EILEEN M. DECKER
United States Attorney
PATRICIA A. DONAHUE
Assistant United States Attorney
Chief, National Security Division
TRACY L. WILKISON (California Bar No. 184948)
Chief, Cyber and Intellectual Property Crimes Section
Assistant United States Attorney
1500 United States Courthouse
312 North Spring Street
Los Angeles, California 90012
Telephone: (213) 894-2400
Facsimile: (213) 894-8601
Email: Tracy.Wilkison@usdoj.gov

Attorneys for Applicant
UNITED STATES OF AMERICA

UNITED STATES DISTRICT COURT

FOR THE CENTRAL DISTRICT OF CALIFORNIA

IN THE MATTER OF THE SEARCH OF AN APPLE IPHONE SEIZED DURING THE EXECUTION OF A SEARCH WARRANT ON A BLACK LEXUS IS300, CALIFORNIA LICENSE PLATE #5KGD203	ED No. CM 16-10 (SP) DECLARATION OF STACEY PERINO IN SUPPORT OF GOVERNMENT'S REPLY IN SUPPORT OF MOTION TO COMPEL AND OPPOSITION TO APPLE INC.'S MOTION TO VACATE ORDER; EXHIBITS 17-30 Hearing Date: March 22, 2016 Hearing Time: 1:00 p.m. Location: Courtroom of the Hon. Sheri Pym

DECLARATION OF STACEY PERINO

I, Stacey Perino, declare as follows:

1. I am an Electronics Engineer with the Federal Bureau of Investigation ("FBI"). I have knowledge of the facts set forth herein and could and would testify to those facts fully and truthfully if called and sworn as a witness.

2. I received a Bachelor of Science in Mechanical Engineering from Colorado State University in 1991. I received a Bachelor of Science Degree in Electrical Engineering from the University of Colorado in 1996. I have been employed as an Electronics Engineer with the Federal Bureau of Investigation since 1996. From 1996 to 2001, I was an electrical engineer in the FBI's Cryptologic and Electronic Analysis Unit ("CEAU") developing both hardware and software solutions to recover data from electronic devices. From 2001 to 2009, I was the Program Manager for the Embedded Engineering Program within that same unit in the FBI. During this time I managed the technical efforts for a team of electrical engineers, computer engineers and computer scientists, composed of both government and contractor personnel with a focus on the recovery and presentation of data from electronic devices. In 2009, I became the Technical Director of the CEAU, a position I still hold.

3. This declaration is made in support of an application seeking an order from the Court compelling Apple Inc. ("Apple") to assist the FBI in its effort to search a cellular telephone, Apple make: iPhone 5C, Model: A1532, P/N:MGFG2LL/A, S/N:FFMNQ3MTG2DJ, IMEI:358820052301412, on the Verizon Network ("Subject Device").

4. In addition to relying on my own education, training, and experience, in preparing this declaration, I have reviewed the following:

a. The Declarations of Christopher Pluhar dated February 16, 2016 ("Initial Pluhar Declaration") and March 9, 2016, the Application filed in Case No. 16-

10 in the Central District of California, and the Court's Order in the same case calling for a software image file or "SIF" to be prepared by Apple (the "Order").

b. The Declaration of Erik Neuenschwander dated February 25, 2016 ("Neuenschwander Declaration").

c. Apple's "iOS Security" for iOS 9.0 or later dated September 2015 ("iOS Security"), attached to the Declaration of Nicola T. Hanna as Exhibit K.

d. Documentation from the website of the information technology company Sogeti, attached hereto as Exhibit 17, available at http://esec-lab.sogeti.com/static/publications/11-hitbamsterdam-iphonedataprotection.pdf.

e. The repository of code stored at https://code.google.com/archive/p/iphone-dataprotection, described as "ios forensics tools," and "Tools and information on iOS 3/4/5/6/7 data protection features."

f. Cellebrite Physical Extraction Manual for iPhone & iPad (Rev 1.3), attached hereto as Exhibit 18.

g. Apple's "Cryptographic Services," attached hereto as Exhibit 19, available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/Security_Overview/CryptographicServices/CryptographicServices.html.

h. Materials from Apple's "Code Signing Guide":

i. Exhibit 20, "About Code Signing," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.

ii. Exhibit 21, "Code Signing Overview," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/AboutCS/AboutCS.html.

iii. Exhibit 22, "Code Signing Tasks," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html.

iv. Exhibit 23, "Code Signing Requirement Language," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/RequirementLang/RequirementLang.html.

i. Materials from Apple's "Cryptographic Services Guide":

i. Exhibit 24, "About Cryptographic Services," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/Introduction/Introduction.html.

ii. Exhibit 25, "Cryptography Concepts In Depth," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/CryptographyConcepts/CryptographyConcepts.html.

iii. Exhibit 26, "Encrypting and Hashing Data," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html.

iv. Exhibit 27, "Managing Keys, Certificates, and Passwords," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/KeyManagementAPIs/KeyManagementAPIs.html.

v. Exhibit 28, "Glossary," available at https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/Glossary/Glossary.html.

j. Apple's "Unauthorized Modification of iOS Can Cause Security Vulnerabilities, Instability, Shortened Battery Life, and Other Issues," attached hereto as Exhibit 29, and available at https://support.apple.com/en-us/HT201954.

k. Apple's "Code Signing," attached hereto as Exhibit 30, and available at https://developer.apple.com/support/code-signing/.

5. This Declaration relies on Apple's publicly disseminated descriptions of how its own devices, operating system, security features, and software operate. Apple's source code is not, however, publicly available. Therefore the descriptions below do not

rely on my having reviewed Apple's source code, rather they rely upon Apple's own description of its devices, operating system, security features, and software, as well as on my training and experience in both observing and/or conducting the tests described in this document, directing the CEAU embedded engineering analysis of Apple devices and software, and reviewing other open source materials describing Apple mobile device technologies.

A. Purpose of this Declaration

6. In this declaration, I discuss the following topics:

a. The SIF called for in the Order could run *only* on the Subject Device. To explain this, I first provide some background on public key cryptography (Part B.1) and Apple's use of it and code signing to prevent the use of unauthorized code on its products (Part B.2). The Order provides that the SIF would only run on the Subject Device. Apple already requires that iOS updates include a unique device identifier for the Subject Device (Part B.3). Because an iPhone requires Apple to have cryptographically "signed" code before an iPhone will run it, and changing a unique device identifier within the SIF would invalidate Apple's signature, the SIF would not run on other iPhones. (Part B.3.)

b. The SIF called for by the Court's Order would perform functions that already exist in open source software for older devices and operating systems. In other words, code already exists that will bypass the auto-erase and time-delay functions and permit electronic submission of passcodes, but would need to be updated and modified for newer operating systems. (Part C.) That software, however, cannot run on the Subject Device without Apple's "signature."

c. The data contained on the Subject Device can be decrypted *only* on the Subject Device. This is because the encryption key includes a unique identifier that exists only on the Subject Device. (Part D.) Because the decryption must occur on the Subject Device, and because only Apple-signed software can run on the Subject Device

(Part B.2), any code or software tools needed to assist in testing passcodes (even code that includes components that already exist, Part C) must be signed by Apple.

d. Because the Subject Device was powered off when it was seized, it was not possible for it to back itself up to iCloud without the passcode. (Part E.)

B. The SIF Called for by the Order Would Run Only on the Subject Device

1. General Background on Public Key Cryptography

7. Generally, encryption and decryption are the processes of first converting intelligible "plaintext" into unintelligible "ciphertext," and second converting the ciphertext back into plaintext, respectively.

8. While encryption is designed to protect the confidentiality of information, a separate issue that arises in cryptology is authentication. Public key cryptography provides a method to both send messages securely, even when using a non-secure channel, and to validate the messages that are received. A properly implemented cryptographic signature gives the receiver reason to believe the message was sent by the person claiming to be the sender. A cryptographic signature also prevents modification of the original message by anyone other than the signer.

9. Public key encryption uses a complex operation that involves two different keys, a public key and a private key. A public key cryptosystem uses one key to encrypt (or to sign) a message and a different key to decrypt (or verify) the same message. (For this reason it is also referred to as asymmetric.) One of the essential properties of a public key cryptosystem is that it is too difficult—computationally infeasible—to determine a person's private key knowing only that person's public key.

10. The public key is made globally available while the private key is kept confidential. This allows anyone who is a member of the system to use the "phone book" of public keys to send a private message to any other member using the recipient's public key, but it allows only the recipient to open it using that person's private key.

Each key pair is unique to an individual member of a properly implemented cryptosystem.

11. One of the other essential properties of a public key cryptosystem is that the encryption operation and the decryption operation used in the cryptosystem are inverse operations.[1] This means that if one started with a message, it would not matter if one used the encryption operation followed by the decryption operation, or the decryption operation followed by the encryption operation, either would yield the original message again.[2]

12. A more detailed example of how the public key cryptosystem works to sign a message is as follows:

a. Alice generates a public-private key pair, and publishes her public key.

b. Alice composes a Message to Bob, and uses her private key to compute or generate the Signature. (This is represented: Signature = D_{pri}(Message) , where D is the decryption operation.)

c. Alice sends Bob both the Message and the Signature.

d. Bob then uses Alice's public key to verify that the message was signed using her private key. Bob does this by running the inverse "encryption" operation on the Signature. (This is represented: E_{pub}(Signature) = Message, where E is the encryption operation.) If the result of that operation is the Message that Alice sent Bob, then Bob knows the message is not a forgery and came from Alice.

[1] This is represented as follows, where E() and D() denote the encryption and decryption operations, and M is the text of the message: M = E(D(M)) = D(E(M)).

[2] (*See* Ex. 19 at 2, diagram (Apple developer website, Cryptographic Services). *See generally* Ex. 25 at 5, 7 (Apple developer website, Cryptographic Concepts in Depth).)

e. In this example, Alice could also have encrypted the message using Bob's public key. Bob could then have decrypted the message using Bob's own private key.

2. Apple's Use of Public Key Encryption to Prevent the Use of Unauthorized Code on Its Products

13. Just as a cryptosystem can be used to "sign" messages, it can be used to "sign" executable code.[3] Specifically, a vendor can embed a public key into a device such that the public key cannot be altered. For any and all executable code modules, the vendor uses its private key to calculate and attach a signature. As the device loads code modules for execution, the device uses the embedded public key to calculate the signature and thus verify the module's integrity and authenticity. As long as the public key cryptosystem is unbroken and the embedded key cannot be modified within the device, the scheme guarantees that only code issued by the vendor (that has been cryptographically signed) will run on the device.

14. Apple implements this system to require that its devices use software that only Apple authorizes. Apple does this by programming the public key into Read Only Memory ("ROM"). ROM is hardwired during the manufacture of the semiconductor device and cannot be changed later through any software means. The firmware in ROM is the first code that executes on the processor when power is applied. According to Apple's Security documentation, Apple products have also stored "the Apple Root CA [certificate authority] public key" within boot ROM.[4] (iOS Security at 5.) The boot ROM code uses the public key to verify that the next code to load (which is stored in

---

[3] In simplified terms, software is generally written by programmers in "source code." That source code is converted (or "compiled") into what is referred to as "executable code" that is in a format that a computer processer can understand and "execute."

[4] Boot ROM is firmware that has been fused or hardwired into the processor during manufacturing. It cannot be changed.

memory outside the processor) has also been signed with Apple's private key. (iOS Security at 5.)

15. This system ensures that Apple controls all code loaded and run on the device from the initial power-on. Apple describes how it has implemented this process in what it refers to as its "Chain of Trust" on pages 5-10 of its iOS Security document, wherein each sequential step needed to boot up the operating system and run application software relies on—and requires—Apple's signature. Specific details include the following:

a. "Each step of the startup process contains components that are cryptographically signed by Apple to ensure integrity and that proceed only after verifying the chain of trust. This includes the bootloaders, kernel, kernel extensions, and baseband firmware." (*Id.* at 5.)[5]

b. "The Boot ROM code contains the Apple Root CA [certificate authority] public key, which is used to verify that the Low-Level Bootloader (LLB) is signed by Apple before allowing it to load. This is the first step in the chain of trust where each step ensures that the next is signed by Apple. When the LLB finishes its tasks, it verifies and runs the next-stage bootloader, iBoot, which in turn verifies and runs the iOS kernel." (*Id.*) A certificate authority is the entity that issues digital

[5] A bootloader is the initial code run on a processor that starts the system's hardware components and peripherals and prepares the hardware for the operating system or higher level code. There may be multiple bootloaders that are executed sequentially at startup. The kernel is the first part of an operating system that loads and is responsible for controlling access to the computer's hardware resources. The kernel generally runs in protected memory to which other parts of the operating system and application code cannot directly read or write. Kernel extensions provide a method for adding or changing functionality of Apple's kernel without recompiling/relinking the source code. A mobile device typically has multiple processors; the application processor running an operating system, such as iOS 9.02, with which the user interacts (via the screen and keyboard), and the baseband processor which handles network communications traffic and protocols. The application processor is responsible for starting (booting) the baseband processor. Therefore, the application processor provides the baseband processor with the code it needs to load and run. Thus, Apple's chain of trust calls for each of these steps to be verified, ensuring that the next steps are authorized by Apple before allowing them to run or execute.

certificates. Certificate authorities create the public/private key pairs, and are responsible for ensuring the security of the private key. Apple has built its own certificate authority and has created its own public/private key pair used in the iPhone. As noted above, the public key is permanently programmed into the ROM of the iPhone, while the private key is controlled and protected by Apple. Because only Apple possesses its private key, only Apple is able to sign software that will be loaded on its devices. By keeping the private key secret, Apple ensures that only software signed by Apple using its private key can be loaded on its devices during the boot process.

c. "This secure boot chain helps ensure that the lowest levels of software are not tampered with and allows iOS to run only on validated Apple devices." (*Id.*) "From initial boot-up to iOS software updates to third-party apps, each step is analyzed and vetted to help ensure that the hardware and software are performing optimally together and using resources properly." (*Id.*)

d. "This architecture is central to security in iOS, and never gets in the way of device usability. The tight integration of hardware and software on iOS devices ensures that each component of the system is trusted, and validates the system as a whole." (*Id.*)

16. "The startup process described above helps ensure that only Apple-signed code can be installed on a device." (*Id.* at 6.) If any component can be made to load code not signed by Apple, the chain of trust is broken. By beginning their chain of trust with the initial code and public key programmed into the device ROM, Apple has made it extremely difficult for anyone to defeat the chain of trust.

17. As a result of these features, an Apple iPhone is designed to only run code (the operating system and the many pieces of firmware and software that may operate within it) that are signed using Apple's keys.

3. Apple "Signs" iOS Updates for Its iPhones that Include a Unique Device Identifier, Ensuring It Only Works on One iPhone

18. While the features described above permit Apple to ensure that the devices it manufactures will use only an operating system or software that Apple has authorized (by signing it), Apple also relies on them to ensure that an operating system will work only on one specific Apple device. Specifically, during an iOS update, recovery, or Device Firmware Update (DFU) process, the device verifies that the code being loaded to it was digitally signed specifically for that device, and not for another device. This feature, enforced by the hardware-based chain of trust, allows Apple to ensure that any code loaded to the phone will only operate on a specific device.

19. Apple implements this process in the following manner. First, the device connects to a computer, for example through iTunes, and provides iTunes with unique information about itself—both its hardware and software. Second, iTunes sends this information from the device to an Apple server that builds the package of code needed to update or recover that device, packages it with the same unique information about the device, and returns it to the computer running iTunes. Third, upon receiving that package from the computer running iTunes, the device is required to read and recognize its own unique information before installing the operating system.

20. Details of this process are as follows:

a. Apple maintains what it refers to as "the Apple installation authorization server," which is referred to herein as the "Installation Server." (iOS Security at 6.)

b. Whenever a device tries to upgrade its version of iOS, through the upgrade or recovery process, the device must first send to that server a set of information from the device. The information sent by the device includes "cryptographic measurements for each part of the bundle to be installed (for example, LLB, iBoot, the kernel, and OS image)." (*Id.*) Those measurements are a digest or partial digest of that component. (A digest can be a cryptographic hash, or the result of a similar algorithm

that generates a unique value, akin to a digital fingerprint, after it processes each part of the bundle.)[6] The device also sends a "nonce," or a random, one-time-use value.

c. Most importantly for ensuring the "personalization" of the software for use on a specific device, the device also sends "the device's unique ID (ECID)." (iOS Security at 6.) The ECID is a unique, device-specific identifier programmed into the phone hardware during manufacture. (*Id.* at 58 (defining ECID as "[a] 64-bit identifier that's unique to the processor in each iOS device. Used as part of the personalization process, it's not considered a secret").) Apple explains the use of these values in their iOS Security document. "These steps ensure that the authorization is for a specific device and that an old iOS version from one device can't be copied to another." (*Id.* at 6.)

d. Once the Apple Installation Server receives this information from the device, it builds a software package and digitally signs it using a private key that is not known to the public. The digital signature includes the ECID, nonce, and other cryptographic measurements in the signed data. Once the device receives the package, the device verifies from the signed data that the package is meant for it.

e. The device is also able to tell that the installation is current and is not a repeat of an older installation (which would result in a "downgrade" of the operating system). The device does so by checking the random, one-time nonce it had sent to the server was the one returned by the server in the signed package. "The nonce prevents an

[6] "In cryptography, hashes are used when verifying the authenticity of a piece of data. Cryptographic hashing algorithms are essentially a form of (extremely) lossy data compression, but they are specifically designed so that two similar pieces of data are unlikely to hash to the same value. . . . . With good hashing algorithms, collisions [messages that hash to the same value] are unlikely if you make small changes to a piece of data. This tamper-resistant nature of good hashes makes them a key component in code signing, message signing, and various other tamper detection schemes." (Ex. 19 at 3 (Apple developer website, Cryptographic Services).) By way of background, data compression that is "lossy" loses some qualities of the original data, such as when a compressed digital image loses resolution or appears "pixelated." In the cryptography context, what is important is that the resulting hash value is unique, not that it be capable of reformulating the entire original piece of data, hence it "loses" data by being reduced to a small but unique string of letters and numbers.

attacker from saving the server's response and using it to tamper with a device or otherwise alter the system software." (*Id.* at 6.)

21. The digital signature prevents any part of the returned package from being changed. If the software in the returned package is altered, the digital signature check will fail and the device will not load it. If the ECID is changed to that of another device, the signature check will fail and the device will not load the code.[7] In other words, unless someone can bypass the digital signature verification, allowing them to load unsigned code, the software cannot be changed to operate on a different device or perform a different function.[8]

22. The Order provides that the SIF would only run on the Subject Device. As shown in the preceding description of Apple's normal code signing process during an iOS update, Apple already has a mechanism in place to do this by including the ECID into the digital signature process. If this same or a similar process were used, the SIF could incorporate the ECID of the Subject Device, and then be signed by Apple. In that case, if the ECID of the SIF were changed to the ECID of another device, the signature check would fail and an Apple device would not load the code.[9]

---

[7] As described on Apple's developer website: "When a piece of code has been signed, it is possible to determine reliably whether the code has been modified by someone other than the signer." (Ex. 21 at 1 (Apple developer website, Code Signing Overview).) Among the purposes of code signing are to "ensure that a piece of code has not been altered," and to "identify code as coming from a specific source (a developer or signer)." (*Id.*)

[8] Because of the significance of the ability to digitally sign code and therefore cryptographically authenticate it, Apple's developer website explains that a "signing identity, no matter how obtained, is completely compromised if it is ever out of the physical control of whoever is authorized to sign code." (Ex. 22 at 2 (Apple developer website, Code Signing Tasks).)

[9] An additional measure to ensure the SIF would only run on the Subject Device could be to program the Subject Device's ECID directly into the software running in the SIF. In this scenario, the SIF would read the ECID of the device on which it was running, and compare that to the ECID of the Subject Device that had been programmed into it; if the two did not match, the software would exit. In other words, while the iOS update scenario described in this Part relies on the *device's* refusal to run the code without a valid Apple signature (which signature would be invalid by changing the ECID), the *SIF* could refuse to fully execute if it did not detect the Subject Device's

(footnote cont'd on next page)

23. For these reasons, the SIF called for by the Order would be permitted to run only on the Subject Device. In other words, the creation of the SIF, tailored and signed with the unique identifier of the Subject Device, would not undermine the security of other iPhones that also require Apple-signed code, because each iPhone has its own unique identifier. The SIF proposed by the Order would therefore not break Apple's chain of trust on its iPhones, or even on the Subject Device; Apple's assistance will keep the chain of trust intact.

24. Importantly, if somebody were to bypass the Apple digital signature process, the chain of trust would be broken. Causing an Apple device to allow itself to run software not signed by Apple is referred to as "jailbreaking" the device. Jailbreaks result from bugs or errors in different programs that can be exploited to run unsigned code on a device. To my knowledge, for the iPhone 5C, jailbreaks have been exclusively performed from a powered-on phone on which the passcode has been entered and the phone unlocked. Thus there are currently no published jailbreaks for an iPhone 5C where the passcode has not been entered at least once since powering on, and hence there are none that could be applied to the Subject Device.

C. Software Already Exists that Performs Similar Functions as the SIF

25. The security features created and implemented by Apple that are described above were challenged by researchers and hackers as previous iterations of iOS were released. Apple's current chain of trust structure has fixed previous issues, but the methods that have been published and used to test earlier versions of iPhones illustrate why the components used in the SIF already exist, and why it, like other previous tools, can be operated from random access memory ("RAM").

26. Paragraph 19 of the Neuenschwander Declaration states that Apple's "current iPhone operating systems designed for consumer interaction do not run in

---

ECID. This example is designed to illustrate that there is more than one way to cause the SIF to only load and execute on the Subject Device.

RAM, but are installed on the device itself. To make them run in RAM, Apple would have to make substantial reductions in the size and complexity of the code." As the discussion below illustrates, the SIF would not be designed for "consumer interaction." Rather, the SIF would be designed only to test passcodes, and other similar tools that have previously been used for this purpose do run in RAM.

27. Those previous tools that are available cannot be used on the Subject Device because they are not signed by Apple, and the current chain of trust on the Subject Device requires Apple to have signed any software that will be allowed to run.

28. A more detailed description is as follows:

a. A previous bug allowed a cold-booted[10] iPhone to load a "minimal" operating system in memory (RAMdisk) that had not been signed by Apple. Previously, Apple iPhone versions 3GS and 4 contained a bug in the Apple boot ROM that allowed unsigned code to be loaded and run through Recovery or DFU mode. This vulnerability was published as the "limera1n" exploit. Other researchers analyzed the Apple boot process and published details of it, including the composition of the RAMdisk (*i.e.*, which software components were bundled into the RAMdisk) used in the Recovery mode and DFU mode process to update device firmware.

b. A passcode-recovery tool has already been developed that uses brute-force techniques. The information technology company Sogeti[11] analyzed Apple's encryption process demonstrating that any passcode "guessing" had to be performed by code running on the device and could not be done externally (further explained below in Part D). Other vulnerability researchers used this result to develop software that could brute force the passcode on a jailbroken device (iphone-dataprotection project[12]).

---

[10] Cold-boot refers to a phone that has been powered off and then powered back on but no passcode has been entered.

[11] (Ex. 17 (http://esec-lab.sogeti.com/static/publications/11-hitbamsterdam-iphonedataprotection.pdf).)

[12] (https://code.google.com/archive/p/iphone-dataprotection, "ios forensics tools," and "Tools and information on iOS 3/4/5/6/7 data protection features.")

c. From this open source research, several forensic tools were developed that combined (1) the boot ROM code signing defeat, and (2) brute-force passcode guessing. Examples include the Cellebrite UFED tool and an FBI-developed tool. Both the Cellebrite[13] and FBI tools utilize the boot ROM exploit, allowing iPhone 3GS and iPhone 4 devices to load and boot an unsigned RAMdisk containing code to brute force the device passcode. The passcode recovery process operated from RAM, and did not alter the system or user data area. The passcode recovery software did not require user interaction, and the entire process ran without use of the "Springboard" graphical user interface. Because these forensic tools ran from a RAMdisk and did not use the operating system that was stored on the device, these tools did not incur time delays or the auto-erase function (which are features implemented by the operating system installed on the device).

d. Apple addressed the bug, and subsequently a jailbreak (*i.e.*, allowing code unsigned by Apple) could only occur on an iPhone after it had been booted and unlocked. As described previously, a jailbroken phone is one that has had the chain of trust broken and can run unsigned code.[14] After Apple corrected the bug present in the

[13] Cellebrite is a private company that makes forensic data recovery tools for mobile devices. While I have not examined the source code for the UFED tool, based on the Cellebrite Physical Extraction Manual for iPhone and iPad (Rev 1.3) and the fact that the Cellebrite tool no longer supports iPhone 4S and later devices, I believe the UFED tool relied on the same ROM exploit. The manual states: "The extraction application does not load iOS but instead loads a special forensic utility to the device. This utility is loaded to the device's memory (RAM) and runs directly from there." The utility is loaded from recovery mode.

