iOS Security

Chawin Sitawarin

14th November 2018

1 Introduction

This note will cover page 1 to 38 of Apple's document on iOS security [1] released in September 2018 supporting their new iOS 12. Today, mobile devices, including smartphones, tablets, and smart watches, have become an integral part of our every life. While facilitating many tasks we do, it also holds a significant amount of our sensitive and personal data. The security of these devices should, therefore, be a top priority for every manufacturer.

While the document does not explicitly describe the threat model, we assume that all components (hardware, software, applications) developed by Apple is trusted, and everything else (third-party applications, etc.) is untrusted. To motivate why smartphone security is necessary, in the simplest case where a device is stolen, there already are a number of attacks the adversary can attempt to extract data from the device. Here is a non-inclusive list of possible attacks.

1. Simply wipe out all of the data on the device (e.g. factory reset).
2. Brute force password or the other authentication methods (FaceID, TouchID).
3. Try to physically read from the memory with specialized equipment.
4. Plug the device into a terminal to reinstall a corrupted OS (discussed in Section 2.1) or to install a malware (discussed in Section 4).
5. Redirect the networking so that firmware updates are handled by the adversary. This includes a firmware downgrade and a corrupted firmware installation (discussed in Section 2.1)

The rest of the paper will, explicitly and implicitly, outline how Apple designs their devices to prevent these attacks. Security of an iOS device expands many components in the system. Figure 1 lists main components in an iOS devices, from the low-level hardware up to the application level, that involves the device's security. However, this document will focus on three main components: system security, encryption and data protection, and app security.

2 System Security

For system security, we cover two main mechanisms: secure boot chain and secure enclave.

2.1 Secure Boot Chain

The secure boot chain ensures that attackers cannot run a modified OS or applications. This relies on a chain of trusts which begins from an embedded key in hardware during fabrication. All components in the boot chain (bootloaders, kernel, kernel extensions, and baseband firmware) are verified by a component one level below and verify a component one level above. If the verification fails, the boot process will stop and

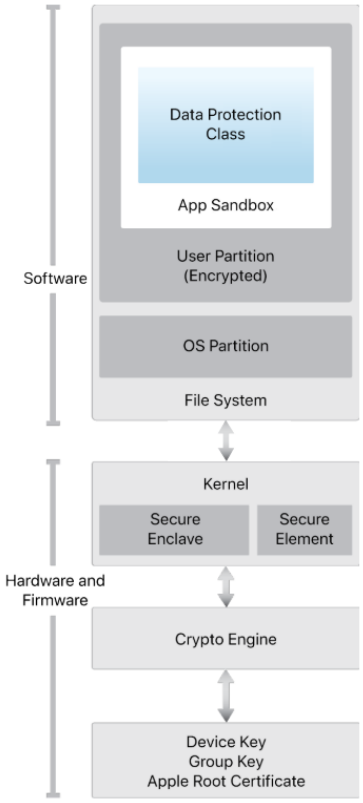

Figure 1: An overview of main components in an iOS device that provide security to the system. This figure is taken from [1].

will not load the unverified component. On a high level, the chain of trusts goes from Boot ROM, iBoot, iOS Kernel, and ends at applications.

Boot ROM is a read-only memory, specified during chip fabrication and installed with the public key of Apple. It verifies and loads iBoot if iBoot is correctly signed by Apple. If the verification fails, the device enters Device Firmware Upgrade (DFU) mode (or the recovery mode) which requires iTunes connection and a factory reset to resolve.

One of the well-known attacks on the system firmware is the *downgrade attack*. The key idea of the attack is to install an older version of iOS (correctly signed by Apple) which is known to have certain vulnerabilities and then exploit that vulnerability with other existing attacks. An adversary can simply download and save the installation package of previous versions of iOS when it is first released. To prevent the attack, Apple uses a method called System Software Authorization which relies on a device's unique identifier called ECID, a number fused into the chip of every iOS device. All iOS updates must include the device's ECID which is also signed by Apple, i.e. $Sign_{Apple}(OS, ECID)$. Updates are allowed to install only if the signed ECID matches with the device's. Since Apple only provides a signature for a new secure iOS, the attacker cannot obtain a valid version downgrade with the target device's ECID.

Nonetheless, assuming that the signature only includes the ECID, the attack might become possible if the adversary manages to access and make a copy of the target device's legitimate updates to the outdated

firmware. Apple prevents this attack by requiring a random nounce as well as other system measurements when the update is requested. All this time-dependent information is included in the signature, and the device is allowed to install an update only if the information matches.

2.2 Secure Enclave

iOS devices are equipped with a secure enclave to provide an isolated environment for computation and management on sensitive data as well as cyptographic protocols. The principle of secure enclave is covered in the past lecture on Intel's secure enclave [2].

The secure enclave has a dedicated Boot ROM similar to that of the application processor. The secure enclave Boot ROM initializes an ephemeral key, which includes the device's UID and an anti-replay counter, during the device starts up. The key is used to encrypt the memory portion used by the secure enclave. The anti-replay services are necessary for revoking data in some events such as Passcode change, TouchID or FaceID change, Apple Pay change, etc.

