKerberosLib/Common/Documentation/KerberosFramework.html. Kerberos version 5 is defined in RFC 4120; PKINIT is defined in RFC 4556; and GSS-API is defined in RFC 2078.

As of Mac OS X v10.5, these Mac OS X services support Kerberos authentication:

- Apple Mail
- Apple Filing Protocol (AFP server)
- Apple FTP server
- Telnet
- SSH
- NFS
- SMB
- CUPS
- VNC
- iCal
- Xgrid
- Macintosh Manager Client Logins

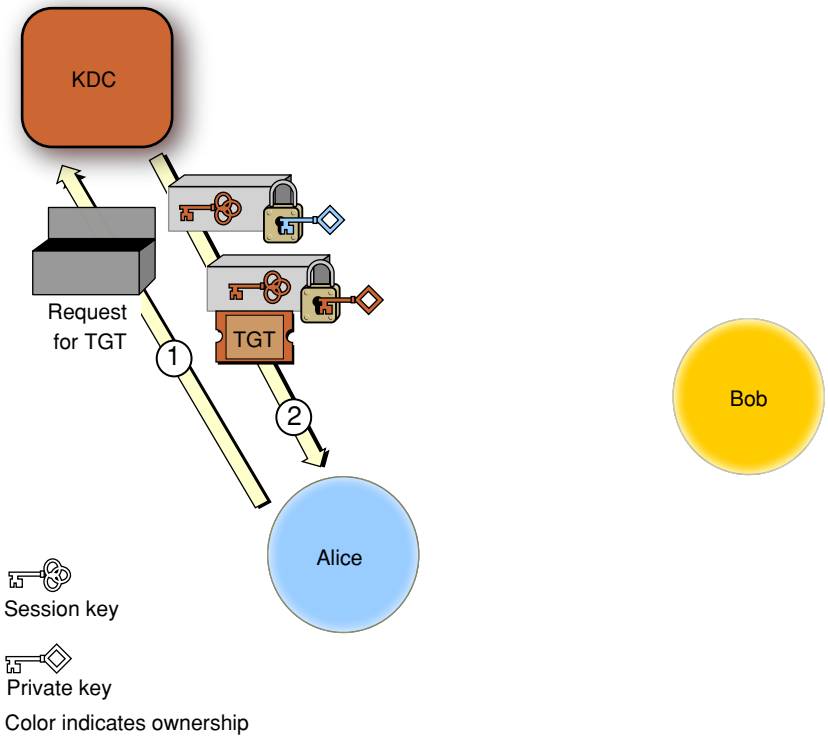
See *Mac OS X Server Open Directory Administration* (at http://www.apple.com/server/documentation/) to learn about the services that support Kerberos and to learn how to implement a Kerberos KDC on your Mac OS X server.

Kerberos Authentication Process

There are several phases to Kerberos authentication. In the first phase, the client obtains **credentials** (blocks of data that identify and authenticate an entity) to be used to request access to kerberized services. In the second phase, the client requests authentication for a specific service. In the final phase, the client presents those credentials to the service. Figure 2-8 and Figure 2-9 (page 40) illustrate this process.

Figure 2-8 shows the first phase, in which the client, labeled Alice in the figure, requests credentials from the Kerberos KDC.

Figure 2-8 Requesting credentials from the KDC



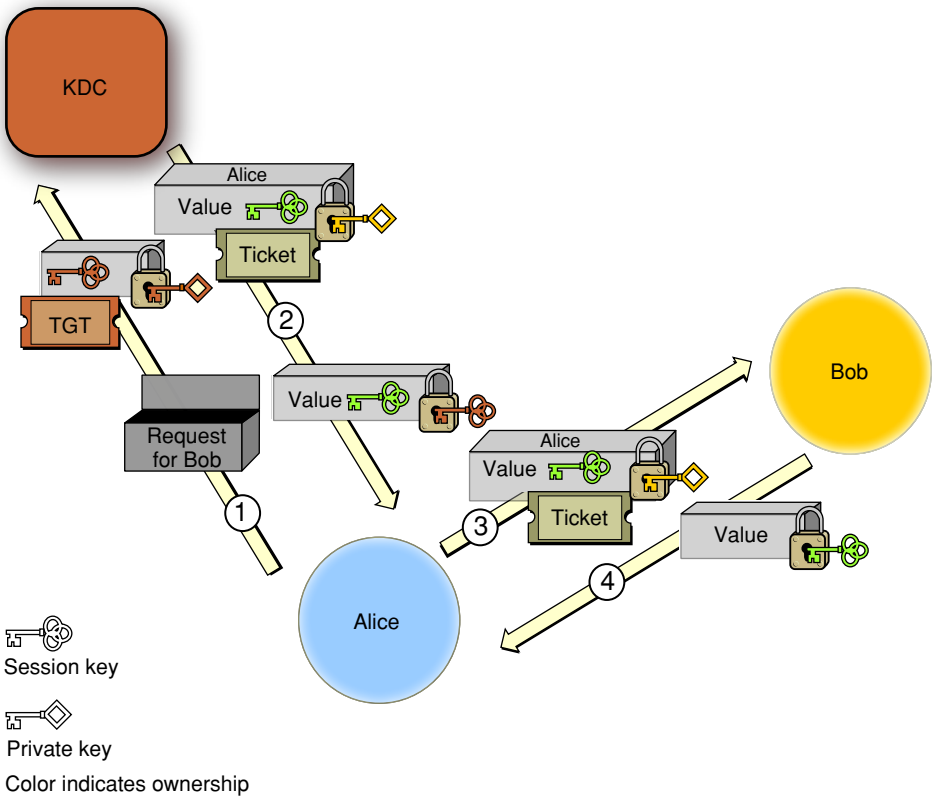
The steps are as follows:

1. Alice sends a request to the KDC for credentials. The KDC prompts Alice for a user name and password (or other authentication information), checks the authentication information against the data in the directory server, and (assuming the authentication is valid) gets Alice's private key from the directory server.

2. The KDC creates an encryption key (called a **session key**) for use by Alice the next time she wants to request service from a kerberized server and encrypts the key with Alice's private key. It also creates an identification credential called a **ticket-granting ticket** (**TGT**), which contains a copy of the session key encrypted with the KDC's private key (plus other information). The KDC sends both credentials to Alice. Alice decrypts the session key and stores it for later. She can't decrypt the TGT or modify it, but saves it for later use as well. Both the session key and the TGT include timestamps and expiration times to limit the chances of their being intercepted and used by unauthorized persons.

In the second phase, Alice uses the TGT to request identification credentials from the KDC in order to use a kerberized service, labeled Bob in the figure. Because Alice has a TGT, the KDC does not have to reauthenticate her, so Alice is not asked again for her password. In the third phase, Alice sends the credentials to Bob, and Bob sends authentication information to Alice. The second and third phases are illustrated in Figure 2-9.

Figure 2-9 Authenticating the client and server with a Kerberos ticket



The steps are as follows:

1. Alice sends to the KDC a request to open a session with Bob, together with the TGT that the KDC issued earlier. Because the TGT is encrypted with the KDC's private key, it cannot have been altered, and the KDC accepts it as proof that Alice has been authenticated.

2. The KDC decrypts the TGT and extracts the session key it issued earlier to Alice. (Recall that when the KDC sent the session key to Alice earlier, it was encrypted with Alice's private key, so only the KDC and Alice can know this session key.) The KDC then generates a random value, encrypts it with the session key, and sends it to Alice. It also creates a ticket for Alice to send to Bob. This ticket contains a new session key, the same random value that was sent to Alice, and an indication that the request for a session came from Alice. This key is encrypted with Bob's private key, so Alice (or an intruder) cannot read or modify it. The KDC sends the ticket to Alice.

3. Alice sends the ticket to Bob. Bob decrypts it with his private key. Because only the KDC and Bob know this key, Bob knows the ticket was issued by the KDC. Bob extracts the random value and the session key, and encrypts the random value with the session key.

4. Bob sends the encrypted value to Alice. Because Alice knows that only she and Bob have this session key, she knows that the credential must have come from Bob. She checks the value and compares it with the one she received earlier from the KDC. If they match, she knows the message was not interfered with, and she accepts that Bob has been authenticated by the KDC.

Note that this procedure does not involve sending either Alice's or Bob's private key over the network. Both Alice and Bob are authenticated to each other, so Bob knows that Alice is a valid user and Alice knows that Bob is the server with which she intended to do business. All credentials are further protected with timestamps and expiration times. Kerberos has other security features as well; for details, see the MIT Kerberos website at http://web.mit.edu/kerberos/.

Kerberos and Authorization

Kerberos is an authentication protocol, not an authorization protocol. That is, it verifies the identities of both the client and the server, but it does not include any information about whether the client has a right to use the services provided by the server. In terms of the preceding discussion, once Bob is satisfied that the request for services really came from Alice, it is up to Bob to determine whether to grant Alice access to those services. The ticket that Bob receives from Alice contains enough information about Alice to enable Bob to make that determination.

Starting with Kerberos version 5, Kerberos tickets provide a mechanism for the tamperproof transmission of authorization information. When the client requests a ticket, it includes information about itself in the request and can request that the KDC include additional authorization in the ticket. The KDC inserts this information into the authorization data field of the ticket and forwards it to the server. Kerberos does not define how this authorization information should be encoded; it provides only a secure mechanism for its transmission. It is up to the client and server to implement the authorization protocol.

Single Signon

Mac OS X uses Kerberos for single signon authentication, which relieves users from entering a name and password separately for every kerberized service. With single signon, after a user enters a name and password in the login window, the user does not have to enter a name and password for Apple file service, mail service, or other services that use Kerberos authentication. In other words, Kerberos authenticates the user once, and thereafter uses tickets to identify the user (see "Authentication, Identification, and Authorization" (page 24)).

To take advantage of the single signon feature, services must be configured for Kerberos authentication and users and services must use the same Kerberos KDC. For Mac OS X Server v10.3 and later, user accounts in an LDAP directory that have a password type of Open Directory use the server's built-in KDC. These user accounts are automatically configured for Kerberos and single signon. The server's kerberized services also use the server's built-in KDC and are automatically configured for single signon. See *Mac OS X Server Open Directory Administration* (at http://www.apple.com/server/documentation/) for details.

Large Networks

In "Kerberos Authentication Process" (page 38), the Kerberos Key Distribution Center (KDC) is treated as a single entity. However, a KDC consists of two separate software processes: the **ticket-granting server** and the **authentication server**. The authentication server verifies a user's identity by prompting the user for a name and password and asking the directory server for the user's password. The authentication server then looks up the user's private key, generates a session key, and creates the ticket-granting ticket (TGT), as shown in Figure 2-8 (page 39). Thereafter, the user sends the TGT to the ticket-granting server whenever the services of a kerberized server are required, and the ticket-granting server issues the ticket, as shown in Figure 2-9 (page 40).

Many networks are too large to efficiently store all the information about users and computers in a single directory server. Instead, a distributed model is used, where there are a number of directory servers, each serving a subset of the network. In Kerberos parlance, this subset is referred to as a **realm**. Each realm has its own ticket-granting server and authentication server. If a user needs a ticket for a service in a different

realm, the authentication server issues a TGT and the user sends the TGT to the authentication server, as before. The authentication server then issues a ticket, not for the desired service but for the remote ticket-granting server for the realm that the service is in. The user then sends the ticket to the remote ticket-granting server to get the ticket for the actual service.

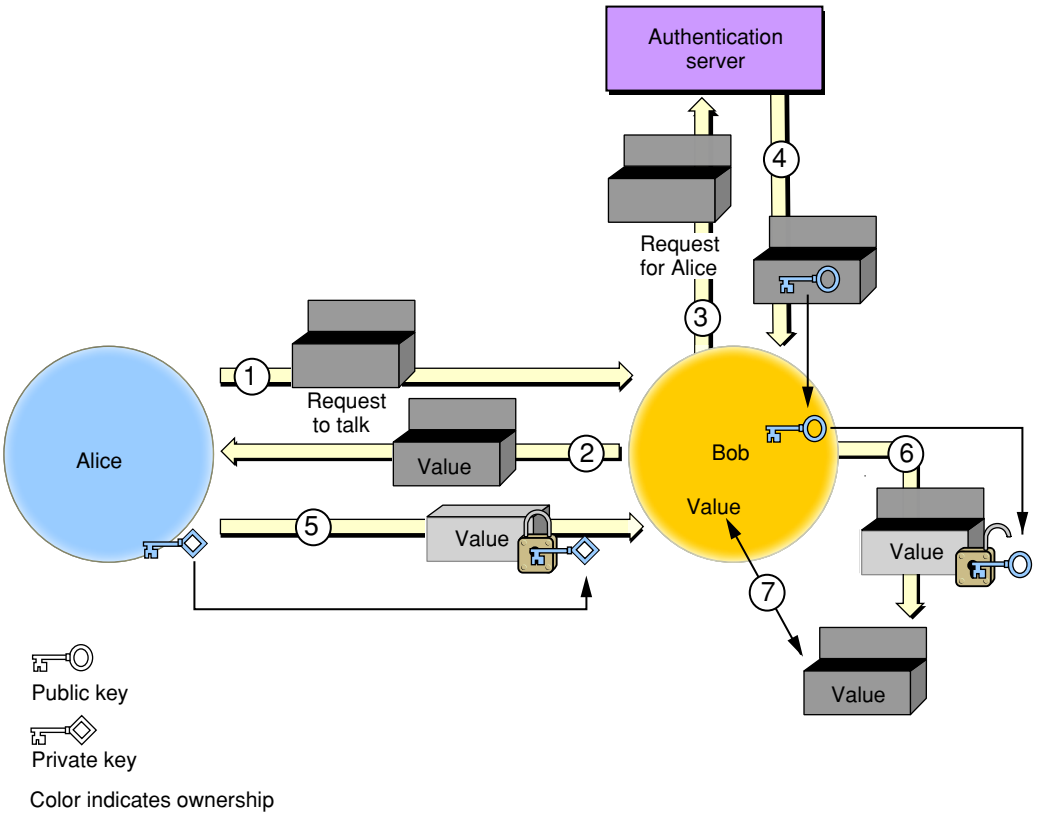
In fact, in a large network, the user might have to contact the remote ticket-granting server in a sequence of realms before finally getting the ticket for the desired service. When a ticket for the application service is finally issued, it contains an enumeration of all the realms consulted in the process of requesting the ticket. An application server that applies strict authorization rules is permitted to reject authentication that passes through realms that it does not trust.

