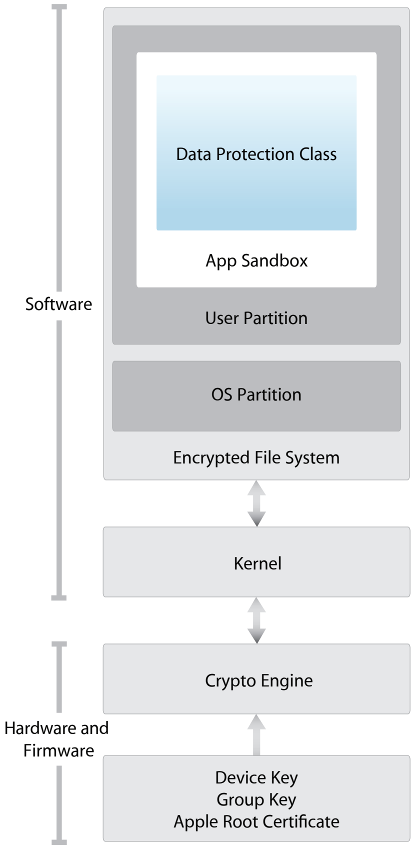
iOS Security

May 2012

Contents

Introduction



Security architecture diagram of iOS provides a visual overview of the different technologies discussed in this document.

Apple designed the iOS platform with security at its core. Keeping information secure on mobile devices is critical for any user, whether they're accessing corporate and customer information or storing personal photos, banking information, and addresses. Because every user's information is important, iOS devices are built to maintain a high level of security without compromising the user experience.

iOS devices provide stringent security technology and features, and yet also are easy to use. The devices are designed to make security as transparent as possible. Many security features are enabled by default, so IT departments don't need to perform extensive configurations. And some key features, like device encryption, are not configurable, so users cannot disable them by mistake.

For organizations considering the security of iOS devices, it is helpful to understand how the built-in security features work together to provide a secure mobile computing platform.

iPhone, iPad, and iPod touch are designed with layers of security. Low-level hardware and firmware features protect against malware and viruses, while high-level OS features allow secure access to personal information and corporate data, prevent unauthorized use, and help thwart attacks.

The iOS security model protects information while still enabling mobile use, third-party apps, and syncing. Much of the system is based on industry-standard secure design principles—and in many cases, Apple has done additional design work to enhance security without compromising usability.

This document provides details about how security technology and features are implemented within the iOS platform. It also outlines key elements that organizations should understand when evaluating or deploying iOS devices on their networks.

- **System architecture:** The secure platform and hardware foundations of iPhone, iPad, and iPod touch.
- **Encryption and Data Protection:** The architecture and design that protects the user's data when the device is lost or stolen, or when an unauthorized person attempts to use or modify it.
- **Network security:** Industry-standard networking protocols that provide secure authentication and encryption of data in transmission.
- **Device access:** Methods that prevent unauthorized use of the device and enable it to be remotely wiped if lost or stolen.

iOS is based on the same core technologies as OS X, and benefits from years of hardening and security development. The continued enhancements and additional security features with each major release of iOS have allowed IT departments in businesses worldwide to rapidly adopt and support iOS devices on their networks.

System Architecture

The tight integration of hardware and software on iOS devices allows for the validation of activities across all layers of the device. From initial boot-up to iOS software installation and through to third-party apps, each step is analyzed and vetted to ensure that each activity is trusted and uses resources properly.

Once the system is running, this integrated security architecture depends on the integrity and trustworthiness of XNU, the iOS kernel. XNU enforces security features at runtime and is essential to being able to trust higher-level functions and apps.

Secure Boot Chain

Each step of the boot-up process contains components that are cryptographically signed by Apple to ensure integrity, and proceeds only after verifying the chain of trust. This includes the bootloaders, kernel, kernel extensions, and baseband firmware.

When an iOS device is turned on, its application processor immediately executes code from read-only memory known as the Boot ROM. This immutable code is laid down during chip fabrication, and is implicitly trusted. The Boot ROM code contains the Apple Root CA 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.

This secure boot chain ensures that the lowest levels of software are not tampered with, and allows iOS to run only on validated Apple devices.

If one step of this boot process is unable to load or verify the next, boot-up is stopped and the device displays the "Connect to iTunes" screen. This is called recovery mode. If the Boot ROM is not even able to load or verify LLB, it enters DFU (Device Firmware Upgrade) mode. In both cases, the device must be connected to iTunes via USB and restored to factory default settings. For more information on manually entering recovery mode, see http://support.apple.com/kb/HT1808.

Entering DFU mode
DFU mode can be entered manually by connecting the device to a computer using the 30-pin Dock Connector to USB Cable, then holding down both the Home and Sleep/Wake buttons. After 8 seconds have elapsed, release the Sleep/Wake button while continuing to hold down the Home button. Note: Nothing will be displayed on the screen when in DFU mode. If the Apple logo appears, the Sleep/Wake button was held down for too long. Restoring a device after entering DFU mode returns it to a known good state with the certainty that only unmodified Apple-signed code is present.

