BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

# Mobile development and security

Session 16

**Karim Karimov**

Lecturer

# iOS architecture

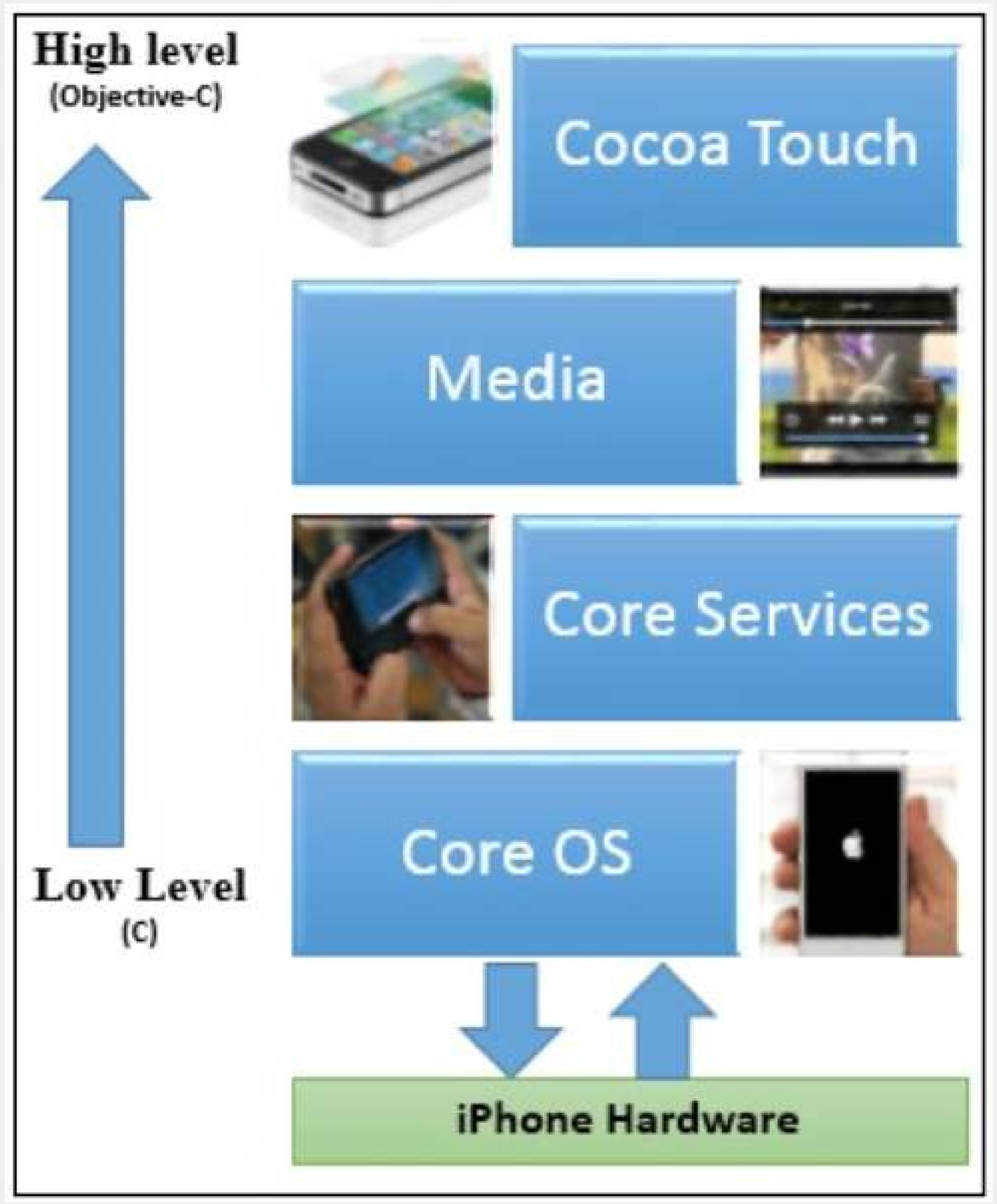
iOS is the operating system that runs on all Apple mobile devices (iPhones, iPads, and iPods), which it shares with the **Darwin foundation**.
Unlike other major operating systems, iOS manages the hardware device and provides the technologies required to build the applications on the platform. There are a few default system apps shipped along with the devices, such as Mail, Calendar, Calculator, Phone, Safari, and so on, which are typically used by users. It is not possible to run iOS and Mac OS X on any other hardware **apart from Apple**'s, and it is restricted to use iOS on any other mobile device apart from Apple's for security and commercial reasons. This has paved the way for jailbreakers to find iOS jailbreak attacks. The attack surface for applications has increased significantly, with more than 1 million applications in App Store (in 2016).