Although limited cross-realm authentication was possible in Kerberos v4, the full implementation of this feature is new in Kerberos v5.

Public Keys

In **public key cryptography**, different keys are used for encryption and decryption. One, the **private key**, is kept secure. The **public key**, on the other hand, can be made publicly available without compromising the private key or the encryption method. In principle, public key authentication works in much the same way as private key authentication, with one major difference: because public keys do not have to be kept secret, there is no need to encrypt them or send them over secure channels. The public key can be provided by a server, in a certificate, or through some other method. Figure 2-10 illustrates public key authentication using an authentication server.

Figure 2-10 Public key authentication



The steps are as follows:

1. Alice sends Bob a request to talk.
2. Bob generates a random value and sends it to Alice as a challenge.
3. Bob requests Alice's public key from the authentication server.
4. The authentication server sends the unencrypted public key to Bob.
5. Alice encrypts the random value with her private key and sends it to Bob.
6. Bob decrypts the value with Alice's public key.
7. Bob compares the decrypted value with the original value to verify that they are identical. Alice has now authenticated herself to Bob.

Bob can authenticate himself to Alice in exactly the same way.

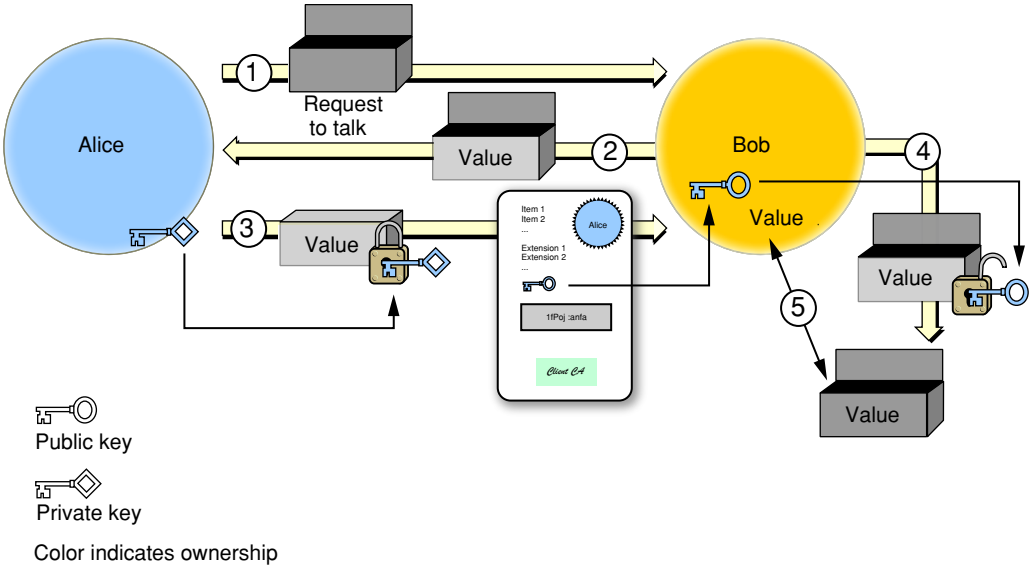
Notice that there is no need for the authentication server to store any sensitive material: the public keys do not have to be stored securely, and the authentication server does not need to hold passwords because it never has to verify one. However, it is necessary to ensure that no one alters the public keys stored in the authentication server. Otherwise, Eve, for example, could substitute her public key for Alice's and then could impersonate Alice. Actual implementations of server-based public key authentication systems, therefore, such as used by Novell Corporation's NDS (Novell Directory Services), include additional security features.

Note, however, that it is not necessary to have an authentication server in order to use public key authentication. Digital certificates can take the place of a central distributor of public keys.

Certificates

The problem of ensuring that a public key actually belongs to the entity you wish to authenticate can be addressed using digital certificates. Authentication using a digital certificate is illustrated in Figure 2-11.

Figure 2-11 Authentication with a digital certificate



The steps are as follows:

1. Alice sends Bob a request to talk.
2. Bob generates a random value and sends it to Alice as a challenge.
3. Alice encrypts the value with her private key and sends it to Bob. She also sends Bob her digital certificate containing her public key.
4. Bob verifies the digital certificate and uses the public key to decrypt the value.
5. Bob compares the decrypted value to the original value, verifying that it was truly Alice who sent him the certificate.

In practice, Alice could digitally sign her response to Bob rather than separately encrypting the challenge. Certificates are described in more detail in "Digital Certificates" (page 31), and digital signatures are discussed in "Digital Signatures" (page 29).

Permissions

An important aspect of security on a computer system is the granting or denying of **permissions** (sometimes called **access rights**). A permission is the ability to perform a specific operation such as to gain access to data or to execute code. Permissions can be granted at the level of directories (folders), subdirectories, files or applications, or specific data within files or functions within applications.

Permissions in Mac OS X are controlled at many levels, from the Mach and BSD components of the kernel, through higher levels of the operating system, and—for networked applications—through the networking protocols. For iOS, see "Sandboxing and the Mandatory Acccess Control Framework" (page 47).

Important: The Mach and BSD access permissions are enforced by the operating system and therefore affect every process running in Mac OS X. In contrast, application-defined security policies must be enforced by those applications. The Mac OS X security APIs discussed in this document are available for that purpose.

This section introduces the basic features of permissions at all of these levels in Mac OS X.

Mach Port Rights

At the deepest level of Mac OS X system architecture, the basis for the operating system's built-in security features is provided by the Mach and BSD components of the kernel. This section provides only a very brief and cursory introduction to Mach. For more information on Mach in Mac OS X and Mach programming, see *Kernel Programming Guide*.

Note: Apple does not support Mach functions for use by external developers and does not guarantee the binary compatibility from one operating system release to the next of code that calls Mach directly. Therefore, Apple recommends against the use of Mach functions by applications. You should use higher-level APIs whenever possible, and call BSD when a low-level interface is required.

Mach security is based on ports and port rights. A Mach **port** is an endpoint of a communication channel between a client who requests a service and a server that provides the service. Mach ports are unidirectional; a reply to a service request must use a second port.

A port has a set of associated port rights, which are owned by tasks. A **port right** specifies which task can use that port. Each port has one receive right, the owner of which can receive messages on that port. Each port also has one or more send rights; the owners of the send rights can send messages to the port. Rights can be transferred between tasks by being attached to a message.

A single task (or other Mach object, such as a thread or the computer itself) may have multiple ports that provide access to resources it supports. For example, a task might have a name port and a control port. Access to the control port allows the task to be manipulated. In contrast, access to the name port merely allows the client to obtain information about the task or perform other nonprivileged operations on it.

Each process has a port right namespace, which maps small integers known as **port right names** to their corresponding port rights. A port right name is meaningful only within that task's port right namespace. A task can transfer a port right to another task by sending it the corresponding port right name. However, unless it sends the name correctly, the receiving task won't be able use the right. The only way to transmit a port right between two tasks is by sending a Mach message and attaching the right name to that message using the correct syntax and message structure.

When you use Mach to create a task, Mach returns a port right name that references a send right for the port (the receive right for a task port is always owned by the kernel). You can send messages to this port to start and stop the task, kill the task, manipulate the task's address space, and so forth. Therefore, whoever owns a send right for a task's port effectively owns the task and can manipulate the task's state without regard to BSD security policies or any higher-level security policies. In other words, an expert in Mach programming with local administrator access to a Mac OS X machine can bypass BSD and higher-level security features. Therefore, it is very important to use strong administrator passwords, keep them secure, and control physical security for any computer containing sensitive information.

BSD

The BSD portion of the Mac OS X kernel enforces access to applications and files. The most familiar aspect of BSD security is the file system security policy, which controls access to files and folders (directories). BSD file permissions are described in "Mac OS X File System Security" (page 48) and in *File System Overview*.

In addition to the file system security policy, BSD defines two other security policies used in special cases: the owner or root security policy, and the root EUID security policy. Each of these policies is described briefly in the following subsections.

The BSD security model is based on matching up attributes of an object (such as a file) with attributes of the process attempting to gain access to that object. For example, suppose a file has an owning user ID of 1234 and the file permissions specify that the owning user has read and write access to that file. Suppose further that Alice has an effective user ID (EUID) of 1234. When Alice attempts to read the file, BSD matches her EUID with the file's owning UID and grants Alice access to read the file.

Each process has three user IDs: the real user ID (RUID), effective user ID (EUID), and saved user ID (SUID). The **RUID** is always inherited from the user or process that executes the process. The **EUID** is normally the same as the RUID, but it can differ in special circumstances as described in "Owner or Root Security Policy" (page 46). In most cases, it is the EUID that BSD checks to determine permissions. The **SUID** is used by BSD to enable a privileged process to switch in and out of privileged mode.

Each process also has real and saved group IDs (RGID and SGID) and up to 16 effective group IDs (EGIDs), which work in a way analogous to the process's user IDs.

For more details on these UIDs and GIDs, see *The Design and Implementation of the 4.4 BSD Operating System* by Marshall McKusick and others.)

Starting in Mac OS X v10.5, the kernel includes an implementation of the TrustedBSD Mandatory Access Control (MAC) framework. Mandatory access control—also known as "sandboxing"—is discussed in "Sandboxing and the Mandatory Acccess Control Framework" (page 47).

File System Security Policy

The BSD file system security policy is described in "Mac OS X File System Security" (page 48).

Owner or Root Security Policy

The owner-or-root security policy is used to control execution of a few specific operations. Under this policy, a specific operation on an object can be performed by any process whose EUID is the same as the object's owner or whose EUID is 0. The user with a UID of 0 is called the **root user** (also called the *superuser*) and a process running with an EUID of 0 is said to be running as `root`.

This policy is used in three primary places:

- Changing permissions on files with the `chmod` system call.

  Only the owner of the file or a process running as `root` can change a file's permissions.
- Deleting files within a directory whose sticky bit is set.

  Only the owner of the file, the owner of the enclosing directory, and `root` can delete the file.
- Sending signals to running processes (including killing the process).

A process can only send a signal to another process if their EUIDs match or if the sending process has an EUID of 0.

Root EUID Security Policy

Under the root EUID security policy, an operation can be performed only by a process with an EUID of 0. Such operations are sometimes referred to as **privileged operations**. Some of the common situations where the root EUID security policy applies are:

- Changing the owner of a file system object
- Binding TCP/IP sockets to low-numbered ports
- Making changes to the network configuration
- Certain I/O Kit operations
- Getting the Mach host privileged special port

Authorization Services and BSD Security Policies

Because a process running with an EUID of 0 has many special privileges, such a process can be a target of malicious hackers. To minimize such risks, you should factor your application into privileged and nonprivileged processes. See "Authorization Services" (page 66) for more information and for references that describe and illustrate this technique.

Processes can change their EUID and EGID by calling `setuid`, `setgid`, and related system calls. For example, a process can run as root temporarily and then switch to a less privileged EUID to minimize exposure to malicious attacks. This technique is complicated by the confusing semantics of the `setuid` call and by the fact that these calls operate somewhat differently on different implementations of UNIX (including different versions of Mac OS X). For a detailed discussion of the issues involved, see *Setuid Demystified* by Chen, Wagner, and Dean (Proceedings of the 11th USENIX Security Symposium, 2002), available at http://www.usenix.org/publications/library/proceedings/sec02/full_papers/chen/chen.pdf. For more information on the system calls, see the man pages for `setuid`, `setreuid`, and `setregid`. (The `setuid` man page includes information about `seteuid`, `setgid`, and `setegid` as well.)

Sandboxing and the Mandatory Acccess Control Framework

Sandboxing provides fine-grained control of the ability of processes to access system resources. For example, you can prevent a process from connecting to any network, from writing any files, or from writing any files outside of specific directories. This feature limits the amount of damage that can be done by a malicious hacker that gains control of an application. For example, if an attacker takes control of an application that is sandboxed so that it can write files only to the folder `/var/tmp`, it is not possible for the hacked program to overwrite system files.

New processes inherit the sandbox restrictions of their parent.

In Mac OS X, sandboxing is provided by the Mac OS X Mandatory Access Control (MAC) framework, which is an implementation of the TrustedBSD MAC framework, documented in http://www.trustedbsd.org/mac.html.

In iOS, each application is put in a sandbox that restricts the application to using only its own files and preferences, and limits the system resources to which the application has access. For example, an application can call the public networking APIs to communicate over a network, but has no direct access to the communications or networking hardware.

It's important to note that a sandbox does not protect an application from a direct attack. For example, if you accept input from the user, don't validate it, and there is an exploitable buffer overflow in your input-handling code, an attacker might be able to cause your program to crash or even take control of the program so that it executes the attacker's code. The sandbox limits the damage an attacker can cause, but cannot prevent attacks.

Mac OS X File System Security

Mac OS X provides file system security policies that limit access to files and directories (including special files and directories such as mount points for volumes, block and character special device files that represent hardware devices, symbolic links, named pipes, UNIX domain sockets, and so on).

Mac OS X provides two file system security schemes: UNIX (BSD) permissions and POSIX access control lists (ACLs). These schemes are described in the sections that follow.

UNIX Permissions

Each file system object has a set of UNIX permissions defined by three attributes:

- **UID**, short for user ID. Commonly referred to as the file's owner.
- **GID**, short for group ID.
- **Flags** that include permission bits and other related attribtues.