[14] The use of jailbroken phones discussed in this Part occurred in a testing environment. Outside of a testing environment, some users have jailbroken their phones to try to use software or services that Apple has not authorized, but Apple cautions that doing so presents "[s]ecurity vulnerabilities": "Jailbreaking your device eliminates security layers designed to protect your personal information and your iOS device. With this security removed from your iOS device, hackers may steal your personal information, damage your device, attack your network, or introduce malware, spyware or viruses." (Ex. 29 (https://support.apple.com/en-us/HT201954).) Furthermore, the jailbreaking process often results in deletion or alteration of data stored on the phone. As discussed in this Part, software already exists that performs certain functions that could be used in the SIF, and to the extent those software components could be used to undermine security, they (like the SIF) would only work on devices that had already assumed security vulnerabilities by being jailbroken.

iPhone 3GS and 4, all known jailbreaks have been applied from within the iPhone user interface, instead of during the boot process. There are publicly known jailbreaks for most recent iPhone OS versions (up to at least version iOS 9.0.2), but they can only be executed from an unlocked iPhone via the user interface, *i.e.*, after the iPhone had booted and had been unlocked. After these jailbreaks are applied, software that has not been signed by Apple may be run.

e. The same brute-force source code still works on jailbroken iPhones. A software project named "iphone-dataprotection" includes a passcode recovery program that can still be compiled, loaded, and run within a jail-broken Apple device. The FBI tool used essentially the same functionality as this project but executed it from a RAMdisk. The FBI recently tested the iphone-dataprotection passcode recovery software on a jailbroken iPhone 6 Plus running iOS 8.4 (in which the passcode had been entered once). With minor modifications this software still functioned and was able to recover the passcode without incurring time delays. The FBI also tested this passcode recovery software on a jailbroken iPad Air 2 running iOS 9.02. In this device the passcode recovery software functioned, but it did incur the time delays and most likely would have erased the device.[15] However, this test does verify that the passcode recovery code works, which has existed for many years and still functions essentially the same. This specific code would not run on the Subject Device "as is," because it is not signed by Apple and also because it would incur time delays and risk causing the device to erase, which would require further development and modifications to the kernel software.[16]

---

[15] It should be noted that the iPhone 6 and iPad Air 2 both use the more advanced A8 processor and the time delay and erase functionality has moved into a separate security controller called the Secure Enclave.

[16] For example, in previous versions of iOS the time delay and password try count resided in the "springboard" user interface, which is in part what allowed the passcode recovery software to work and to bypass the time-delay and auto-wipe features. In approximately iOS 8.4, that functionality moved from the Springboard and would require further modification to bypass the delay and wipe functions.

f. Only Apple can produce and sign the RAMdisk needed to run the passcode guessing code without first unlocking the iPhone. Beginning with the release of the iPhone 4S in 2011, Apple fixed the bug in the boot ROM. Since that time, the Apple chain of trust—which governs the boot process on an iPhone—has remained intact, preventing loading of unsigned RAMdisks. (The jailbreaks that have occurred on iPhones 5C or later have occurred after the boot-up process has occurred, and after a passcode has been entered; the chain of trust through the boot-up process remains intact on those phones.) However, the steps used in the Apple Recovery and DFU mode boot processes have not changed substantially since that time, and Apple's use of a RAMdisk to perform the updates and device recovery processes appear consistent with the methodology of the earlier devices. Without assistance from Apple to digitally sign the code, however, it has not been possible to continue development of these tools for newer devices. The passcode-guessing software employed by these tools has been tested within jailbroken devices running an iOS that has already been booted and unlocked; neither the FBI, nor others to my knowledge, however, have been able to integrate the software into a RAMdisk to test passcodes from a cold-booted iPhone device since the iPhone 4.

29. As set forth above in the previous paragraph, there are already software components available that perform some of the functions of the SIF called for by the Court's Order. Although code similar to what would be in the SIF already exists, it cannot be used on the Subject Device without Apple's signature because of Apple's robust security and code-signing practices.

D. The Encrypted Data on the Subject Device Must Be Decrypted on the Subject Device Itself

30. As described in paragraph 12 of the Initial Pluhar Declaration, an iPhone 5C running iOS 9 is encrypted using a combination of two components: one user-determined passcode, and one unique 256-bit key (referred to as a "UID") fused into the

phone itself during manufacture. (iOS Security at 12; *id.* 11 (diagram); Neuenschwander Decl. ¶ 13.) These two different components are discussed below.

31. According to Apple's documentation, the UID is unique to each device, is fused into the hardware, and is not known to Apple or anyone else, as described on page 10 of iOS Security:

> The device's unique ID (UID) . . . [is] fused . . . into the application processor and Secure Enclave during manufacturing. No software or firmware can read them directly . . . . The UIDs are unique to each device and are not recorded by Apple or any of its suppliers. . . . The UID allows data to be cryptographically tied to a particular device. For example, the key hierarchy protecting the file system includes the UID, so if the memory chips are physically moved from one device to another, the files are inaccessible. The UID is not related to any other identifier on the device.

32. I know from Supervisory Special Agent ("SSA") Pluhar that the Subject Device was powered off when the FBI found it. When the Subject Device was powered on, it displays a numerical keypad (like that on a telephone), and a prompts for four numbers to be entered.

33. With a four-digit numerical pin, there are only 10,000 possible passcodes. Testing 10,000 passcodes electronically would likely take less than a day, depending on how the SIF were configured.

34. Apple's iOS Security also explains that because its passcodes are permitted to be weak in that they can be only four numbers, Apple has included additional features to discourage brute-force attacks. These features are described in paragraphs 13 and 14 of the Initial Pluhar Declaration, and on page 12 of Apple's iOS Security (noting that iOS 9 iPhones (1) escalate time delays between failed passcodes, and can (2) be configured to wipe their contents after ten failed passcodes, to "discourage brute-force passcode attacks").

35. The UID is itself a strong encryption key. It is fused into the hardware and is both unknowable and unchangeable: it is always used the same way to create the encryption key. The only variable is the passcode.

36. Because both the UID, which is unique and embedded in the device itself, is a part of the encryption key (along with the user-generated passcode), the data that is stored on the Subject Device will need to be decrypted on the Subject Device. Because only Apple-signed software can run on the iPhone, and the decryption must occur on the Subject Device, any code or software tools needed to assist in testing passcodes must be signed using Apple's encryption keys.

E. Apple's iCloud Backup

37. I know from SSA Pluhar that the Subject Device was found in a powered-off state. Based on Apple's published documentation, open source research relating to Apple's encryption, and Apple press releases about iOS 8 and later encryption, I believe that (1) the device would not connect to a WiFi network until the passcode was entered, and (2) even if the device could be forced to perform an iCloud backup, the user data would still be encrypted with the encryption key formed from the 256 bit UID and the user's passcode.

38. Subsequent to seizing the Subject Device, the FBI performed several tests on exemplar phones to test whether a cold-booted iPhone could connect to a trusted WiFi network and perform a backup. The result of that testing was that cold-booted iPhones would not connect to a WiFi network.

a. To the best of my knowledge, a cold-booted iPhone will not connect to WiFi networks trusted by the Subject Device such as a home or work network until the passcode is entered. However, according to Apple and verified by the FBI, there are some WiFi networks inherently trusted by iOS, such as those operated by iPhone sponsors (referred to as carrier-sponsored WiFi). For example, an AT&T iPhone can automatically connect to an AT&T hotspot.

b. When the FBI tested a locked AT&T phone on which the passcode had been entered once by taking it to an area with an AT&T hotspot, the phone connected automatically to the hotspot, as indicated by the WiFi indicator on the top

banner of the lock screen display. Additionally the "Find My iPhone" service was used and was able to locate the iPhone, verifying that a phone in which the passcode has been entered will connect, even when screen-locked, to a trusted WiFi network.

c. The same test was also done with the phone first powered off and restarted, but with the passcode not having been entered. In this scenario, the test phone did not show any indication it was connected to the AT&T hotspot through the banner. Additionally, the "Find My iPhone" service was unable to locate the device. The results of these tests show that WiFi is not enabled on the device until after the passcode is entered.

d. Further tests were conducted by myself and a colleague in CEAU by taking an iPhone 5 running iOS 9.02 and an iPhone 6 Plus running iOS 9.2 into a radio-frequency shielded chamber to test their electronic emissions. Both iPhones were fully charged, connected to power and had their WiFi enabled. The same series of tests was done on both phones with identical results. When the iPhone was not protected by a passcode and was powered on in that chamber, it began to emit signals in the frequency band of 2.4 gigahertz (GHz), a common band for WiFi connections. This is consistent with the iPhone trying to detect a WiFi network. When the iPhone *was* protected by a passcode and was powered on in the same chamber without entering the passcode, no emissions in the 2.4 GHz frequency band were detected. This indicates that the WiFi was not active. When the passcode was entered, WiFi 2.4GHz emissions were detected. The phone was allowed to screen lock after the passcode had been entered. Again, 2.4GHz emissions were detected. Each phone was rebooted, no passcode entered, and left overnight in the chamber. No 2.4GHz signals were observed. These tests indicate the WiFi is not active on a cold-booted device until the passcode has been entered at least once.

e. The FBI does not know of any way to force an iPhone that has not had the passcode entered at least once since being powered on to perform an iCloud backup.

39. This result is consistent with Apple's security documentation, which states that data stored on the device is encrypted using a key that is a combination of both the UID (the device-specific unique identifier) and the passcode generated by the user. Unless the passcode is entered by the user, the entire encryption key could not be used to decrypt the data, and the data therefore could not be backed-up to an iCloud—at least in a state that could be recovered outside the device.

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct and that this declaration is executed at Virginia, on March 9, 2016.

Stacey Perino

STACEY PERINO

Exhibit 17

iPhone data protection in depth

Jean-Baptiste Bédrune
Jean Sigwald
Sogeti / ESEC
jean-baptiste.bedrune(at)sogeti.com
jean.sigwald(at)sogeti.com

SOGETI

Introduction

Motivation

- Mobile privacy is a growing concern
- iPhone under scrutiny
  - iPhoneTracker (O'Reilly)
  - "Lost iPhone? Lost Passwords!" (Fraunhofer)

Agenda

- iOS 4 data protection
- Storage encryption details
- iTunes backups

iPhone forensics

Trusted boot vulnerablities

- Chain of trust starting from BootROM
- BootROM runs USB DFU mode to allow bootstrapping of restore ramdisk
- Unsigned code execution exploits through DFU mode
  - Pwnage/steaks4uce/limera1n (dev team/pod2g/geohot)
  - All devices except iPad 2

Custom ramdisk techniques

- Zdziarski method, msft_guy ssh ramdisk
- Modify ramdisk image from regular firmware, add sshd and command line tools
- Boot (unsigned) ramdisk and kernel using DFU mode exploits
- Dump system/data partition over usb (usbmux)

iPhone crypto

Embedded AES keys

- UID key : unique for each device
- GID key : shared by all devices of the same model
  - Used to decrypt IMG3 firmware images (bootloaders, kernel)
  - Disabled once kernel boots
- IOAESAccelerator kernel extension
  - Requires kernel patch to use UID key from userland

UID key

- Encrypts static nonces at boot to generate unique device keys
  - `key0x835 = AES(UID, "01010101010101010101010101010101")`
  - `key0x89B = AES(UID, "183e99676bb03c546fa468f51c0cbd49")`
- Also used for passcode derivation in iOS 4

iOS 3.x data protection

Hardware Flash memory encryption

- Introduced with iPhone 3GS
- Allows fast remote wipe
- Data still accessible transparently from custom ramdisk

Keychain

- SQLite database for passwords, certificates and private keys
- Each table has an encrypted data column
- All items encrypted with key 0x835
- Format : IV + AES128(key835, data + SHA1(data), iv)

iOS 4

Data protection

- Set of features to protect user data
- Phone passcode used to protect master encryption keys
- Challenges for iOS 4 forensics :
  - Keychain encryption has changed
  - Some protected files cannot be recovered directly from custom ramdisk
  - Raw data partition image cannot be read with standard tools
  - New encrypted iTunes backup format

Our work

- Keychain tools
- Passcode bruteforce
- Data partition encryption scheme
- iTunes backup tools

Plan

Data protection

Objectives

- Protect data at rest (phone locked or powered off)
  - Limit impact from custom ramdisk attacks
- Encrypted data protected by user's passcode
  - Limit bruteforce attacks speed with custom passcode derivation function

Design

- Data availability
  - When unlocked
  - After first unlock
  - Always
- Protection Classes for files and keychain items
- Master keys for protection classes stored encrypted in a keybag
  - 3 keybag types : System, Escrow, Backup

Data protection

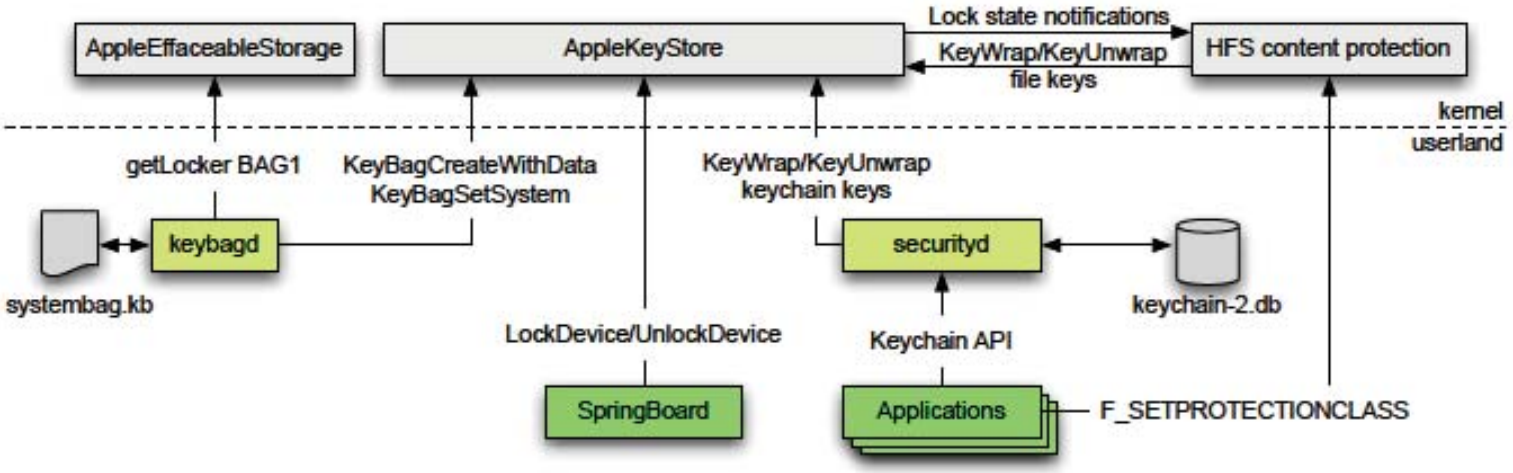
Protection classes

Availability	Filesystem	Keychain
When unlocked	NSProtectionComplete	WhenUnlocked
After first unlock		AfterFirstUnlock
Always	NSProtectionNone	Always

Implementation

- keybagd daemon
- AppleKeyStore kernel extension
  - MobileKeyBag private framework (IOKit user client)
- AppleKeyStore clients :
  - Keychain
  - HFS content protection (filesystem)

Data protection components & interactions



Keybagd

Description

- System daemon, loads system keybag into AppleKeyStore kernel service at boot
- Handles system keybag persistance and passcode changes

System keybag

- Stored in /private/var/keybags/systembag.kb
- Binary plist with encrypted payload
- Encryption key pulled from AppleEffaceableStorage kernel service
  - Stored in "BAG1" effaceable locker
- Tag-Length-Value payload

Keybag binary format

Example keybag hexdump

```
0000000: 4441 5441 0000 0444 5645 5253 0000 0004  DATA...DVERS....
0000010: 0000 0002 5459 5045 0000 0004 0000 0000  ....TYPE........
0000020: 5555 4944 0000 0010 ceea c20d cf52 40e0  UUID.........R@.
0000030: ac0e dd52 915d 38bc 484d 434b 0000 0028  ...R.]8.HMCK...(
0000040: 6785 4e94 bc50 f2e4 541b c51d 8f46 ad59  g.N..P..T....F.Y
0000050: 3af3 cdcb 201a 2e53 6424 b728 3775 788f  :... ..Sd$.(7ux.
0000060: cd2e 28f8 b692 2bac 5752 4150 0000 0004  ..(...+.WRAP....
0000070: 0000 0001 5341 4c54 0000 0014 8bda 11d7  ....SALT........
0000080: 43bb 669c e451 646c 2ea9 ac0b 6658 ff9d  C.f..Qdl....fX..
0000090: 4954 4552 0000 0004 0000 c350 5555 4944  ITER.......PUUID
00000a0: 0000 0010 02ed b2ea c187 49b2 b9f1 7925  ..........I...y%
00000b0: ddaa daae 434c 4153 0000 0004 0000 000b  ....CLAS........
00000c0: 5752 4150 0000 0004 0000 0001 5750 4b59  WRAP........WPKY
00000d0: 0000 0020 8f81 980c a483 2ae4 e978 4cc8  ... ......*..xL.
00000e0: f715 f4e3 44ac 71cc b568 22e6 e119 6983  ....D.q..h"...i.
00000f0: b156 e25e 5555 4944 0000 0010 d8e0 f7a2  .V.^UUID........
```

Keybag binary format

Header

- `VERS` : 1 or 2
  - Version 2 was introduced in iOS 4.3
  - Minor changes in passcode derivation function
- `TYPE`: Keybag type
  - 0 : System
  - 1 : Backup
  - 2 : Escrow
- `UUID, ITER, SALT, WRAP`
- `HMCK` : encrypted HMAC key for integrity check
- `SIGN = HMAC_SHA1(DATA, AES_UNWRAP(key835, HMCK))`
  - HMAC parameters inverted, DATA is the HMAC key (?!)

Keybag binary format

Wrapped class keys

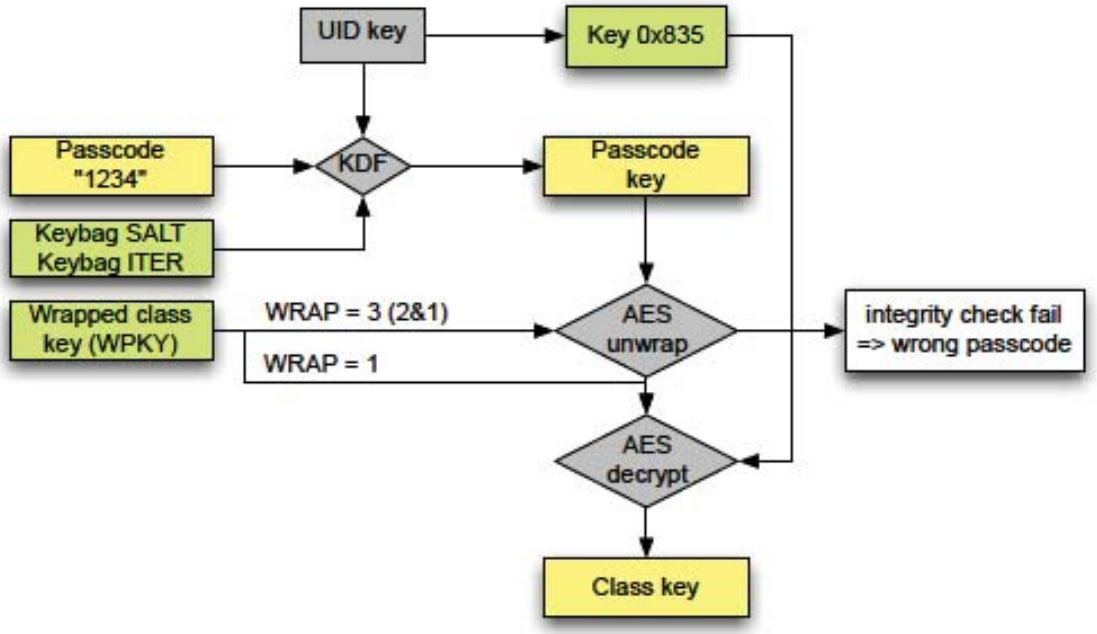
- `UUID` : Key uuid
- `CLAS` : Class number
- `WRAP` : Wrap flags
  - 1 : AES encrypted with key 0x835
  - 2 : AES wrapped with passcode key (RFC 3394)
- `WPKY` : Wrapped key

Class keys identifiers

Class keys

Id	Class name	Wrap
1	NSProtectionComplete	3
2	(NSFileProtectionWriteOnly)	3
3	(NSFileProtectionCompleteUntilUserAuthentication)	3
4	NSProtectionNone (stored in effaceable area)	x
5	unused ? (NSFileProtectionRecovery ?)	3
6	kSecAttrAccessibleWhenUnlocked	3
7	kSecAttrAccessibleAfterFirstUnlock	3
8	kSecAttrAccessibleAlways	1
9	kSecAttrAccessibleWhenUnlockedThisDeviceOnly	3
10	kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly	3
11	kSecAttrAccessibleAlwaysThisDeviceOnly	1

Keybag unlock



Escrow Keybags

Definition

- Copy of the system keybag, protected with random 32 byte passcode
- Stored off-device
- Escrow keybags passcodes stored on device
  - `/private/var/root/Library/Lockdown/escrow_records`

Usage

- iTunes, allows backup and synchronization without entering passcode
  - Device must have been paired (plugged in while unlocked) once
  - Stored in `%ALLUSERSPROFILE%\Apple\Lockdown`
- Mobile Device Management
  - Sent to MDM server during check-in, allows remote passcode change

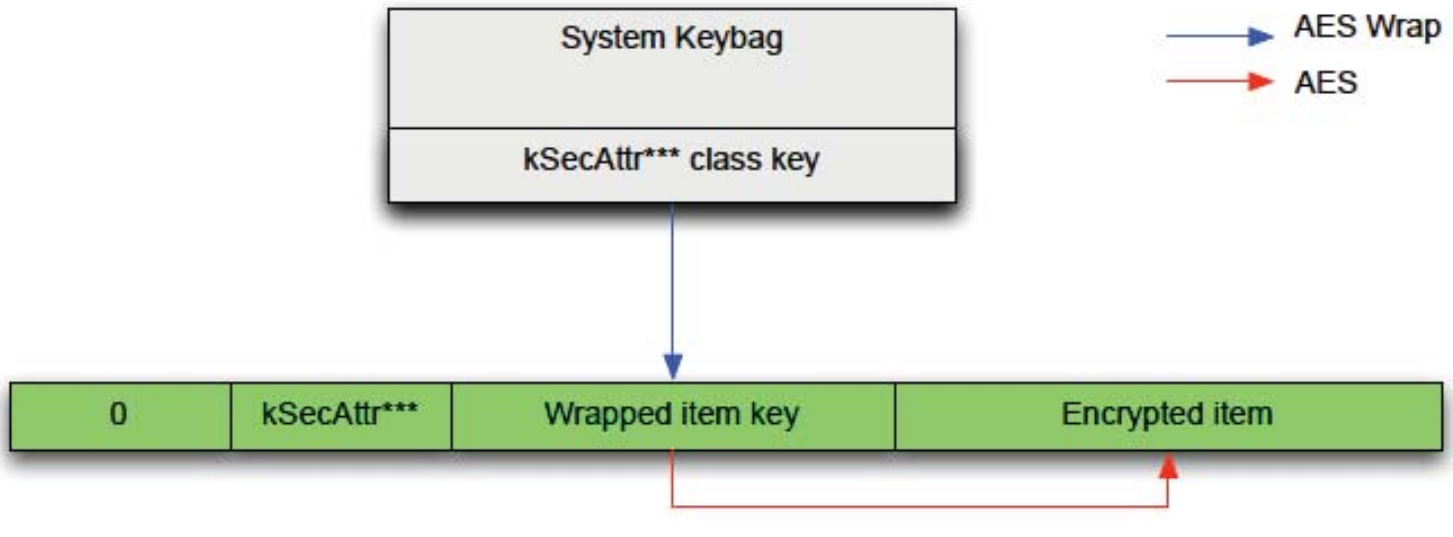
Keychain

Description

- SQLite database (`keychain-2.db`)
- 4 tables : genp, inet, cert, keys
- `securityd` daemon handles database access
- Keychain API : IPC calls to `securityd`
- Access control : access group from caller's entitlements (application identifier)
  - `WHERE agrp=...` clause appended to SQL statements
- On iOS 4, applications can specify a protection class (kSecAttrAccessible***) for their secrets
  - Each protection class has a `ThisDeviceOnly` variant
- Secrets encrypted with unique key, wrapped by class key

Keychain

Data column format



Keychain

Protection for build-in applications items

Item	Accessibility
Wi-Fi passwords	Always
IMAP/POP/SMTP accounts	AfterFirstUnlock
Exchange accounts	Always
VPN	Always
LDAP/CalDAV/CardDAV accounts	Always
iTunes backup password	WhenUnlockedThisDeviceOnly
Device certificate & private key	AlwaysThisDeviceOnly

Keychain Viewer

Description

- Graphical application for jailbroken devices
- Inspect Keychain items content and attributes
- Show items protection classes

Implementation

- Access `keychain-2.db` directly (read only)
- Calls AppleKeyStore KeyUnwrap selector to get items keys
  - Requires `com.apple.keystore.access-keychain-keys` entitlement
- Has to run as root (source code available)

Passcode derivation

Description

- AppleKeyStore exposes methods to unlock keybags
  - UnlockDevice, KeyBagUnlock
- Passcode derivation is done in kernel mode
- Transforms user's passcode into passcode key
- Uses hardware UID key to tie passcode key to the device
  - Makes bruteforce attacks less practical
- Resulting passcode key is used to unwrap class keys
  - If AES unwrap integrity check fails, then input passcode is wrong
- Bruteforce possible with unsigned code execution, just use the AppleKeyStore interface