System Software Personalization

Apple regularly releases software updates to address emerging security concerns; these updates are provided for all supported devices simultaneously. Users receive iOS update notifications on the device and through iTunes, and updates are delivered wirelessly, encouraging rapid adoption of the latest security fixes.

The boot process described above ensures that only Apple-signed code can be installed on a device. To prevent devices from being downgraded to older versions that lack the latest security updates, iOS uses a process called System Software Personalization. If downgrades were possible, an attacker who gains possession of a device could install an older version of iOS and exploit a vulnerability that's been fixed in the newer version.

iOS software updates can be installed using iTunes or over-the-air (OTA) on the device. With iTunes, a full copy of iOS is downloaded and installed. OTA software updates are provided as deltas for network efficiency.

During an iOS install or upgrade, iTunes (or the device itself, in the case of OTA software updates) connects to the Apple installation authorization server (gs.apple.com) and sends it a list of cryptographic measurements for each part of the installation bundle to be installed (for example LLB, iBoot, the kernel, and OS image), a random anti-replay value (nonce), and the device's unique ID (ECID).

The server checks the presented list of measurements against versions for which installation is permitted, and if a match is found, adds the ECID to the measurement and signs the result. The complete set of signed data from the server is passed to the device as part of the install or upgrade process. Adding the ECID "personalizes" the authorization for the requesting device. By authorizing and signing only for known measurements, the server ensures that the update is exactly as provided by Apple.

The boot-time chain-of-trust evaluation verifies that the signature comes from Apple and that the measurement of the item loaded from disk, combined with the device's ECID, matches what was covered by the signature.

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. The nonce prevents an attacker from saving the server's response and using it to downgrade a user's device in the future.

App Code Signing

Once the iOS kernel has booted, it controls which user processes and apps can be run. To ensure that all apps come from a known and approved source and have not been tampered with, iOS requires that all executable code be signed using an Apple-issued certificate. Apps provided with the device, like Mail and Safari, are signed by Apple. Third-party apps must also be validated and signed using an Apple-issued certificate. Mandatory code signing extends the concept of chain of trust from the OS to apps, and prevents third-party apps from loading unsigned code resources or using self-modifying code.

In order to develop and install apps on iOS devices, developers must register with Apple and join the iOS Developer Program. The real-world identity of each developer, whether an individual or a business, is verified by Apple before their certificate is issued. This certificate enables developers to sign apps and submit them to the App Store for distribution. As a result, all apps in the App Store have been submitted by an identifiable person or organization, serving as a deterrent to the creation of malicious apps. They have also been reviewed by Apple to ensure they operate as described and don't contain obvious bugs or other problems. In addition to the technology already discussed, this curation process gives customers confidence in the quality of the apps they buy.

Businesses also have the ability to write in-house apps for use within their organization and distribute them to their employees. Businesses and organizations can apply to the iOS Developer Enterprise Program (iDEP) with a D-U-N-S number. Apple approves applicants after verifying their identity and eligibility. Once an organization becomes a member of iDEP, it can register to obtain a provisioning profile that permits in-house apps to run on devices it authorizes. Users must have the provisioning profile installed in order to run the in-house apps. This ensures that only the organization's intended users are able to load the apps onto their iOS devices.

Unlike other mobile platforms, iOS does not allow users to install potentially malicious unsigned apps from websites, or run untrusted code. At runtime, code signature checks of all executable memory pages are made as they are loaded to ensure that an app has not been modified since it was installed or last updated.

Runtime Process Security

Once an app is verified to be from an approved source, iOS enforces security measures to ensure that it can't compromise other apps or the rest of the system.

All third-party apps are "sandboxed," so they are restricted from accessing files stored by other apps or from making changes to the device. This prevents apps from gathering or modifying information stored by other apps. Each app has a unique home directory for its files, which is randomly assigned when the app is installed. If a third-party app needs to access information other than its own, it does so only by using application programming interfaces (APIs) and services provided by iOS.

System files and resources are also shielded from the user's apps. The majority of iOS runs as the non-privileged user "mobile," as do all third-party apps. The entire OS partition is mounted read-only. Unnecessary tools, such as remote login services, aren't included in the system software, and APIs do not allow apps to escalate their own privileges to modify other apps or iOS itself.

Access by third-party apps to user information and features such as iCloud is controlled using declared entitlements. Entitlements are key/value pairs that are signed in to an app and allow authentication beyond runtime factors like unix user ID. Since entitlements are digitally signed, they cannot be changed. Entitlements are used extensively by system apps and daemons to perform specific privileged operations that would otherwise require the process to run as root. This greatly reduces the potential for privilege escalation by a compromised system application or daemon.