The flags for a file or directory are a 16-bit value that is often represented as a three-digit or four-digit octal value (with the top four or seven bits dropped):

- Bits 12-15: Flags indicating the type of the file. These bits are immutable and are omitted when representing permissions.
- Bits 9-11: Special permissions bits described in Table 2-2 (page 49). Usually 0; may be omitted if not set.
- Bits 6-8: Owner rights bits. These bits limit access by any process whose effective user ID (EUID) is equal to the UID of the file or directory).

  These bits have the highest precedence.
- Bits 3-5: Group rights bits. These bits limit access by any process with an effective group ID (EGID) matching the GID of the file or directory.

  These rights do *not* apply to any process whose EUID matches the UID of the file or directory. These bits have lower precedence than the Owner rights, but higher precedence than the Other rights.
- Bits 0-2: Other rights bits. These bits apply to any process that matches neither the UID nor GID of the file or directory.

The Owner, Group, and Other bit sets contain three bits: read, write, execute (`rwx` for short). The effect of these bits differs for files and directories, as shown in Table 2-1.

Table 2-1 File permission bits in BSD

Bit	File	Directory
read	Can open file for read	Can list directory contents
write	Can open file for write	Can modify directory contents (move, rename, or delete enclosed files or directories)
execute	Can treat file as a program to run	Can search through the directory (to access files or directories inside it)

In addition to the r, w, and x bits, each file system object also has three ancillary permission bits: setuid, setgid, and sticky.

Table 2-2 Special filesystem permissions bits

Bit	File	Directory
setuid	For a binary executable, when executed, the EUID of the resulting process is set to the file's UID instead of the EUID of the parent process. The RUID of the resulting process is still set to the EUID of the parent process as usual. This flag has no effect for interpreted scripts or non-executable files.	The UID of any file or directory created within the directory is set to the UID of the directory.
setgid	For a binary executable, when executed, the EGID of the resulting process is set to the file's GID instead of the EGID of the parent process. The RGID of the resulting process is still set to the EGID of the parent process as usual. This flag has no effect for interpreted scripts or non-executable files.	The GID of any file or directory created within the directory is set to the GID of the directory.
sticky	No effect in Mac OS X. For portability, you should avoid setting this bit, as it does have an effect in other UNIX variants.	Restricts deletion of enclosed files or directories to the following three users: ■ The file or directory's owner (EUID = file UID) ■ `root` (EUID = 0) ■ The owner of the sticky directory (EUID = directory UID)

For example, if the owner of a binary executable file is the root user and the setuid bit is set, the program always runs with an EUID of 0. Because such a program runs with root privileges when executed by someone other than `root`, it can create a security vulnerability. Therefore, it is important to restrict the creation and use of setuid and setgid programs.

A user can change the permissions only on files owned by that user. Therefore, only the root user can set the setuid bit on a program owned by `root`.

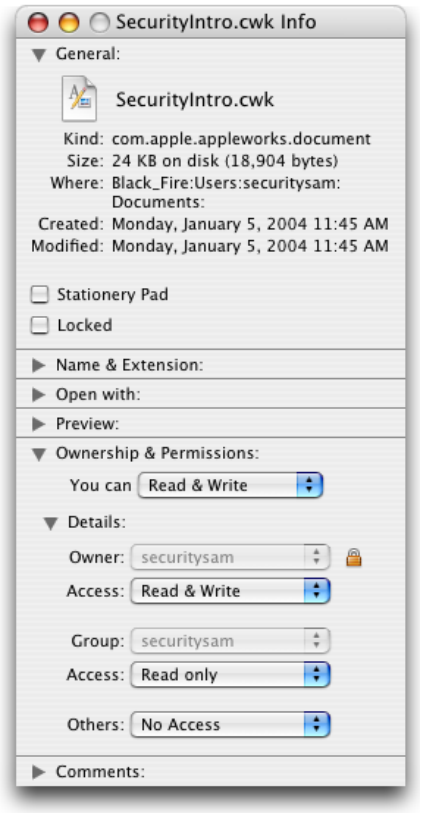
UNIX permissions are visible to users in Terminal and in the Finder. In Terminal, they look like this:

```
$ ls -ld filename dirname
drwxr-xr-x  2 username  groupname  68 Jun 16 13:40 dirname
-rw-r--r--  1 username  groupname   0 Jun 16 13:40 filename
```

The format of the permissions (at left) is described further in the "Shell Script Security" chapter of *Shell Scripting Primer*.

In Finder, UNIX permissions take the form of the Ownership and Permissions information in a file or folder's Info dialog (Figure 2-12).

Figure 2-12 Ownership and Permissions information



BSD File Flags

In addition to the standard UNIX file permissions, Mac OS X supports several BSD file flags provided by the `chflags` API and the related `chflags` command. These flags override the UNIX permissions.

C flag name Command-line name	Hex Value	Meaning
UF_NODUMP nodump	0x1	Do not back up the file when using the UNIX dump command. This flag is largely superfluous in Mac OS X. This flag can be changed by either the file's owner or the superuser (root).
UF_IMMUTABLE uchg, uchange, or uimmutable	0x2	File cannot be moved, renamed, or deleted (except by root in single-user mode). This flag can be changed by either the file's owner or the superuser (root).
UF_APPEND uappnd or uappend	0x4	Software can only append to the file, not modify the existing data. This flag can be changed by either the file's owner or the superuser (root).
UF_OPAQUE opaque	0x8	Directory is opaque with respect to union mounts. This means that if a directory in the underlying file system exists and has the same name, its contents are not visible. This flag can be changed by either the file's owner or the superuser (root).
----------	0x10	Reserved.
UF_COMPRESSED (no command equivalent)	0x20	File is compressed at the file system level. This flag can be changed by either the file's owner or the superuser (root).
----------	0x40–0x4000	Reserved.
UF_HIDDEN hidden	0x8000	Hint that the file should be hidden in the GUI. This flag can be changed by either the file's owner or the superuser (root).
SF_ARCHIVED arch or archived	0x10000	File has been archived. This flag can be changed only by the superuser (root).
SF_IMMUTABLE schg, schange, or simmutable	0x20000	File cannot be moved, renamed, or deleted (except by root in single-user mode). This flag can be changed only by the superuser (root).
SF_APPEND sappnd or sappend	0x40000	Software can only append to the file, not modify the existing data. This flag can be changed only by the superuser (root).

Note: To disable a flag with `chflags`, add `no` before the flag name or drop the leading `no`, as appropriate.

POSIX ACLs

Starting with Mac OS X v10.4, the Mach and BSD permissions policies are supplemented by support in the kernel for **ACLs** (access control lists), which are data structures that provide much more detailed control over permissions than does BSD. For example, ACLs allow the system administrator to specify that a specific user can delete a file but cannot write to it. ACLs also provide compatibility with Active Directory and with the SMB/CIFS networks used by the Windows operating system. For more information on ACL support in Mac OS X for different network file systems, see "Network File Systems" (page 59).

An ACL consists of an ordered list of **ACEs** (access control entries), each of which associates a user or group with a set of permissions and specifies whether each permission is allowed or denied. ACEs also include attributes related to inheritance (see "Inheritance of Permissions" (page 54)).

Note: File system ACLs are not related to the ACLs used by keychains, as described in *Keychain Services Programming Guide*.

File System Access Control Policy

You can use file system ACLs to implement more detailed and complex access control policies than are possible using only BSD permissions. They do so by using many more permission bits than the three used by BSD and by implementing both allow and deny associations for each permission for each user or group. Table 2-3 shows the permission bits used by ACLs. Compare these to the BSD permission bits shown in Table 2-1 (page 49).

Table 2-3 File permission bits using ACLs

Bit	File	Directory
read	Open file for read	List directory contents
write	Open file for write	Add a file entry to the directory
execute	Execute file	Search through the directory (to access files or directories within it)
delete	Delete file	Delete directory
append	Append to file	Add subdirectory to directory
delete child	—	Remove a file or subdirectory entry from the directory
read attributes	Read basic attributes	Read basic attributes
write attributes	Write basic attributes	Write basic attributes
read extended	Read extended (named) attributes	Read extended (named) attributes
write extended	Write extended (named) attributes	Write extended (named) attributes

Bit	File	Directory
read permissions	Read file permissions (ACL)	Read directory permissions (ACL)
write permissions	Write file permissions (ACL)	Write directory permissions (ACL)
take ownership	Take ownership	Take ownership

Notice that the right to change permissions is itself controlled by a permission.

ACLs and User IDs

One of the main reasons for implementing ACLs in Mac OS X is to support network file systems such as SMB/CIFS (see "SMB/CIFS" (page 60)). In order to be able to identify users and groups throughout the network, each file or directory must have universally unique identifiers (UUIDs) in addition to the locally-unique UID and GID used by BSD. Each file or directory that has associated ACLs, therefore, has four associated identities, two to support BSD and two to support ACLs:

- user ID (UID)
- group ID (GID)
- owner UUID
- group UUID

Unlike BSD, which specifies three permissions for each file (one for the file's owner, one for members of the file's group, and one for everyone else), an ACL can specify different permissions for each ACE. Another contrast between ACLs and BSD is that, whereas in BSD the file owner must be an individual, in the ACL permission scheme the file owner can be either a user or a group. If a file is owned by a group, its GID (used by BSD) and group UUID are always coherent (that is, there is always a simple, 1:1 mapping between them). However, because BSD does not support the concept of a group as owner of a file, in this case the system assigns a special UID that identifies the file as owned by "not a user" and the owner UUID represents a group. If the file is owned by a single individual, its UID and owner UUID are coherent.

The owner of a file using an ACL has certain unrevokable permissions (read and write permissions) regardless of the contents of the ACL. If the file is owned by an individual, the group UUID associates a group with a file system object and affects the inheritance of certain ACEs (see "Inheritance of Permissions" (page 54)) but does not confer any special permissions on the group.

Evaluating Permissions

Each ACE in an ACL either allows or a denies some set of permissions. It is very important to understand that a deny ACE is not the same as the absence of an allow ACE. Rather, the system evaluates the ACEs in sequence until either all requested permissions are allowed or any requested permission is denied. A request for authorization includes a credential (which identifies the requesting entity) and the permissions required for the operation. Mac OS X v10.4 and later evaluates permissions using the following algorithm (also, see "Inheritance of Permissions" (page 54) for a discussion of inherited permissions):

1. If the requested permissions would change the object, and the file system is read-only or the object is marked as immutable, the operation is denied.

2. If the entity making the request is the root user, the operation is allowed.

3. If the entity making the request is the object's owner, the requestor is given Read Permissions and Write Permissions access. If that is sufficient to satisfy the request, the operation is allowed.

4. If the object has an ACL, the ACEs in the ACL are scanned in order. (Those with deny associations are usually placed before those with allow associations.) Each ACE is evaluated according to the following criteria until either a required permission has been denied, all required permissions have been allowed, or the end of the ACL is reached:

    a. The ACE is checked for applicability. The ACE is not considered applicable if it does not refer to any of the requested permissions. In addition, the requesting entity must be the same as the entity named in the ACE, or the requestor must be a member of a group named in the ACE. (Groups may be nested and an external directory service may be used to resolve group membership.) Non-applicable ACEs are ignored.

    b. If the ACE denies any of the requested permissions, then the request is denied. (Note that Read Permissions and Write Permissions are granted to the object's owner, regardless of whether allowed or denied by ACEs.)

    c. If the ACE allows any of the requested permissions, the system adds this permission to the list of granted permissions. If the granted permissions include all the requested permissions, the request is allowed and the process stops. If the list is not complete, the system goes on to check the next ACE.

5. If the end of the ACL is reached without finding all of the required permissions, and if the object also has BSD permissions, then the system checks the unsatisfied permissions against the BSD permissions. If these are sufficient to grant all required permissions, the request is allowed. If the permission requested has no BSD equivalent (such as "take ownership"), then it is considered still outstanding and the request is denied.

6. If the file system object has no ACL, then permissions are evaluated according to the BSD security policies, as described in "BSD" (page 46).

The credential of the requesting entity is equivalent to the effective UID (that is, the EUID) of the program attempting to open or execute a file. The EUID is normally the same as the UID of the user or process that executes the process., but it can differ in special circumstances (involving the setuid bit) as described in "Owner or Root Security Policy" (page 46).

Inheritance of Permissions

BSD permissions are assigned only on a per-file basis, so that the permissions assigned to a directory do not affect the permissions of a new file or subdirectory created in that directory. Although you can apply the permissions of a directory to enclosed items, doing so is a one-time operation. Any newly created files or subdirectories are not affected—they are created with default permissions.

With ACLs, by contrast, newly created files and subdirectories can inherit permissions from their enclosing directory. Each ACE on a directory can contain any combination of the following inheritance flags:

- Inherited (this ACE was inherited).
- File Inherit (this ACE should be inherited by files created within this directory.
- Directory Inherit (this ACE should be inherited by directories created within this directory).
- Inherit Only (this ACE should not be checked during authorization).

- No Propagate Inherit (this ACE should be inherited only by direct children; that is, the ACE should lose any Directory Inherit or File Inherit bit when inherited).

When it creates a new file, the kernel goes through the entire access control list of the parent directory and copies to the file's ACL any ACEs that are marked for file inheritance. Similarly, when it creates a new subdirectory, the kernel copies to the subdirectory's ACL any ACEs that are marked for directory inheritance.

If a file is copied and pasted into a directory, the kernel replicates the contents of the source file into a new file at the destination. Because it is creating a new file, the system checks the ACL of the parent directory and adds any inherited ACEs to whatever ACEs were in the original file. If a file is moved into a directory, on the other hand, the original file is not replicated and no ACEs are inherited. In this case, the parent directory's ACEs are added to the moved file only if the administrator specifically propagates ACEs from the parent directory through contained files and subdirectories. Similarly, once a file has been created, changing the ACL of the parent directory does not affect the ACL of contained files and subdirectories unless the administrator specifically propagates the change.

In BSD, applying a directory's permissions to enclosed files and subdirectories completely replaces the permissions of the enclosed objects. With ACLs, in contrast, inherited ACEs are added to other ACEs already on the file or directory.