The secure enclave is also dedicated to handling processing of TouchID and FaceID data. This ensures that the sensitive biometric information stays within the secure enclave and cannot be read by any other part of the device including the OS itself.

3 Encryption and Data Protection

Data encryption and decryption are made efficient by using a dedicated hardware called AES-256 crypto engine which sits on the bus between the main memory and the flash storage. This design also ensures that data is always encrypted before written to the memory. The device's UIDs and GIDs are used as the AES-256 keys. They are created during fabrication and only readable by the AES. As a result, only the device that encrypts the data can decrypt it, and even if the memory is physically removed and placed on other devices, they cannot read the data.

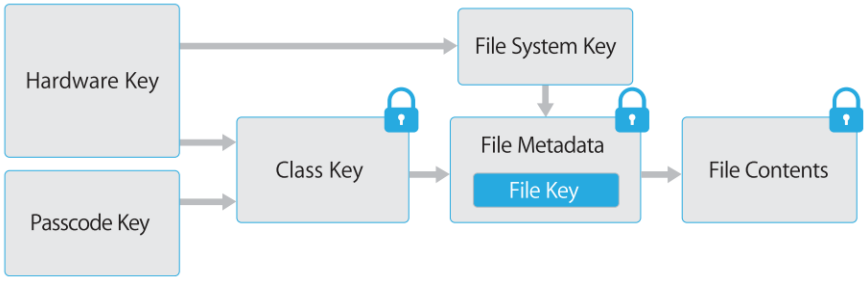

Figure 2: A diagram of the file encryption in an iOS device displaying relationship between the file system key, the class key, and the file key. This figure is taken from [1].

Data protection uses a hierarchy of keys (also called key wrapping) to encrypt files on the device. More specifically, each file is encrypted with a unique secret per-file key, k_f. The file keys are in turn encrypted by the main file system key, k_e, and the class keys, k_c. Only when the class condition is met and the system key is available, the file keys can be decrypted and then used to decrypt the files. The class keys are associated with *circumstances* the device is in. For instance, *Complete Protection* class, the most secure class, discards the decryption key after the device locks, but *Protected Unless Open* class allows continuous access to the file even after the device is locked. Allowed classes associated with each file are contained in the file metadata

encrypted by the file system key. Figure 2 summarizes the hierarchy of the keys.

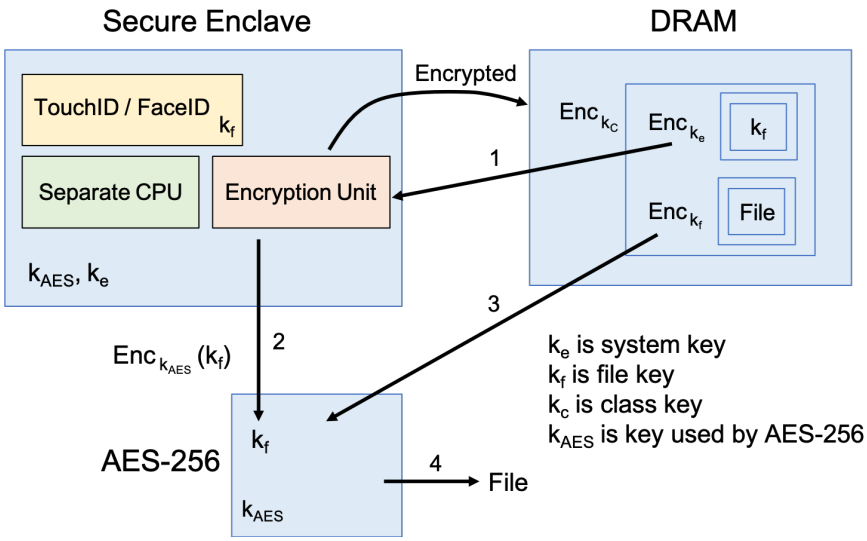

Figure 3: This figure, taken from the lecture, shows hardware components and keys involving in the data protection scheme. It also visualizes the steps necessary for file decryption.

As shown in Figure 3, the system key (k_e) is stored in the secure enclave. On a high level, the data decryption process is visualized in steps 1 to 4 in the figure, assuming that the class key is unlocked. The steps 1 to 4 are as follows:

1. The secure enclave retrieves and decrypts k_f using k_e it has.
2. The secure enclave encrypts k_f with k_{AES} and sends it to AES-256.
3. The file encrypted with k_f is also sent to AES-256.
4. AES-256 decrypts k_f from the secure enclave using its k_{AES}. Then, it uses k_f to decrypt the file.

At this point, one might wonder why the system does not use only one master key, i.e. the system key, to encrypt all of the files. The main reason is that if one file key is compromised, the adversary will only be able to read the file encrypted with that key but not any other files on the device. On the other hand, if all the files use the same decryption key, then once the key is compromised, the adversary can access everything. Additionally, having a hierarchy of keys also simplifies fine-grained key management and key revocation.

