iOS Security Framework: Understanding the Security of Mobile Phone Platforms

William Enck, Răzvan Deaconescu, Mihai Chiroiu, Luke Deshotels

Synonyms

mobile application security, mobile platform security, mobile device security

Definitions

iOS is one of the major mobile operating systems. Desgined and implemented by Apple, it powers iPhone and iPod touch.

Background

The wide-spread adoption of smartphones resulted from technological advancements in processing power, storage, communication, and power efficiency. Smartphones go beyond the simple features of storing contacts, making calls and sending text messages. They are defined by their ability for users to download and run third-party applications (or "apps"). These apps allow users to browse the web, engage in video calls, stream media content, play games, and organize information, all in a device that fits in a pocket.

iOS is the second most used mobile operating system, after Android. iOS is developed by Apple and runs on Apple iPhone and iPod touch. At its core, Apple iOS uses Darwin operating system and the XNU kernel, which is also used by Apple's other operating systems: iOS (used for Apple mobile devices), macOS (used for laptops, desktop systems and servers), tvOS (used for Apple TV), an watchOS (used for Apple Watch). iOS uses Darwin similar to the way Android uses Linux.

Unlike Android, iOS largely consists of proprietary, closed-source software. While Darwin is open source, a number of features relevant to iOS security

(e.g., the sandbox implementation) exist as proprietary kernel extensions. Another distinction from Android is the distribution model. Whereas Android is updated and modified by many different vendors (which leads to so called "fragmentation"), iOS only ships on Apple devices and hence there is greater continuity across the ecosystem.

The real power of a mobile platform are its applications. iOS applications are written using Objective-C or Swift. Application code is compiled as an executable and packed together with resources as an application package with a `.ipa` filename extension. Third-party applications are distributed to users via Apple's App Store, which is itself an application running on iOS.

New versions of iOS are released yearly by Apple, usually in September. At the time of writing, the latest major version is iOS 13, released in September 2019.

Application

The following discussion presents iOS security components following a timeline of developing and using a mobile application. We begin with application development and security features available to the developer. We then discuss runtime security components that prevent applications from abusing resources and leaking data. We end with security features that protect user data when the device is physically accessible to the attacker.

App Development

iOS applications are developed using Apple's official IDE, Xcode, and are written in either the Objective-C or Swift programming languages. Developers have access to a rich set of development frameworks provided by Apple, such as UIKit for defining the user interface, or AddressBook for accessing user contacts.

Developers are permitted to use "public frameworks," also called public APIs. However, iOS contains "private frameworks" (private APIs). Private frameworks used by Apple-provided apps and services. They are also sometimes indirectly called by public frameworks. Xcode does not allow application developers to call private frameworks directly. However, it is possible to bypass limitations and abuse iOS services and resources, as shown by Wang et al (2013).

During runtime, applications access resources and make requests to system services. Access to certain iOS services and resources requires an application to be provisioned with a *capability* specified by the developer in Xcode. Capabilities are implemented through *entitlements*; entitlements are key-value pairs added in the application binary executable file. The presence of an entitlement grants particular permissions to the application, such as accessing contacts or health information.

Before deploying the application, the developer registers for a developer ID with Apple. Using the developer ID, the developer creates a developer certificate for signing an application's binary executable. The developer packs the executable code and resource files (images, configuration files) into an

application package with the `.ipa` extension (iPhone Archive). Once signed, the application package may be deployed and run on local devices that have been provisioned with the developer ID, or it may be submitted to be published in the Apple App Store.

Apple performs compliance checking (sometimes called "app review" or "vetting") for all applications before publishing them to the App Store. After vetting, the application is signed by Apple and made available on the Apple App Store. Only apps signed by Apple may be run on any device. Apple provides guidelines for reviewing an app before submitting, including the requirement to only use public APIs.

When a user requests to download an application from the Apple App Store, the application code is encrypted with a unique key specific to the user's device. The new package is then downloaded and installed on the device. It is only decrypted at runtime using the device key.