The order in which ACEs are placed in an ACL—and therefore the order in which they are evaluated to determine permissions—is as follows:

1. Explicitly specified deny associations
2. Explicitly specified allow associations
3. Inherited associations, in the same order in which they appeared in the parent

Therefore, any explicitly specified ACEs take precedence over all inherited ACEs. For more information on how ACEs are evaluated, see "Evaluating Permissions" (page 53).

Because ACEs can be inherited, administrators can control the fine-grained permissions of files created in a directory by assigning inheritable ACEs to the directory. Doing so saves the work of assigning ACEs to each file individually. In addition, because ACEs can apply to groups of users, administrators can assign permissions to groups rather than having to specify permissions for each individual. Applying access security to directories and groups rather than to files and individuals saves administrator time and gives better file system performance in many circumstances.

For application programmers, it is important to note that the automatic inheritance of permissions from directories means that it is not necessary for an application to create an ACL for each new file it creates or to maintain inherited ACEs when a file is saved, because the kernel creates the ACL for the file using inherited ACEs. (Note that assignment and inheritance of BSD permissions are not affected by ACLs. If ACLs are not supported, the BSD permissions are used. For more information on the way permissions are evaluated when both ACLs and BSD permissions are set, see "Evaluating Permissions" (page 53).)

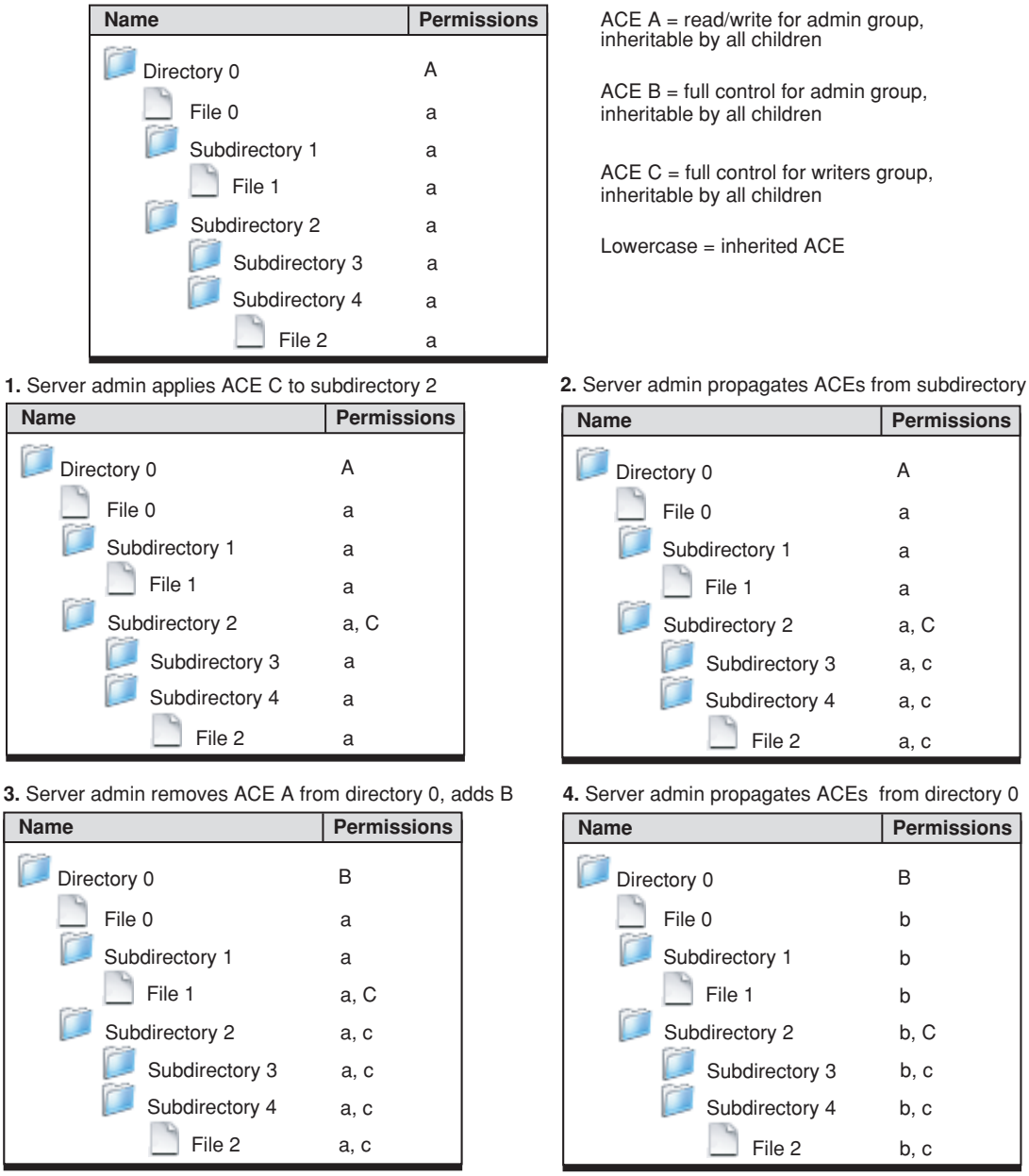
In Mac OS X Server v10.4, the server administrator can perform the following operations:

- Copy permissions from a parent directory to all files and directories below it in the hierarchy. This makes permissions uniform in the directory tree and should be used only for BSD permissions.

- Propagate permissions from a parent directory to all files and directories below it in the hierarchy. In this case, explicitly specified ACEs are unchanged and ACEs inherited from them are unchanged. Files and subdirectories inherit ACEs as if they had been newly created in place under the directories that have explicitly specified ACEs, as illustrated in Figure 2-13.
- Apply inheritance from a parent directory to a specific directory or file.
- Make inherited ACEs in directories explicit.
- Remove all ACEs from directories and files.
- Enable or disable ACLs on a volume.

The server GUI cannot directly manipulate ACEs of files. There is no GUI in the Finder to set or change ACEs. ACEs can be read and set both on the server and client using the command-line tools `ls` and `chmod`.

Figure 2-13 Propagating permissions



Mac OS X

This section discusses how Mac OS X uses BSD permissions.

The Root User

The **root user** owns many of the primary system processes and has unlimited access to the file system objects on the devices attached to the computer. For example, the root user can:

- Read, write, and execute any file
- Copy, move, and rename any file or folder
- Transfer ownership and reset permissions for any file

A major difference between standard BSD permission semantics and the Mac OS X implementation is that in Mac OS X the root user is disabled after system installation. In most cases, it is not necessary for an administrator to run as `root` (see "The Admin Group" (page 58)). You may also assume root power by using the `sudo` utility. Although the `sudo` utility does not require you to enable the root user, you can use it only from the Terminal application; that is, you must have physical access to the machine to use it. See the `sudo` man page for more information on its use.

The root user should not be enabled on user systems. If your application needs to perform operations as the root user, you must use Authorization Services. For more information, see *Authorization Services C Reference* and *Authorization Services Programming Guide* in Security Documentation.

Note: In almost all cases you can run as a member of the `admin` group or use `sudo` rather than enabling the root user. If you absolutely must enable the root user, run the NetInfo Manager utility and authenticate yourself as the local administrator. Then choose Enable Root User from the Security menu. This menu item is enabled only if you are a member of the local `admin` group—a group with special administrative privileges—and you have been previously authenticated in the local domain. Once you've enabled the root user, the password is blank, so you should give the root user a password by selecting Change Root Password from the Security menu. After you've completed the task requiring root access, you should relinquish root user privileges by choosing Disable Root User from the Security menu.

Whereas most user permissions apply across networks, setuid and setgid are often ignored on network volumes, as is the concept of a root user.

For example, when accessing remote volumes over NFS, by default, the root user is mapped to **nobody**—a special user with very little access. This prevents the root user on one computer from becoming the root user on another computer.

The Wheel Group

There is a special group in BSD called the **wheel group**. Membership in the wheel group confers on users the ability to become the root user by using the `su` utility on the command line. Users who are not in the wheel group can't become the root user, even if they have the correct password.

In Mac OS X v 10.3 and later, the wheel group is not used. Its functions have been assumed by the `admin` group.

The Admin Group

Mac OS X provides the `admin` group in place of the root user. A member of the **admin group** (referred to as an **administrator**) can perform almost all functions the root user can, and can do them using the Finder—that is, without resorting to the command line. The only thing the administrator is prevented from doing is directly adding, modifying, or deleting files in the system domain. An administrator can use special applications such as Installer or Software Update for this purpose, however.

The user who installs Mac OS X on a system becomes automatically the first administrator for the system. Thereafter, this user (or any other administrator) can use Accounts preferences to create accounts on the local system for new users and can grant administrative privileges to any user on the system.

Network File Systems

This section discusses the use of permissions by network file server protocols. Mac OS X supports four network file server protocols:

- **AFP**—Apple Filing Protocol, the principal file-sharing protocol in Mac OS 9 systems, used by AppleShare servers and clients.
- **NFS**—Network File System, the main file-sharing protocol used by UNIX systems.
- **SMB/CIFS**—Server Message Block/Common Internet File System, a file-sharing protocol used on Windows and UNIX systems
- **WebDAV**—Web-based Distributed Authoring and Versioning, an extension of HTTP that allows collaborative file management on the web.

AFP

If the AppleShare client and server both support AFP 3.0, the actual BSD permissions are transported over the connection. If the file or directory on the AFP server has an ACL, the ACL is transported over the connection and the effective permissions are displayed by the Finder. However, enforcement of permissions is done only on the server, not on the client. See "Mac OS X File System Security" (page 48) for more information on the Mac OS X implementation of ACLs.

If the connection is using AFP 2.x, you should be aware of the differences in how permissions work:

- BSD supports permissions on files, whereas AFP 2.x does not.
- BSD implements a "best match" permissions policy: if you're the owner, you get the owner permissions; if you're not the owner but you're in the file's group, you get the group permissions; otherwise, you get the other permissions. AFP implements a cumulative permissions policy: your permissions are the union of the permissions you derive from the owner, group, and other permissions. For example, if a folder is writable by the group but not by the owner, AFP permissions let the owner modify the folder but BSD permissions do not.
- BSD interprets the rwx bits for folders as shown in Table 2-1 (page 49). AFP permissions define them as "See Files", "See Folders", and "Make Changes". When dealing with an AppleShare 2.x server, the Mac OS X AppleShare client maps between these privilege models. A similar mapping applies when you connect to a Mac OS X server using an AppleShare 2.x client.
- ACLs are not supported by AFP 2.x.

AFP excludes a process having an EUID of 0 (that is, one running as `root`) from accessing any data over the network.

NFS

In general, NFS is not a secure protocol, because most NFS servers trust their clients. That is, if a client says that this file operation is done on behalf of user Bob, the server does the operation on behalf of user Bob. However, if you have root access on the client, you can pretend to be user Bob and access any of Bob's files on the NFS server. To maintain some security, most NFS servers map the root user to a special user, `nobody`, which owns no files or directories. For this reason, if your EUID is 0 you can, in general, access only those files on an NFS server that allow access to "other".

SMB/CIFS

SMB is a networking protocol for file sharing commonly used on Windows networks. CIFS is often used as a synonym for SMB. **Samba** is software that implements an SMB/CIFS server on UNIX. Therefore, this file sharing protocol is variously referred to as SMB, CIFS, SMB/CIFS, Samba, and Windows file sharing.

Mac OS X v10.4 and later implements SMB/CIFS-compatible access control lists (ACLs). Although individual users cannot set or alter ACLs, server administrators can do so. (Administrators can use the SMB server command line to manipulate ACLs, but only if both the client and server are bound to the same Active Directory domain.) However, enforcement of permissions is done only on the server, not on the client. See "Mac OS X File System Security" (page 48) for more information on the Mac OS X implementation of ACLs.

For Mac OS X v10.3 and earlier, all of the SMB access controls in Mac OS X are implemented on the server, not the client. Consequently, when a Mac OS X user mounts an SMB file server, the volume, directory, or file mounted appears in the Finder to allow read, write, and execute access and to be owned by the user. However, when the user attempts to open a folder or file, the server evaluates the user's access permissions and either allows access or prompts the user for a new user name and password before granting access.

For more information on SMB/CIFS permissions and to learn how to modify their behavior, see the man page for SMB (`man 5 smb.conf`).

WebDAV

The **WebDAV** protocol is an extension to the HTTP protocol that allows users to write and edit web content remotely; that is, over a network connection. The Mac OS X WebDAV file system uses WebDAV and HTTP requests to access resources on a WebDAV-enabled HTTP server as files and directories.

The WebDAV protocol does not support users and groups. Furthermore, a WebDAV client cannot determine access permissions for files and directories on a WebDAV server before attempting to access them. Therefore, the WebDAV file system in Mac OS X sets the user and group IDs to `unknown` for all files and directories and the permissions default to `read`, `write`, and `execute` for everyone: user, group, and other.

When the WebDAV file system sends a request to a WebDAV-enabled HTTP server, the server determines whether authorization is required. If no authorization is required, the server accepts the request. If authorization is required, the server checks for authentication credentials (such as a user name and password) and, if they are present and correct, the server authorizes the client and allows access. If authorization is required and no credentials were sent or the credentials are not correct, the server rejects the request with a challenge for authentication. If the user cannot supply the correct credentials, the WebDAV file system refuses access.

For more information on the protocols used by the WebDAV file system, see the following documents:

- *Hypertext Transfer Protocol–HTTP/1.1* http://www.ietf.org/rfc/rfc2616.txt
- *HTTP Authentication: Basic and Digest Access Authentication* http://www.ietf.org/rfc/rfc2617.txt

- *HTTP Extensions for Distributed Authoring–WEBDAV*http://www.ietf.org/rfc/rfc2518.txt

Authorization

Mac OS X uses authorization to control access to files and programs. The iOS uses sandboxing for this purpose (see "Sandboxing and the Mandatory Acccess Control Framework" (page 47)). Before a user or service on Mac OS X can execute a program or gain access to data, it must be authorized to do so. Authorization is normally a three-step process:

1. Authenticate a user or service (see "Authentication and Identification Methods" (page 35)).
2. Determine the user's or service's permissions (see "Permissions" (page 44)).
3. Decide whether to give the user or service access to the data or allow them to execute the program.