In addition, apps can only perform background processing through system-provided APIs. This enables apps to continue to function without degrading performance or dramatically impacting battery life. Apps can't share data directly with each other; sharing can be implemented only by both the receiving and sending apps using custom URL schemes, or through shared keychain access groups.

Address space layout randomization (ASLR) protects against the exploitation of memory corruption bugs. Built-in apps use ASLR to ensure that all memory regions are randomized upon launch. Additionally, system shared library locations are randomized at each device startup. Xcode, the iOS development environment, automatically compiles third-party programs with ASLR support turned on.

Further protection is provided by iOS using ARM's Execute Never (XN) feature, which marks memory pages as non-executable. Memory pages marked as both writable and executable can be used only by apps under tightly controlled conditions: The kernel checks for the presence of the Apple-only "dynamic-codesigning" entitlement. Even then, only a single mmap call can be made to request an executable and writable page, which is given a randomized address. Safari uses this functionality for its JavaScript JIT compiler.

Encryption and Data Protection

The secure boot chain, code signing, and runtime process security all help to ensure that only trusted code and apps can run on a device. iOS has additional security features to protect user data, even in cases where other parts of the security infrastructure have been compromised (for example, on a device with unauthorized modifications). Like the system architecture itself, these encryption and data protection capabilities use layers of integrated hardware and software technologies.

Hardware Security Features

On mobile devices, speed and power efficiency are critical. Cryptographic operations are complex and can introduce performance or battery life problems if not designed and implemented correctly.

Every iOS device has a dedicated AES 256 crypto engine built into the DMA path between the flash storage and main system memory, making file encryption highly efficient. Along with the AES engine, SHA-1 is implemented in hardware, further reducing cryptographic operation overhead.

The device's unique ID (UID) and a device group ID (GID) are AES 256-bit keys fused into the application processor during manufacturing. No software or firmware can read them directly; they can see only the results of encryption or decryption operations performed using them. The UID is unique to each device and is not recorded by Apple or any of its suppliers. The GID is common to all processors in a class of devices (for example, all devices using the Apple A5 chip), and is used as an additional level of protection when delivering system software during installation and restore. Burning these keys into the silicon prevents them from being tampered with or bypassed, and guarantees that they can be accessed only by the AES engine.

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.

Apart from the UID and GID, all other cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on Yarrow. System entropy is gathered from interrupt timing during boot, and additionally from internal sensors once the device has booted.

Securely erasing saved keys is just as important as generating them. It's especially challenging to do so on flash storage, where wear-leveling might mean multiple copies of data need to be erased. To address this issue, iOS devices include a feature dedicated to secure data erasure called Effaceable Storage. This feature accesses the underlying storage technology (for example, NAND) to directly address and erase a small number of blocks at a very low level.

File Data Protection

In addition to the hardware encryption features built into iOS devices, Apple uses a technology called Data Protection to further protect data stored in flash memory on the device. This technology is designed with mobile devices in mind, taking into account the fact that they may always be turned on and connected to the Internet, and may receive phone calls, text, or emails at any time.

Data Protection allows a device to respond to events such as incoming phone calls without decrypting sensitive data and downloading new information while locked. These individual behaviors are controlled on a per-file basis by assigning each file to a class, as described in the Classes section later in document.

Data Protection protects the data in each class based on when the data needs to be accessed. Accessibility is determined by whether the class keys have been unlocked.

Data Protection is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies previously described.

Architecture overview

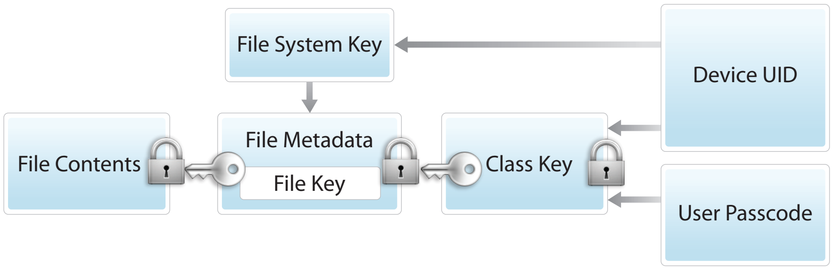
Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the "per-file" key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. The initialization vector (IV) is the output of a linear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

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

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

The metadata of all files in the file system are encrypted with a random key, which is created when iOS is first installed or when the device is wiped by a user. The file system key is stored in Effaceable Storage. Since it's stored on the device, this key is not used to maintain the confidentiality of data; instead, it's designed to be quickly erased on demand (by the user, with the "Erase all content and settings" option, or by a user or administrator issuing a remote wipe command from a Mobile Device Management server, Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.

Erase all content and settings
The "Erase all content and settings" option in Settings obliterates all the keys in Effaceable Storage, rendering all user data on the device cryptographically inaccessible. Therefore, it's an ideal way to be sure all personal information is removed from a device before giving it to somebody else or returning it for service. Important: Do not use the "Erase all content and settings" option until the device has been backed up, as there is no way to recover the erased data.