The application package is installed in the iOS filesystem in an application home folder, typically located under `/var/containers/Bundle/Application/<UUID>/`[1]. The `<UUID>` is a unique ID for the application that is persistent across reboots; however, it changes at each new install of the application.

The application home folder contains a subfolder with the application's name and the `.app` prefix (e.g. `Facebook.app`). This folder stores application files, including the encrypted executable. When the application is started, the binary executable code is decrypted using the device decryption key and then loaded in memory and executed. Because the executable code resides decrypted in the device memory at runtime, it can be dumped. Security researchers commonly use a Jailbroken device and a debugger such as GDB to attach to a running process and dump the decrypted memory contents. This functionality has been automated by a tool called Clutch (2020).

[1] This location is subject to change in different iOS versions.

App Runtime

Users generally start an application by tapping its icon on the device home screen. The application is confined by iOS runtime access control mechanisms. First, all application processes run as the user `mobile` and are limited to actions allowed to this user. Second, the application home folder in `/var/container/Bundle/Application/<UUID>` acts as a filesystem container for the application, with most of its file operations being allowed only in this folder.

Application confinement is achieved using sandboxing, privacy settings, and hardcoded checks in system services. Sandboxing prevents collusion with other applications and unauthorized access to sensitive data. The sandbox implementation is generally referred to as the Apple Sandbox, though it was originally called Seatbelt. The Apple Sandbox is also used by macOS.

iOS sandboxes both third-party applications from the Apple App Store and as well as most system applications. When sandboxing an application, it is attached to a sandbox profile, i.e. a set of rules that define allowed actions. Rules in sandbox profiles allow or deny low-level actions, generally

mapped to system calls: file operations, inter-process communication, network operations. Sandbox rules are enforced in Darwin's XNU kernel, in the sandbox kernel extension.

Sandbox profiles are stored as binary blobs for efficient storage and processing. In recent iOS versions, sandbox profiles are stored in the sandbox kernel extension. Analysis of sandbox profiles is made difficult by their availability as only binary blobs; however, Sandblaster, created by Deaconescu et al (2016), can be used to reverse the binary sandbox profiles to a human readable policy.

Most system applications use a sandbox profile specific to the application. For example `BTServer` (the Bluetooth daemon process) uses the `BTServer` sandbox profile. Third-party applications, on the other hand, all use the same sandbox profile, named `container`. The `container` sandbox profile is by far the largest and most complex sandbox profile, as it is applied to all third-party applications and to some system applications.

Due to their size and complexity, sandbox profiles may contain flaws: incomplete or incorrect rules that allow applications to intentionally or non-intentionally abuse system resources or leak sensitive information. Deshotels et al (2016) proposed SandScout to analyze sandbox profiles, discovering six CVEs in the `container` sandbox profile.

Given that the `container` profile applies to multiple applications, it needs to adapt to specific application requirements needs. Sandbox profiles differentiate an application's access using *entitlements* and *sandbox extensions*. As mentioned above, entitlements are key-value pairs that are hardcoded in the signed application executable. On the other hand, sandbox extensions are non-permanent tokens that are granted to or revoked from an application during runtime. For example, the `tcc.kTCCServicePhotos` sandbox extension grants an application access to photos. This privilege may be revoked from an app, denying it access to photos.

Dynamic access control for applications is configured through iOS's privacy settings, which is accessed from the *Privacy* screen in the iOS `Settings` system app. Here, the user can grant an application access to resources or services such as Location, Photos, or Contacts. The user action of enabling or disabling a privacy setting is passed to the *Transparency, Consent and Control* (TCC) system. TCC stores privacy-related configurations in a database file (`TCC.db`) managed by the `tccd` service. Enabling a privacy setting for an application will request the TCC service to either "poke a hole" in the application sandbox profile by granting the application a sandbox extension, or to configure a target service to reply to the application request.