The first step, authentication, may be omitted in special circumstances, such as when a user is using a ticket (see "Kerberos and Authorization" (page 41).)

Authentication and determination of permissions are facilitated by Authorization Services (see "Authorization Services" (page 66)). Each individual process, whether the Finder or your application, must then make its own determination of whether to allow access to the data or code it controls. The BSD permissions structure provides a basic level of access permission. You can implement more sophisticated or fine-grained access permissions based on user access lists or security policies of your own design. For example, you can let any user run your application but allow only users who are members of the `admin` group to change application preferences.

Authorization Services can be used to implement such security policies. For details, see *Authorization Services C Reference* and *Authorization Services Programming Guide* in Security Documentation.

Secure Storage

Secure storage is the protection of data through either access permissions or encryption. Access permissions can prevent users who are not computer experts from gaining access to data but cannot prevent someone who is capable of bypassing the operating system from reading data off the disk or out of memory. Therefore, highly sensitive data must be stored in an encrypted form.

Mac OS X and iOS provide an API, called **Keychain Services**, for securely storing small amounts of data, such as passwords or other short text strings. Mac OS X includes a utility that allows users to store and read the data in the keychain, called Keychain Access. In Mac OS X v10.3 and later, users can encrypt their entire Home folder by using FileVault, available through Security preferences. In Mac OS X, there is no high-level API for general encryption or for encryption of files or directories. You must use the CSSM API to perform such operations. For examples of code using CSSM for common encryption tasks, see the *CryptoSample* sample code. In iOS, the Certificate, Key, and Trust Services API includes functions for encrypting and decrypting blocks of data.

In iOS, backups of data to the user's computer are stored in plaintext, with the exception of passwords and other secrets on the keychain, which remain encrypted in the backup. It is therefore important to use the keychain to store passwords and other data (such as cookies) that are used to access secure web sites, as otherwise this data might be compromised if an unauthorized person gains access to the backup data.

Secure Communication

One important aspect of computer security is the secure communication of data over a network. Although you can devise your own security protocols and use low-level APIs such as BSD sockets and CSSM to implement them, it is usually much more convenient to use standard protocols and higher-level APIs when they are available. Mac OS X uses the SSL and TLS protocols and provides the Secure Transport, CFNetwork, and URL Loading System APIs for secure communication. The CFNetwork API is available on iOS as well.

Protocols for Secure Communication

SSL and **TLS** are versions of a security protocol that provides secure communication over a network. They are commonly used over TCP/IP connections such as the Internet. They use certificate-based authentication ("Digital Certificates" (page 31)) to ensure that you are communicating with a valid server, they validate data to prevent tampering, and they can use public-key cryptography ("Asymmetric Keys" (page 26)) to guard against eavesdropping or message forgery. SSL is built into all major browsers and web servers (the most recent versions also include TLS). Whenever you use a secure website—for example, to send your credit card number to a vendor over the Internet—and see a protocol identifier of `https` rather than `http` at the beginning of the URL, you are using SSL or TLS for communication.

Although the TLS protocol is not interoperable with SSL, the Mac OS X and iOS implementation of these protocols, Secure Transport, switches to SSL 3.0 if it cannot negotiate a TLS session with the other end of the connection.

Secure Transport uses certificate management and cryptography services provided by CDSA ("CDSA" (page 12)). Secure Transport has no transport-layer dependencies; it can be used with BSD sockets, Open Transport, or any other transport-layer protocol available.

For more information on the SSL standard, see http://wp.netscape.com/eng/ssl3/ and for the TLS standard, see http://www.ietf.org/html.charters/tls-charter.html.

Secure Communication APIs

There are three secure communication APIs in Mac OS X: Secure Transport, CFNetwork, and URL Loading System. The CFNetwork API is also available on iOS.

Secure Transport is the Mac OS X implementation of SSL and TLS. Applications are responsible for setting up the network connection and must provide callback functions that Secure Transport calls to perform I/O operations over the network. For more information about the capabilities and use of Secure Transport, see "Secure Transport" (page 70). For complete documentation of the Secure Transport API, see *Secure Transport Reference* in Security Documentation.

CFNetwork is a high-level C API that makes it easy to create, send, and receive serialized HTTP messages. Because CFNetwork is built on top of Secure Transport, you can encrypt the data stream using any of a variety of SSL or TLS protocol versions.

For more information about CFNetwork, see "CFNetwork" (page 71).

URL Loading System is a higher-level Mac OS X API built on CFNetwork. Rather than creating and maintaining a connection or a data stream, URL Loading System allows you to access the contents of a specific URL, including a secure `https://` URL. See "URL Loading System" (page 72).

Security Services

Mac OS X provides several security APIs to make it easy for developers to add such features as authorization and evaluation of digital certificates to their applications. This chapter describes security features provided automatically by Mac OS X, the high-level security APIs provided by Mac OS X and shown in Figure 1-3 (page 18), and user-level security features. The lower-level APIs provided by Apple's implementation of CSSM are not described here. Those APIs are fully documented in *Common Security: CDSA and CSSM*, version 2 (with corrigenda), from the Open Group (http://www.opengroup.org/security/cdsa.htm).

Restrictions On Code Execution

A common way for a hacker to gain control of a system is to exploit a buffer overflow in a running program. A buffer overflow occurs when a program does not validate its input and accepts more data than can fit in the memory that the program reserved for that data. The data then overwrites memory owned by the system or by some other program. In some circumstances, the hacker can insert executable code directly into memory this way; in other cases, the hacker can cause a jump of execution to another location in memory. For more information how such a buffer overflow is exploited, see "Types of Security Vulnerabilities" in *Secure Coding Guide*.

In order to make such exploits more difficult, starting in Mac OS X v10.4, a change was made to prevent the execution of code in the region of memory known as the *stack* on Intel-based Macintosh computers by default. In Mac OS X v10.5, the restrictions on executing code were extended in two ways: First, the system now disallows the execution of stack-based code on both PowerPC-based and Intel-based Macintosh computers by default. Second, for 64-bit programs, the system now disallows attempts to execute code in any portion of memory unless that portion is explicitly marked as executable. Most developers may not notice these changes because code compiled and linked statically is automatically marked as executable by the linker. A 64-bit application that generates code dynamically, however, must explicitly mark that code as executable or the program receives a SIGBUS signal and exits when trying to execute that code. A program can use the `mprotect` system call with the `PROT_EXEC` option to grant execute permissions to a block of memory containing dynamically generated code. For information on how to use this call, see the `mprotect` manual page.

To limit the damage in case a program is hijacked, both Mac OS X and iOS provide process sandboxing ("Sandboxing and the Mandatory Acccess Control Framework" (page 47)). In Mac OS X, a limited high-level sandboxing interface is provided by the command-line function `sandbox_init`. See the `sandbox_init` manual page for documentation. In iOS, every application is sandboxed during installation. The application, its preferences, and its data are restricted to a unique location in the file system and no application can access another application's preferences or data. In addition, an application running in iOS can see only its own keychain items.

Because every iOS application is sandboxed, your application's data and preferences cannot be read or modified by other applications, even if they have been compromised by an attacker. If your application is compromised, the attacker cannot use it to take control of the device or to attack other applications.

File Quarantine

Starting in Mac OS X v10.5, applications that download files from the Internet or receive files from external sources (such as email attachments) can assign quarantine attributes to the file using a function in Launch Services. The attributes associate basic information with the file, such as its type, when it was received, and the URL from which it came. When the Finder or any other program uses Launch Services to open a quarantined file, Launch Services inspects the file to see if it appears to be an application, script, or other executable file type. If so, the system displays an alert informing the user that the file is an application and asking for confirmation that it should be executed. The alert lets the user open the URL from which the file was downloaded, launch the program, or cancel. If the user proceeds to open the file, Launch Services removes the quarantine attributes from that file.

If you are developing a web browser or email program, or if your software somehow deals with files from unknown sources, you should use the Quarantine feature as part of your program's basic security procedures. Quarantine is part of the Launch Services API, which is itself part of the Core Services framework. For more information about the Quarantine feature, see *Launch Services Release Notes*.

Authentication

Authentication is the process of verifying the identity of a user or service. Authentication is normally done only as a step in authorization. Authentication answers the question "Is this entity who it claims to be?" before authorization asks "Does this entity have permission to perform this operation?" Therefore, Mac OS X has no separate authentication API. Some applications and operating system components carry out their own authentication; for example, see "Movie Toolbox Access Keys" (page 74). Mac OS X Authorization Services handles authentication for you when necessary (see "Authorization Services" (page 66); there is no authorization API in iOS). If you are using digital certificates for authentication—for example, when you need to authenticate a web server— use the functions in Certificate, Key, and Trust Services. See "Certificate, Key, and Trust Services" (page 67) for a description of that API. To exchange certificates over a secure connection, use the Secure Transport API described in "Secure Transport" (page 70) (Mac OS X only) or one of the high-level APIs that call Secure Transport—see "CFNetwork" (page 71) (Mac OS X or iOS) or "URL Loading System" (page 72) (Mac OS X only). To authenticate with a directory server, use the Open Directory API. See *Open Directory Programming Guide* in Networking Documentation for details.

Authorization Services

Authorization is the process by which an entity such as a user or a server gets the right to perform a restricted operation. (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. See "Authorization" (page 61) for more information about this process.

The principal Mac OS X API for authorization is **Authorization Services**. Authorization Services is implemented by the Security Server daemon ("Security Server" (page 15)) and built on top of BSD. Unlike BSD, however, which can control access at the level of individual files or programs, Authorization Services lets you control access to specific features or data within an application. The security daemon uses a policy database to determine the rights of a given authenticated user. Authorization Services includes functions to read, add,

edit, and delete policy database items. There is no authorization API in iOS—instead, the user can set a PIN to protect the device and each application is prohibited from accessing data belonging to another application (see "Restrictions On Code Execution" (page 65)).

When you request that Authorization Services authorize a user to perform a certain action, the Security Server authenticates the user if necessary. Because the interaction with the user is handled automatically for you, your code is not affected by the authentication method in use. For example, if at some time in the future Apple provides a user interface to support smart card authentication, your application will automatically gain the benefits of this new feature without any changes in your code.

Occasionally a Mac OS X application needs to perform some operation that requires running with root privileges, for example, when installing new software. In order to avoid having the entire application run as `root`, in this case the developer creates a separate helper tool that runs with root privileges only as long as is necessary. Applications in iOS cannot run with root privileges.

To learn how to use Mac OS X Authorization Services, start with *Authorization Services Programming Guide* and then look at *Authorization Services C Reference*, both in Security Documentation. There are also technical notes, Q&As, and sample code for Authorization Services available from the Reference Library > Security page on the ADC website.

Cryptography

You can use Keychain Services to encrypt and store small amounts of data (see "Keychain Manager and Keychain Services" (page 69)). In Mac OS X, if you want to encrypt or decrypt larger amounts of data, you can use the CSSM Cryptographic Services Manager (see "CDSA" (page 12)). This manager also has functions to create and verify digital signatures, generate cryptographic keys, and create cryptographic hashes. To see exactly which security protocols and algorithms are supported by Apple's Cryptographic Service Provider (CSP) implementation, see the documentation provided with the Open Source security code, which you can download at http://developer.apple.com/opensource/security/index.html. In iOS, CSSM is not available, but many of these functions are provided by the Certificate, Key, and Trust Services API.

The sample code *CryptoSample* contains source code and program examples for a library intended to facilitate the use of the Cryptographic Services Manager, specifically for symmetric encryption and message digest calculation.

Starting with Mac OS X v10.5, you can use the Cryptographic Message Syntax Services programming interface to encrypt or add a digital signature to S/MIME messages. See *Cryptographic Message Syntax Services Reference* for details.

Certificate, Key, and Trust Services

Certificate, Key, and Trust Services is a C API for managing certificates, public and private keys, and trust policies. You can use these services in your application to:

- Add a certificate to a keychain
- Retrieve information from a certificate
- Convert between a binary representation of a certificate and a certificate object usable by the API

- Retrieve the certificate associated with a specific cryptographic identity
- Retrieve the private key that's associated with a specific cryptographic identity
- Create a pair of asymmetric keys
- Retrieve the value of a trust policy
- Evaluate the trust associated with a specific certificate and trust policies
- Set anchor certificates

In Mac OS X, functions are also available to:

- Retrieve anchor certificates
- Set user-specified settings for trust policies for a given certificate (see Figure 3-3 (page 73) for examples of trust settings)

In iOS, additional functions are provided to:

- Use a private key to generate a digital signature for a block of data
- Use a public key to verify a signature
- Use a public key to encrypt a block of data
- Use a private key to decrypt a block of data

Certificate, Key, and Trust Services operates on certificates that conform to the X.509 ITU standard, uses the keychain for storage and retrieval of certificates and keys, and uses the trust policies provided by Apple. See the Security Release Notes for details about Apple's certificate trust policies.

Because certificates are used by SSL and TLS for authentication, the Mac OS X Secure Transport API includes a variety of functions to manage the use of certificates and root certificates in a secure connection. See "Secure Transport" (page 70) for more information about Secure Transport.

To display the contents of a certificate in a Mac OS X user interface, you can use the `SFCertificatePanel` and `SFCertificateView` classes in the Security Objective-C API. In addition, the `SFCertificateTrustPanel` class displays trust decisions and lets the user edit trust decisions. See "Security Objective-C API" (page 72) for more information about this API.

Code Signing

Starting with Mac OS X v10.5, developers are expected to add a digital signature ("Digital Signatures" (page 29)) to each Mac OS X program. Digital signatures are required on all applications for iOS. In addition, Apple adds its own signature before distributing an iOS application. The addition of a digital signature to an application or block of code is often referred to as **code signing**.