Passcode derivation algorithm

Initialization

- A = A1 = PBKDF2(passcode, salt, iter=1, outputLength=32)

Derivation (390 iterations)

- XOR expand A to 4096 bytes
  - B = A $\oplus$ 1 | A $\oplus$ 2 | ...
  - Keybag V2 : B = A1 $\oplus$ counter++ | A1 $\oplus$ counter++ | ...
- AES encrypt with hardware UID key
  - $C = AES\_ENCRYPT\_UID(B)$: must be done on the target device
  - Last encrypted block is reused as IV for next round
- XOR A with AES output
  - A = A $\oplus$ C

Bruteforce attack

Using MobileKeyBag framework

```
//load and decrypt keybag payload from systembag.kb
CFDictionaryRef kbdict = AppleKeyStore_loadKeyBag("/mnt2/keybags",
                                                  "systembag");

CFDataRef kbkeys = CFDictionaryGetValue(kbdict, CFSTR("KeyBagKeys"));

//load keybag blob into AppleKeyStore kernel module
AppleKeyStoreKeyBagCreateWithData(kbkeys, &keybag_id);
AppleKeyStoreKeyBagSetSystem(keybag_id);

CFDataRef data = CFDataCreateWithBytesNoCopy(0, passcode, 4, NULL);
for(i=0; i < 10000; i++)
{
    sprintf(passcode, "%04d", i);
    if (!MKBUnlockDevice(data))
    {
            printf("Found passcode: %s\n", passcode);
            break;
    }
}
```

Bruteforce attack

Bruteforce speed

Device	Time to try 10000 passcodes
iPad 1	~16min
iPhone 4	~20min
iPhone 3GS	~30min

Implementation details

- MobileKeyBag framework does not export all the required functions (AppleKeyStore***)
  - Easy to re-implement
- No passcode set : system keybag protected with empty passcode
- Passcode "keyboard complexity" stored in configuration file
  - `/var/mobile/Library/ConfigurationProfiles/UserSettings.plist`

Bruteforce attack - Custom ramdisk

Ramdisk creation

- Extract restore ramdisk from any 4.x ipsw
- Add msft_guy sshd package (ssh.tar)
- Add bruteforce/key extractor tools

Ramdisk bootstrap

- Chronic dev team syringe injection tool (DFU mode exploits)
- Minimal cyanide payload patches kernel before booting
  - Patch IOAESAccelerator kext to allow UID key usage
  - Once passcode is found we can compute the passcode key from userland
- Same payload and ramdisk works on all A4 devices and iPhone 3GS

Bruteforce attack - Ramdisk tools

Custom restored daemon

- Initializes usbmux, disables watchdog
- Forks sshd
- Small plist-based RPC server
- Python scripts communicate with server over usbmux
- Plist output

Bruteforce attack - Ramdisk tools

Bruteforce

- Decrypt system keybag binary blob
- Load in AppleKeyStore kernel extension
- Try all 4-digit passcodes, if bruteforce succeeds :
  - Passcode, Passcode key (derivation funtion reimplemented)
  - Unwrapped class keys
  - Keychain can be decrypted offline
  - Protected files access through modified HFSExplorer
  - In-kernel keybag unlocked, protected files can also be retrieved directly using scp or sftp

Escrow keybags

- Get escrow keybag passcode from device
- Compute passcode key
- Get class keys without bruteforce

Plan

iPhone storage

Introduction

- iPhone 3GS and below use NOR + NAND memory
- Newer devices only use NAND (except iPad 1)
- NAND encryption done by DMA controller (CDMA)
- Software Flash Translation Layer (FTL)
  - Bad block management, wear levelling
  - Only applies to filesystem area

NAND terminology

- Page : read/write unit
- Block : erase unit

Filesystem encryption

Algorithm

- AES in CBC mode
- Initialization vector depends on logical block number
- Hardcoded key for system partition (f65dae950e906c42b254cc58fc78eece)
- 256 bit key for data partition (EMF key)

IV computation

```
void iv_for_lbn(unsigned long lbn, unsigned long *iv)
{
    for(int i = 0; i < 4; i++)
    {
        if(lbn & 1)
            lbn = 0x80000061 ^ (lbn >> 1);
        else
            lbn = lbn >> 1;
        iv[i] = lbn;
    }
}
```

Data partition encryption

iOS 3

- MBR partition type 0xAE (Apple_Encrypted)
- EMF key stored in data partition last logical block
- Encrypted with key 0x89B

iOS 4

- GPT partition table, EMF GUID
- EMF key stored in effaceable area
- Encrypted with key 0x89B
- HFS content protection

Data partition encryption - iOS 3

Encrypted key format

```
struct crpt_ios3
{
    uint32_t magic0; // 'tprc'

    struct encryted_data //encrypted with key89b CBC mode zero iv
    {
        uint32_t magic1; // 'TPRC'
        uint64_t partition_last_lba; //end of data partition
        uint32_t unknown;//0xFFFFFFFF
        uint8_t filesystem_key[32]; //EMF key
        uint32_t key_length; //=32
        uint32_t pad_zero[3];
    };
};
```

iOS 4 NAND layout

Container partitions

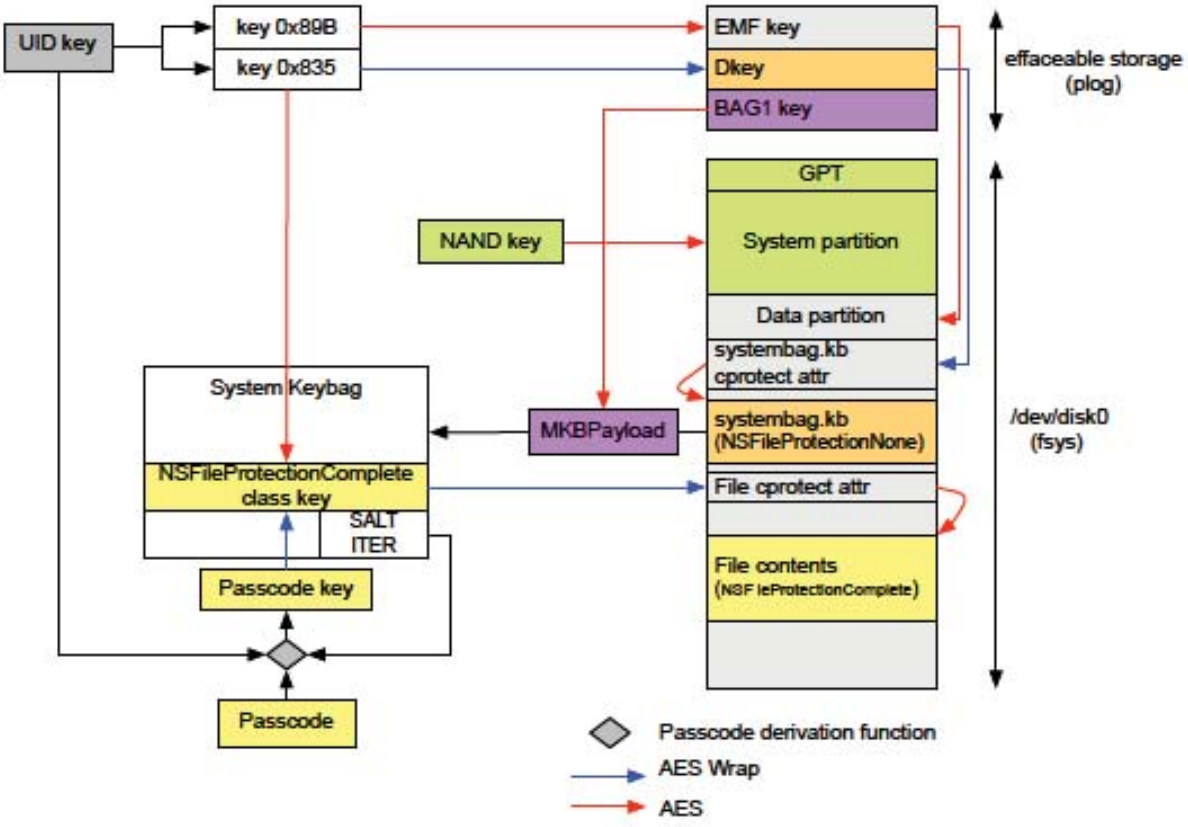
- boot : Low Level Bootloader (LLB) image
- plog : Effaceable area
- nvrm : nvram, contains environments variables
- firm : iBoot, device tree, boot logos (IMG3 images)
- fsys : Filesystem partition, mapped as /dev/disk0

16 Gb iPhone 4 NAND layout

boot block 0	plog block 1	nvrm blocks 2 - 7	firm blocks 8 - 15	fsys blocks 16 - 4084	reserved blocks 4085 - 4100

- 4 banks of 4100 blocks of 128 pages of 8192 bytes data, 448 bytes spare

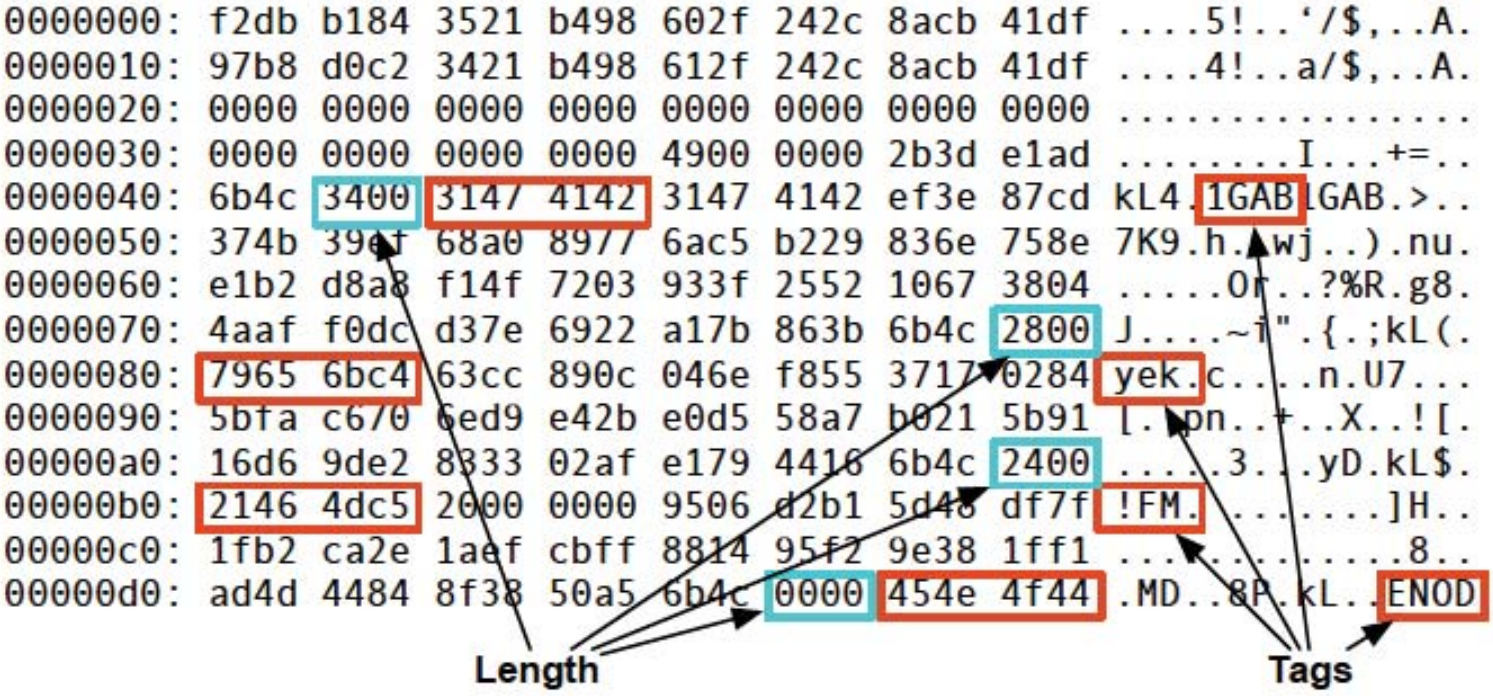
iOS 4 Storage encryption overview



Effaceable area

Plog partition

- Stores small binary blobs ("lockers")
- Abstract AppleEffaceableStorage kernel service
- Two implementations : AppleEffaceableNAND, AppleEffaceableNOR
- AppleEffaceableStorage organizes storage in groups and units
- For AppleEffaceableNAND, 4 groups (1 block in each bank) of 96 units (pages)


0000000: f2db b184 3521 b498 602f 242c 8acb 41df ....5!..‘/$,..A.
0000010: 97b8 d0c2 3421 b498 612f 242c 8acb 41df ....4!..a/$,..A.
0000020: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000030: 0000 0000 0000 0000 4900 0000 2b3d e1ad ....I...+=..
0000040: 6b4c 3400 3147 4142 3147 4142 ef3e 87cd kL4.1GAB1GAB.>..
0000050: 374b 39ef 68a0 8977 6ac5 b229 836e 758e 7K9.h..wj..).nu.
0000060: e1b2 d8a8 f14f 7203 933f 2552 1067 3804 .....Or..?%R.g8.
0000070: 4aaf f0dc d37e 6922 a17b 863b 6b4c 2800 J....~i".{.;kL(.
0000080: 7965 6bc4 63cc 890c 046e f855 3717 0284 yek.c....n.U7...
0000090: 5bfa c670 6ed9 e42b e0d5 58a7 b021 5b91 [..pn..+..X..![.
00000a0: 16d6 9de2 8333 02af e179 4416 6b4c 2400 .....3...yD.kL$.
00000b0: 2146 4dc5 2000 0000 9506 d2b1 5d48 df7f !FM. ....]H..
00000c0: 1fb2 ca2e 1aef cbff 8814 95f2 9e38 1ff1 ....8..
00000d0: ad4d 4484 8f38 50a5 6b4c 0000 454e 4f44 .MD..8P.kL..ENOD
Length
Tags

Plog structures

Plog Unit Header

- header[0:16] XOR header[16:31] = 'ecaF' + 0x1 + 0x1 + 0x0
- generation : incremented at each write
- crc32 (headers + data)

Plog lockers format

Effaceable lockers

EMF!

- Data partition encryption key, encrypted with key 0x89B
- Format: length (0x20) + AES(key89B, emfkey)

Dkey

- NSProtectionNone class key, wrapped with key 0x835
- Format: AESWRAP(key835, Dkey)

BAG1

- System keybag payload key
- Format : magic (BAG1) + IV + Key
- Read from userland by keybagd to decrypt systembag.kb
- Erased at each passcode change to prevent attacks on previous keybag

AppleEffaceableStorage

AppleEffaceableStorage IOKit userland interface

Selector	Description	Comment
0	getCapacity	960 bytes
1	getBytes	requires PE_i_can_has_debugger
2	setBytes	requires PE_i_can_has_debugger
3	isFormatted	
4	format	
5	getLocker	input : locker tag, output : data
6	setLocker	input : locker tag, data
7	effaceLocker	scalar input : locker tag
8	lockerSpace	?

HFS Content Protection

Description

- Each file data fork is encrypted with a unique file key
- File key is wrapped and stored in an extended attribute
  - `com.apple.system.cprotect`
- File protection set through `F_SETPROTECTIONCLASS` fcntl
- Some headers appear in the opensource kernel
  - `http://opensource.apple.com/source/xnu/xnu-1504.9.37/bsd/sys/cprotect.h`

Protection for build-in applications files

Files	Accessibility
Mails & attachments	NSProtectionComplete
Minimized applications screenshots	NSProtectionComplete
Everything else	NSProtectionNone

HFS Content Protection

cprotect extended attribute format

```
struct cprotect_xattr
{
    uint16_t xattr_version; // =2 (version?)
    uint16_t zero; // =0
    uint32_t unknown; // leaks stack dword in one code path :)
    uint32_t protection_class_id;
    uint32_t wrapped_length; // 40 bytes (32 + 8 bytes from
                             // aes wrap integrity)
    uint8_t wrapped_key[1]; // wrapped_length
};
```

HFSExplorer

Motivation

- Standard dd image of iOS 4 data partition yields unreadable files
- When reading data partition from block device interface, each block is decrypted using the EMF key
  - Files data forks decrypted incorrectly

HFSExplorer additions

- Support for inline extended attributes
- Reads EMF, Dkey and other class keys from plist file
- Unwraps cprotect attributes to get file keys
- For each block in data fork :
  - Encrypt with EMF key to get original ciphertext
  - Decrypt with file key
  - (HFS allocation block size == NAND page size)

Data Wipe

Trigger

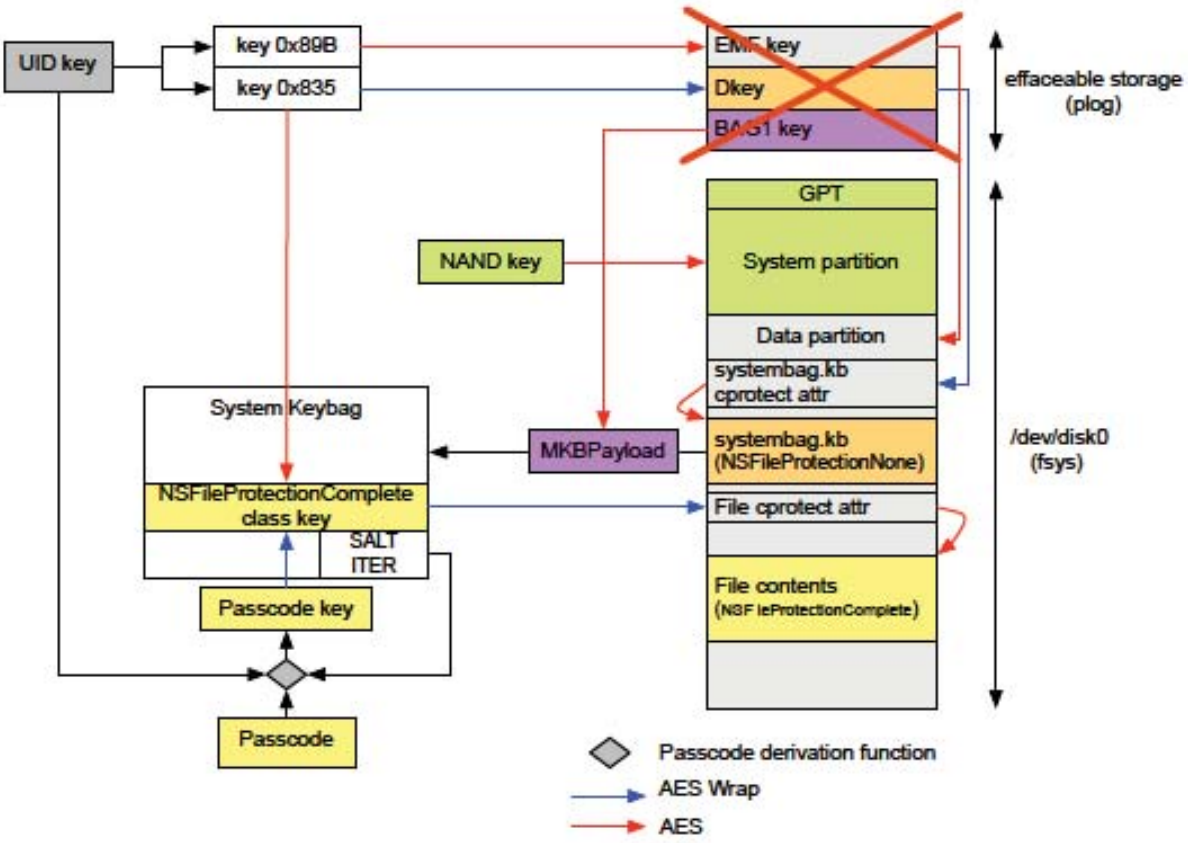
- Preferences → General → Reset → Erase All Content and Settings
- Erase data after *n* invalid passcode attempts
- Restore firmware
- MobileMe Find My iPhone
- Exchange ActiveSync
- Mobile Device Managment (MDM) server

Data Wipe

Operation

- `mobile_obliterator` daemon
- Erase DKey by calling `MKBDeviceObliterateClassDKey`
- Erase EMF key by calling selector `0x14C39` in `EffacingMediaFilter` service
- Reformat data partition
- Generate new system keybag
- High level of confidence that erased data cannot be recovered

iOS 4 Data wipe



Plan

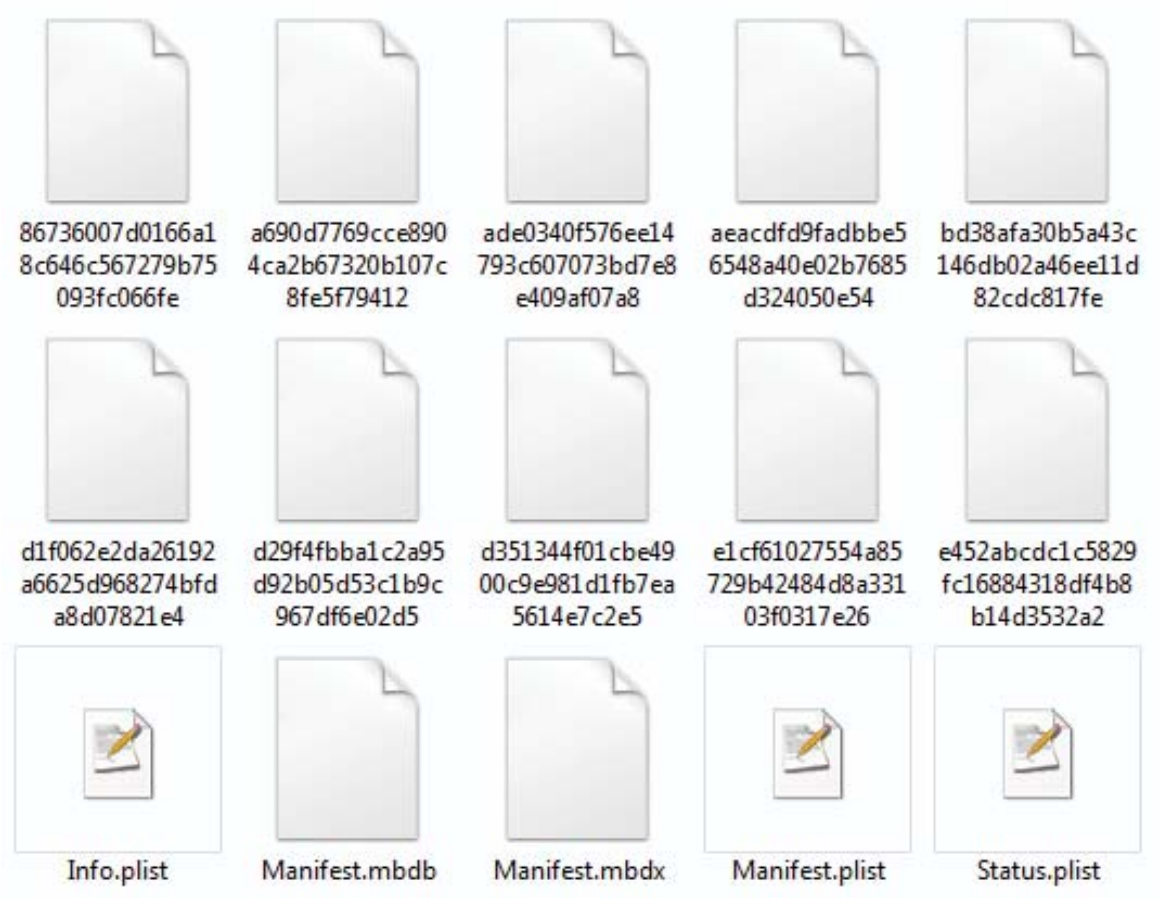
Backed up files

Backup storage

- One directory per backup
- `%APPDATA%/Apple Computer/MobileSync/Backup/<udid>`
- **Can be password protected**
- Each file stored in a separate file
  - Encrypted (AES-256 CBC)
  - Filenames : SHA1 hashes

Database: MBDB

- Custom format
- Two files: `Manifest.mbdb`, `Manifest.mbdx`
- Contains information to restore files correctly
  - Filenames, size, permissions, extended attributes, etc.