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

Passcodes

By setting up a device passcode, the user automatically enables Data Protection. iOS supports four-digit and arbitrary-length alphanumeric passcodes. In addition to unlocking the device, a passcode provides the entropy for encryption keys, which are not stored on the device. This means an attacker in possession of a device can't get access to data in certain protection classes without the passcode.

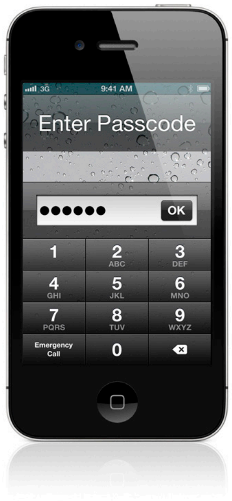
The passcode is "tangled" with the device's UID, so brute-force attempts must be performed on the device under attack. A large iteration count is used to make each attempt slower. The iteration count is calibrated so that one attempt takes approximately 80 milliseconds. This means it would take more than 5½ years to try all combinations of a six-character alphanumeric passcode with lowercase letters and numbers, or 2½ years for a nine-digit passcode with numbers only.

To further discourage brute-force passcode attacks, the iOS interface enforces escalating time delays after the entry of an invalid passcode at the Lock screen. Users can choose to have the device automatically wiped after 10 failed passcode attempts. This setting is also available as an administrative policy through Mobile Device Management (MDM) and Exchange ActiveSync, and can also be set to a lower threshold.

Creating strong Apple ID passwords
Apple IDs are used to connect to a number of services including iCloud, FaceTime, and iMessage. To help users create strong passwords, all new accounts must contain the following password attributes:

- At least eight characters
- At least one letter
- At least one uppercase letter
- At least one number
- No more than three consecutive identical characters
- Not the same as the account name

Passcode considerations
If a long password that contains only numbers is entered, a numeric keypad is displayed at the Lock screen instead of the full keyboard. A longer numeric passcode may be easier to enter than a shorter alphanumeric passcode, while providing similar security.



Classes

When a new file is created on an iOS device, it's assigned a class by the app that creates it. Each class uses different policies to determine when the data is accessible. The basic classes and policies are as follows:

Complete Protection
(NSFileProtectionComplete): The class key is protected with a key derived from the user passcode and the device UID. Shortly after the user locks a device (10 seconds, if the Require Password setting is Immediately), the decrypted class key is discarded, rendering all data in this class inaccessible until the user enters the passcode again.

The Mail app implements Complete Protection for messages and attachments. App launch images and location data are also stored with Complete Protection.

Protected Unless Open
(NSFileProtectionCompleteUnlessOpen): Some files may need to be written while the device is locked. A good example of this is a mail attachment downloading in the background. This behavior is achieved by using asymmetric elliptic curve cryptography (ECDH over Curve25519). Along with the usual per-file key, Data Protection generates a file public/private key pair. A shared secret is computed using the file's private key and the Protected Unless Open class public key, whose corresponding private key is protected with the user's passcode and the device UID. The per-file key is wrapped with the hash of this shared secret and stored in the file's metadata along with the file's public key; the corresponding private key is then wiped from memory. As soon as the file is closed, the per-file key is also wiped from memory. To open the file again, the shared secret is re-created using the Protected Unless Open class's private key and the file's ephemeral public key; its hash is used to unwrap the per-file key, which is then used to decrypt the file.

Protected Until First User Authentication
(NSFileProtectionCompleteUntilFirstUserAuthentication): This class behaves in the same way as Complete Protection, except that the decrypted class key is not removed from memory when the device is locked. The protection in this class has similar properties to desktop full-disk encryption, and protects data from attacks that involve a reboot.

No Protection
(NSFileProtectionNone): This class key is protected only with the UID, and is kept in Effaceable Storage. This is the default class for all files not otherwise assigned to a Data Protection class. Since all the keys needed to decrypt files in this class are stored on the device, the encryption only affords the benefit of fast remote wipe. If a file is not assigned a Data Protection class, it is still stored in encrypted form (as is all data on an iOS device).

The iOS Software Development Kit (SDK) offers a full suite of APIs that make it easy for third-party and in-house developers to adopt Data Protection and ensure the highest level of protection in their apps. Data Protection is available for file and database APIs, including NSFileManager, CoreData, NSData, and SQLite.

Keychain Data Protection

Many apps need to handle passwords and other short but sensitive bits of data, such as keys and login tokens. The iOS keychain provides a secure way to store these items.

The keychain is implemented as a SQLite database stored on the file system in the No Protection class, while its security is provided by a different key hierarchy that runs parallel to the key hierarchy used to protect files. There is only one database; the *securityd* daemon determines which keychain items each process or app can access. Keychain access APIs result in calls to the securityd framework, which queries the app's "keychain-access-groups" and the "application-identifier" entitlement. Rather than limiting access to a single process, access groups allow keychain items to be shared between apps.