In general, applications require access to system resources (such as files) and system services (such as location services). Access to system resources is controlled by general UNIX permissions, such as running under the user `mobile` and file access permissions, and by sandbox rules, such as allowing or denying reading of information from a given file. Requests to system services are enabled through XPC (*Cross Process Communication*), an Apple-specific inter-process communication mechanism. Access to system services is controlled by sandbox rules and by specific rules encoded in the system

service implementation. Both sandbox rules and specific system service rules may allow or deny a request based on entitlements held by an application. Kobold, Deshotels et al (2020), uses a fuzzing-based approach to interrogate system services and highlight faulty configurations that result in misbehavior of system services, identifying three CVEs.

Protecting User Data

iOS also protects against physical attacks to the device. Here, an attacker may have direct access to the device, or indirect access by being near the device or convincing the user to use a malicious USB charging port or cable.

iOS prevents the execution of unsigned apps: only applications that have been signed by Apple or by a provisioned developer profile are allowed to run. Furthermore, Apple devices use secure boot to ensure the integrity of the operating system kernel. Secure boot is a trusted boot chain starting with the *bootrom*, a read-only program part of the hardware. The bootrom stores the Apple root CA public key that is used to verify the integrity of the bootloader, *iBoot*, and, if verification is passed, loads *iBoot*. Similar to the bootrom, *iBoot* checks the integrity of the kernel image and, if verification is passed, loads the kernel. In this way, only certified pieces of code can be run on the device, even if an attacker has tampered with the device.

Bypassing the kernel integrity protection of a device is called *jailbreaking*, which is similar to *rooting* an Android device. Jailbreaking a device provides full access to the root account and the filesystem and is usually used to install custom apps not found in the Apple App Store. It is also used by security researchers and analysts to study iOS security. Jailbreaking requires a flaw in the system that usually patches the kernel after the secure boot verification.

iOS protects user data using hardware-based encryption. On boot, the decryption key is only made available if the user enteres the correct passcode (also known as a PIN). Failure to insert the correct passcode several times results in authentication being disabled for increasing amounts of time. After booting, iOS provides more accessible ways of authenticating, including fingerprint authentication, also called Touch ID, and more recently, face recognition, also called Face ID, which was introduced with the iPhone X,

The passcode is not stored on the device. Instead, iOS uses a one-way function to derive a key from the passcode and compares that to a pattern stored on the device. In contrast, Touch ID and Face ID are stored on the device and compared to user input. As they are sensitive information, Touch ID and Face ID are stored in a hardware-secluded area in the device called the *Secure Enclave*.

The Secure Enclave is a hardware-based key manager, used for storing encryption keys, the Touch ID and the Face ID. The Secure Enclave runs on a specific processor called the Secure Enclave Processor (SEP), which is separated from the application processor. Communication between the application processor to the SEP is tightly controlled at the hardware level, preventing malicious behavior of the application software (including the kernel) from accessing the critical information. The

Secure Enclave is used by Apple Pay for payments, as Touch ID or Face ID is required to authorize a payment. Protecting Touch ID and Face ID in the Secure Enclave prevents flaws in software running on the application processor from affecting these financial transactions.

Cross-References

TODO

References

Clutch (2020) Clutch: Fast ios executable dumper. https://github.com/KJCracks/Clutch, accessed: 2020-02-07

Deaconescu R, Deshotels L, Bucicoiu M, Enck W, Davi L, Sadeghi AR (2016) Sandblaster: Reversing the apple sandbox. 1608.04303

Deshotels L, Deaconescu R, Chiroiu M, Davi L, Enck W, Sadeghi AR (2016) Sandscout: Automatic detection of flaws in ios sandbox profiles. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '16, p 704–716, DOI 10.1145/2976749.2978336, URL https://doi.org/10.1145/2976749.2978336

Deshotels L, Carabas C, Beichler J, Deaconescu R, Enck W (2020) Kobold: Evaluating decentralized access control for remote nsxpc methods on ios, to appear

Wang T, Lu K, Lu L, Chung S, Lee W (2013) Jekyll on ios: When benign apps become evil. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), USENIX, Washington, D.C., pp 559–572, URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_tielei