86736007d0166a1
8c646c567279b75
093fc066fe
a690d7769cce890
4ca2b67320b107c
8fe5f79412
ade0340f576ee14
793c607073bd7e8
e409af07a8
aeacdfd9fadbbe5
6548a40e02b7685
d324050e54
bd38afa30b5a43c
146db02a46ee11d
82cdc817fe
d1f062e2da26192
a6625d968274bfd
a8d07821e4
d29f4fbba1c2a95
d92b05d53c1b9c
967df6e02d5
d351344f01cbe49
00c9e981d1fb7ea
5614e7c2e5
e1cf61027554a85
729b42484d8a331
03f0317e26
e452abcdc1c5829
fc16884318df4b8
b14d3532a2
Info.plist
Manifest.mbdb
Manifest.mbdx
Manifest.plist
Status.plist

Database format

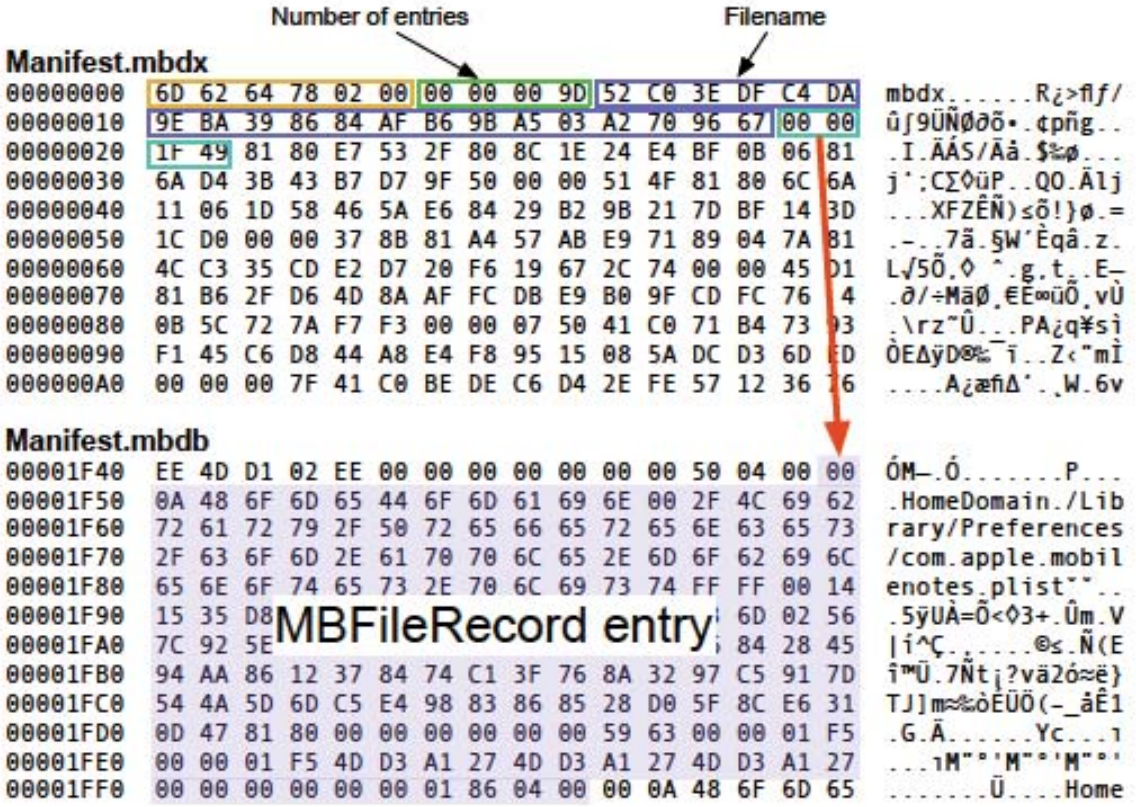
mbdx = index

- hex filenames
- file information offset in mbdb

mbdb = data

- Sequence of `MBFileRecord`
- Path, digest, etc.
- Encryption key, different for each file
  - ...and wrapped by class keys from backup keybag

Database format



Backup keybag

- Same format as before
- Stored in `Manifest.plist`
  - `BackupKeyBag` section
- Random class keys for each backup
  - Different from system keybag keys

Not all the keys can be retrieved

Backup keychain

- Stored in `keychain-backup.plist`
- Same structure as `keychain-2.db`, but in a plist
- Before accessing it:
  - Backup needs to be decrypted
  - Filenames need to be recovered
- Decrypt items using keychain class keys from backup keybag

iTunes backup decrypter

Requirements

- Needs password if protected
- Wrote a bruteforcer (slow)

Implementation

- Decrypted files in a new directory
- Filenames can be restored or not
- `MBFileRecord` fully documented
- Integrated keychain viewer

Plan

Conclusion

Data protection

- Significant improvement over iOS 3
- Derivation algorithm uses hardware key to prevent attacks
- Bruteforce attack only possible due to BootROM vulnerabilities
- Only Mail files are protected by passcode
  - Should be adopted by other build-in apps (Photos, etc.)
  - Might be difficult in some cases (SMS database)

Tools & Source code

- `http://code.google.com/p/iphone-dataprotection/`

Thank you for your attention
Questions ?

References

- Apple WWDC 2010, Session 209 - Securing Application Data
- The iPhone wiki, http://www.theiphonewiki.com
- msftguy ssh ramdisk http://msftguy.blogspot.com/
- AES wrap, RFC 3394 http://www.ietf.org/rfc/rfc3394.txt
- NAND layout, CPICH http://theiphonewiki.com/wiki/index.php?title=NAND
- HFSExplorer, Erik Larsson http://www.catacombae.org/hfsx.html
- syringe, Chronic dev team https://github.com/Chronic-Dev/syringe
- cyanide, Chronic dev team https://github.com/Chronic-Dev/cyanide
- usbmux enable code, comex https://github.com/comex/bloggy/wiki/Redsn0w%2Busbmux
- restored_pwn, Gojohnnyboi https://github.com/Gojohnnyboi/restored_pwn

References

- xpwn crypto tool, planetbeing `https://github.com/planetbeing/xpwn`
- iPhone backup browser `http://code.google.com/p/iphonebackupbrowser/`

Exhibit 18

Cellebrite Physical Extraction Manual for iPhone & iPad

July 3rd, 2011

Revision 1.3

cellebrite
mobile data secured

Table of Contents

Introduction

This manual provides an overview of the steps required to extract data from an iPhone or iPad using the UFED Physical Analyzer.

The UFED Physical Analyzer allows you to extract, decode and analyze the following devices running iOS version 3.0 or higher:

- **iPhone (original)**
- **iPhone 3G**
- **iPhone 3GS**
- **iPhone 4 GSM**
- **iPhone 4 CDMA**
- **iPad 1**

Before You Start

You will need:

- A UFED Physical Analyzer installed on a PC with Windows XP/Vista/7 Operating Systems (iPhone/iPad physical extraction is not designed to be used in Virtual Machine environments).
- An iPhone or iPad.
- UFED Cable Number 110.

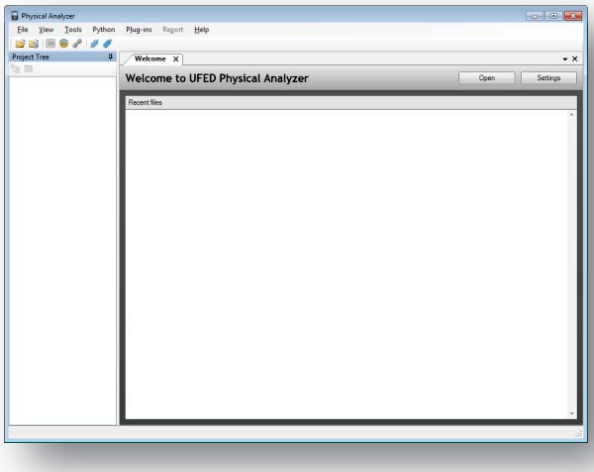
An Internet connection is required before the first use for the installation of updates. Access to the Internet is used to download relevant software and may be carried out through any computer with Internet connection.

Performing an Extraction

The following steps will guide you through the extraction process.

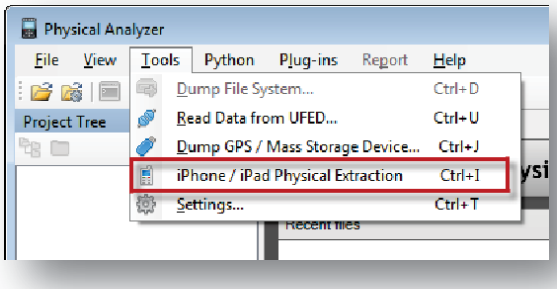
Step 1: Launch the UFED Physical Analyzer

1. Launch UFED Physical Analyzer by clicking the application icon or program shortcut. The default location of UFED Physical Analyzer is: C:\Program Files\Cellebrite Mobile Synchronization\UFED Physical Analyzer.



Step 2: Open iPhone / iPad Physical Extraction

1. Click the *Tools* menu and click *iPhone/iPad Physical Extraction.* "UFED iPhone Physical" will then launch.



On first use

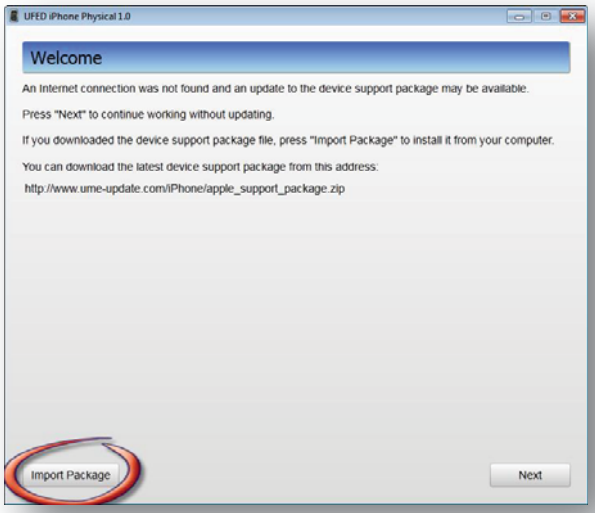
On the first use of UFED iPhone Physical you will be required to download the Apple Device Support Package. The support package contains the newest utilities that enable UFED iPhone Physical to be compatible with a variety of devices. The download may take a while, depending on your Internet connection speed.

No Internet connection?

If your computer is not connected to the Internet you can download the support package on a different computer and manually copy it to your computer.

1. Click this link[1] to download the latest Apple Device Support Package:

2. Copy the file to your computer.
3. Click the *Import Package* button and locate the file on your computer.

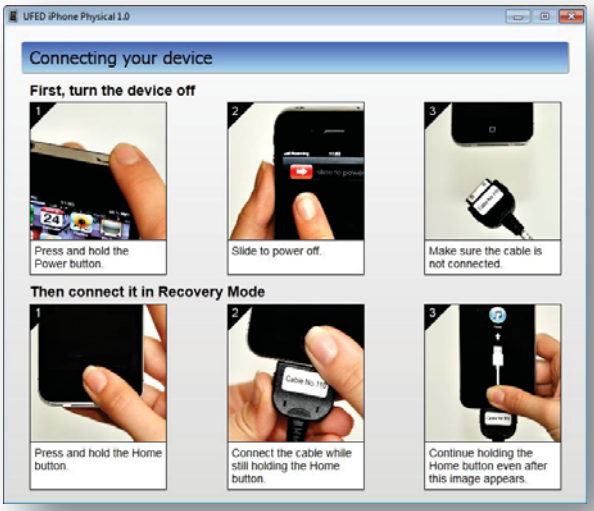


[1] http://www.ume-update.com/iPhone/apple_support_package.zip

Step 3: Connect the device in Recovery Mode to your PC

1. Follow the steps on the screen to connect the device in Recovery Mode.

   Note: connect your device to the PC using cable # 110 or the iPhone/iPad data cable.

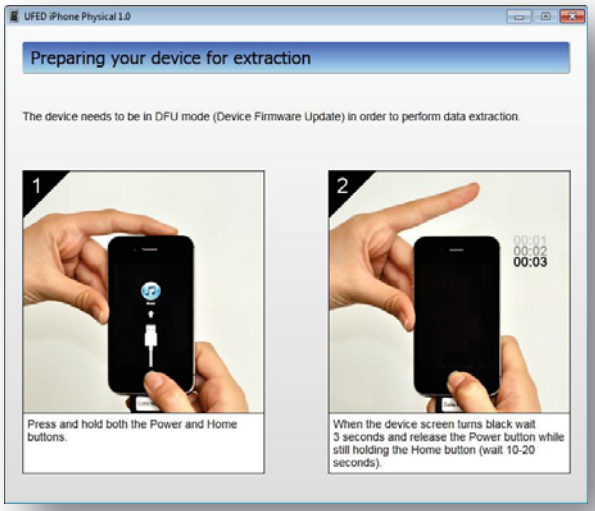


2. After connecting the device in Recovery Mode, UFED iPhone Physical will display certain device information, such as serial number, IMEI, hardware version, iOS version and more. You can copy that information to the clipboard by clicking the *Copy* link.

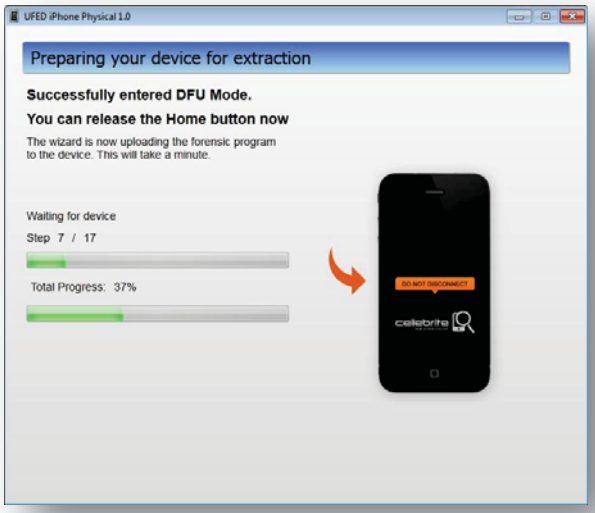
Note: In case a range of versions are displayed, the version of the specific device connected may be any version within the displayed range. In the example above the iOS version may be 4.0, 4.0.1 or 4.0.2.

Step 4: Setting the Device to DFU Mode

1. Click *Next* on the screen with the device info.
2. Follow the instructions on the screen to set the device to DFU (Device Firmware Upgrade) mode. Be assured that UFED iPhone Physical will not affect the device firmware or user data.



3. When you have succeeded, the following screen will be displayed.

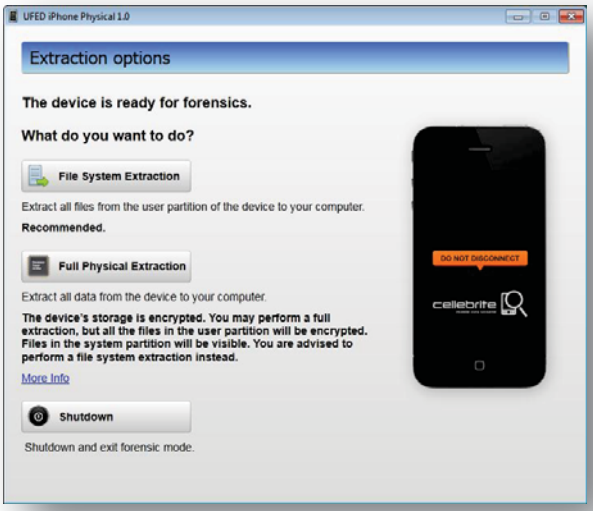


UFED iPhone Physical will upload the forensics program required to extract data from the device. As mentioned above, this will not affect the data, memory or firmware of the device.

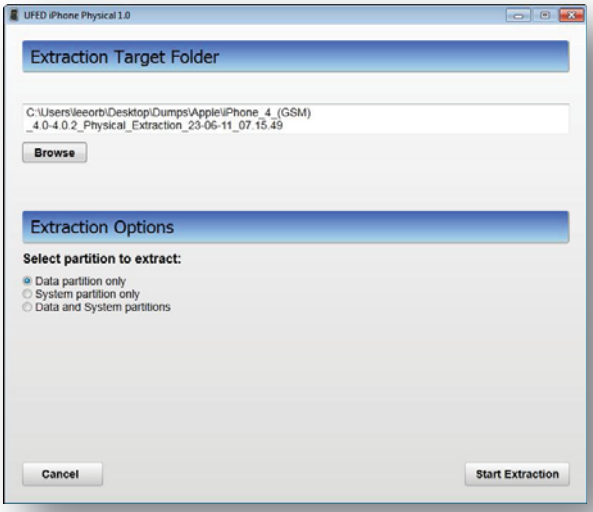
Step 5: Extract Data

Now the device is ready for forensic extraction.

1. Choose the desired extraction method (Full Physical or File System). We recommend reading the Extraction and Encryption FAQ appendix to make the best of your iPhone and iPad extraction.
2. Choose the location you wish to save the extraction to. You can save it on your computer or on a removable storage device.

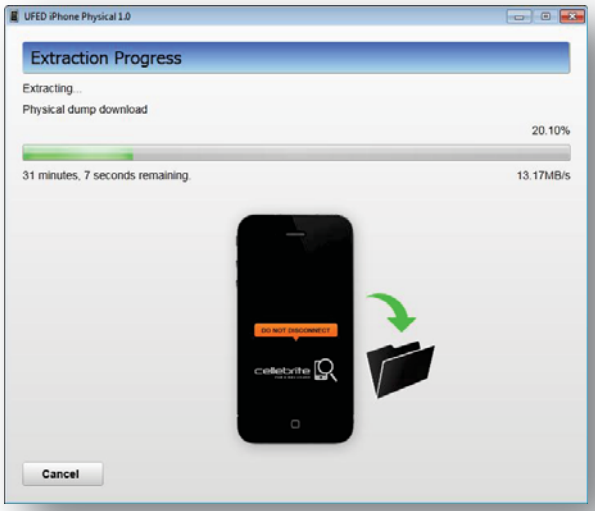


3. While performing Full Physical Extraction, you will be required to choose the relevant partition for extraction. Select the Data partition, System partition or both partitions.
4. Click Start Extraction.

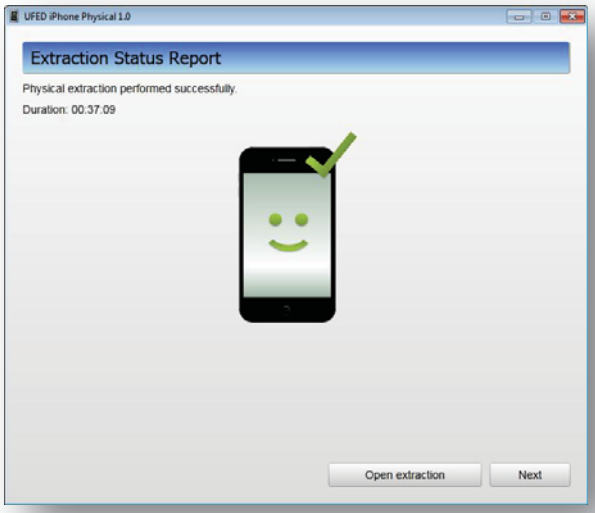


Step 6: Wait

1. Wait until the extraction is completed. The extraction duration varies depending on the extraction method, the device used, the quantity of data on the device, your computer and other parameters.

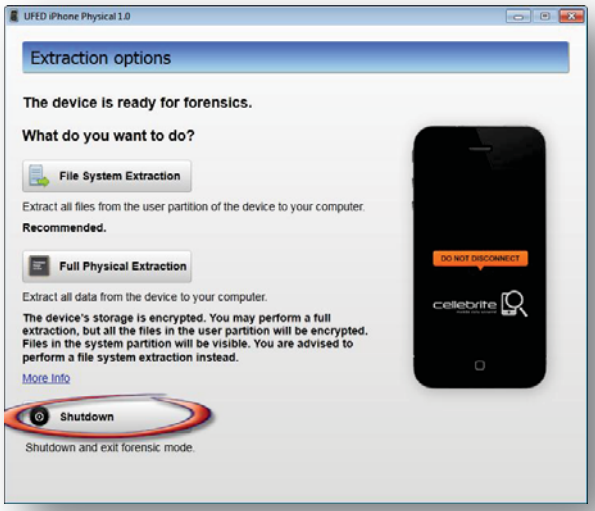


2. When the extraction is completed you will see this screen.
3. Clicking *Open extraction* will load the extraction file in UFED Physical Analyzer.
4. Clicking *Next* will take you back to the extraction options screen.

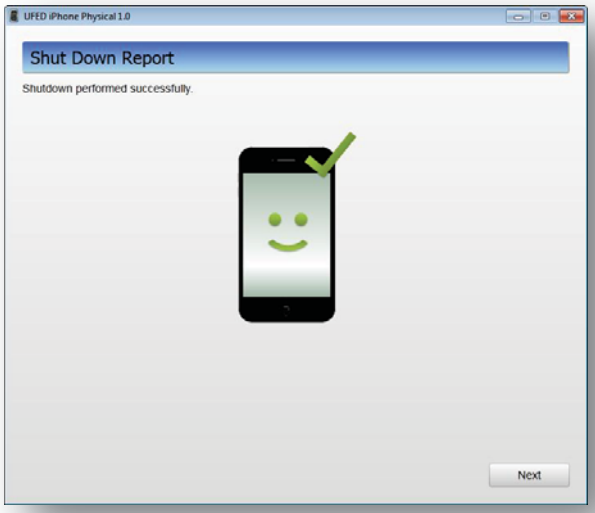


Step 7: Shutdown the Device

1. When extraction is complete, you may click *Shutdown* to safely turn off the device and set it back to normal mode.



2. The *Shut Down Report* screen will indicate your device has successfully been shut down.



cellebrite
mobile data secured

Appendix - UFED iPhone Physical Extraction and Encryption FAQ

Is it possible to extract data from user locked iPhone devices?

Yes. The UFED iPhone Physical Extraction solution enables extraction of the device image and file system even when user lock is active.

What is "physical extraction"?

Physical extraction is performed by imaging the device's partitions. This recovers the device's entire file system which can then be decoded by UFED Physical Analyzer. On devices that have data encryption, the contents of the files may be encrypted (explanation below).

What is "low-level file system extraction"?

Apple iOS devices have two partitions: The system partition (normally 1GB) and the user data partition (the rest of the flash memory). The system partition contains the operating system files. The user data partition contains all user-generated content (photos, messages, etc.)

Low-level file system extraction reads the entire directory tree of the user partition and puts it in a simple "tar" file. The user data will not be encrypted in a low-level file system extraction, even if encryption is enabled on the device. However, some "protected" files cannot be fully extracted.

On devices that have data encryption, some files may be protected and inaccessible. Protected files are only readable when the device is turned on regularly and unlocked. Low-level file system extraction cannot extract the contents of those files; only their metadata. Among the protected files are some of the email files.

The system partition is never encrypted, even if encryption is enabled on the device.

cellebrite
mobile data secured

What devices have data encryption enabled?

Device	Data Encryption
iPhone (Original), iPhone 3G, iPod Touch 1st and 2nd generation*	Disabled
iPhone 3GS iPod Touch 3rd generation* iPad 1	In some cases. See paragraph below.
iPhone 4 iPod Touch 4th generation* iPad 2*	Enabled

* Extraction from this device is not currently supported.

iPhone 3GS, iPod Touch 3rd Generation and iPad 1 were originally manufactured and shipped with iOS version 3.x. The data encryption feature was added in iOS 4.x.

Simply updating an iOS 3.x device to iOS 4.x (or later) does not enable data encryption. Data encryption will be enabled on these devices only if the user has "restored" the device with iOS 4.x. (or later) "Restore" is a feature in iTunes which reformats the file system (making it encryption-ready) and reinstalls iOS.

If the device had iOS 4.x (or later) preinstalled on it when it was bought, encryption will be enabled.

What type of extracted data will be encrypted?

If data encryption is disabled, all data on the device will be unencrypted and readable. However, if data encryption is enabled, the data that's encrypted varies between the different types of extractions:

Extraction type	If data encryption enabled
Physical extraction - system partition	Will be extracted and not encrypted
Physical extraction - user partition	File contents will be encrypted. Directory tree, file names, modification dates, etc. will not be encrypted
Low-level file system extraction Non-protected files	Will be extracted and not encrypted
Low-level file system extraction Protected files	File contents will not be extracted. Only 0's will appear. File names, modification dates, etc. will be extracted and not encrypted

What is the best way to extract data from an encrypted device?

The best way to extract data from a device with encryption enabled is to perform a low-level file system extraction. You will be able to retrieve all user content except protected files (among which are some of the email files).

Can jailbreaking help extract data from an encrypted device?

Unfortunately, jailbreaking does not help circumvent the data encryption. The Cellebrite UFED solution performs extraction without Jailbreaking the device. Both Jailbroken and non-jailbroken devices are supported.

Does data extraction affect the storage or data on the device?

No.

The extraction application does not load iOS, but instead loads a special forensic utility to the device. This utility is loaded to the device's memory (RAM) and runs directly from there. Therefore, it does not modify the device's storage and does not leave any footprints.

Exhibit 19

Cryptographic Services

Cryptographic services form the foundation of securing data in transit (secure communications) and data at rest (secure storage). Using sophisticated mathematics, they allow you to:

- Encrypt and decrypt data so that it cannot be understood by an outside observer
- Verify that data has not been modified since it was originally sent by hashing, signing, and verifying

This chapter describes these cryptographic techniques and briefly summarizes the technologies that OS X and iOS provide to help you use cryptography in your own application.

Encryption and Decryption

Encryption is a means of protecting data from interception by transforming it into a form that is not readable except by someone who knows how to transform it back.

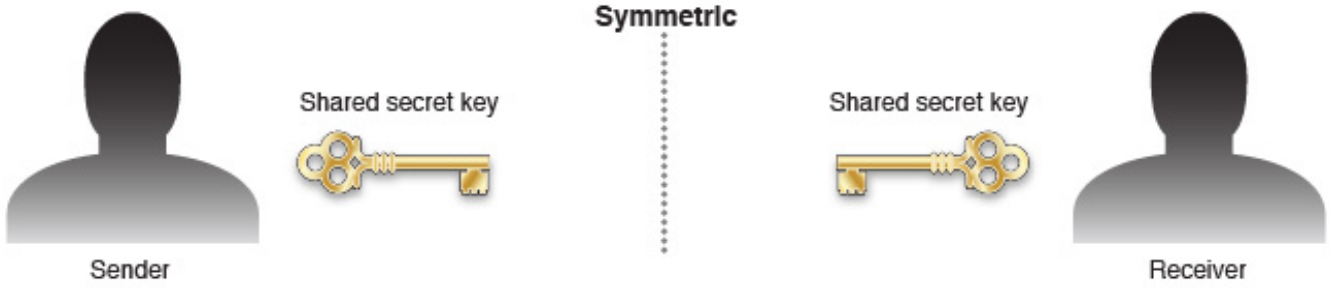
Encryption is commonly used to protect data in transit and data at rest. When information must be sent across an untrusted communication channel, it is the responsibility of the two endpoints to use encryption to secure the communication. Similarly, when storing information on a local disk, an app may use encryption to ensure that the information is not readable by third parties even if the computer is stolen.