Components of a keychain item
Along with the access group, each keychain item contains administrative metadata (such as "created" and "last updated" time stamps). It also contains SHA-1 hashes of the attributes used to query for the item (such as the account and server name) to allow lookup without decrypting each item. And finally, it contains the encryption data, which includes the following:

- Version number
- Value indicating which protection class the item is in
- Per-item key wrapped with the protection class key
- Dictionary of attributes describing the item (as passed to SecItemAdd), encoded as a binary plist and encrypted with the per-item key

The encryption is AES 128 in GCM (Galois/Counter Mode); the access group is included in the attributes and protected by the GMAC tag calculated during encryption.

Keychain items can only be shared between apps from the same developer. This is managed by requiring third-party apps to use access groups with a prefix allocated to them through the iOS Developer Program. The prefix requirement is enforced through code signing and provisioning profiles.

Keychain data is protected using a class structure similar to the one used in file Data Protection. These classes have behaviors equivalent to file Data Protection classes, but use distinct keys and are part of APIs that are named differently.

Availability	File Data Protection	Keychain Data Protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSFileProtectionNone	kSecAttrAccessibleAlways

Each keychain class has a "This device only" counterpart, which is always protected with the UID when being copied from the device during a backup, rendering it useless if restored to a different device.

Apple has carefully balanced security and usability by choosing keychain classes that depend on the type of information being secured and when it's needed by the OS. For example, a VPN certificate must always be available so the device keeps a continuous connection, but it's classified as "non-migratory," so it can't be moved to another device.

For keychain items created by iOS, the following class protections are enforced:

Item	Accessible
Wi-Fi passwords	After first unlock
Mail accounts	After first unlock
Exchange accounts	After first unlock
VPN certificates	Always, non-migratory
VPN passwords	After first unlock
LDAP, CalDAV, CardDAV	After first unlock
iTunes backup	When unlocked, non-migratory
Voicemail	Always
Safari passwords	When unlocked
Bluetooth keys	Always, non-migratory
Apple Push Notification Service Token	Always, non-migratory
iCloud certificates and private key	Always, non-migratory
iMessage keys	Always, non-migratory
Certificates and private keys installed by Configuration Profile	Always, non-migratory
SIM PIN	Always, non-migratory

Keybags

The keys for both file and keychain Data Protection classes are collected and managed in keybags. iOS uses the following four keybags: System, Backup, Escrow, and iCloud Backup.

System keybag is where the wrapped class keys used in normal operation of the device are stored. For example, when a passcode is entered, the NSFileProtectionComplete key is loaded from the system keychain and unwrapped. It is a binary plist stored in the No Protection class, but whose contents are encrypted with a key held in Effaceable Storage. In order to give forward security to keybags, this key is wiped and regenerated each time a user changes a passcode. The System keybag is the only keybag stored on the device. The AppleKeyStore kernel extension manages the System keybag, and can be queried regarding a device's lock state. It reports that the device is unlocked only if all the class keys in the System are accessible, having been unwrapped successfully.

Backup keybag is created when an encrypted backup is made by iTunes and stored on the computer to which the device is backed up. A new keybag is created with a new set of keys, and the backed-up data is re-encrypted to these new keys. As explained earlier, non-migratory keychain items remain wrapped with the UID-derived key, allowing them to be restored to the device they were originally backed up from, but rendering them inaccessible on a different device.

The keybag is protected with the password set in iTunes, run through 10,000 iterations of PBKDF2. Despite this large iteration count, there's no tie to a specific device, and therefore a brute-force attack parallelized across many computers can be attempted on the backup keybag. This threat can be mitigated with a sufficiently strong password.

If a user chooses to not encrypt an iTunes backup, the backup files are not encrypted regardless of their Data Protection class, but the keychain remains protected with a UID-derived key. This is why keychain items migrate to a new device only if a backup password is set.

Escrow keybag is used for iTunes syncing and Mobile Device Management (MDM). This keybag allows iTunes to back up and sync without requiring the user to enter a passcode, and it allows an MDM server to remotely clear a user's passcode. It is stored on the computer that's used to sync with iTunes, or on the MDM server that manages the device.

The Escrow keybag improves the user experience during device synchronization, which potentially requires access to all classes of data. When a passcode-locked device is first connected to iTunes, the user is prompted to enter a passcode. The device then creates an Escrow keybag and passes it to the host. The Escrow keybag contains exactly the same class keys used on the device, protected by a newly generated key. This key is needed to unlock the Escrow keybag, and is stored on the device in the Protected Until First User Authentication class. This is why the device passcode must be entered before backing up with iTunes for the first time after a reboot.

iCloud Backup keybag is similar to the Backup keybag. All the class keys in this keybag are asymmetric (using Curve25519, like the Protected Unless Open Data Protection class), so iCloud backups can be performed in the background. For all Data Protection classes except No Protection, the encrypted data is read from the device and sent to iCloud. The corresponding class keys are protected by iCloud keys. The keychain class keys are wrapped with a UID-derived key in the same way as an unencrypted iTunes backup.