Code signing ensures the integrity of the program and allows the system to recognize updated versions as the same program as the original. Once a program is signed, any change in the code not intended by the developer—whether introduced accidently or by hackers—can be detected by the system. On the other hand, a signature on an updated version of a program tells the system to treat the new version exactly as it

treated the old. On Mac OS X, this means that users are not asked to give their keychain password before the system executes the program. On iPhone, users are never asked for a password, but an application that hasn't been signed by Apple will not execute.

Because signatures are ignored by Mac OS X versions prior to Mac OS X v10.5, signing applications does not affect whether they are backward compatible with earlier versions of the operating system. On the other hand, because Mac OS X v10.5 and later systems expect all code to be signed, any code that is not signed does not behave in the same manner as the majority of the programs on the user's system. In particular, the user is likely to be bothered with additional dialog boxes and prompts for unsigned code that they don't see with signed code, and unsigned code might not work as expected with some system components, such as parental controls. On iOS, Apple does not sign applications that have not been signed by the developer, and applications not signed by Apple simply will not run.

The procedure for signing code is not time consuming and requires few resources. See *Code Signing Guide* for details.

Keychain Manager and Keychain Services

The **keychain** provides secure storage of passwords, keys, certificates, and notes for users. Applications can use the keychain to store and retrieve data, passwords, encryption keys, and certificates. In Mac OS X, the user is prompted for permission when an application needs to access the keychain; if the keychain is locked, the user is asked for a password to unlock it. In iOS, an application can access only its own items in the keychain—the user is never asked for permission or for a password. There are two Mac OS X APIs for the keychain: Keychain Manager and Keychain Services. Only Keychain Services is available in iOS.

The **Keychain Manager** is part of the Carbon framework and is maintained for compatibility with older versions of Mac OS X. For newer versions of Mac OS X (starting with Mac OS X v10.2), Keychain Manager functions call Keychain Services rather than calling CSSM directly. In addition, because all new API development is being done on the **Keychain Services** API, it is a richer and more flexible API than the Keychain Manager. Therefore, Keychain Services is the preferred API unless you are writing code to run on Mac OS X v10.1 or earlier.

Keychain Services allows you to create keychains, add, delete, and edit keychain items, and—in Mac OS X only—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.

Backups of iOS data are stored in plaintext, with the exception of passwords and other secrets on the keychain, which remain encrypted in the backup. It is therefore important to use the keychain to store passwords and other data (such as cookies) that are used to access secure web sites, as otherwise this data might be compromised if an unauthorized person gains access to the backup data.

To get started using Keychain Services, see *Keychain Services Programming Guide* and *Keychain Services Reference*.

In Mac OS X, the Keychain Access application provides a user interface to the keychain. See "Keychain Access" (page 77) for more information about this application.

Smart Card Services

A **smart card** is a plastic card similar in size to a credit card that has memory and a microprocessor embedded in it and is therefore capable of both storing information and processing it. For security purposes, smart cards can store passwords, certificates, and keys. A smart card normally requires a personal identification number (PIN) or biometric measurement (such as a fingerprint) as an additional security measure. Because it contains a microprocessor, a smart card can carry out its own authentication evaluation offline before releasing information. Smart cards can exchange information with a personal computer through a smart card reader.

The PC/SC Workgroup (http://www.pcscworkgroup.com/) has established a standard for accessing cards and writing card reader drivers.

Apple provides a Smart Card Services software development kit (SDK), which contains source code that you can use to implement a PC/SC-compliant application or driver. The PC/SC framework, with header files, is included in Mac OS X at `/System/Library/Frameworks/PCSC.framework`. Apple's smart card support is based on the Movement for the Use of Smart Cards in a Linux Environment (MUSCLE) Open Source implementation of the PC/SC standard. The MUSCLE homepage is http://www.linuxnet.com/ and the MUSCLE PC/SC Lite API Toolkit API Reference Documentation is available at http://pcsclite.alioth.debian.org/pcsc-lite/.

The ADC Security homepage at http://developer.apple.com/security/ includes a link to the sourcecode for Apple's Smart Card Services project, which is at http://www.opensource.apple.com/darwinsource/Current/SmartCardServices/. You must agree to the Apple Public Source License (APSL) before you can download the code. The code includes drivers and the pcscd command line tool, which launches a PC/SC smart card daemon. See the manual page for `pcscd` for more information on this daemon.

The preferred means to work with smart cards on Mac OS X v10.4 and later is by using Keychain Services. Mac OS X v10.4 and later implements the Tokend interface that allows smart card developers to make their cards appear to be keychains. Tokend is not yet an official API, but it is supported by DTS. DTS has a sample Tokend project and related documentation available on request. Keychain Services is documented in *Keychain Manager Reference* and *Keychain Services Programming Guide*.

Secure Transport

Secure Transport is Apple's implementation of SSL and TLS, used to create secure connections over TCP/IP connections such as the Internet (see "Secure Communication" (page 62)). In iOS, there is no API for Secure Transport; use CFNetwork for secure connections. In Mac OS X, you can use the Secure Transport API to set parameters for a secure session, open and maintain a session, and close a session. Functions provided by Mac OS X Secure Transport allow you to:

- Set which SSL/TLS protocol versions are allowed
- Specify which cipher suites should be enabled for a session
- Specify whether client-side authentication should be required
- Specify certificates to be used for the session
- Specify whether expired certificates are allowed
- Specify trusted root certificates for the session
- Specify whether unknown or expired root certificates are allowed

- Specify Diffie-Hellman parameters
- Specify the domain name of the other end of the connection
- Set up and open an SSL or TLS session
- Determine which SSL/TLS protocol was negotiated for the session
- Determine which cipher suite was negotiated for the session
- Obtain the current ID data of the other end of the connection
- Read and write data in a session
- Terminate the session

To get started with Secure Transport, see *Secure Transport Reference*. For sample code, see *SSLSample*.

OpenSSL Note: Although Mac OS X includes a low-level command-line interface to the OpenSSL open-source cryptography toolkit, this interface is not available on the iOS. For iOS development, use the CFNetwork API for secure networking and the Certificate, Key, and Trust Services API for cryptographic services.

CFNetwork

CFNetwork is an API for creating, sending, and receiving serialized messages over a network. It is a high-level interface that can be used to set up and maintain a secure SSL or TLS networking session. CFNetwork includes the following security-related components:

- CFHTTPMessage, which you can use to create, serialize, deserialize, and manage HTTP protocol messages. This component lets you add authentication information to a message.
- CFHTTPAuthentication, which applies authentication credentials to challenged HTTP messages.
- CFStream Socket Additions, which allocates read and write streams and provides constants used with the CFReadStream and CFWriteStream APIs to set security protocols.
- CFFTPStream, which you can use to peform FTP file transfers. This component lets you send passwords to FTP servers.

In addition to the CFNetwork API, you use the CFReadStream and CFWriteStream APIs in the Core Foundation framework to create and manage the read and write streams that CFNetwork depends on. You can specify an SSL or TLS protocol version to encrypt and decrypt the data stream. Note that CFReadStream and CFWriteStream are "toll-free bridged" with their Cocoa Foundation counterparts, the classes `NSInputStream` and `NSOutputStream`. This means that each Core Foundation type is interchangeable in function or method calls with the corresponding bridged Foundation object, so you can use either C or Objective C interfaces, whichever is most convenient for you.

URL Loading System

The URL Loading System is a very high-level Mac OS X API that you can use to access the contents of `HTTP://`, `HTTPS://`, and `FTP://` URLs. Because `HTTPS://` websites use SSL or TLS to protect data transfers, you can use the URL Loading System as a secure transport API. See *URL Loading System Programming Guide* for information about this API.

Kerberos

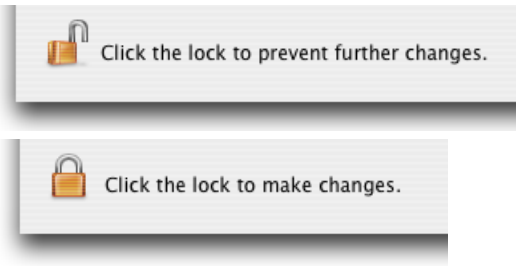
Kerberos is an industry-standard protocol used to provide authentication over a network. Kerberos is highly secure, can be used for any number of users and servers on a network, and provides **single-signon** authentication—that is, users provide authentication credentials (such as user ID and password) only once, after which they can access additional services without reauthenticating. Mac OS X Kerberos works with all common directory servers, including **LDAP** (Lightweight Directory Access Protocol) servers and Microsoft Windows Active Directory servers. Mac OS X Server v10.2 and later can host Kerberos authentication services and Mac OS X v10.5 and later includes a full Kerberos implementation, so that any user's computer can serve as either a Kerberos server or client.

Security Objective-C API

The Security Objective-C API (available only on Mac OS X) includes the`SFAuthorization` class, which provides an Objective-C interface for Authorization Services. The other classes in this API provide security-related UI elements, as follows:

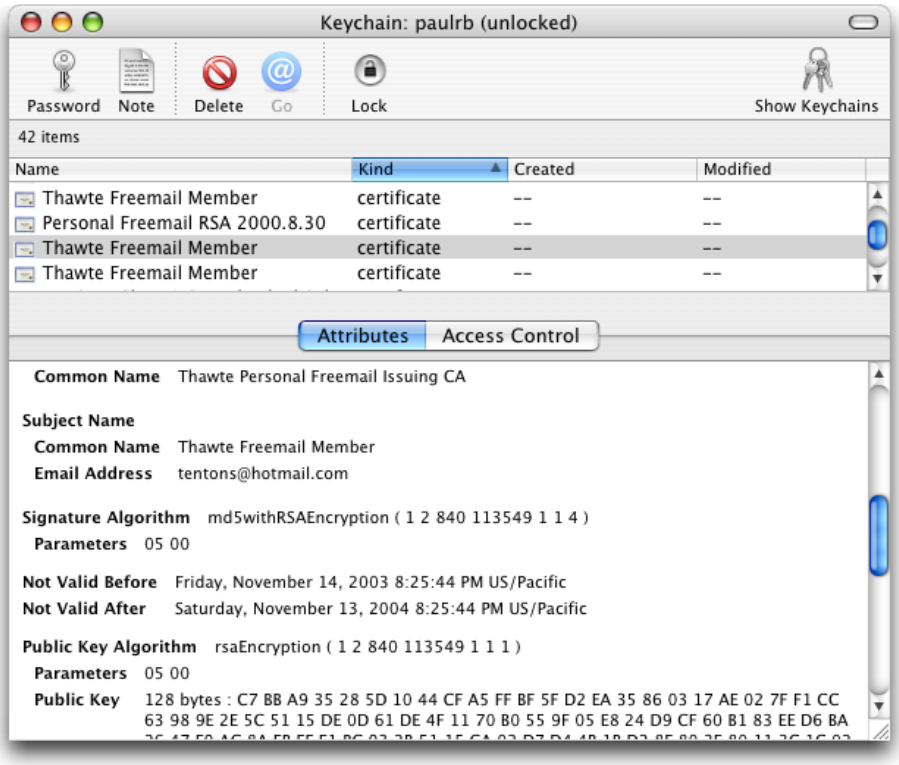
- The`SFAuthorizationView` class implements an authorization view in a window. An authorization view is a lock icon and accompanying text that indicates whether an operation can be performed (Figure 3-1). If the lock is closed, when the user clicks it, an authorization dialog displays. Once the user is authorized, the lock icon appears open. When the user clicks the open lock, Authorization Services restricts access again and changes the icon to the closed state.

  Figure 3-1 Authorization view



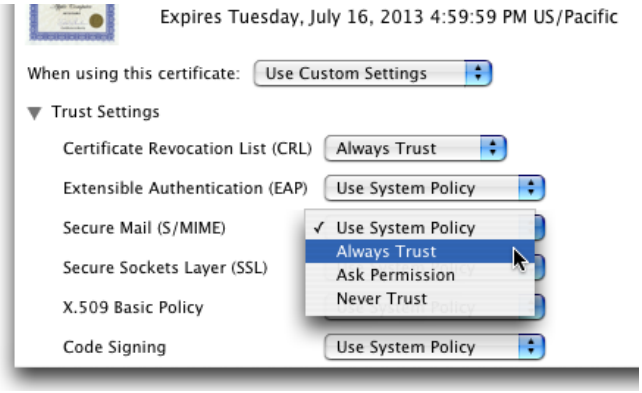
- The `SFCertificateView` and `SFCertificatePanel` classes display the contents of a certificate. The `SFCertificateView` class is used by the Keychain Access application, for example (Figure 3-2).

Figure 3-2 Certificate view



- The `SFCertificateTrustPanel` class displays and optionally lets the user edit the trust settings in a certificate. Figure 3-3 shows this feature as used by the Keychain Access application.

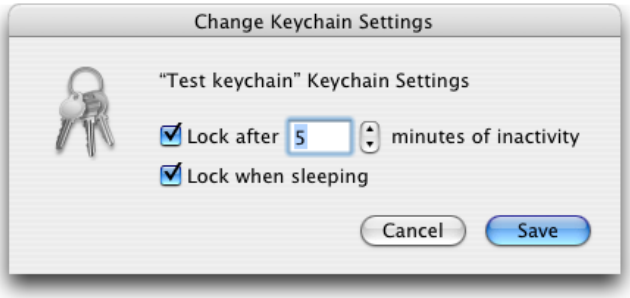
Figure 3-3 Editable trust settings



- The `SFChooseIdentityPanel` class displays a list of identities in the system and lets the user select one. In this context, **identity** refers to the combination of a private key and its associated certificate. If a user had two or more certificates, for example, each with its own private key, the user's email application could use this interface to let the user select which identity to use to sign a specific letter.

- The `SFKeychainSavePanel` class adds an interface to an application that lets the user save a new keychain. The user interface is nearly identical to that used for saving a file. The difference is that this class returns a keychain in addition to a filename and lets the user specify a password for the keychain.
- The `SFKeychainSettingsPanel` class displays an interface that lets the user change keychain settings. Figure 3-4 shows this interface in the Keychain Access application.