There are many different encryption techniques, called *ciphers*, that work in different ways and can serve different purposes. Ciphers generally work by combining the original information (the *cleartext*, or *plaintext*) with a second piece of information (a key) in some fashion to produce an encrypted form, called the *ciphertext*.

Modern encryption techniques can be grouped into three broad categories: symmetric encryption, asymmetric encryption, and steganography.

Symmetric Encryption

In symmetric encryption, a single key (usually a long string of random bytes) is used to mathematically transform a piece of information and is later used in reverse to retrieve the original information.



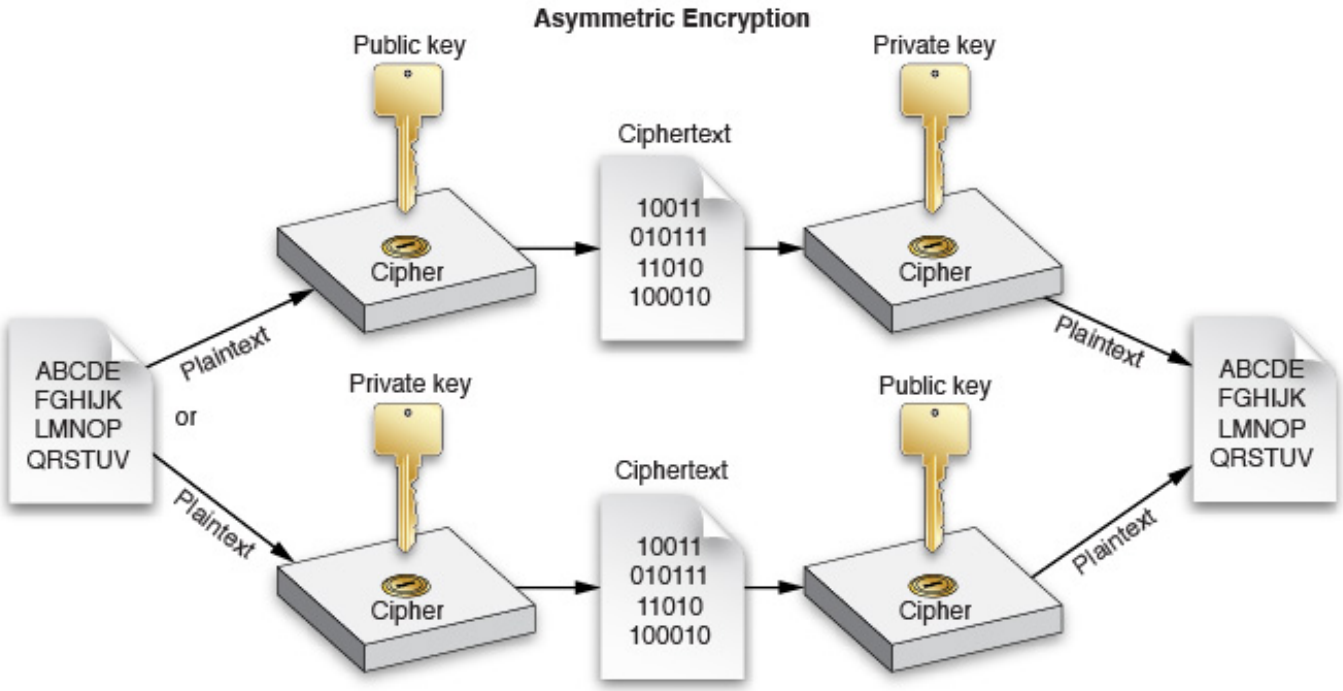
Symmetric encryption is often used for secure communication. However, because both endpoints must

know the same secret key, symmetric encryption is not sufficient by itself.

Asymmetric Encryption

In asymmetric encryption, two mathematically related keys are used to transform a piece of information. Information encrypted with one key can be decrypted only with the other key and vice versa. Generally speaking, one of these keys (the private key) is kept secret, and the other key (the public key) is made broadly available. For this reason, asymmetric encryption is also called *public key cryptography*.

Note: Although the two keys are mathematically related, it is considered computationally infeasible to derive one key from the other. The security of public key cryptography depends on this being the case.

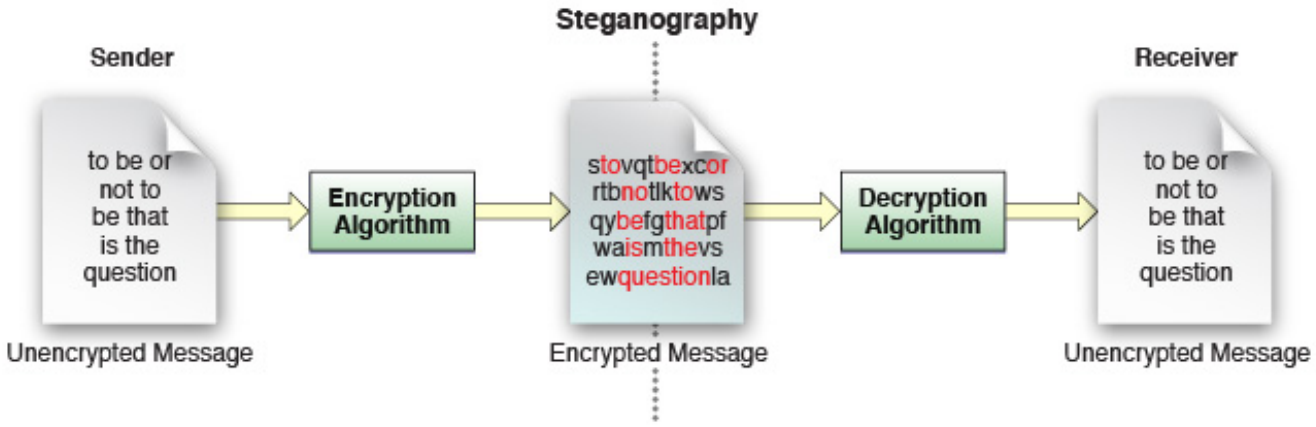


Asymmetric encryption is often used for establishing a shared communication channel. Because asymmetric encryption is computationally expensive, the two endpoints often use asymmetric encryption to exchange a symmetric key, and then use a much faster symmetric encryption algorithm for encrypting and decrypting the actual data.

Asymmetric encryption can also be used to establish trust. By encrypting information with your private key, someone else can read that information with your public key and be certain that it was encrypted by you.

Steganography

Steganography means hiding information in less important bits of another piece of information.



Steganography is commonly used for storing copyright information into photographs in such a way that is largely indistinguishable from noise unless you know how to look for it.

Steganography can also be used for storing encrypted volumes underneath other encrypted or unencrypted volumes (either by using the unused blocks or by taking advantage of error correction in subtle ways).

Hashing

A hash value, or hash, is a small piece of data derived from a larger piece of data that can serve as a proxy for that larger piece of data. In cryptography, hashes are used when verifying the authenticity of a piece of data. Cryptographic hashing algorithms are essentially a form of (extremely) lossy data compression, but they are specifically designed so that two similar pieces of data are unlikely to hash to the same value.

For example, two schoolchildren frequently passed notes back and forth while deciding when to walk home together. One day, a bully intercepted the note and arranged for Bob to arrive ten minutes early so that he could steal Bob's lunch money. To ensure that their messages were not modified in the future, they devised a scheme in which they computed the remainder after dividing the number of letters in the message by the sum of their ages, then wrote that many dots in the corner of the message. By counting the number of letters, they could (crudely) detect certain modifications to each other's messages.

This is, of course, a contrived example. A simple remainder is a *very* weak hashing algorithm. With good hashing algorithms, collisions are unlikely if you make small changes to a piece of data. This tamper-resistant nature of good hashes makes them a key component in code signing, message signing, and various other tamper detection schemes.

At a high level, hashing is also similar to checksumming (a technique for detecting and correcting errors in transmitted data). However, the goals of these techniques are very different, so the algorithms used are also very different. Checksums are usually designed to allow detection and repair of a single change or a small number of changes. By contrast, cryptographic hashes must reliably detect a large number of changes to a single piece of data but need not tell you how the data changed.

For example, the following command in the shell demonstrates a common hashing algorithm:

```
$ echo "This is a test.  This is only a test." | sha1sum
```

```
7679a5fb1320e69f4550c84560fc6ef10ace4550  -
```

OS X provides a number of C language APIs for performing hashing. These are described further in the documents cited at the end of this chapter.

Signing and Verifying

A signature is a way to prove the authenticity of a message, or to verify the identity of a server, user, or other entity.

In olden days, people sometimes stamped envelopes with a wax seal. This seal not only proved who sent the message but also proved that no one had opened the message and potentially modified it while in transit.

Modern signing achieves many of the same benefits through mathematics. In addition to the data itself, signing and verifying require two pieces of information: the appropriate half of a public-private key pair and a digital certificate.

The sender computes a hash of the message and encrypts it with the private key. The recipient also computes a hash and then uses the corresponding public key to decrypt the sender's hash and compares the hashes. If they are the same, the data was not modified in transit, and you can safely trust that the data was sent by the owner of that key.

The sender's digital certificate is a collection of data that contains a public key and other identifying information, at the sender's discretion, such as a person's name, a company name, a domain name, and a postal address. The purpose of the certificate is to tie a public key to a particular person. If you trust the certificate, you also trust that messages signed by the sender's private key were sent by that person.

To provide a means of determining the legitimacy of a certificate, the sender's certificate is signed by someone else, whose certificate is in turn signed by someone else, and so on, forming a chain of trust to a certificate that the recipient inherently trusts, called an *anchor certificate*. This certificate may be a root certificate—a self-signed certificate that represents a known certificate authority and thus the root of the tree of certificates originating from that authority—or it may be any arbitrary certificate that the user or application developer has explicitly designated as a trusted anchor.

Because the recipient trusts the anchor certificate, the recipient knows that the certificate is valid and, thus, that the sender is who he or she claims to be. The degree to which the recipient trusts a certificate is defined by two factors:

- Each certificate can contain one or more *certificate extensions* that describe how the certificate can be used. For example, a certificate that is trusted for signing email messages might not be trusted for signing executable code.
- The *trust policy* allows you to trust certificates that would otherwise be untrusted and vice versa.

A certificate can also be used for authentication. By signing a nonce (a randomly generated challenge string created specifically for this purpose), a user or server can prove that he, she, or it is in possession of the private key associated with that certificate. If that certificate is considered trusted (by evaluating its chain of trust), then the certificate and signed nonce prove that the user or server must be who he, she, or it claims to be.

Secure Storage

OS X and iOS provide a number of technologies for secure storage. Of these, the three most commonly used technologies are keychains, FileVault, and data protection.

Keychains

In concept, a keychain is similar to a physical key ring in that it is a place where keys and other similarly small pieces of data can be stored for later use in performing cryptographic tasks, but the similarity ends there. With a physical key ring, the owner can take the key and use it to unlock something. With a keychain, apps usually do not access the actual key data itself, so they do not risk exposing the keys even if compromised. Instead, they use a unique identifier to identify those keys, and the actual encryption is performed in a separate process called the Security Server (described later in this document).

Thus, a keychain is in some ways more like a heavily armed security guard in full body armor who carries a key ring. You can ask that guard to unlock a door for you if you are authorized to enter, but you usually can't unlock the door yourself.

OS X also includes a utility that allows users to store and read the data in the keychain, called *Keychain Access*. This utility is described in more detail later, in Keychain Access.

FileVault

In OS X, FileVault uses encryption to provide encrypted storage for the user's files. When FileVault is enabled, the disk is decrypted only after an authorized user logs in. (Note that prior to OS X v10.7, FileVault protected only a user's home directory.)

FileVault and its configuration UI are described in more detail later, in End-User Security Features.

Data Protection

iOS provides APIs that allow an app to make files accessible only while the device is unlocked to protect their contents from prying eyes. With data protection, files are stored in encrypted form and are decrypted only after the user enters his or her passcode.

For apps that run in the background, there are also settings that allow the file to remain available until the user shuts down the device.

To Learn More

For a more detailed conceptual overview of authentication and authorization in OS X, read *Cryptographic Services Guide*.

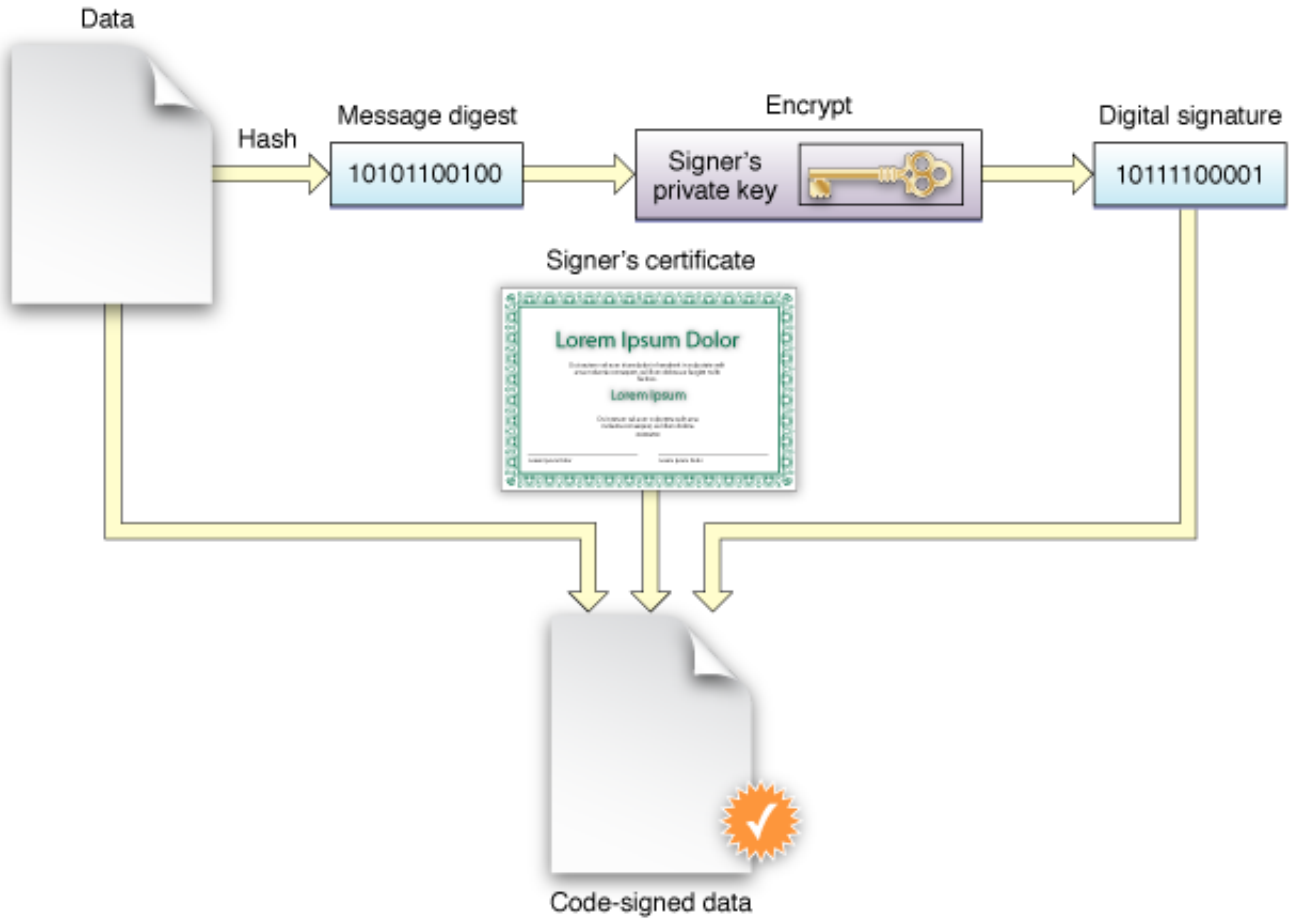
To learn more about creating signing certificates, read Creating Your Signing Certificates in *App Distribution Guide*.

You can also learn about other Apple and third-party security books in Other Security Resources.

Exhibit 20

About Code Signing

Code signing is a security technology, used in OS X, that allows you to certify that an app was created by you. Once an app is signed, the system can detect any change to the app—whether the change is introduced accidentally or by malicious code.



Users appreciate code signing. After installing a new version of a code-signed app, a user is not bothered with alerts asking again for permission to access the keychain or similar resources. As long as the new version uses the same digital signature, OS X can treat the new app exactly as it treated the previous one.

Other OS X security features, such as App Sandbox and parental controls, also depend on code signing.

In most cases, you can rely on Xcode's automatic code signing (described in *App Distribution Guide*), which requires only that you specify a code signing identity in the build settings for your project. This document is for readers who must go beyond automatic code signing—perhaps to troubleshoot an unusual problem, or to incorporate the `codesign(1)` tool into a build system.

At a Glance

The elements of code signing include code signatures, code signing identities, code signing certificates, and security trust policies. Be sure to understand these concepts if you need to perform code signing outside of Xcode.

Relevant chapter: Code Signing Overview

Before you can sign code, you must obtain or create a code signing identity. You then sign your code and prepare it for distribution.

Relevant chapter: Code Signing Tasks

To specify recommended criteria for verifiers to use when evaluating your app's code signature, you use a requirements language specific to the `codesign(1)` and `csreq(1)` commands. You then save your criteria to a binary file as part of your Xcode project.

Relevant chapter: Code Signing Requirement Language

Prerequisites

Read *Security Overview* to understand the place of code signing in the OS X security picture.

See Also

For descriptions of the command-line tools for performing code signing, see the `codesign(1)` and `csreq(1)` man pages.

Exhibit 21

Code Signing Overview

Code signing is a security technique that can be used to ensure code integrity, to determine who developed a piece of code, and to determine the purposes for which a developer intended a piece of code to be used. Although the code signing system performs policy checks based on a code signature, it is up to the caller to make policy decisions based on the results of those checks. When it is the operating system that makes the policy checks, whether your code will be allowed to run in a given situation depends on whether you signed the code and on the requirements you included in the signature.

This chapter describes the benefits of signing code and introduces some of the basic concepts you need to understand in order to carry out the code signing process.

Before you read this chapter, you should be familiar with the concepts described in *Security Overview*.

The Benefits Of Signing Code

When a piece of code has been signed, it is possible to determine reliably whether the code has been modified by someone other than the signer. The system can detect such alteration whether it was intentional (by a malicious attacker, for example) or accidental (as when a file gets corrupted). In addition, through signing, a developer can state that an app update is valid and should be considered by the system as the same app as the previous version.

For example, suppose a user grants the SurfWriter app permission to access a keychain item. Each time SurfWriter attempts to access that item, the system must determine whether it is indeed the same app requesting access. If the app is signed, the system can identify the app with certainty. If the developer updates the app and signs the new version with the same unique identifier, the system recognizes the update as the same app and gives it access without requesting verification from the user. On the other hand, if SurfWriter is corrupted or hacked, the signature no longer matches the previous signature; the system detects the change and refuses access to the keychain item.

Similarly, if you use Parental Controls to prevent your child from running a specific game, and that game has been signed by its manufacturer, your child cannot circumvent the control by renaming or moving files. Parental Controls uses the signature to unambiguously identify the game regardless of its name, location, or version number.

All sorts of code can be signed, including tools, applications, scripts, libraries, plug-ins, and other “code-like” data.

Code signing has three distinct purposes. It can be used to:

- ensure that a piece of code has not been altered
- identify code as coming from a specific source (a developer or signer)
- determine whether code is trustworthy for a specific purpose (for example, to access a keychain item).

To enable signed code to fulfill these purposes, a code signature consists of three parts:

- A seal, which is a collection of checksums or hashes of the various parts of the code, such as the identifier, the `Info.plist`, the main executable, the resource files, and so on. The seal can be used to detect alterations to the code and to the app identifier.
- A digital signature, which signs the seal to guarantee its integrity. The signature includes information that can be used to determine who signed the code and whether the signature is valid.
- A unique identifier, which can be used to identify the code or to determine to which groups or categories the code belongs. This identifier can be derived from the contents of the `Info.plist` for the app, or can be provided explicitly by the signer.

For more discussion of digital signatures, see the following section, Digital Signatures and Signed Code.

To learn more about how a code signature is used to determine the signed code's trustworthiness for a specific purpose, see Code Requirements.

Note that code signing deals primarily with running code. Although it can be used to ensure the integrity of stored code (on disk, for example), that's a secondary use.

To fully appreciate the uses of code signing, you should be aware of some things that signing *cannot* do:

- It can't guarantee that a piece of code is free of security vulnerabilities.
- It can't guarantee that an app will not load unsafe or altered code—such as untrusted plug-ins—during execution.
- It is not a digital rights management (DRM) or copy protection technology. Although the system could determine that a copy of your app had not been properly signed by you, or that its copy protection had been hacked, thus making the signature invalid, there is nothing to prevent a user from running the app anyway.

Digital Signatures and Signed Code

As explained in *Security Overview*, a digital signature uses public key cryptography to ensure data integrity. Like a signature written with ink on paper, a digital signature can be used to identify and authenticate the signer. However, a digital signature is more difficult to forge, and goes one step further: it can ensure that the signed data has not been altered. This is somewhat like designing a paper check or money order in such a way that if someone alters the written amount of money, a watermark with the text "Invalid" becomes visible on the paper.

To create a digital signature, the signing software computes a special type of checksum called a hash (or digest) based on a piece of data or code and encrypts that hash with the signer's private key. This encrypted hash is called a signature.

To verify that signature, the verifying software computes a hash of the data or code. It then uses the signer's public key to decrypt the signature, thus obtaining the original hash as computed by the signer. If the two hashes match, the data has not been modified since it was signed by someone in possession of the signer's private key.

Signed code contains several digital signatures:

- If the code is universal, the object code for each slice (architecture) is signed separately. This

signature is stored within the binary file itself.

- Various components of the application bundle (such as the `Info.plist` file, if there is one) are also signed. These signatures are stored in a file called `_CodeSignature/CodeResources` within the bundle.

Code Requirements

It is up to the system or program that is launching or loading signed code to decide whether to verify the signature and, if it does, to determine how to evaluate the results of that verification. The criteria used to evaluate a code signature are called *code requirements*. The signer can specify requirements when signing the code; such requirements are referred to as *internal requirements*. A verifier can read any internal requirements before deciding how to treat signed code. However, it is up to the verifier to decide what requirements to use. For example, Safari could require a plug-in to be signed by Apple in order to be loaded, regardless of whether that plug-in's signature included internal requirements.

One major purpose of code signatures is to allow the verifier to identify the code (such as a program, plug-in, or script) to determine whether it is the same code the verifier has seen before. The criteria used to make this determination are referred to as the code's *designated requirement*. For example, the designated requirement for Apple Mail might be "was signed by Apple and the identifier is `com.apple.Mail`".

To see how this works in practice, assume the user has granted permission to the Apple Mail application to access a keychain item. The keychain uses Mail's designated requirement to identify it: the keychain records the identifier (`com.apple.Mail`) and the signer of the application (Apple) to identify the program allowed to access the keychain item. Whenever Mail attempts to access this keychain item, the keychain looks at Mail's signature to make sure that the program has not been corrupted, that the identifier is `com.apple.Mail`, and that the program was signed by Apple. If everything checks out, the keychain gives Mail access to the keychain item. When Apple issues a new version of Mail, the new version includes a signature, signed by Apple, that identifies the application as `com.apple.Mail`. Therefore, when the user installs the new version of Mail and it attempts to access the keychain item, the keychain recognizes the updated version as the same program and does not prompt the user for verification.

Architecturally, a code requirement is a script, written in a dedicated language, that describes conditions (restrictions) the code must satisfy to be acceptable for some purpose. It is up to you whether to specify internal requirements when you sign code.

The program identifier or the entire designated requirement can be specified by the signer, or can be inferred by the `codesign` tool at the time of signing. In the absence of an explicitly specified designated requirement, the `codesign` utility typically builds a designated requirement from the name of the program found in its `Info.plist` file and the chain of signatures securing the code signature.

Note that validation of signed code against a set of requirements is performed only when the system or some other program needs to determine whether it is safe to trust that code. For example, unsigned code injected into an application through a buffer overflow can still execute because it was not part of the application at launch time. Similarly, an app with an invalid code identifier may still run (depending on policy), but does not get automatic access to keychain items created by previous versions of the app.

The Role of Trust in Code Signing

Trust is determined by policy. A security trust policy determines whether a particular identity should be accepted for allowing something, such as access to a resource or service. Various parts of OS X have different policies, and make this determination differently. For example, a specialized client application might include a set of root certificates that it trusts when communicating with a specific set of servers. However, these root certificates would not be trusted if those same servers were accessed using a web browser.

In much the same way, many parts of OS X (the OS X keychain and parental controls, for example) do not care what entity signed an application; they care only whether the signer has changed since the last time the signature was checked. They use the code signature's designated requirement for this purpose.