# iOS software stack

The iOS architecture is layered, and technologies are packaged as frameworks. A framework typically contains all the necessary libraries that are shared dynamically, and it also consists of images and header files. The following image illustrates the layers of the iOS software stack:

It consists of four abstraction layers:

- **Cocoa Touch**
- **Media**
- **Core Services**
- **Core OS**



# Cocoa Touch

The Cocoa Touch layer is bundled with a crucial set of frameworks, written in Objective-C, and developed based on the **Mac OS X Cocoa API**. The appearance of any app that you see in iOS is developed using the Cocoa Touch framework. Notifications, multi-tasking, touch-specific inputs, all the high-level system services, and other key technologies are supported by this layer and it also provides basic infrastructure support for an app.
The following is the list of important frameworks that are extensively used in this layer:

- The Address Book UI framework
- The Event Kit UI framework
- The Game Kit framework
- The Map Kit framework
- The Message UI framework and so on.

## Media

We often comment on multimedia experiences, particularly on sound clarity and video quality. This role is basically played by the media layer in the iOS stack, which provides the iOS with audio, video, graphics, and AirPlay (over-the-air) capabilities. As with the Cocoa Touch layer, the media layer includes a set of frameworks that can be utilized by developers:

- The Assets Library framework
- The AV Foundation framework
- The Core Audio framework
- The Core Graphics framework
- The Core Image framework
- The Core MIDI framework
- The Core Text framework
- The Core Video framework and so on.

## Core services

The core services layer provides the fundamental services that all applications can use. Like other layers, the core services layer provides a list of frameworks:

- The Accounts framework
- The Address Book framework
- The Ad Support framework
- The CFNetwork framework
- The Core Data framework
- The Core Foundation framework
- The Core Location framework
- The Core Media framework
- The Core Motion framework
- The Event Kit framework and so on

## Core OS

Core OS contains low-level fundamental services and technologies for end users. It comprises the OS X kernel. It taps the I/O reads between the CPUs and device. This is the layer that sits on top of the device hardware, which provides low-level networking, access to external accessories, and fundamental system services such as memory management, filesystem, and so on.
Core OS contains the following frameworks:

- The Accelerate framework
- The Core Bluetooth framework
- The External Accessory framework
- The Generic Security Services framework
- The Security framework

# iOS SDK and Xcode

The iOS software development kit provides resources, technologies, and tools to developers that can help them make better choices about how to design and implement apps. Developed and supported by Apple Inc. and released in February 2008 to develop native apps for devices, it was previously called the iPhone SDK. Xcode is the integrated development environment (IDE) suite developed by Apple for the development of iOS apps.

The following restrictions apply:

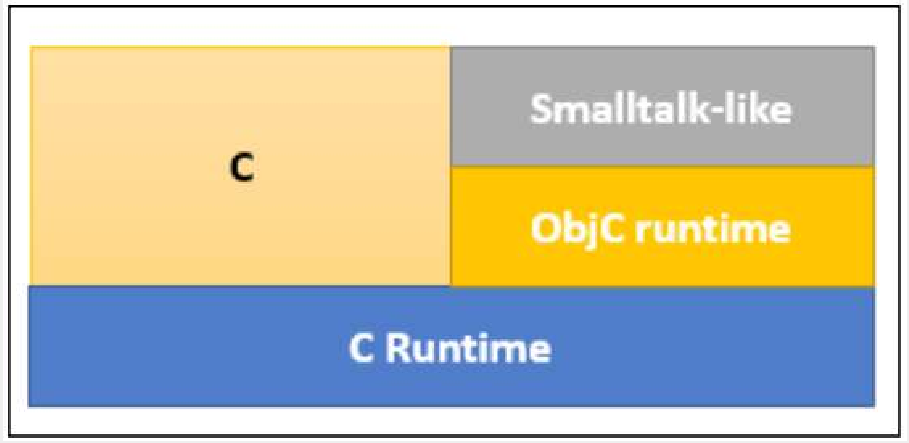
- These SDKs can only be installed on Mac OS X
- Apple does not impose a license on computers that are not running Mac OS X or are not Apple branded.

## iOS application programming languages

A majority of the apps developed for iOS are native apps; these are developed in Objective-C and, since 2015, Swift. Apple has mandated the use of Swift for developing apps. This would be easy for those who have some background in object-oriented programming languages.

# Objective-C

Objective-C is a strict superset of and augmentation to C; it is an object-oriented language that adds Smalltalk-style (an object-oriented, dynamically typed, reflective programming language) messaging to the C programming language and was created by Brad Cox and Tom Love in the early 1980s. This means that the Objective-C compiler can also compile C programs. The following diagram provides the sample Objective-C runtime and its components.
In Objective-C, one does not call the object one sends a message to. This language is mainly used on the Mac OS X and iOS operating systems and their APIs. The apps are compiled to native code and linked against the iOS SDK and Cocoa Touch frameworks.