4 Presentation: iOS Application Security

The main threat model in this section is still the same as the previous: everything designed by Apple, both system and applications, is considered secure, and all third-party apps are not trusted by default. This section will cover app code signing, sandboxing, extensions, and remote access.

4.1 App Code Signing

Every app writer must register for an Apple Developer account which costs 100 dollars per year in order to obtain a certificate from Apple. Developers must sign their applications with the certificate which will be

verified by Apple during the app submission/review and by the device during every load. This ensures that all code running on an iOS device is signed by a certificate authorized by Apple.

Why do we need code signing if we assume that the OS is trusted?: code signing has several benefits in both security and utility.

1. Attribution: all code is linked to a developer.
2. Allow dynamic linking to libraries written by the same developer, i.e. multiple apps from the same developer can share libraries.
3. Prevent installation of third-party apps that have not passed the review process.
4. Prevent apps from being modified at any time since the signature is checked on every load.

4.1.1 Masque Attack

Despite the code signing, in iOS 7 and 8, any unauthorized app can still be installed on iOS devices with Masque attack [3, 4]. The attack exploits the fact that iOS did not check certificates for apps with the same bundle identifier during updates. With some phishing, an adversary can lure users to install a malicious app placed in a bundle with the same bundle identifier as that of one of the legitimate apps already on the user's device. The malicious app will replace the legitimate one, but all user's data is maintained and thus, readable by the malicious app.

4.2 iOS Sandbox

All third-party apps run in iOS sandboxes. Interfaces to other apps and system resources are managed by iOS. Apps always run in the user mode, and the OS partition is mounted as read-only. In order to communicate with OS, Apple uses *entitlements* instead of setting a *uid bit* due to the risk of privilege escalation attacks.

4.2.1 Entitlements

Entitlements are key-value pairs set at compile time and used to authenticate an application's access to system features or data. iOS API verifies that an app has a necessary entitlement for the calls it tries to make instead of giving it a privilege mode. Entitlements are signed with developer's certificates so that they cannot be changed later on and are also verified during the reviewing process.

4.2.2 ARM's Execute Never

iOS further prevents execution of malicious code by using ARM's Execute Never (XN) feature. In a high level, a bit is assigned to each of the memory pages marking them as non-executable by the processor. This feature mitigates dynamically loaded shellcode such as buffer overflow, but does not prevent Return-Oriented Programming [5]. Memory pages can be written and executed by certain apps under a very controlled circumstance, requiring specific Apple-only dynamic code-signing entitlements. An example of an app that is allowed this access is Safari for its JavaScript JIT compiler.

4.3 Extensions

An app can provide its functionality to other apps without leaking the sensitive data they contain through a mechanism called *extensions*. Extensions are signed executable binaries that come with an app and can be activated by other apps. The system automatically detects and finds a matching extension if possible. Extensions are run by the system and communicate with the app that activates it through inter-process

communications also managed by the system.

Similarly to applications, extensions run in sandboxes and have a separate address space. They are isolated from each other, from the app that activates it, and from the app that contains it. They, however, do share the same access control as the containing app. For instance, custom keyboards provide a specific type of extensions that is heavily restricted on their access to network and files. Since a keyboard extension can see everything the user types, it is rid of any capability to leak this information.

4.4 Remote Access

Secure remote access is a broad topic, but here we will focus on Apple's HomeKit and HomeKit accessories. HomeKit application uses Ed25519 public-private key pairs generated on iOS devices as its identity and encryption keys. The keys are stored in the system's Keychain. Homekit accessories also generate their own Ed25519 keys used for communicating with iOS devices, and the keys are newly generated after every factory reset.

To establish the first connection between an iOS device and a HomeKit accessory, the keys exchange protocol, Secure Remote Password, is used. It relies on a specific eight-digit code embedded in the accessory during its manufacture. The user is given the code during purchase of the accessory and must correctly input the code on the iOS device to establish the key exchange. On the accessory's side, the key exchange is handled by Apple's custom MFi IC chip. After the first connection is established, the communications are encrypted by the exchanged keys, and the eight-digit code is no longer needed.

HomeKit data is stored and synchronized using iCloud. All data stored locally and on the cloud are always encrypted using the user's HomeKit identity and a random nounce. The key only lives on the user's iOS device so the synchronized data cannot be read by the cloud or during transmission.

References

[1] ios security guide. `https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf`, September 2018. (Accessed on 11/14/2018).

[2] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[3] Apple ios "masque attack" technique — fireeye. `https://www.us-cert.gov/ncas/alerts/TA14-317A`, November 2014. (Accessed on 11/21/2018).

[4] Hui Xue, Tao Wei, and Yulong Zhang. Masque attack: All your ios apps belong to us masque attack: All your ios apps belong to us — fireeye inc. `https://www.fireeye.com/blog/threat-research/2014/11/masque-attack-all-your-ios-apps-belong-to-us.html`, November 2014. (Accessed on 11/21/2018).

[5] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.