Components of a keybag

A header containing:

- Version (set to 3 in iOS 5)
- Type (System, Backup, Escrow, or iCloud Backup)
- Keybag UUID
- An HMAC if the keybag is signed
- The method used for wrapping the class keys: tangling with the UID or PBKDF2, along with the salt and iteration count

A list of class keys:

- Key UUID
- Class (which file or keychain Data Protection class this is)
- Wrapping type (UID-derived key only; UID-derived key and passcode-derived key)
- Wrapped class key
- Public key for asymmetric classes

Network Security

In addition to the measures Apple has taken to protect data stored on iOS devices, there are many network security measures that organizations can take to safeguard information as it travels to and from an iOS device.

Mobile users must be able to access corporate information networks from anywhere in the world, so it's important to ensure they are authorized and that their data is protected during transmission. iOS uses—and provides developer access to—standard networking protocols for authenticated, authorized, and encrypted communications. iOS provides proven technologies and the latest standards to accomplish these security objectives for both Wi-Fi and cellular data network connections.

On other platforms, firewall software is needed to protect numerous open communication ports against intrusion. Because iOS achieves a reduced attack surface by limiting listening ports and removing unnecessary network utilities such as telnet, shells, or a web server, it doesn't need firewall software. Additionally, communication using iMessage, FaceTime, and the Apple Push Notification Server is fully encrypted and authenticated.

SSL, TLS

iOS supports Secure Socket Layer (SSL v3) as well as Transport Layer Security (TLS v1.1, TLS v1.2) and DTLS. Safari, Calendar, Mail, and other Internet applications automatically use these mechanisms to enable an encrypted communication channel between the device and network services. High-level APIs (such as CFNetwork) make it easy for developers to adopt TLS in their apps, while low-level APIs (SecureTransport) provide fine-grained control.

VPN

Secure network services like virtual private networking typically require minimal setup and configuration to work with iOS devices. iOS devices work with VPN servers that support the following protocols and authentication methods:

- Juniper Networks, Cisco, Aruba Networks, SonicWALL, Check Point, and F5 Networks SSL-VPN using the appropriate client app from the App Store. These apps provide user authentication for the built-in iOS support.
- Cisco IPSec with user authentication by Password, RSA SecurID or Cryptocard, and machine authentication by shared secret and certificates. Cisco IPSec supports VPN On Demand for domains that are specified during device configuration.
- L2TP/IPSec with user authentication by MS-CHAPV2 Password, RSA SecurID or Cryptocard, and machine authentication by shared secret.
- PPTP with user authentication by MS-CHAPV2 Password and RSA SecurID or Cryptocard.

iOS supports VPN On Demand for networks that use certificated-based authentication. IT policies specify which domains require a VPN connection by using a configuration profile.

For more information on VPN server configuration for iOS devices, see http://help.apple.com/iosdeployment-vpn/.

Wi-Fi

iOS supports industry-standard Wi-Fi protocols, including WPA2 Enterprise, to provide authenticated access to wireless corporate networks. WPA2 Enterprise uses 128-bit AES encryption, giving users the highest level of assurance that their data remains protected when sending and receiving communications over a Wi-Fi network connection. With support for 802.1X, iOS devices can be integrated into a broad range of RADIUS authentication environments. 802.1X wireless authentication methods supported on iPhone and iPad include EAP-TLS, EAP-TTLS, EAP-FAST, EAP-SIM, PEAPv0, PEAPv1, and LEAP.

Bluetooth

Bluetooth support in iOS has been designed to provide useful functionality without unnecessary increased access to private data. iOS devices support Encryption Mode 3, Security Mode 4, and Service Level 1 connections. iOS supports the following Bluetooth profiles:

- Hands-Free Profile (HFP 1.5)
- Phone Book Access Profile (PBAP)
- Advanced Audio Distribution Profile (A2DP)
- Audio/Video Remote Control Profile (AVRCP)
- Personal Area Network Profile (PAN)
- Human Interface Device Profile (HID)

Support for these profiles varies by device. For more information, see http://support.apple.com/kb/ht3647.

Device Access

iOS supports flexible security policies and configurations that are easily enforced and managed. This enables enterprises to protect corporate information and ensure that employees meet enterprise requirements, even if they are using devices they've provided themselves.

Passcode Protection

In addition to providing the cryptographic protection discussed earlier, passcodes prevent unauthorized access to the device's UI. The iOS interface enforces escalating time delays after the entry of an invalid passcode, dramatically reducing the effectiveness of brute force attacks via the Lock screen. Users can choose to have the device automatically wiped after 10 failed passcode attempts. This setting is available as an administrative policy and can also be set to a lower threshold through MDM and Exchange ActiveSync.