Other parts of OS X constrain acceptable signatures to only those drawn from certificate authorities (root certificates) that are trusted anchors on the system performing the validation. For those checks, the nature of the identity used matters. The Application Firewall is one example of this type of policy. Self-signed identities and self-created certificate authorities do not work for these purposes unless the user has explicitly told the operating system to trust the certificates.

You can modify the code signing polices of OS X with the `spctl(8)` command.

Exhibit 22

Code Signing Tasks

This chapter gives procedures and examples for the code signing process. It covers what you need to do before you begin to sign code, how to sign code, and how to ship the code you signed.

Obtaining a Signing Identity

To sign code, you need a code signing identity, which is a private key plus a digital certificate. The digital certificate must have a usage extension that enables it to be used for signing and it must contain the public key that corresponds to the private key. You can use more than one signing identity, each for its own purpose, such as one to be used for beta seeds and one for final, released products. However, most organizations use only one identity.

You can obtain two types of certificates from Apple using the developer portal: Developer ID certificates (for public distribution) and distribution certificates (for submitting to the Mac App Store). To learn more about this, read *Tools Workflow Guide for Mac*.

Note: Apple uses the industry-standard form and format of code signing certificates. Therefore, if your company already has a third-party signing identity that you use to sign code on other systems, you can use it with the OS X `codesign` command. Similarly, if your company is a certificate issuing authority, contact your IT department to find out how to get a signing certificate issued by your company.

If you do not have an existing identity, you should first create one using the Certificate Assistant, which is provided as part of the Keychain Access application. This tool creates a public key, puts it into your keychain, and optionally can produce a certificate signing request that you can then send to Apple (or another certificate authority). The certificate authority then sends you a certificate that, in combination with your private key, completes your digital identity.

To import a signing certificate with Keychain Access

Note: If the original private key is not already in your keychain (for example, if you are moving from one development machine to another), you must also import the private key in the same way.

Before you obtain a code signing identity and sign your code, consider the following points:

- Do not ship applications signed by self-signed certificates. A self-signed certificate created with the Certificate Assistant is not recognized by users' operating systems as a valid certificate for any purpose other than validating the designated requirement of your signed code. Because a self-signed certificate has not been signed by a recognized root certificate authority, the user can only

verify that two versions of your application came from the same source; they cannot verify that your company is the true source of the code. For more information about root authorities, see Security Concepts.

- Depending on your company's internal policies, you might have to involve your company's Build and Integration, Legal, and Marketing departments in decisions about what sort of signing identity to use and how to obtain it. You should start this process well in advance of the time you need to actually sign the code for distribution to customers.
- Any signed version of your code that gets into the hands of users will appear to have been endorsed by your company for use. Therefore, you might not want to use your "final" signing identity to sign code that is still in development.
- A signing identity, no matter how obtained, is completely compromised if it is ever out of the physical control of whoever is authorized to sign the code. That means that the signing identity's private key must never, under any circumstances, be given to end users, and should be restricted to one or a small number of trusted persons within your company. Before obtaining a signing identity and proceeding to sign code, you must determine who within your company will possess the identity, who can use it, and how it will be kept safe. For example, if the identity must be used by more than one person, you can keep it in the keychain of a secure computer and give the password of the keychain only to authorized users, or you can put the identity on a smart card to which only authorized users have the PIN.
- A self-signed certificate created by the Certificate Assistant is adequate for internal testing and development, regardless of what procedures you put in place to sign released products.

To use the Certificate Assistant to create a self-signed signing identity

Adding an Info.plist to Single-File Tools

As discussed in Code Requirements, the system often uses the `Info.plist` file of an application bundle to determine the code's designated requirement. Although single-file tools don't normally have an `Info.plist`, you can add one. To do so, use the following procedure:

1. Add an `Info.plist` file to your project (including adding it to your source control).
2. Make sure the `Info.plist` file has the following keys:
   - `CFBundleIdentifier`
   - `CFBundleName`
3. The value for `CFBundleIdentifier` is used as the default unique name of your program for Code Signing purposes. Because the `CFBundleIdentifier` value is also used when your application accesses resources in the application bundle, it may sometimes be necessary to use a non-unique `CFBundleIdentifier` value for a helper. If you do this, you must provide a different, unique identifier for code signing purposes by passing the `-i` or `--identifier` flag to the `codesign` command.

   The identifier used for signing must be globally unique. To ensure uniqueness, you should

include your company's name in the value. The usual form for this identifier is a hierarchical name in reverse DNS notation, starting with the top level domain, followed by the company name, followed by the organization within the company, and ending with the product name. For example, the `CFBundleIdentifier` value for the `codesign` command is `com.apple.security.codesign`.

4. The value for `CFBundleName` shows up in system dialogs as the name of your program, so it should match your marketing name for the product.
5. Add the following arguments to your linker flags:

   `-sectcreate __TEXT __info_plist` *Info.plist_path*

   where *Info.plist_path* is the complete path of the `Info.plist` file in your project.

   In Xcode, for example, you would add these linker flags to the `OTHER_LDFLAGS` build variable (Other Linker Flags in the target's build rules).

For example, here are the contents of the `Info.plist` file for the `codesign` command:

```
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.security.codesign</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>codesign</string>
    <key>CFBundleVersion</key>
    <string>0.3</string>
</dict>
</plist>
```

Signing Your Code

You use the `codesign` command to sign your code. This section discusses what to sign and gives some examples of the use of `codesign`. See the `codesign(1)` manual page for a complete description of its use.

What to Sign

You should sign every executable in your product, including applications, tools, hidden helper tools,

utilities and so forth. Signing an application bundle covers its resources, but not its subcomponents such as tools and sub-bundles. Each of these must be signed independently.

If your application consists of a big UI part with one or more little helper tools that try to present a single face to the user, you can make them indistinguishable to code signing by giving them all the exact same code signing identifier. (You can do that by making sure that they all have the same `CFBundleIdentifier` value in their `Info.plist`, or by using the `-i` option in the `codesign` command, to assign the same identifier.) In that case, all your program components have access to the same keychain items and validate as the same program. Do this only if the programs involved are truly meant to form a single entity, with no distinctions made.

A universal binary (bundle or tool) automatically has individual signatures applied to each architecture component. These are independent, and usually only the native architecture on the end user's system is verified.

In the case of installer packages (.pkg and .mpkg bundles), everything is implicitly signed: The CPIO archive containing the payload, the CPIO archive containing install scripts, and the bill of materials (BOM) each have a hash recorded in the XAR header, and that header in turn is signed. Therefore, if you modify an install script (for example) after the package has been signed, the signature will be invalid.

You may also want to sign your plug-ins and libraries. Although this is not currently required, it will be in the future, and there is no disadvantage to having signatures on these components.

Important: When code signing a framework, you must sign a particular *version* of the framework, not the framework as a whole. For example:

```
codesign -s my-signing-identity ../MyCustomFramework/Versions/A
```

Depending on the situation, `codesign` may add to your Mach-O executable file, add extended attributes to it, or create new files in your bundle's Contents directory. None of your other files is modified.

When to Sign

You can run `codesign` at any time on any system running OS X v10.5 or later, provided you have access to the signing identity. You can run it from a shell script phase in Xcode if you like, or as a step in your Makefile scripts, or anywhere else you find suitable. Signing is typically done as part of the product mastering process, after quality assurance work has been done. Avoid signing pre-final copies of your product so that no one can mistake a leaked or accidentally released incomplete version of your product for the real thing.

Your final signing must be done after you are done building your product, including any post-processing and assembly of bundle resources. Code signing detects any change to your program after signing, so if you make any changes at all after signing, your code will be rejected when an attempt is made to verify it. Sign your code before you package the product for delivery.

Because each architecture component is signed independently, it is all right to perform universal-binary operations (such as running the `lipo` command) on signed programs. The result will still be validly signed as long as you make no other changes.

Using the codesign Command

The `codesign` command is fully described in the `codesign(1)` manual page. This section provides some examples of common uses of the command. Note that your signing identity must be in a keychain for these commands to work.

Signing Code

To sign the code located at `<code-path>`, using the signing identity `<identity>`, use the following command:

```
codesign -s <identity> <code-path> …
```

The `<code-path>` value may be a bundle folder or a specific code binary. See What to Sign for more details.

The identity can be named with any (case sensitive) substring of the certificate's common name attribute, as long as the substring is unique throughout your keychains. (Signing identities are discussed in Obtaining a Signing Identity.)

As is typical of Unix-style commands, this command gives no confirmation of success. To get some feedback, include the `-v` option:

```
codesign -s <identity> -v <code-path> …
```

Use the `-r` option to specify an internal requirement. With this option you can specify a text file containing the requirements, a precompiled requirements binary, or the actual requirement text prefixed with an equal sign (=). For example, to add an internal requirement that all libraries be signed by Apple, you could use the following option:

```
-r="library => anchor apple"
```

The code requirement language is described in Code Signing Requirement Language.

If you have built your own certificate hierarchy (perhaps using Certificate Assistant—see Obtaining a Signing Identity), and want to use your certificate’s anchor to form a designated requirement for your program, you could use the following command:

```
codesign -s signing-identity -r="designated => anchor /my/anchor/cert and identifier
com.mycorp.myprog"
```

Note that the requirement source language accepts either an SHA1 hash of a certificate (for example `H"abcd...."`) or a path to the DER encoded certificate in a file. It does not currently accept a reference to the certificate in a keychain, so you have to export the certificate before executing this command.

You can also use the `csreq` command to write the requirements out to a file, and then use the path to that file as the input value for the `-r` option in the `codesign` command. See the manual page for `csreq(1)` for more information on that command.

Here are some other samples of requirements:

- `anchor apple` –the code is signed by Apple
- `anchor trusted` –the anchor is trusted (for code signing) by the system
- `certificate leaf = /path/to/certificate` –the leaf (signing) certificate is the one specified
- `certificate leaf = /path/to/certificate and identifier "com.mycorp.myprog"` –the leaf certificate and program identifier are as specified
- `info[mykey] = myvalue` – the `Info.plist` key `mykey` exists and has the value `myvalue`

Except for the explicit `anchor trusted` requirement, the system does not consult its trust settings database when verifying a code requirement. Therefore, as long as you don't add this designated requirement to your code signature, the anchor certificate you use for signing your code does not have to be introduced to the user's system for validation to succeed.

Adding Entitlements for Sandboxing

If you want to enable App Sandbox for an application, you must add an entitlement property list during the signing process. To do this, add the `--entitlements` flag and an appropriate property list. For example:

```
codesign --entitlements /path/to/entitlements.plist -s <identity> <code-path> …
```

For a list of entitlement keys that can appear in the entitlement property list, see *Entitlement Key Reference*.

Verifying Code

To verify the signature on a signed binary, use the `-v` option with no other options:

```
codesign -v <code-path> …
```

This checks that the code binaries at `<code-path>` are actually signed, that the signature is valid, that all the sealed components are unaltered, and that the whole thing passes some basic consistency checks. It does not by default check that the code satisfies any requirements except its own designated requirement. To check a particular requirement, use the `-R` option. For example, to check that the Apple Mail application is identified as Mail, signed by Apple, and secured with Apple's root signing certificate, you could use the following command:

```
codesign -v -R="identifier com.apple.mail and anchor apple" /Applications/Mail.app
```

Note that, unlike the `-r` option, the `-R` option takes only a single requirement rather than a requirements collection (no `=>` tags). Add one or more additional `-v` options to get details on the validation process.

If you pass a number rather than a path to the verify option, `codesign` takes the number to be the process ID (pid) of a running process, and performs dynamic validation instead.

Getting Information About Code Signatures

To get information about a code signature, use the `-d` option. For example, to output the code signature's internal requirements to standard out, use the following command:

```
codesign -d -r code-path
```

Note that this option does not verify the signature.

Using the spctl Tool to Test Code Signing

The `spctl(8)` tool can be used to test your code signatures against various system policies that the user may set. The basic syntax for code signing assessment is shown below:

```
# Assess an application or tool
spctl --assess --type execute myTool


# Assess an installer package
spctl --assess --type install myInstallerPackage.pkg
```

If your application or package signature is valid, these tools exit silently with an exit status of `0`. (Type `echo $?` to display the exit status of the last command.) If the signature is invalid, these tools print an error message and exit with a nonzero exit status.

For more detailed information about why the assessment failed, you can add the `--verbose` flag. For example:

```
spctl --assess --verbose=4 /bin/ls
```

This prints the following output:

```
    /bin/ls: accepted

    source=Apple System
```

To see *everything* the system has to say about an assessment, pass the `--raw` option. With this flag, the `spctl` tool prints a detailed assessment as a property list.

To whitelist a program (exactly as if the UI did it), type:

```
spctl --add --label mytest /some/program
```

The --label is an optional tag that you can add to your own rules. This tag allows you to remove the rule easily by typing:

```
spctl --remove --label mytest
```

Note that this removes all rules that match the label, which means that it is a handy way to clean up after testing. You can also temporarily suspend your rules by typing:

```
spctl --disable --label mytest
```

and reenable them later by typing:

```
spctl --enable --label mytest
```

To see a list of the current assessment rules, use the `--list` flag. For example:

```
spctl --list --type execute
```

The resulting list of rules might look like this:

```
    3[Apple System] P0 allow execute

        anchor apple

    4[Mac App Store] P0 allow execute

        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.9]
exists

    5[Developer ID] P0 allow execute

        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
and certificate leaf[field.1.2.840.113635.100.6.1.13] exists

    7[UNLABELED] P0 allow execute [/var/tmp/firefly/RUN-FIREFLY-JOBS/test1.app]

        cdhash H"f34c03450da53c07ac69282089b68723327f278a"

    8[UNLABELED] P0 allow execute [/var/tmp/firefly/RUN-FIREFLY-JOBS/test1.app]

        identifier "org.tpatko.Run-Firefly-Job-X-Cores" and certificate root =
H"5056a3983e3b7f44e17e3db8e483b35b6745b236"
```

Notice that the list above includes a number of predefined rules that describe the handling of certain classes of code. For example, rule 5 captures all applications signed by a Developer ID. You can disable those applications by typing:

```
spctl --disable --label "Developer ID"
```

This command tells the system to no longer allow execution of any Developer ID–signed applications that the user has not previously run. This is exactly what happens when you use the preference UI to switch to "Mac App Store only".

Each rule in the list has a unique number that can be used to address it. For example, if you type:

```
spctl --list --label "Developer ID"
```

you might get a list of rules that looks like this:

```
    5[Developer ID] P0 allow execute

        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
and certificate leaf[field.1.2.840.113635.100.6.1.13] exists

    6[Developer ID] P0 allow install

        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
and certificate leaf[field.1.2.840.113635.100.6.1.14] exists
```

Notice that there are separate rules for execution (5) and installation (6), and you can enable and disable them separately. For example, to enable installation of new applications signed with a Developer ID, you can type:

```
spctl --enable --rule 6
```

Finally, `spctl` allows you to enable or disable the security assessment policy subsystem. By default, assessment is turned off, which means that missing or invalid code signatures do not prevent an application from launching. However, it is strongly recommended that you test your application with assessment enabled to ensure that your application works correctly.

To enable or disable assessment, issue one of the following commands.

```
sudo spctl --master-enable   # enables assessment

sudo spctl --master-disable  # disables assessment

spctl --status               # shows whether assessment is enabled
```

For more information, see the manual page for `spctl(8)`.

Shipping and Updating Your Product

The only thing that matters to the code signing system is that the signed code installed on the user's system identical to the code that you signed. It does not matter how you package, deliver, or install your product as long as you don't introduce any changes into the product. Compression, encoding, encrypting, and even binary patching the code are all right as long as you end up with exactly what you started with. You can use any installer you like, as long as it doesn't write anything into the product as it installs it. Drag-installs are fine as well.

When you have qualified a new version of your product, sign it just as you signed the previous version, with the same identifier and the same designated requirement. The user's system will consider the new version of your product to be the same program as the previous version. In particular, the keychain will not distinguish older and newer versions of your program as long as both were signed and the unique Identifier hasn't changed.

You can take a partial-update approach to revising your code on the user's system. To do so, sign the new version as usual, then calculate the differences between the new and the old signed versions, and transmit the differences. Because the differences include the new signature data, the result of installing the changes on the end-user's system will be the newly signed version. You cannot patch a signed application in the field. If you do so, the system will notice that the application has changed and will invalidate the signature, and there is no way to re-validate or resign the application in the

field.

Exhibit 23

Code Signing Requirement Language

When you use the `codesign` command to sign a block of code, you can specify internal requirements; that is, the criteria that you recommend should be used to evaluate the code signature. It is up to the verifier to decide whether to apply the internal requirements or some other set of requirements when deciding how to treat the signed code. You use the code requirement language described in this chapter when specifying requirements to the `codesign` or `csreq` command (see the manual pages for `codesign(1)` and `csreq(1)`).

This chapter describes the requirement language source code. You can compile a set of requirements and save them in binary form using the `csreq` command. You can provide requirements to the `codesign` command either as source code or as a binary file. Both the `codesign` and `csreq` commands can convert a binary requirement set to text. Although there is some flexibility in the source code syntax (for example, quotes can always be used around string constants but are not always required), conversion from binary to text always uses the same form:

- Parentheses are placed (usually only) where required to clarify operator precedence.
- String constants are quoted (usually only) where needed.
- Whether originally specified as constants or through file paths, certificate hashes are always returned as hash constants.
- Comments in the original source are not preserved in the reconstructed text.

Language Syntax

Some basic features of the language syntax are:

- Expressions use conventional infix notation (that is, the operator is placed between the two entities being acted on; for example *quantity* < *constant*).
- Keywords are reserved, but can be quoted to be included as part of ordinary strings.
- Comments are allowed in C, Objective C, and C++.
- Unquoted whitespace is allowed between tokens, but strings containing whitespace must be quoted.
- Line endings have no special meaning and are treated as whitespace.

Evaluation of Requirements

A requirement constitutes an expression without side effects. Each requirement can have any number of subexpressions, each of which is evaluated with a Boolean (succeed-fail) result. There is no defined order of evaluation. The subexpressions are combined using logical operators in the expression to

yield an overall Boolean result for the expression. Depending on the operators used, an expression can succeed even if some subexpressions fail. For example, the expression

```
anchor apple or anchor = "/var/db/yourcorporateanchor.cert"
```

succeeds if either subexpression succeeds—that is, if the code was signed either by Apple or by your company—even though one of the subexpressions is sure to fail.

If an error occurs during evaluation, on the other hand, evaluation stops immediately and the `codesign` or `csreq` command returns with a result code indicating the reason for failure.

Constants

This section describes the use of string, integer, hash-value, and binary constants in the code signing requirement language.

String Constants

String constants must be enclosed by double quotes (`" "`) unless the string contains only letters, digits, and periods (.), in which case the quotes are optional. Absolute file paths, which start with a slash, do not require quotes unless they contain spaces. For example:

```
com.apple.mail                      //no quotes are required
"com.apple.mail"                    //quotes are optional
"My Company's signing identity"     //requires quotes for spaces and apostrophe
/Volumes/myCA/root.crt              //no quotes are required
"/Volumes/my CA/root.crt"           //space requires quotes
"/Volumes/my_CA/root.crt"           //underscore requires quotes
```

It's never incorrect to enclose the string in quotes—if in doubt, use quotes.

Use a backslash to "escape" any character. For example:

```
"one \" embedded quote"             //one " embedded quote
"one \\ embedded backslash"         //one \ embedded backslash
```

There is nothing special about the single quote character (').

Integer Constants

Integer constants are written as decimal constants are in C. The language does not allow radix prefixes (such as `0x`) or leading plus or minus (`+` or `–`) signs.

Hash Constants

Hash values are written either as a hexadecimal number in quotes preceded by an `H`, or as a path to a file containing a binary certificate. If you use the first form, the number must include the exact number of digits in the hash value. A SHA-1 hash (the only kind currently supported) requires exactly 40 digits; for example:

```
H"0123456789ABCDEFFEDCBA98765432100A2BC5DA"
```

You can use either uppercase or lowercase letters (`A..F` or `a..f`) in the hexadecimal numbers.

If you specify a file path, the compiler reads the binary certificate and calculates the hash for you. The compiled version of the requirement code includes only the hash; the certificate file and the path are not retained. If you convert the requirement back to text, you get the hexadecimal hash constant. The file path must point to a file containing an X.509 DER encoded certificate. No container forms (PKCS7, PKCS12) are allowed, nor is the OpenSSL "PEM" form supported.

Variables

There are currently no variables in the requirement language.

Logical Operators

The requirement language includes the following logical operators, in order of decreasing precedence:

- `!` (negation)
- `and` (logical AND)
- `or` (logical OR)

These operators can be used to combine subexpressions into more complex expressions. The negation operator (`!`) is a unary prefix operator. The others are infix operators. Parentheses can be used to override the precedence of the operators.

Because the language is free of side effects, evaluation order of subexpressions is unspecified.

Comparison Operations

The requirement language includes the following comparison operators:

- `=` (equals)
- `<` (less than)
- `>` (greater than)

- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `exists` (value is present)

The value-present (`exists`) operator is a unary suffix operator. The others are infix operators.

There are no operators for non-matches (not equal to, not greater than, and so on). Use the negation operator (`!`) together with the comparison operators to make non-match comparisons.

Equality

All equality operations compare some value to a constant. The value and constant must be of the same type: a string matches a string constant, a data value matches a hexadecimal constant. The equality operation evaluates to `true` if the value exists and is equal to the constant. String matching uses the same matching rules as `CFString` (see *CFString Reference*).

In match expressions (see Info, Part of a Certificate, and Entitlement), substrings of string constants can be matched by using the `*` wildcard character:

- `value = *constant*` is `true` if the value exists and any substring of the value matches the constant; for example:
  - `thunderbolt = *under*`
  - `thunderbolt = *thunder*`
  - `thunderbolt = *bolt*`
- `value = constant*` is `true` if the value exists and begins with the constant; for example:
  - `thunderbolt = thunder*`
  - `thunderbolt = thun*`
- `value = *constant` is `true` if the value exists and ends with the constant; for example:
  - `thunderbolt = *bolt`
  - `thunderbolt = *underbolt`

If the constant is written with quotation marks, the asterisks must be outside the quotes. An asterisk inside the quotation marks is taken literally. For example:

- `"ten thunderbolts" = "ten thunder"*` is `true`
- `"ten thunder*bolts" = "ten thunder*"*` is `true`
- `"ten thunderbolts" = "ten thunder*"` is `false`

Inequality

Inequality operations compare some value to a constant. The value and constant must be of the same type: a string matches a string constant, a data value matches a hexadecimal constant. String comparisons use the same matching rules as `CFString` with the `kCFCompareNumerically` option flag; for example, `"17.4"` is greater than `"7.4"`.

Existence

The existence operator tests whether the value exists. It evaluates to `false` only if the value does not exist at all or is exactly the Boolean value `false`. An empty string and the number `0` are considered to exist.

Constraints

Several keywords in the requirement language are used to require that specific certificates be present or other conditions be met.

Identifier

The expression

`identifier =` *constant*

succeeds if the unique identifier string embedded in the code signature is exactly equal to *constant*. The equal sign is optional in identifier expressions. Signing identifiers can be tested only for exact equality; the wildcard character (`*`) can not be used with the identifier constraint, nor can identifiers be tested for inequality.

Info

The expression

`info [`*key*`]`*match expression*

succeeds if the value associated with the top-level key in the code's `info.plist` file matches *match expression*, where *match expression* can include any of the operators listed in Logical Operators and Comparison Operations. For example:

```
info [CFBundleShortVersionString] < "17.4"
```

or

```
info [MySpecialMarker] exists
```

You must specify *key* as a string constant.

If the value of the specified key is a string, the match is applied to it directly. If the value is an array, it must be an array of strings and the match is made to each in turn, succeeding if any of them matches. Substrings of string constants can be matched by using any match expression (see Comparison Operations).

If the code has no `info.plist` file, or the `info.plist` does not contain the specified key, this expression evaluates to `false` without returning an error.

Certificate

Certificate constraints refer to certificates in the certificate chain used to validate the signature. Most uses of the `certificate` keyword accept an integer that indicates the position of the certificate in the chain: positive integers count from the leaf (0) toward the anchor. Negative integers count backward from the anchor (-1). For example, `certificate 1` is the intermediate certificate that was used to sign the leaf (that is, the signing certificate), and `certificate -2` indicates the certificate that was directly signed by the anchor. Note that this convention is the same as that used for array indexing in the Perl and Ruby programming languages:

Anchor	First intermediate	Second intermediate	Leaf
`certificate 3`	`certificate 2`	`certificate 1`	`certificate 0`
`certificate -1`	`certificate -2`	`certificate -3`	`certificate -4`

Other keywords include:

- `certificate root`—the anchor certificate; same as certificate 0
- `anchor`—same as `certificate root`
- `certificate leaf`—the signing certificate; same as `certificate -1`

Note: The short form `cert` is allowed for the keyword `certificate`.

If there is no certificate at the specified position, the constraint evaluates to `false` without returning an error.

If the code was signed using an ad-hoc signature, there are no certificates at all and all certificate constraints evaluate to `false`. (An ad-hoc signature is created by signing with the pseudo-identity `-` (a dash). An ad-hoc signature does not use or record a cryptographic identity, and thus identifies exactly and only the one program being signed.)

If the code was signed by a self-signed certificate, then the leaf and root refer to the same single certificate.

Whole Certificate

To require a particular certificate to be present in the certificate chain, use the form

`certificate` *position* `=` *hash*

or one of the equivalent forms discussed above, such as `anchor =` *hash*. Hash constants are described in Hash Constants.