Figure 3-4 Keychain settings



Documentation for the Security Objective-C API is in *Security Interface Framework Reference*.

Movie Toolbox Access Keys

Movie Toolbox Access Keys is a QuickTime API that provides password protection to QuickTime data. You can add password protection to a QuickTime movie—so that only users who know the password can run the movie— or you can add password protection to data, so that only an application that has registered that access key can get access to the data.

For documentation on Movie Toolbox Access Keys, see "Movie Toolbox Access Keys" in *QuickTime Movie Internals Guide*.

User-Level Security Features

There are many security features built into Mac OS X and iOS, including industry-standard digital signatures and encryption for Apple's Mail application and authentication for the Safari web browser. The four features most visible to users of Mac OS X are:

- Security system preferences
- File Vault, which users can configure through Security system preferences
- Accounts system preferences
- the Keychain Access application

Security System Preferences

Security system preferences in Mac OS X let the user configure FileVault (discussed next) and control some aspects of authorization on the computer (Figure 3-5).

Figure 3-5 Security system preferences

The Security system preferences dialog lets the user specify whether authorization should be required:

- To wake the computer from sleep or a screen saver
- For every account on login
- To unlock each lockable system preference

At the bottom of the dialog is the lock icon provided by the authorization view (see "Security Objective-C API" (page 72)). When this icon shows a closed lock, authorization is required before the user can change the settings in this system preferences dialog.

FileVault

When the user turns on the FileVault operating system feature (see Figure 3-5 (page 75)), Mac OS X uses 128-bit **AES encryption** to encrypt everything in the user's home folder. As long as the user is authenticated and logged in, the system automatically unencrypts any file the user opens. However, no other user can gain access to these files.

AES (Advanced Encryption Standard) is a symmetric-key algorithm adopted by the National Institute of Standards and Technology (NIST) as a standard for government and private use to protect sensitive, nonclassified data. It enables very fast and highly secure encryption and decryption of data. Because it is a symmetric-key algorithm, keys are stored securely on the user's computer.

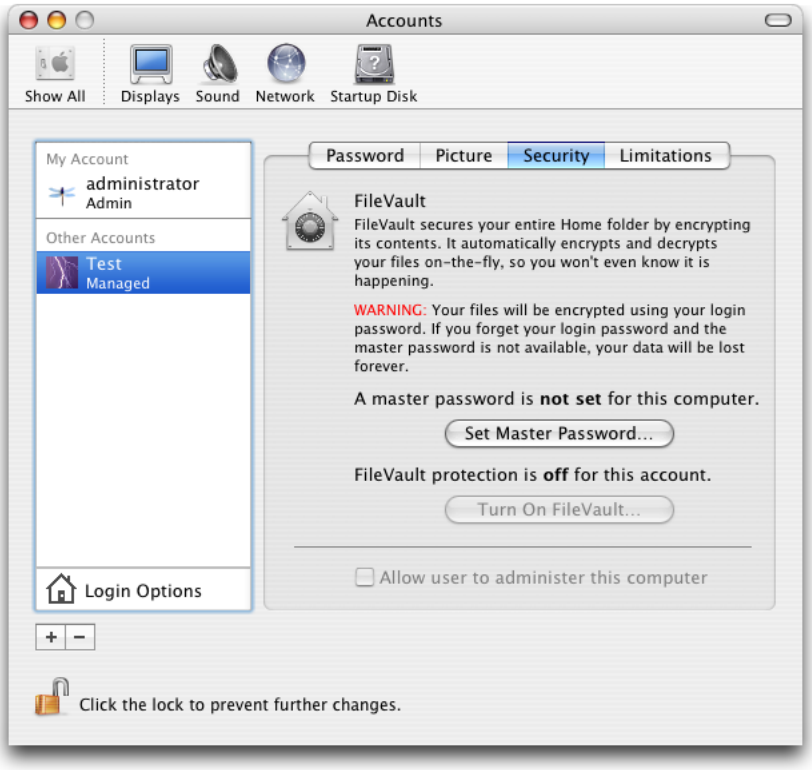
Full documentation of the AES algorithm is available on the NIST website at http://csrc.nist.gov/CryptoToolkit/aes/rijndael/.

Accounts System Preferences

When a user installs Mac OS X on a computer, that user automatically becomes a member of the `admin` group ("The Admin Group" (page 58)). Subsequently, the user or any other member of the `admin` group can use Accounts system preferences to add new users to the system.

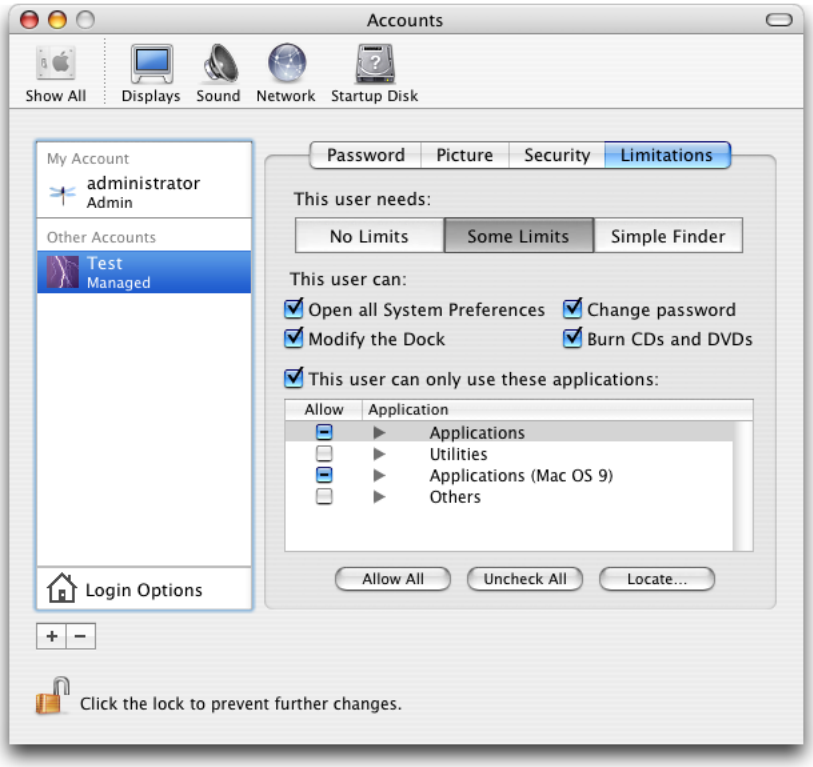
For each new user, the administrator can specify whether that user is a member of the `admin` group (Figure 3-6). If a FileVault master password has been set, the administrator can also turn on FileVault for the new account.

Figure 3-6 Accounts system preferences Security pane



If the new user is not a member of the `admin` group, the administrator can limit the system features and applications to which that user has access (Figure 3-7).

Figure 3-7 Accounts system preferences Limitations pane



Keychain Access

Keychain Access is a Mac OS X utility that gives users access to Keychain Services ("Keychain Manager and Keychain Services" (page 69)). A user can see the passwords, certificates, and other data that are stored in their keychain. They can create new keychains, add and delete keychain items, lock and unlock keychains, and select one keychain to be the default.

Keychain access lets the user see what certificates are available for use by email and web applications, who owns each certificate, and who issued each certificate. Certificates are described in "Digital Certificates" (page 31).

The user can see and change passwords stored for various applications and can securely store other secrets such as passwords, credit card numbers, and notes. When a keychain is locked and an application needs to gain access to a keychain item, Keychain Services prompts the user for a password.

In addition, the Keychain Access menu includes items to open the **Certificate Assistant** and **Kerberos Ticket Viewer** utilities. The Certificate Assistant enables users to create certificates, request certificates from a certificate authority, create a public/private key pair, or evaluate a certificate. The Kerberos Ticket Viewer lets users see any Kerberos tickets in use on the system, and enables them to renew or destroy a ticket as well as change a ticket's password.

Document Revision History

This table describes the changes to *Security Overview*.

Date	Notes
2010-07-13	Reorganized file system permissions into a single section and enhanced the content.
2008-10-15	Added a link to the iPhone OS Technology Overview.
2008-06-26	Added information about security in iPhone OS.
2008-02-08	Added descriptions of new features for Mac OS X v10.5.
	See "Sandboxing and the Mandatory Acccess Control Framework" (page 47); "Cryptography" (page 67); "Code Signing" (page 68); "Restrictions On Code Execution" (page 65); and "File Quarantine" (page 66).
2005-04-29	Updated for Mac OS X v10.4. Added information about file-system access control lists (ACLs). Changed "Rendezvous" to "Bonjour."
2004-02-23	New document that introduces computer security concepts and describes the security features and APIs in Mac OS X.

Document Revision History

Glossary

access control entry See ACE.

access control list See ACL.

access permissions See permissions.

access rights See permissions.

ACE Abbreviation for access control entry. An ACE is a component of an ACL that associates a user or group with a set of permissions and specifies whether each permission is allowed or denied. See also ACL.

ACL Abbreviation for access control list. A set of permissions associated with a user or group. An ACL consists of an ordered list of ACEs.

admin group A group with special administrative privileges. For example, only members of the `admin` group can open locked system preferences or install software. See also wheel group.

administrator A member of the `admin` group.

AES encryption Abbreviation for Advanced Encryption Standard encryption. A Federal Information Processing Standard (FIPS), described in FIPS publication 197. AES has been adopted by the U.S. government for the protection of sensitive, non-classified information. The algorithm was developed by Dr. Joan Daemen and Dr. Vincent Rijmen and was named the Rijndael algorithm. It is a symmetric-key algorithm that can use key sizes of 128, 192, or 256 bits. Apple has adopted the 128-bit version of AES for FileVault. There are approximately 3.4 x 10**38 possible 128-bit keys.

AFP Abbreviation for Apple Filing Protocol. The principal file-sharing protocol in Mac OS 9 systems, used by AppleShare servers and clients.

algorithm A sequence of actions to accomplish some task. In cryptography, refers to a sequence of actions, usually mathematical calculations, performed on data to encrypt or decrypt it.

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

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. Compare symmetric keys.

authentication The process by which a person or other entity (such as a server) proves that it is who (or what) it says it is. Compare authorization; identification.

authentication server A server that has access to a store of authentication information and that can authenticate users. For example, an authentication server might verify a user's identity by prompting the user for a name and password and comparing that information to the names and passwords in a database. In Kerberos authentication, the authentication server also looks up the user's private key, generates a session key, and creates a ticket-granting ticket (TGT). See also ticket-granting server.

authorization The process by which an entity such as a user or a server gets the right to perform a privileged operation. (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. Compare authentication.

Authorization Services A Mac OS X API that applications can use to restrict access to files or services.

BSD Berkeley Software Distribution. BSD is a form of the UNIX operating system and provides the basis for the Mac OS X file system, including file access permissions.

CA See certification authority (CA).

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. In addition, Mac OS X includes its own security APIs that call the CDSA API for you. See also CDSA plug-in.

CDSA plug-in A software module that connects to CDSA through a standard interface and that implements or extends CDSA security services for a particular operating system and hardware environment.

certificate See digital certificate.

Certificate Assistant A utility available through the Keychain Access Utility that can be used to create certificates and keys, request certificates from a certificate authority, and evaluate certificates.

certificate authority See certification authority (CA).

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

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

Certificate, Key, and Trust Services An API you can use to create, manage, and read certificates; add certificates to a keychain; create encryption keys; and manage trust policies. In iOS, you can also use this API to encrypt, decrypt, and sign data.

certificate subject The entity associated with the public key that is in the certificate.

certification authority (CA) The issuer of a digital certificate. 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.

CFHTTP An API that you can use to create, serialize, deserialize, and manage HTTP protocol messages, including secure HTTPS messages. This component lets you add authentication information to a message. CFHTTP is a component of CFNetwork and is built on top of CFStream.

CFNetServices An API that allows you to use Bonjour. Bonjour enables applications to discover services that are available on the network and find all access information (such as name and IP address) needed to use each service. CFNetServices is a component of CFNetwork. This component has no security features.

CFNetwork A high-level API used for creating, sending, and receiving serialized messages over a network. CFNetwork is built on top of Secure Transport, and so can use the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) secure networking protocols.

CFStream An API that creates and manages the read and write streams that CFHTTP depends on. CFStream is a component of CFNetwork and is built on top of Secure Transport. You can specify a Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocol version to encrypt and decrypt the data stream.

CIFS Acronym for Common Internet File System. A file-sharing protocol used widely on Windows and UNIX systems. CIFS is an extension of the SMB protocol. CIFS has been given to the Internet Engineering Task Force (IETF), making it an Internet standard. Unlike SMB, CIFS runs only over TCP/IP. See also Samba.

ciphertext Text or other data that has been encrypted. Compare plaintext.

code signing The addition of a digital signature to an application or block of code.

credentials Data that can be used to identify, authenticate, or authorize an entity. For example, a user name and password constitute authentication credentials. A Kerberos ticket, consisting of an encrypted session key and other information, is an identification credential. In Kerberos version 5 and later, tickets can also carry authorization information.

cryptographic hash function An algorithm that takes any amount of data and transforms it into a fixed-size output value. For a cryptographic hash function to be useful for security, it has to be extremely difficult or impossible to reconstruct the original data from the hash value, and it must be extremely unlikely that the same output value could result from any other input data. See also message digest.

cryptographic hashing The process whereby data is transformed using a cryptographic hash function.

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

decryption The transformation of ciphertext back into the original plaintext. Compare encryption. See also asymmetric keys; symmetric keys.

Diffie-Hellman key exchange A protocol that provides a way for two ends of a communication session to generate symmetric private keys through the exchange of public keys.

digest See message digest.