By default, the user's passcode can be defined as a four-digit PIN. Users can specify a longer, alphanumeric passcode by turning on Settings > General > Passcode > Complex Passcode. Longer and more complex passcodes are harder to guess or attack, and are recommended for enterprise use.

Administrators can enforce complex passcode requirements and other policies using MDM or Exchange ActiveSync, or by requiring users to manually install configuration profiles. The following passcode policies are available:

- Allow simple value
- Require alphanumeric value
- Minimum passcode length
- Minimum number of complex characters
- Maximum passcode age
- Passcode history
- Auto-lock timeout
- Grace period for device lock
- Maximum number of failed attempts

For details about each policy, see the iPhone Configuration Utility documentation at http://help.apple.com/iosdeployment-ipcu/.

Configuration Enforcement

A configuration profile is an XML file that allows an administrator to distribute configuration information to iOS devices. Settings that are defined by an installed configuration profile can't be changed by the user. If the user deletes a configuration profile, all the settings defined by the profile are also removed. In this manner, administrators can enforce settings by tying policies to access. For example, a configuration profile that provides an email configuration can also specify a device passcode policy. Users won't be able to access mail unless their passcodes meet the administrator's requirements.

An iOS configuration profile contains a number of settings that can be specified:

- Passcode policies
- Restrictions on device features (disabling the camera, for example)
- Wi-Fi settings
- VPN settings
- Email server settings
- Exchange settings
- LDAP directory service settings
- CalDAV calendar service settings
- Web clips
- Credentials and keys
- Advanced cellular network settings

Configuration profiles can be signed and encrypted to validate their origin, ensure their integrity, and protect their contents. Configuration profiles are encrypted using CMS (RFC 3852), supporting 3DES and AES-128.

Configuration profiles can also be locked to a device to completely prevent their removal, or to allow removal only with a passcode. Since many enterprise users personally own their iOS devices, configuration profiles that bind a device to an MDM server can be removed—but doing so will also remove all managed configuration information, data, and apps.

Users can install configuration profiles directly on their devices using the iPhone Configuration Utility. Configuration profiles can be downloaded via email or over-the-air using an MDM server.

Mobile Device Management

iOS support for MDM allows businesses to securely configure and manage scaled iPhone and iPad deployments across their organizations. MDM capabilities are built on existing iOS technologies such as Configuration Profiles, Over-the-Air Enrollment, and the Apple Push Notification service. Using MDM, IT departments can enroll iOS devices in an enterprise environment, wirelessly configure and update settings, monitor compliance with corporate policies, and even remotely wipe or lock managed devices. For more information on Mobile Device Management, visit www.apple.com/business/mdm.

Device Restrictions

Administrators can restrict device features by installing a configuration profile. The following restrictions are available:

- Allow app installs
- Allow use of camera
- Allow FaceTime
- Allow screen capture
- Allow voice dialing
- Allow automatic sync while roaming
- Allow in-app purchases
- Force user to enter store password for all purchases
- Allow multiplayer gaming
- Allow adding Game Center Friends
- Allow Siri

- Allow Siri while device is locked
- Allow use of YouTube
- Allow use of iTunes Store
- Allow use of Safari
- Enable Safari autofill
- Force Fraudulent Website Warning
- Enable JavaScript
- Block pop-ups
- Accept cookies
- Allow iCloud backup
- Allow iCloud document sync
- Allow Photo Stream
- Allow diagnostics to be sent to Apple
- Allow user to accept untrusted TLS certificates
- Force encrypted backups
- Restrict media by content rating

Remote Wipe

iOS devices can be erased remotely by an administrator or user. Instant remote wiping is achieved by securely discarding the block storage encryption key from Effaceable Storage, rendering all data unreadable. Remote wiping can be initiated by MDM, Exchange, or iCloud.

When remote wiping is triggered by MDM or iCloud, the device sends an acknowledgment and performs the wipe. For remote wiping via Exchange, the device checks in with the Exchange Server before performing the wipe.

Users can also wipe devices in their possession using the Settings app. And as mentioned, devices can be set to automatically wipe after a series of failed passcode attempts.

Conclusion

A Commitment to Security

Each component of the iOS security platform, from hardware to encryption to device access, provides organizations with the resources they need to build enterprise-grade security solutions. The sum of these parts gives iOS its industry-leading security features, without making the device difficult or cumbersome to use.

Apple uses this security infrastructure throughout iOS and the iOS apps ecosystem. Hardware-based storage encryption provides instant remote wipe capabilities when a device is lost, and ensures that users can completely remove all corporate and personal information when a device is sold or transferred to another owner. For the collection of diagnostic information, unique identifiers are created to identify a device anonymously.