For Apple's own code, signed by Apple, you can use the short form

```
anchor apple
```

For code signed by Apple, including code signed using a signing certificate issued by Apple to other developers, use the form

```
anchor apple generic
```

Part of a Certificate

To match a well-defined element of a certificate, use the form

`certificate` *position*`[`*element*`]` *match expression*

where *match expression* can include the `*` wildcard character and any of the operators listed in Logical Operators and Comparison Operations. The currently supported elements are as follows:

Note: Case is significant in element names.

Element name	Meaning	Comments
`subject.CN`	Subject common name	Shown in Keychain Access utility
`subject.C`	Subject country name	
`subject.D`	Subject description	
`subject.L`	Subject locality	
`subject.O`	Subject organization	Usually company or organization
`subject.OU`	Subject organizational unit	
`subject.STREET`	Subject street address	

Certificate field by OID

To check for the existence of any certificate field identified by its X.509 object identifier (OID), use the form

`certificate` *position* `[field.`*OID*`] exists`

The object identifier must be written in numeric form (*x*`.`*y*`.`*z*...) and can be the OID of a certificate extension or of a conventional element of a certificate as defined by the CSSM standard (see Chapter 31 in *Common Security: CDSA and CSSM*, version 2 (with corrigenda) by the Open Group (http://www.opengroup.org/security/cdsa.htm)).

Trusted

The expression

`certificate` *position* `trusted`

succeeds if the certificate specified by *position* is marked trusted for the code signing certificate policy in the system's Trust Settings database. The *position* argument is an integer or keyword that indicates the position of the certificate in the chain; see the discussion under Certificate.

The expression

`anchor trusted`

succeeds if any certificate in the signature's certificate chain is marked trusted for the code signing certificate policy in the system's Trust Settings database, provided that no certificate closer to the leaf certificate is explicitly untrusted.

Thus, using the `trusted` keyword with a certificate position checks only the specified certificate, while using it with the `anchor` keyword checks all the certificates, giving precedence to the trust setting found closest to the leaf.

> **Important:** The syntax `anchor trusted` is *not* a synonym for `certificate anchor trusted`. Whereas the former checks all certificates in the signature, the latter checks only the anchor certificate.

Certificates can have per-user trust settings and system-wide trust settings, and trust settings apply to specific policies. The `trusted` keyword in the code signing requirement language causes trust to be checked for the specified certificate or certificates for the user performing the validation. If there are no settings for that user, then the system settings are used. In all cases, only the trust settings for the code-signing policy are checked. Policies and trust are discussed in *Certificate, Key, and Trust Services Programming Guide*.

> **Important:** If you do not include an expression using the `trusted` keyword in your code signing requirement, then the verifier does not check the trust status of the certificates in the code signature at all.

Entitlement

The expression

`entitlement [`*key*`]` *match expression*

succeeds if the value associated with the specified key in the signature's embedded entitlement dictionary matches *match expression*, where *match expression* can include the `*` wildcard character and any of the operators listed in Logical Operators and Comparison Operations. You must specify *key* as a string constant. The entitlement dictionary is included in signatures for certain platforms.

Code Directory Hash

The expression

`cdhash` *hash-constant*

computes a SHA-1 hash of the program's CodeDirectory resource and succeeds if the value of this hash exactly equals the specified hash constant.

The CodeDirectory resource is the master directory of the contents of the program. It consists of a versioned header followed by an array of hashes. This array consists of a set of optional special hashes for other resources, plus a vector of hashes for pages of the main executable. The CodeSignature and CodeDirectory resources together make up the signature of the code.

You can use the codesign utility with (at least) three levels of verbosity to obtain the hash constant of a program's CodeDirectory resource:

```
$ codesign -dvvv /bin/ls
...
CodeDirectory v=20001 size=257 flags=0x0(none) hashes=8+2 location=embedded
CDHash=4bccbc576205de37914a3023cae7e737a0b6a802
...
```

Because the code directory changes whenever the program changes in a nontrivial way, this test can be used to unambiguously identify one specific version of a program. When the operating system signs an otherwise unsigned program (so that the keychain or Parental Controls can recognize the program, for example), it uses this requirement.

Requirement Sets

A requirement set is a collection of distinct requirements, each indexed (tagged) with a type code. The expression

tag `=>` *requirement*

applies *requirement* to the type of code indicated by *tag*, where possible tags are

- `host`—this requirement is applied to the direct host of this code module; each code module in the hosting path can have its own host requirement, where the hosting path is the chain of code signing hosts starting with the most specific code known to be running, and ending with the root of trust (the kernel)
- `guest`—this requirement is applied to each code module that is hosted by this code module
- `library`—this requirement is applied to all libraries mounted by the signed code
- `designated`—this is an explicitly specified designated requirement for the signed code; if there is no explicitly specified designated requirement for the code, then there is no `designated` internal requirement

The primary use of requirement sets is to represent the internal requirements of the signed code. For example:

```
    codesign -r='host => anchor apple and identifier com.apple.perl designated =>
anchor /my/root and identifier com.bar.foo'
```

sets the internal requirements of some code, having a host requirement of `anchor apple and identifier com.apple.perl` ("I'm a Perl script and I want to be run by Apple's Perl interpreter") and an explicit designated requirement of `anchor /my/root and identifier com.bar.foo`. Note that this command sets no guest or library requirements.

You can also put the requirement set in a file and point to the file:

```
    codesign -r myrequirements.rqset
```

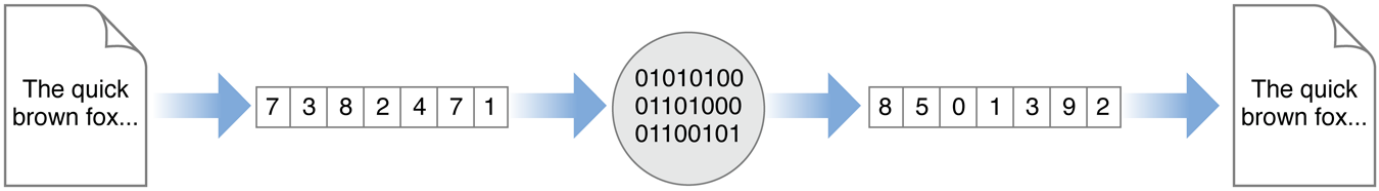
where the file `myrequirements.rqset` might contain:

```
    //internal requirements

      host => anchor apple and identifier com.apple.perl //require Apple's Perl
interpreter

      designated => anchor /my/root and identifier com.bar.foo
```

Exhibit 24

About Cryptographic Services

OS X and iOS provide a number of technologies that provide cryptographic services—encryption and decryption, hashing, random number generation, secure network communication, and so on. These technologies can be used to secure data at rest (when stored on your hard drive or other media), secure data in transit, determine the identity of a third party, and build additional security technologies.



At a Glance

Some of the cryptographic services provided by iOS and OS X include:

- Encryption and decryption (both general-purpose and special-purpose)
- Key management using keychains
- Cryptographically strong random number generation
- Secure communication (SSL and TLS)
- Secure storage using FileVault and iOS File Protection

Encryption, Signing and Verifying, and Digital Certificates Can Protect Data from Prying Eyes

There are two main types of encryption: symmetric encryption, in which a single shared key is used for encrypting and decrypting data, and asymmetric encryption, in which you use one key to encrypt data and a separate (but related) key to decrypt the data. You can use a hash to detect modifications to a piece of data. You can combine hashes with asymmetric keys to create a digital signature that, when verified against a digital certificate, proves the source of a piece of data. Digital certificates, in turn, are verified by verifying the signature of the party that signed the certificate, then verifying that party's certificate, and so on until you reach a certificate that you trust inherently, called an *anchor certificate*.

Relevant Chapter: Cryptography Concepts In Depth

OS X and iOS Provide Encryption and Hashing APIs

OS X and iOS provide a number of APIs for encrypting and hashing data, including Keychain Services;

Cryptographic Message Syntax Services; Certificate, Key, and Trust Services; Common Crypto; and Security Transforms.

Relevant Chapter: Encrypting and Hashing Data

Keychains Help You Store Secret Information

If your app must store encryption keys, passwords, certificates, and other security-related information, it should use a keychain. Keychains provide secure storage for small pieces of information so that is not accessible by other apps running on the system, and so that it is accessible only after the user has logged in or unlocked the device. OS X and iOS provide two APIs for working with the keychain and keys obtained from the keychain: the Certificate, Key, and Trust Services API and the Keychain Services API.

Relevant Chapter: Managing Keys, Certificates, and Passwords

OS X and iOS Provide Cryptographically Secure Random Number Generators

Some cryptographic tasks require you to generate cryptographically strong pseudorandom numbers. OS X can provide these numbers through the `/dev/random` device node. iOS can provide these numbers through the Randomization Services API.

Relevant Chapter: Generating Random Numbers

OS X and iOS Provide Secure Network Communication APIs

Transmitting data securely requires a secure communications channel. OS X and iOS provide a number of APIs for establishing secure communications channels, including the URL Loading System, socket streams in Core Foundation and Foundation, and Secure Transport.

Relevant Chapter: Transmitting Data Securely

Deprecated Technologies

Although the CDSA and CSSM API is deprecated in OS X v10.7 and later, you may still need to use it in a few situations. For this reason, its documentation is provided as an appendix.

Relevant Chapter: CDSA Overview

Prerequisites

Before reading this document, you should be familiar with the concepts in *Security Overview* and *Secure Coding Guide*.

See Also

For more information about OS X authentication and authorization (built on top of encryption technologies), read *Authentication, Authorization, and Permissions Guide*.

Exhibit 25

Cryptography Concepts In Depth

The word cryptography (from Greek *kryptos*, meaning hidden) at its core refers to techniques for making data unreadable to prying eyes. However, cryptography can also be used for other purposes. Cryptography includes a range of techniques that can be used for verifying the authenticity of data (detecting modifications), determining the identity of a person or other entity, determining who sent a particular message or created a particular piece of data, sending data securely across a network, locking files securely behind a password or passphrase, and so on.

This chapter describes a number of these techniques, beginning with basic encryption, then moving on to other cryptographic constructs built on top of it.

Note: This chapter repeats many of the concepts in *Security Overview*, but with additional detail and depth. You may find it helpful to read that document before reading this chapter.

What Is Encryption?

Encryption is 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 (cleartext or plaintext) form is called *decryption*. Most of the security APIs in 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.

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 only needs to 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.

Types of Encryption

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 secure transport protocols), and cryptographic hash functions.

Symmetric Keys

Symmetric key cryptography (also called *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 the Caesar cipher, the key used to encrypt and decrypt the message is simply the number of places by which the alphabet is rotated and the direction of that rotation. 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 who 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 (for example, Diffie-Hellman key exchange, described later in this chapter), a more practical solution for use in computer communications came about with the invention of practical algorithms for asymmetric key cryptography.

Asymmetric Keys

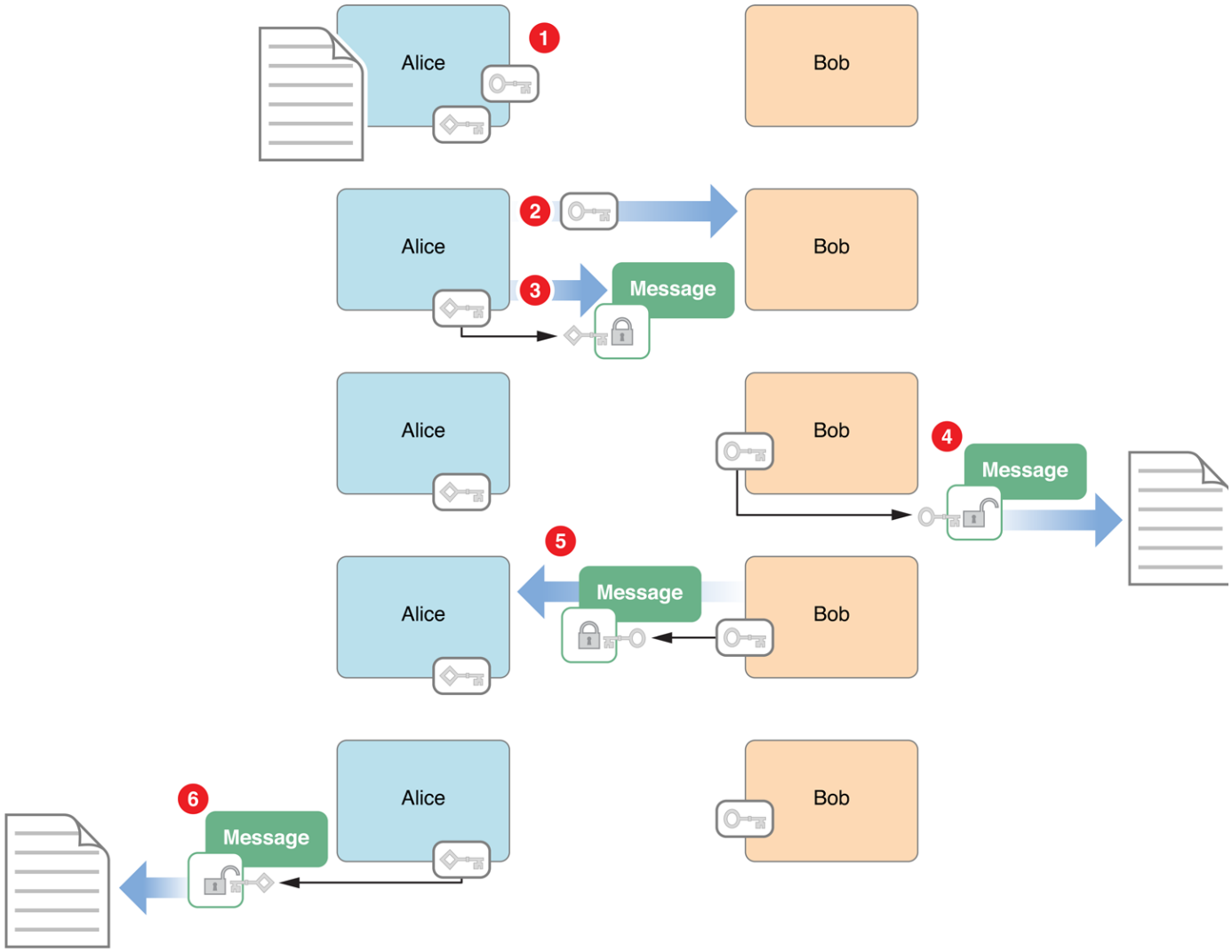
In asymmetric key cryptography, different keys are used for encrypting and decrypting a message. The asymmetric key algorithms that are most useful are those in which neither key can be deduced from the other. In that case, one key can be made public while the other is kept secure. This arrangement is often referred to as *public key cryptography*, and provides some distinct advantages over symmetric encryption: 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 1-1.

Figure 1-1 Asymmetric key encryption



The secure message exchange illustrated in Figure 1-1 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*, in which 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, digital certificates and digital signatures can help address these security problems. These techniques are described later in this chapter.

Diffie-Hellman Key Exchange

The *Diffie-Hellman key exchange* protocol is a way for two ends of a communication session to generate a shared symmetric key securely over an insecure channel. Diffie-Hellman is usually implemented using mathematics similar to RSA public key encryption. However, a similar technique can also be used with elliptic curve encryption. The basic steps are listed below:

1. Alice and Bob exchange public keys.
    - For RSA, these keys must have the same modulo portion, *p*.
    - For elliptic curve encryption, the domain parameters used for encryption must be agreed upon.

    As a rule, you should use the modulo or domain parameter values specified in RFC 5114.
2. Alice and Bob each encrypt a shared, non-secret value, *g*, using their private keys, and they exchange these encrypted values.

    The value for *g* is also usually taken from RFC 5114, but if another value is chosen when using RSA, the value for *g* must be a primitive root mod *p*—that is, any number that shares no common divisors with *p* other than 1, is congruent to a power of *g* mod *p*.
3. Alice encrypts the encrypted value received from Bob with her private key, and vice versa. These values are used as a shared session key.

At this point, 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 resulting 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, Bob knows the message has not been corrupted or tampered with.

The most common hash function you will use is SHA-1, an algorithm developed and published by the U.S. Government that produces a 160-bit hash value from any data up to 2**64 bits in length. There are also a number of more exotic algorithms such as SHA-2, elliptic-curve-based algorithms, and so on.

For compatibility with existing systems and infrastructure, you may occasionally need to use older algorithms such as MD5, but they are not recommended for use in new designs because of known weaknesses.

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.

Before a signer can create a digital signature, the signer must first have a digital *identity*—a public-private key pair and a corresponding digital certificate that proves the authenticity of the signer's public key.

The signer generates a message digest of the data and then uses the private key to encrypt the digest. The signer includes the encrypted digest and information about the signer's digital certificate along with the message. The combination of the encrypted digest and the digital certificate is a digital signature.

The certificate can later be used by the recipient to verify the signature; the certificate 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 same algorithm to create a digest of the message as received, then uses the public key to decrypt the encrypted digest from the message 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 he or she claims to be, the certificate is also signed—in this case by the certification authority who issued the certificate. (More on certification authorities later.)

Digital signatures play a key role in code signing. Developers are encouraged to sign their apps. On execution, each app's signature is checked for validity. Digital signatures are required on all apps for iOS. Read *Code Signing Guide* for details about how code signing is used by OS X and iOS.

Figure 1-2 illustrates the creation of a digital signature.

Figure 1-2 Creating a digital signature

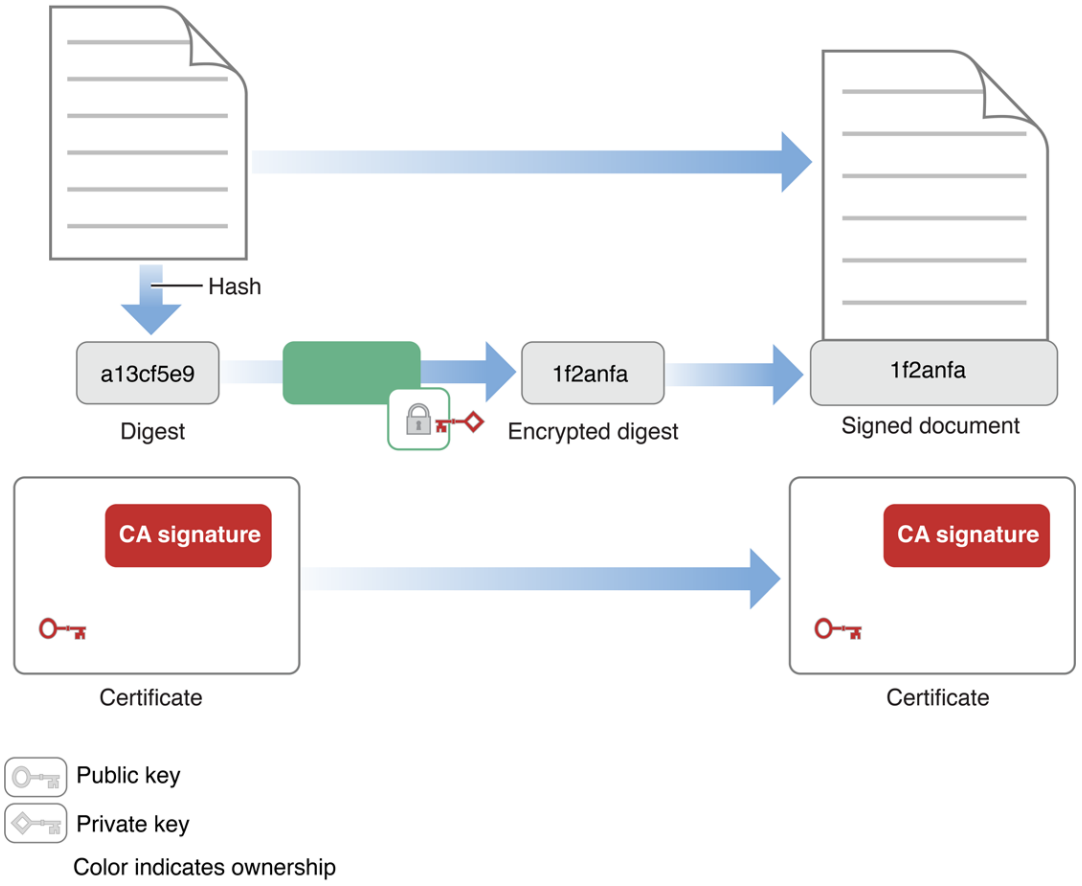
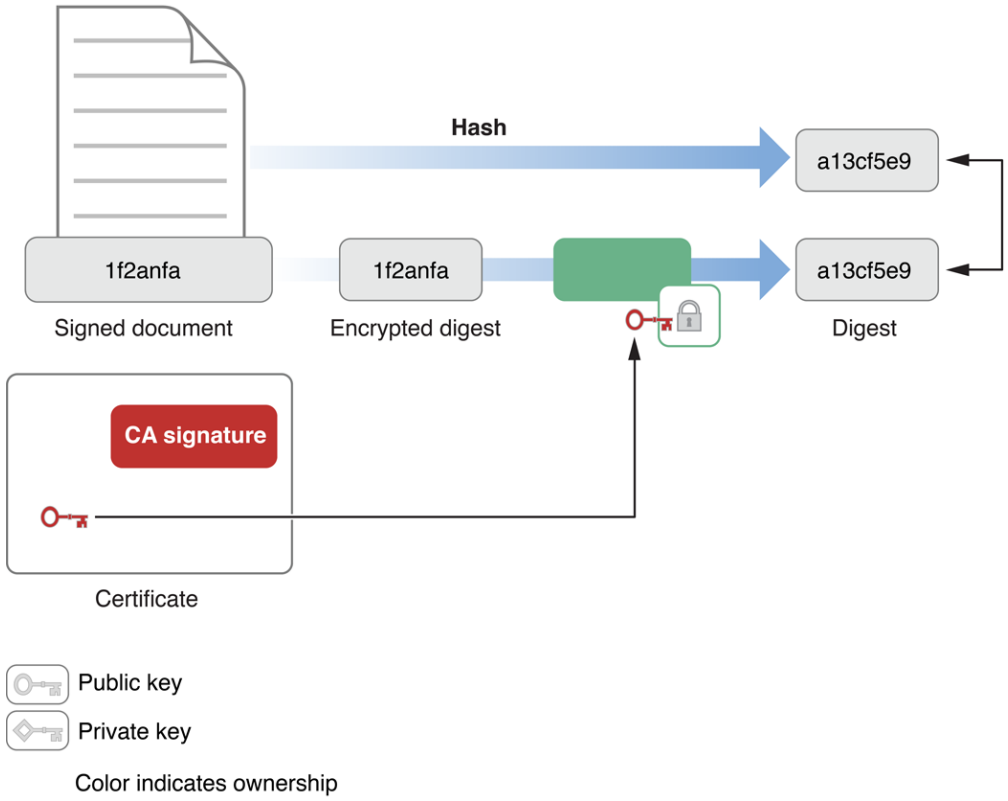


Figure 1-3 illustrates the verification of a digital signature. The recipient gets the signer's public key from the signer's certificate and uses that to decrypt the digest. Then, using the algorithm indicated in the certificate, the recipient creates a new digest of the data and compares the new digest 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 1-3 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.

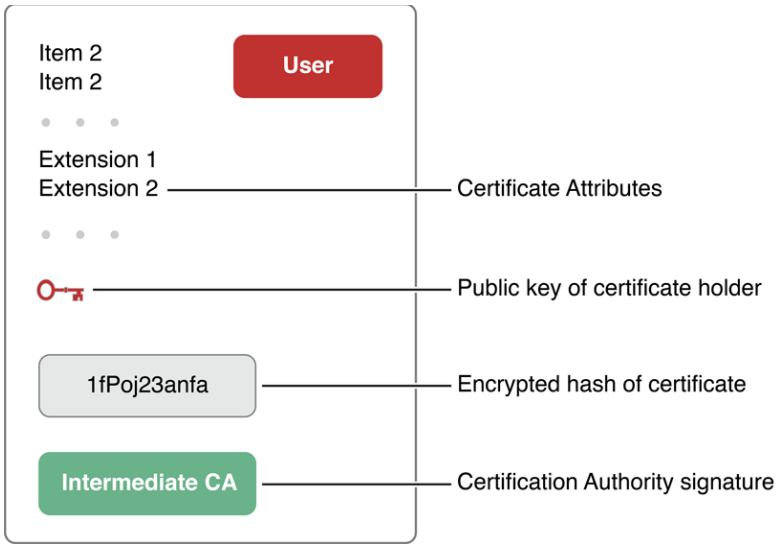
For example, an X.509 certificate contains such information as:

- Structural information—version, serial number, the message digest algorithm used to create the signature, and so on
- A digital signature from a *certification authority* (CA)—a person or organization that issued the certificate—to ensure that the certificate has not been altered and to indicate the identity of the issuer
- Information about the certificate holder—name, email address, company name, the owner's public key, and so on
- Validity period (the certificate is not valid before or after this period)
- *Certificate extensions*—attributes that contain additional information such as allowable uses for this certificate

The careful reader will have noticed that a digital signature includes the certificate of the signer, and that the signer's certificate, in turn, contains a digital signature that includes another certificate.

In general, each certificate is verified through the use of another certificate, creating a *chain of trust*—a chain of certificates, each of which is digitally signed by the next certificate in the chain, ending with a *root certificate*. The owner of this root certificate is called the *root certification authority*. Figure 1-4 illustrates the parts of a digital certificate.