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 and iOS support the X.509 standard for digital certificates. See also certificate chain.

digital ID See digital certificate.

digital signature A way to ensure the integrity of a message or other data using public key cryptography. To create a digital signature, the signer generates a message digest of the data and then uses a private key to encrypt the digest. The signature includes the encrypted digest and identifies the signer. Anyone wanting to verify the signature uses the signer's digital certificate, which contains the public key needed to decrypt the digest and specifies the algorithm used to create the digest.

encryption The transformation of data into a form in which it cannot be made sense of without the use of some key. Such transformed data is referred to as *ciphertext*. Use of a key to reverse this process and return the data to its original (or *plaintext*) form is called *decryption*.

EGID Abbreviation for Effective Group ID. See GID.

EUID Abbreviation for Effective User ID. See UID.

file GID The GID associated with a file system object. Each file system object has a user ID (the file UID, commonly referred to as the file's owner), a group ID (the file GID, commonly referred to as the file's group), and three sets of permission bits, known as owner, group, and other permissions. The first set of bits controls access to the object by the owner, the second controls access by members of the group, and the third controls access by everyone else. See also process GID.

file's group See file GID.

file's owner See file UID.

file UID The UID of a file system object, used to determine the object's permissions. Each file system object has a user ID (the file UID, commonly referred to as the file's owner), a group ID (the file GID, commonly referred to as the file's group), and three sets of permission bits, known as owner, group, and other permissions. The first set of bits controls access to the object by the owner (any process whose effective UID is equal to the file UID); the second controls access by members of the group; and the third controls access by everyone else.

GID Abbreviation for group ID, a unique identifier for a collection of users. In BSD, each user can belong to one or more groups. Each file system object has an associated GID that is used to determine the object's permissions. Each process has an associated group list. See also process GID.

group See GID.

group list The list of groups to which the owner of a process belongs plus any additional groups added to the list programatically (for example, using the `setgid` command). If the file GID of a file system object matches the GID of any group in the group list, that group has group permissions for the object. See also file UID.

GSS-API Generic Security Service Application Program Interface; an open-source API that can be used to adapt an application to use Kerberos.

hash algorithm See cryptographic hash function.

identification The process by which a process verifies that a person or entity is the same one it communicated with previously. Identification is in general faster than authentication and does not require interaction with the user. In Kerberos, for example, the authentication server authenticates a user and issues a credential (called a *ticket-granting ticket*), which can be used later for identification so that reauthentication is not necessary.

identity A digital certificate together with an associated private key.

KDC See key distribution center (KDC).

kerberized service A service that has been configured to accept Kerberos tickets for identification.

Kerberos An industry-standard protocol created by the Massachusetts Institute of Technology (MIT) to provide authentication over a network. It is a symmetric-key, server-based protocol and is used widely in Macintosh, Windows, and UNIX networks.

Kerberos ticket A credential used to identify a user who has been previously authenticated so that reauthentication is not needed. In Kerberos, the Kerberos key distribution center (KDC) issues the user a ticket-granting ticket (TGT) when they first authenticate. Thereafter, when they need to access a secure server, they present the ticket-granting ticket to the KDC and are issued a ticket, which they present to the secure server as identification. See also authentication; identification.

Kerberos Ticket Viewer A utility available through the Keychain Access utility that shows any Kerberos tickets in use on the system and enables the user to renew or destroy a ticket or change a ticket's password

key A piece of secret information required to decode an encrypted message. In modern cryptographic methods, it is usually a lengthy integer.

keychain A database in Mac OS X and iOS 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.

Keychain Access A Mac OS X utility that enables users to view and modify the data stored in the keychain.

Keychain Manager An API for securely storing small amounts of data on the keychain, kept for compatibility with older versions of the operating system. New code should use Keychain Services instead.

Keychain Services An API for securely storing small amounts of data on the keychain.

key distribution center (KDC) A Kerberos term referring to the sum of two separate software processes: the ticket-granting server and the authentication server.

LDAP Acronym for Lightweight Directory Access Protocol. A standard client-server protocol for accessing online directory services.

level of trust The confidence you can have in the validity of a certificate, based on the certificates in its certificate chain and on the certificate extensions the certificate contains. The level of trust for a certificate is used together with the trust policy to answer the question "Should I trust this certificate for this action?"

Mach The lowest level of the Mac OS X and iOS kernels. Mach provides such basic services and abstractions as threads, tasks, ports, interprocess communication, scheduling, address space management, virtual memory, and timers.

man-in-the-middle attack An attack on a communication channel in which the attacker can intercept messages going between two parties without the communicating parties' knowledge. Typically, the man in the middle substitutes messages and even cryptographic keys to impersonate one party to the other.

message digest The result of applying a cryptographic hash function to a message or other data. A cryptographically secure message digest cannot be transformed back into the original message and cannot (or is very unlikely to) be created from a different input. Message digests are used to ensure that a message has not been corrupted or altered. For example, they are used for this purpose in digital signatures. The digital signature includes a digest of the original message, and the recipient prepares their own digest of the received message. If the two digests are identical, then the recipient can be confident that the message has not been altered or corrupted.

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

Movie Toolbox Access Keys A QuickTime API that can be used to add password protection to QuickTime data.

NFS Abbreviation for Network File System. The main file-sharing protocol used by UNIX systems.

nobody A special user with very little access. To prevent someone running as `root` or as an administrator on one system from gaining control over another system through a network connection, such users are often mapped to the `nobody` user on the remote system.

one-time pad authentication A form of shared secret authentication in which both parties have an identical list of pairs of numbers, words, or symbols and each pair is used only once.

owner See UID.

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 ACL; authorization; UID.

PKI See public key infrastructure (PKI).

PKINIT A protocol that defines the use of public key cryptography for initial authentication in Kerberos.

plaintext Ordinary, unencrypted data. Compare ciphertext.

plug-in A code module that uses a standard interface to implement certain features of a program or extend the program. See also CDSA plug-in.

port In Mach, a port is an endpoint of a communication channel between a client who requests a service and a server who provides the service. Mach ports are unidirectional; a reply to a service request must use a second port. See also port right.

port right In Mach, a specification of which task can send to or receive from a particular port.

port right name A small integer used to identify a Mach port right. Each process has a port right namespace, which maps port right names to their corresponding port rights. A port right name is meaningful only within that task's port right namespace.

private key A cryptographic key that must be kept secret. Whereas a pair of identical private keys can be used as symmetric keys, asymmetric keys consist of one private key and one public key.

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

process GID The GID of a process. Each process has three group IDs: the real group ID (RGID), effective group ID (EGID), and saved group ID (SGID). The RGID is always inherited from the user or process who executes the process. The EGID is the first GID in the group list. The SGID is used by BSD to enable a privileged process to switch in and out of privileged mode.

process UID The UID of a process. Each process has three user IDs: the real user ID (RUID), effective user ID (EUID), and saved user ID (SUID). The RUID is always inherited from the user or process who executes the process. The EUID is normally the same as the RUID but can differ in special circumstances. It is the EUID that BSD checks to determine permissions. The SUID is used by BSD to enable a privileged process to switch in and out of privileged mode.

pseudorandom number A number generated by an algorithm that produces a series of numbers with no discernible pattern. It should be impossible or nearly impossible to deduce the algorithm from such a series. However, unlike a truly random number generator, a pseudorandom number generator always produces the same series if the algorithm is given the same starting value or values.

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

public key certificate See digital certificate.

public key cryptography A cryptographic method using asymmetric keys 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. See also digital certificate; digital signature.

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

quantum computer A computer in which the logic gates are based on quantum phenomena such as electron spin rather than mechanical or conventional electronic components. Because of the superposition of quantum states (a consequence of the Heisenberg Uncertainty Principle), a properly designed quantum computer can in principle perform simultaneously certain types of calculations that require a huge number of sequential operations in a classic computer. Consequently, factoring large numbers should be several orders of magnitude faster on a quantum computer than on present-day supercomputers. Because the strength of most modern cryptographic methods depends on the difficulty of making such calculations, a practical quantum computer would break most cryptographic schemes in common use. Although small proof-of-concept quantum computers have been constructed, no such machine capable of solving practical problems has yet been demonstrated.

Randomization Services An iOS API that produces cryptographically secure pseudorandom numbers.

realm A subset of a large network served by its own Kerberos authentication server and ticket-granting server.

RGID Abbreviation for Real Group ID. See GID.

root certificate A certificate 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. Compare anchor certificate.

root certification authority The owner of the root certificate.

root user The user on a UNIX system with a UID of 0. A process running with an EUID of 0 is said to be *running as root*. The root user owns many of the primary system processes and has unlimited access to the file system objects on the devices attached to the computer.

RSA encryption A system of public key cryptography, named for its inventors: Ron Rivest, Adi Shamir, and Leonard Adleman. The RSA algorithm takes two large prime numbers, finds their product, and then derives asymmetric keys from the prime numbers and their product. Because the public key includes the product, the private key could be derived from the public key if the product could be factored. No easy method for factoring products of large prime numbers is currently known, but it has not been mathematically proven that no such method is possible. Therefore, the discovery of a fast way to factor such numbers, or the development of quantum computers, would break RSA.

RUID Abbreviation for Real User ID. See UID.

Samba Software that implements SMB/CIFS on a UNIX server.

sandboxing A system feature that provides fine-grained control of the ability of processes to access system resources, therefore limiting the amount of damage that can be done by a malicious hacker that gains control of an application.

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; public key cryptography.

Secure Sockets Layer (SSL) A protocol that provides secure communication over a TCP/IP connection such as the Internet. It uses digital certificates for authentication and digital signatures to ensure message integrity, and can use public key cryptography 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. SSL has been superseded by Transport Layer Security (TLS).

secure storage Storage of encrypted data on disk or another medium that persists when the power is turned off.

Secure Transport The Mac OS X and iPhone implementation of Secure Sockets Layer (SSL) and Transport Layer Security (TLS), used to create secure connections over TCP/IP connections such as the Internet. On Mac OS X, Secure Transport includes an API that is independent of the underlying transport protocol. The CFNetwork and URL Loading System APIs use the services of Secure Transport.

Security Agent In Mac OS X, a process used by the Security Server to communicate with the user through dialogs and other user interface elements.

Security Objective-C API A Mac OS X API providing a set of Objective-C methods that are wrappers for the Authentication Services functions plus a set of classes that display security-related UI elements.

Security Server A daemon running in Mac OS X and iOS that implements security protocols for such purposes as encryption, decryption, and authorization computation. The use of the Security Server to perform actions with cryptographic keys allows the keys to be maintained in a separate address space from the client application, keeping them more secure. In Mac OS X, the Security Server uses a process called the *Security Agent* to communicate with the user through dialogs and other user interface elements.

session key A cryptographic key calculated or issued for use only for the duration of a specific communication session. Session keys are used, for example, by the Diffie-Hellman key exchange and Kerberos protocols.

SGID Abbreviation for Saved Group ID. See GID.

shared secret authentication An authentication method based on a secret known to only the two parties involved. Verification of passwords is a commonly used shared secret authentication method.

single signon A feature of a security system whereby users provide authentication credentials (such as user ID and password) only once, after which they can access additional services without reauthenticating. See also authentication; ticket.

smart card A plastic card similar in size to a credit card that has memory and a microprocessor embedded in it. A smart card can store and process information, including passwords, certificates, and keys. A smart card normally requires a personal identification number (PIN) or biometric measurement (such as a fingerprint) before releasing information and can carry out its own authentication evaluation. Smart cards can exchange information with a personal computer through a smart card reader.

SMB Abbreviation for Server Message Block. A file-sharing protocol used on Windows and UNIX systems. SMB can also be used to share printers and has calls to authenticate users. It runs over several different types of networks, including TCP/IP. For most purposes, SMB has been superseded by CIFS. See also Samba.

SMB/CIFS Abbreviation for Server Message Block/Common Internet File System. See CIFS; SMB. See also Samba.

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

SSL See Secure Sockets Layer (SSL).

strength A measure of the amount of effort required to break a security system. For example, the strength of RSA encryption is believed to be related to the difficulty of factoring the product of two large prime numbers.

SUID See UID.

superuser The root user.

symmetric keys A pair of identical keys used to encrypt and decrypt data. See also private key. Compare asymmetric keys.

TGT See ticket-granting ticket (TGT).

ticket A credential that a user can use to prove their identity. See also Kerberos ticket; authentication; identification.

ticket-granting server In Kerberos, the server that issues a ticket when presented with a ticket-granting ticket (TGT). See also key distribution center (KDC).

ticket-granting ticket (TGT) In Kerberos, a credential presented to the ticket-granting server in order to obtain a ticket. The ticket can then be used to gain access to a secure server. The use of TGTs and tickets enable the *single signon* feature, whereby the user need authenticate only once, after which they can access additional services without reauthenticating (by reentering their password, for example). See also authentication; identification.

time-based authentication A form of shared secret authentication in which the secret is changed periodically in a way known only to the two parties involved.

TLS See Transport Layer Security (TLS).

Transport Layer Security (TLS) A protocol that provides secure communication over a TCP/IP connection such as the Internet. It uses digital certificates for authentication and digital signatures to ensure message integrity, and can use public key cryptography 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. Although the TLS and SSL protocols are not interoperable, Secure Transport can back down to SSL 3.0 if a TLS session cannot be negotiated.

trust See level of trust.

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

UID Abbreviation for user ID. In BSD, the UID is a unique attribute of a user account that is used to identify the user. Each file system object and each process has an associated UID. See also file UID; GID; UUID.

URL Loading System An API that you can use to access the contents of http://, https://, and ftp:// URLs. Because https:// websites use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to protect data transfers, you can use the URL Loading System as a secure transport API. The URL Loading System is layered on top of CFNetwork.

UUID Abbreviation for Universally Unique Identifier. A type of UID or GID that is unique across all systems and all networks.

WebDAV Acronym for Web-based Distributed Authoring and Versioning. An extension of HTTP that allows collaborative file management on the web.

wheel group In BSD, a special group, membership in which confers on users the ability to become the root user by using the `su` utility on the command line. Users who are not in the wheel group can't become the root user, even if they have the correct password. In Mac OS X, starting with version 10.3, the admin group is used for this purpose rather than the wheel group.

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).

Index

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

X