Safari offers safe browsing with its support for OCSP, EV certificates, and certificate verification warnings. Mail leverages certificates for authenticated and encrypted email by supporting S/MIME. iMessage and FaceTime provide client-to-client encryption as well.

The combination of required code signing, sandboxing, and entitlements in apps provides solid protection against viruses, malware, and other exploits that compromise the security of other platforms. The App Store submission process works to further protect users from these risks by reviewing every app before it’s made available for sale.

Businesses are encouraged to review their IT and security policies to ensure they are taking full advantage of the layers of security technology and features offered by the iOS platform.

Apple maintains a dedicated security team to support all Apple products. The team provides security auditing and testing for products under development as well as released products. The Apple team also provides security tools and training, and actively monitors for reports of new security issues and threats. Apple is a member of the Forum of Incident Response and Security Teams (FIRST). For information about reporting issues to Apple and subscribing to security notifications, go to apple.com/support/security.

Apple is committed to incorporating proven encryption methods and creating modern mobile-centric privacy and security technologies to ensure that iOS devices can be used with confidence in any personal or corporate environment.

Glossary

Address space layout randomization (ASLR)

A technique employed by iOS to make the successful exploitation of a software bug much more difficult. By ensuring memory addresses and offsets are unpredictable, exploit code can’t hard code these values. In iOS 5, the position of all system apps and libraries are randomized, along with all third-party apps compiled as position-independent executables.

Boot ROM

The very first code executed by a device’s processor when it first boots. As an integral part of the processor, it can’t be altered by either Apple or an attacker.

Data Protection

File and keychain protection mechanism for iOS. It can also refer to the APIs that apps use to protect files and keychain items.

DFU

A mode in which a device’s Boot ROM code waits to be recovered over USB. The screen is black when in DFU mode, but upon connecting to a computer running iTunes, the following prompt is presented: “iTunes has detected an iPad in recovery mode. You must restore this iPad before it can be used with iTunes.”

ECID

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.

Effaceable Storage

A dedicated area of NAND storage, used to store cryptographic keys, that can be addressed directly and wiped securely. While it doesn’t provide protection if an attacker has physical possession of a device, keys held in Effaceable Storage can be used as part of a key hierarchy to facilitate fast wipe and forward security.

File system key

The key that encrypts each file’s metadata, including its class key. This is kept in Effaceable Storage to facilitate fast wipe, rather than confidentiality.

GID

Like the UID but common to every processor in a class.

iBoot

Code that’s loaded by LLB, and in turn loads XNU, as part of the secure boot chain.

Keybag

A data structure used to store a collection of class keys. Each type (System, Backup, Escrow, or iCloud Backup) has the same format:

- A header containing:
  - Version (set to 3 in iOS 5)
  - Type (System, Backup, Escrow, or iCloud Backup)
  - Keybag UUID
  - An HMAC if the keybag is signed
  - The method used for wrapping the class keys: tangling with the UID or PBKDF2, along with the salt and iteration count
- A list of class keys:
  - Key UUID
  - Class (which file or keychain Data Protection class this is)
  - Wrapping type (UID-derived key only; UID-derived key and passcode-derived key)
  - Wrapped class key
  - Public key for asymmetric classes

Keychain The infrastructure and a set of APIs used by iOS and third-party apps to store and retrieve passwords, keys, and other sensitive credentials.

Key wrapping Encrypting one key with another. iOS uses NIST AES key wrapping, as per RFC 3394.

Low-Level Bootloader (LLB) Code that's invoked by the Boot ROM, and in turn loads iBoot, as part of the secure boot chain.

Per-file key The AES 256-bit key used to encrypt a file on the file system. The per-file key is wrapped by a class key and is stored in the file's metadata.

Provisioning profile A plist signed by Apple that contains a set of entities and entitlements allowing apps to be installed and tested on an iOS device. A development provisioning profile lists the devices that a developer has chosen for ad hoc distribution, and a distribution provisioning profile contains the app ID of an enterprise-developed app.

Tangling The process by which a user's passcode is turned into a cryptographic key and strengthened with the device's UID. This ensures that a brute-force attack must be performed on a given device, and thus is rate limited and cannot be performed in parallel. The tangling algorithm is PBKDF2, which uses AES as the pseudorandom function (PRF) with a UID-derived key.

UID A 256-bit AES key that's burned into each processor at manufacture. It cannot be read by firmware or software, and is used only by the processor's hardware AES engine. To obtain the actual key, an attacker would have to mount a highly sophisticated and expensive physical attack against the processor's silicon. The UID is not related to any other identifier on the device including, but not limited to, the UDID.

XNU The kernel at the heart of the iOS and OS X operating systems. It's assumed to be trusted, and enforces security measures such as code signing, sandboxing, entitlement checking, and ASLR.

Yarrow A cryptographically secure pseudorandom number generator algorithm. An implementation of Yarrow in iOS takes entropy generated by various system events and produces unpredictable random numbers that can be used, for example, as encryption keys.