Figure 1-4 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 1-5 and Figure 1-6 illustrate how a chain of certificates is created and used. Figure 1-5 shows how the root certification authority creates its own certificate and then creates a certificate for a secondary certification authority.

Figure 1-5 Creating the certificates for the root CA and a secondary CA

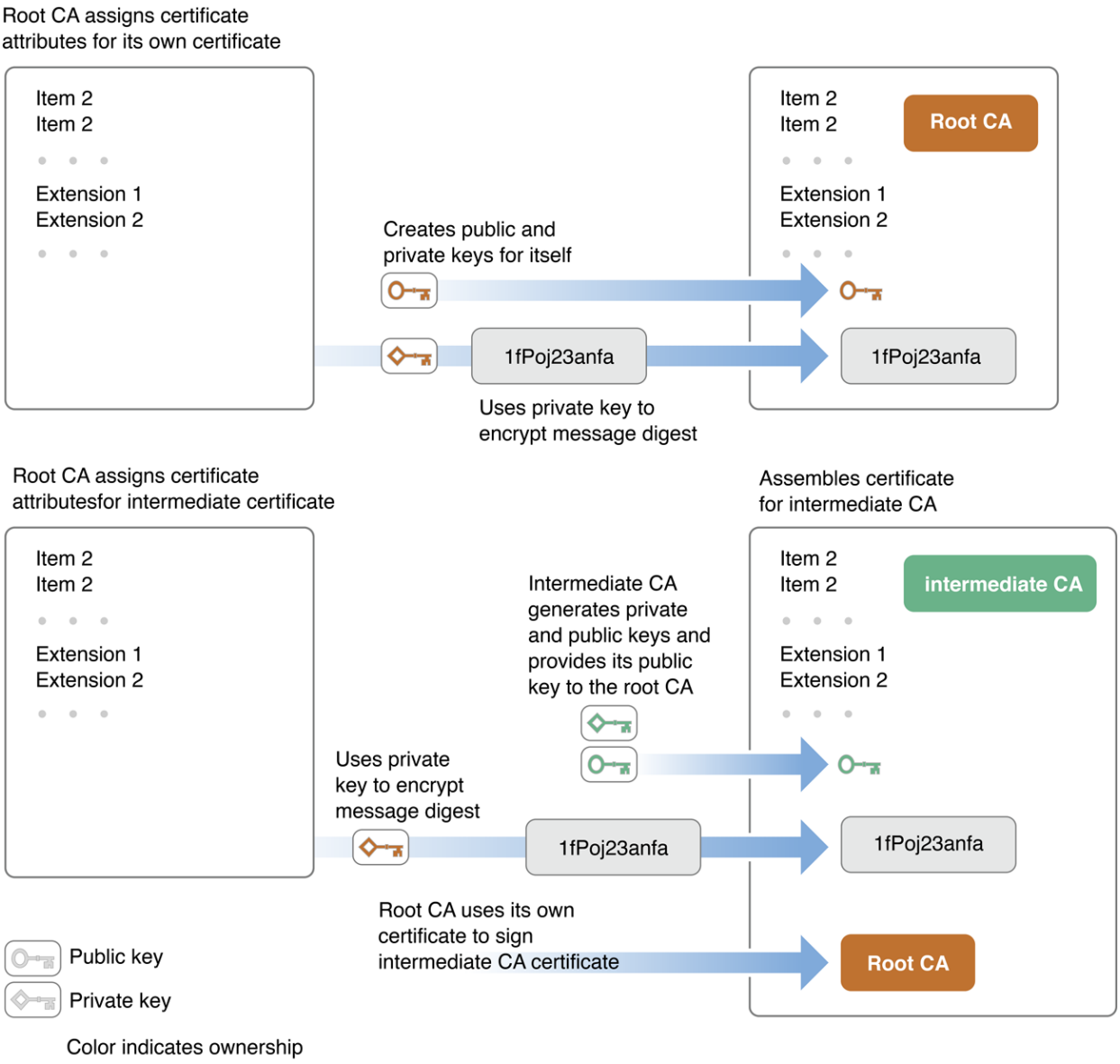
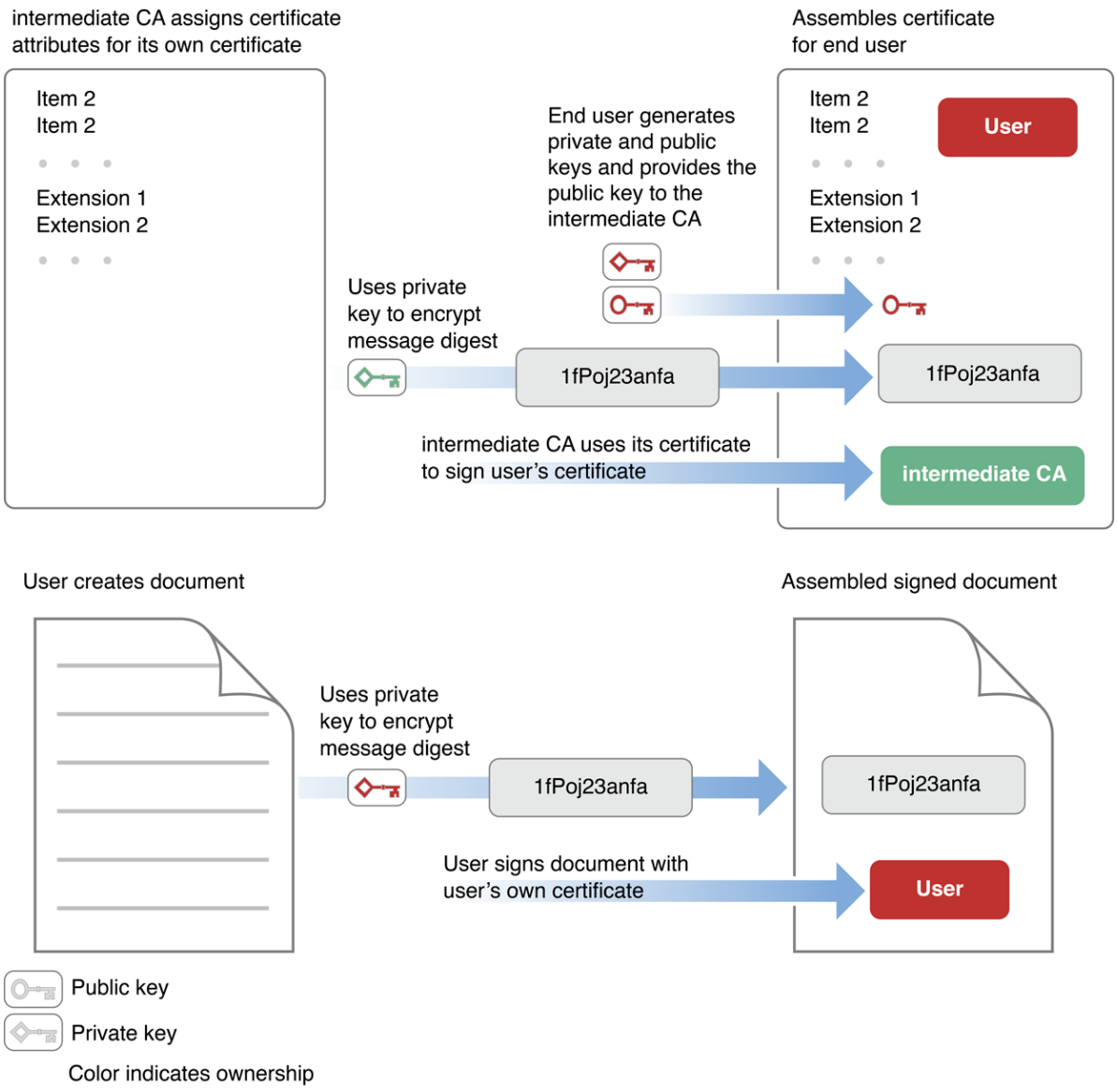


Figure 1-6 shows how the secondary certification authority creates a certificate for an end user and how the end user uses it to sign a document.

Figure 1-6 Creating the certificate for an end user and signing a document with it



In Figure 1-6, 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 (described earlier in Digital Signatures). 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 1-5 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 app 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 1-5.

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 app 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 OS X and iOS, all certificates you receive are stored in your keychain. In OS X, you can use the Keychain Access utility to view them.

Certain attributes of a digital certificate (known as *certificate extensions*) provide additional information about the certificate. Some of these extensions describe how the certificate was intended to be used. For example, a certificate extension might indicate that a key can be used for code signing, or might provide a list of additional domain names for which a TLS certificate is valid. Other extensions provide signed time stamps indicating when the certificate was used to sign a particular document, thus allowing you to verify that a now-expired certificate was valid when it was used to sign the document. Still others provide information used for checking whether a certificate has been revoked. And so on.

These certificate extensions are interpreted in the context of a *trust policy*—a set of rules that specify how a particular extension affects whether the certificate should be trusted for a given use. For example, a trust policy might specify that in order to be trusted to verify a digitally signed email message, a certificate must contain an email address that matches the address of the sender of the email.

Exhibit 26

Encrypting and Hashing Data

Both symmetric and asymmetric key encryption schemes can be used to encrypt data. Asymmetric encryption is most commonly used for sending data across trust boundaries, such as one person sending another person an encrypted email. It is also often used for sending a symmetric session key across an insecure communication channel so that symmetric encryption can then be used for future communication. Symmetric encryption is most commonly used for data at rest (on your hard drive for example) and as a session key in a number of encrypted networking schemes.

OS X and iOS provide a number of different APIs for encryption and decryption. This chapter describes the recommended APIs.

Encryption Technologies Common to iOS and OS X

OS X and iOS provide a number of encryption technologies. Of these, three APIs are available on both iOS and OS X:

- Keychain Services API—provides secure storage for passwords, keys, and so on
- Cryptographic Message Syntax—provides (nonstreaming) symmetric and asymmetric encryption and decryption
- Certificate, Key, and Trust Services—provides cryptographic support services and trust validation

The sections that follow describe these technologies.

Keychain Services

The Keychain Services API is commonly used to store passwords, keys, certificates, and other secrets in a special encrypted file called a keychain. You should always use the keychain to store passwords and other short pieces of data (such as cookies) that are used to grant access to secure web sites, as otherwise this data might be compromised if an unauthorized person gains access to a user's computer, mobile device, or a backup thereof.

Although this is mostly used for storing passwords and keys, the keychain can also store small amounts of arbitrary data. The keychain is described further in Managing Keys, Certificates, and Passwords.

OS X also includes a utility that allows users to store and read the data in the keychain, called *Keychain Access*. For more information, see Keychain Access in *Security Overview*.

Cryptographic Message Syntax Services

The Cryptographic Message Syntax Services programming interface allows you to encrypt or add a digital signature to S/MIME messages. (S/MIME is a standard for encrypting and signing messages, most commonly used with email.) It is a good API to use when signing or encrypting data for store-

and-forward applications, such as email. See *Cryptographic Message Syntax Services Reference* for details.

Certificate, Key, and Trust Services

The Certificate, Key, and Trust Services API provides trust validation and support functions for cryptography. These features are described further in Managing Keys, Certificates, and Passwords.

In iOS, this API also provides basic encryption capabilities, as described in Encryption in iOS.

Common Crypto

In OS X v10.5 and later and iOS 5.0 and later, Common Crypto provides low-level C support for encryption and decryption. Common Crypto is not as straightforward as Security Transforms, but provides a wider range of features, including additional hashing schemes, cipher modes, and so on.

For more information, see the manual page for `CommonCrypto`.

Encryption Technologies Specific to OS X

In addition to Keychain Services and Cryptographic Message Syntax Services, OS X provides four additional APIs for performing encryption:

- Security Transforms API—a Core-Foundation-level API that provides support for signing and verifying, symmetric cryptography, and Base64 encoding and decoding
- Common Crypto—a C-level API that can perform most symmetric encryption and decryption tasks
- CDSA/CSSM—a legacy API that should be used only to perform tasks not supported by the other two APIs, such as asymmetric encryption

These APIs are described in the sections that follow.

Security Transforms

In OS X v10.7 and later, the Security Transforms API provides efficient and easy-to-use support for performing cryptographic tasks. Security transforms are the recommended way to perform symmetric encryption and decryption, asymmetric signing and verifying, and Base64 encoding and decoding in OS X.

Based on the concept of data flow programming, the Security Transforms API lets you construct graphs of transformations that feed into one another, transparently using Grand Central Dispatch to schedule the resulting work efficiently across multiple CPUs. As the `CFDataRef` (or `NSData`) objects pass through the object graph, callbacks within each individual transform operate on that data, then pass it on to the transform's output, which may be connected to the input of another transform object, and so on.

The transform API also provides a file reader transform (based on `CFReadStreamRef` or `NSInputStream` objects) that can be chained to the input of other transforms.

Using the built-in transforms, the Security Transforms API allows you to read files, perform symmetric encryption and decryption, perform asymmetric signing and verifying, and perform Base64 encoding. The Security Transforms API also provides support for creating custom transforms that perform other operations on data. For example, you might create a transform that byte swaps data prior to encrypting it or a transform that encodes the resulting encrypted data for transport.

For more information, read *Security Transforms Programming Guide* and *Security Transforms Reference*.

CDSA/CSSM

Important: CDSA (including CSSM) is deprecated and should not be used for new development. It is not available in iOS.

CDSA is an Open Source security architecture adopted as a technical standard by the Open Group. Apple has developed its own Open Source implementation of CDSA, available as part of Darwin at Apple's Open Source site. This API provides a wide array of security services, including fine-grained access permissions, authentication of users' identities, encryption, and secure data storage.

Although CDSA has its own standard programming interface, it is complex and does not follow standard Apple programming conventions. For this reason, the CDSA API is deprecated as of OS X version 10.7 (Lion) and is not available in iOS. Fortunately, OS X and iOS include their own higher-level security APIs that abstract away much of that complexity.

Where possible, you should use one of the following instead of using CDSA directly:

- The Security Objective-C API for authentication (in OS X). See Security Objective-C API in *Security Overview* for details.
- The Security Transforms API for symmetric encryption and decryption, asymmetric signing and verifying, and other supported tasks in OS X v10.7 and later. See Security Transforms for details.
- The Certificate, Key, and Trust Services API for general encryption, key management, and other tasks. See Encryption in iOS for details.

If these APIs do not meet your needs, you can still use CDSA in OS X, but please file bugs at http://bugreport.apple.com/ to request the additional functionality that you need. For more information, read CDSA Overview.

OpenSSL

Although OpenSSL is commonly used in the open source community, OpenSSL does not provide a stable API from version to version. For this reason, although OS X provides OpenSSL libraries, the OpenSSL libraries in OS X are deprecated, and OpenSSL has never been provided as part of iOS. Use of the OS X OpenSSL libraries by apps is strongly discouraged.

If your app depends on OpenSSL, you should compile OpenSSL yourself and statically link a known version of OpenSSL into your app. This use of OpenSSL is possible on both OS X and iOS. However, unless you are trying to maintain source compatibility with an existing open source project, you should generally use a different API.

Common Crypto and Security Transforms are the recommended alternatives for general encryption. CFNetwork and Secure Transport are the recommended alternatives for secure communications.

Encryption in iOS

In iOS, in addition to providing support functions for encoding and decoding keys, the Certificate, Key, and Trust Services API also provides basic encryption, decryption, signing, and verifying of blocks of data using the following `SecKey` functions:

`SecKeyEncrypt`—encrypts a block of data using the specified key.

`SecKeyDecrypt`—decrypts a block of data using the specified key.

`SecKeyRawSign`—signs a block of data using the specified key.

`SecKeyRawVerify`—verifies a signature against a block of data and a specified key.

You can find examples of how to use these functions in Certificate, Key, and Trust Services Tasks for iOS in *Certificate, Key, and Trust Services Programming Guide*.

For detailed reference content, read *Certificate, Key, and Trust Services Reference*.

Exhibit 27

Managing Keys, Certificates, and Passwords

The *keychain* provides storage for passwords, encryption keys, certificates, and other small pieces of data. After an app requests access to a keychain, it can store and retrieve sensitive data, confident that untrusted apps cannot access that data without explicit action by the user.

In OS X, the user is prompted for permission when an app needs to access the keychain; if the keychain is locked, the user is asked for a password to unlock it.

In iOS, an app can access only its own items in the keychain—the user is never asked for permission or for a password.

There are two recommended APIs for accessing the keychain:

- Certificate, Key, and Trust Services
- Keychain Services

Certificate, Key, and Trust Services

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

- Create certificates and asymmetric keys
- Add certificates and keys to keychains, remove them from keychains, and use keys to encrypt and decrypt data
- Retrieve information about a certificate, such as the private key associated with it, the owner, and so on
- Convert certificates to and from portable representations
- Create and manipulate trust policies and evaluate a specific certificate using a specified set of trust policies
- Add anchor certificates

In OS X, functions are also available to retrieve anchor certificates and set user-specified settings for trust policies for a given certificate.

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.

Because certificates are used by SSL and TLS for authentication, the Secure Transport API includes a variety of functions to manage the use of certificates and root certificates in a secure connection.

To display the contents of a certificate in an 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.

Keychain Services

In OS X and iOS, Keychain Services allows you to create keychains, add, delete, and edit keychain items, and—in OS X only—manage collections of keychains. In most cases, a keychain-aware app does not have to do any keychain management and only has to call a few functions to store or retrieve passwords.

By default, backups of iOS data are stored in cleartext, 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. 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 OS X, the Keychain Access application provides a user interface to the keychain. See Keychain Access in *Security Overview* for more information about this application.

To Learn More

For more information about using Keychain Services to store and retrieve secrets and certificates, read *Keychain Services Programming Guide* and *Keychain Services Reference*.

For more information about Secure Transport, read Secure Transport.

For more information about the certificate user interface API, read Security Objective-C API in *Security Overview*.

Exhibit 28

Glossary

anchor certificate A digital certificate trusted to be valid, which can then be used to verify other certificates. An anchor certificate can be a root certificate, a cross-certified certificate (that is, a certificate signed with more than one certificate chain), or a locally defined source of trust.

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, OS X includes its own security APIs that call the CDSA API for you.

certificate See digital certificate.

certificate chain See chain of trust.

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.

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.

chain of trust A set of digital certificates in which each certificate signs the next certificate, ending in a root certificate that is also a trusted anchor certificate. A chain of trust can be used to verify the validity of a digital certificate.

cipher A scheme for encrypting data.

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

cleartext Ordinary, unencrypted data. Compare ciphertext.

cryptographic hashing The process whereby data is transformed into a much smaller value that can take the place of the original data for cryptographic purposes. A hashing algorithm 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 similar input data. See also message digest.

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 encrypted data back into the original cleartext. Compare

encryption.

Diffie-Hellman key exchange A protocol that provides a way for two ends of a communication session to generate a symmetric shared secret key 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. OS X and iOS support the X.509 standard for digital certificates. See also certificate chain.

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 (cleartext) form is called decryption.

hash algorithm See cryptographic hashing.

identity A digital certificate together with an associated private key.

keychain A database in 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. Apps can use the Keychain Services API (or the legacy Keychain Manager API) to manipulate data in the keychain. Users can also access keychain data using the Keychain Access utility.

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.

plaintext See cleartext.

private key A cryptographic key that must be kept secret, usually used in the context of public key cryptography. Although this term can also be used in the context of symmetric key cryptography, the term "secret key" (or "shared secret") is preferred.

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-private key pair A pair of mathematically related keys that cannot be derived from one another used in public key cryptography. One of these keys (the public key) is made public while the other (the private key) is kept secure. Data encrypted with one key must be decrypted with the other.

public key A cryptographic key that can be shared or made public without compromising the cryptographic method—generally, the public portion of a public-private key pair. See also public key cryptography.

public key certificate See digital certificate.

public key cryptography A cryptographic method using a public-private key pair. 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, such data can be used for authentication. See also digital certificate; digital signature. Compare symmetric key cryptography.

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 certification authority that owns the root certificate.

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 a public-private key pair from the prime numbers and their product. The strength of this algorithm depends on the difficulty of factoring the resulting product and upon reasonable assurance of the primality of the values used in constructing the keys.

secret key A cryptographic key that cannot be made public without compromising the security of the cryptographic method. In symmetric key cryptography, a secret key is used both to encrypt and decrypt data, and is often called a *shared secret*. Although the term “secret key” can be used in the context of public key cryptography, the term “private key” is preferred.

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 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. 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.

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 SSL and Kerberos protocols, and are often obtained using Diffie-Hellman key exchange.

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.

symmetric key cryptography Cryptography that uses a single shared key to encrypt and decrypt data. See also secret key. Compare public key cryptography.

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 certificates for authentication and 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 policy A set of rules that specify the appropriate uses for a certificate based on its certificate extensions and other trust criteria. For example, a standard trust policy specifies that the user should be prompted for permission to trust an expired certificate. However, a custom trust policy might override that behavior in some specific set of circumstances, such as when verifying the signature on a document that you know was generated while the certificate was still valid.

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 apps based on a public key infrastructure (PKI).

Exhibit 29

Unauthorized modification of iOS can cause security vulnerabilities, instability, shortened battery life, and other issues

This article is about adverse issues experienced by customers who have made unauthorized modifications to iOS (this hacking process is often called "jailbreaking").

iOS is designed to be reliable and secure from the moment you turn on your device. Built-in security features protect against malware and viruses and help to secure access to personal information and corporate data. Unauthorized modifications to iOS ("jailbreaking") bypass security features and can cause numerous issues to the hacked iPhone, iPad, or iPod touch, including:

Security vulnerabilities: Jailbreaking your device eliminates security layers designed to protect your personal information and your iOS device. With this security removed from your iOS device, hackers may steal your personal information, damage your device, attack your network, or introduce malware, spyware or viruses.

Instability: Frequent and unexpected crashes of the device, crashes and freezes of built-in apps and third-party apps, and loss of data.

Shortened battery life: The hacked software has caused an accelerated battery drain that shortens the operation of an iPhone, iPad, or iPod touch on a single battery charge.

Unreliable voice and data: Dropped calls, slow or unreliable data connections, and delayed or inaccurate location data.

Disruption of services: Services such as Visual Voicemail, Weather, and Stocks have been disrupted or no longer work on the device. Additionally, third-party apps that use the Apple Push Notification Service have had difficulty receiving notifications or received notifications that were intended for a different hacked device. Other push-based services such as iCloud and Exchange have experienced problems synchronizing data with their respective servers.

Inability to apply future software updates: Some unauthorized modifications have caused damage to iOS that is not repairable. This can result in the hacked iPhone, iPad, or iPod touch becoming permanently inoperable when a future Apple-supplied iOS update is installed.

Apple strongly cautions against installing any software that hacks iOS. It is also important to note that unauthorized modification of iOS is a violation of the iOS end-user software license agreement and because of this, Apple may deny service for an iPhone, iPad, or iPod touch that has installed any unauthorized software.

Last Modified: Sep 22, 2015

Helpful? Yes No

82% of people found this helpful.

Additional Product Support Information

 iPod touch

 iPod

 iPad

Start a Discussion
in Apple Support Communities

Ask other users about this article

Submit my question to the community

See all questions on this article See all questions I have asked

Contact Apple Support

Need more help? Save time by starting your support request online and we'll connect you to an expert.
Get started

Support Unauthorized modification of iOS can cause security vulnerabilities, instability, shortened battery life, and other issues

More ways to shop: Visit an Apple Store, call 1-800-MY-APPLE, or find a reseller.

 Privacy Policy | Terms of Use | Sales and Refunds | Site Map | Contact Apple United States (English)

Exhibit 30

Code Signing

Code signing your app assures users that it is from a known source and the app hasn't been modified since it was last signed. Before your app can integrate app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple. For more information on how to request certificates and code sign your apps, review the App Distribution Guide.

Common Tasks

To avoid potential issues with common tasks involving code signing, follow these best practices:

Signing and Running Development Builds

- Launching Your iOS App on a Device
- Launching Your Mac App

Beta Testing

- Beta Testing Your iOS App
- How to reproduce bugs reported against Mac App Store submissions

Distribution

- Submitting Your App
- Distributing Enterprise Apps for iOS Devices (in-house, internal use)

Essential Guides and Documentation

- App Distribution Guide
- Code Signing Troubleshooting
- Troubleshooting Push Notifications
- Developer ID and Gatekeeper (OS X)
- Code Signing Guide (OS X)

Frequently Asked Questions

- **How do I transfer my code signing certificates and provisioning profiles to another Mac?**
  Review the instructions in
  Exporting and Importing Certificates and Provisioning Profiles.
- **What does “Valid Signing Identity Not Found” mean and how do I resolve it?**
  Follow the steps outlined in
  Your Certificates Are Invalid Because You're Missing Private Keys.
- **How do I resolve a code signing build error?**
  See the list of published solutions in Build and Code Signing Issues.
- **How do I revoke or delete my certificates and start over?**
  Use the process outlined in
  Re-Creating Certificates and Updating Related Provisioning Profiles.
- **Do I need to define a custom Code Signing Entitlements file in Xcode?**
  To understand when entitlements are required and how to configure them properly, see Adding Capabilities.

Apple Developer Forum Discussions

- Capabilityies, Certificates, Identifiers & Profiles
- App Submission and Review

Developer Forums

Post questions and share thoughts with fellow developers and Apple engineers.

Discuss with other developers

Contact Us

Get personalized help with enrollment, membership, tools, and more.

Contact Apple Developer Support