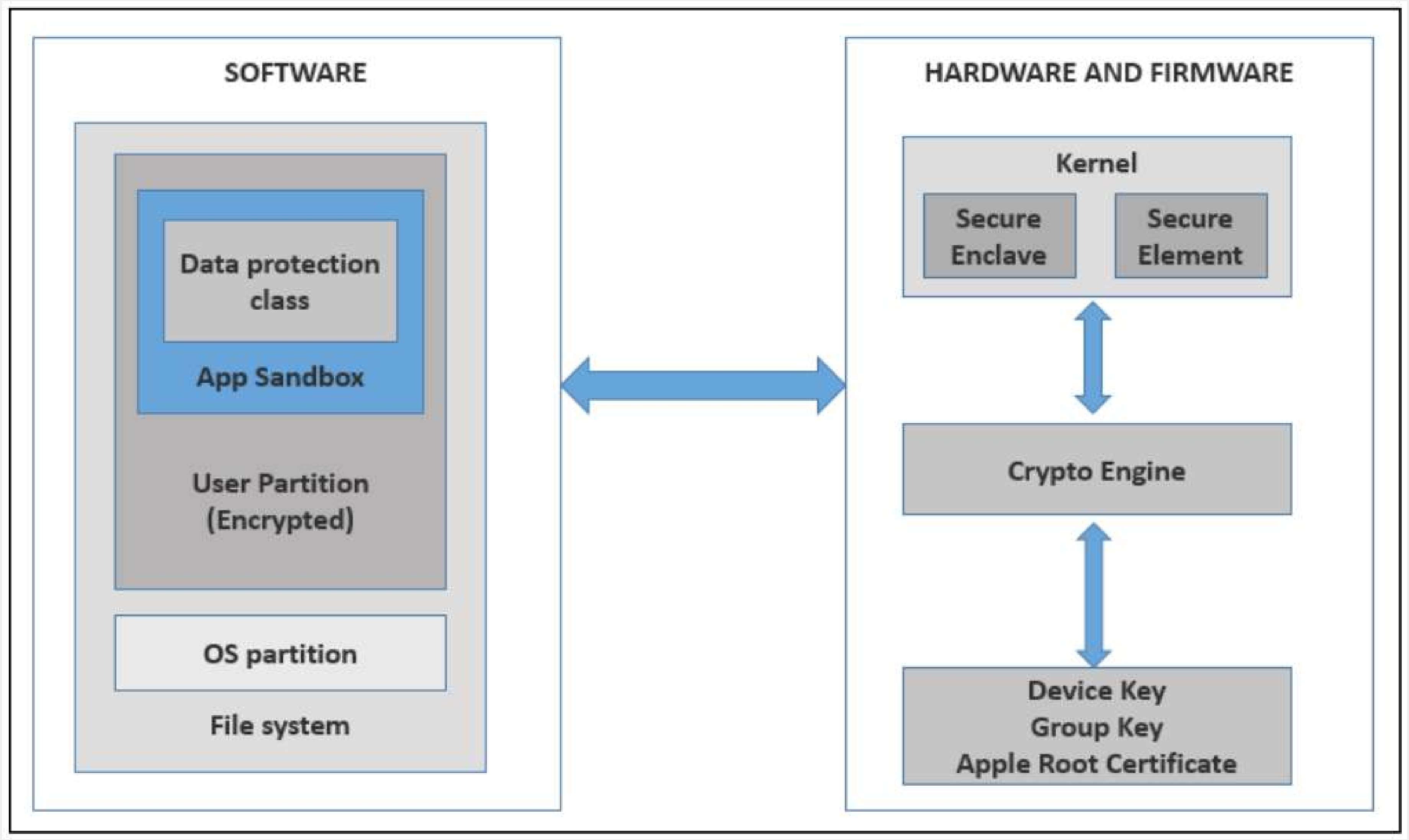
## Swift

Swift is a new programming language created by Apple Inc. specifically for iOS, OS X, and watchOS and is a replacement for Objective-C. It was first released on June 2, 2014, with a stable release on September 15, 2015.
This language is published open source now and you can contribute it at https://github.com/apple/swift

# Apple's iOS security model

The following diagram shows the security architecture of an iOS device and also provides an overview of security features implemented from the hardware level to software stack:

- Device-level security
- System-level security
- Data-level security
- Network-level Security
- Application-level security
- Hardware-level security



## Device-level security

At the device level, the security model ensures that unauthorized personnel cannot use a user's device. It enforces a device-level lock such as a **PIN** or **passcode**, remote wipe using mobile device management (MDM), and options such as activation lock and finding your phone. Strategically, Apple allows the signing of configuration profiles, thereby allowing companies to centrally distribute all configurations to the device in a secure way.
These kinds of configurations can restrict the device by applying a particular policy, for example, making it impossible to open an application on a device that is jailbroken.

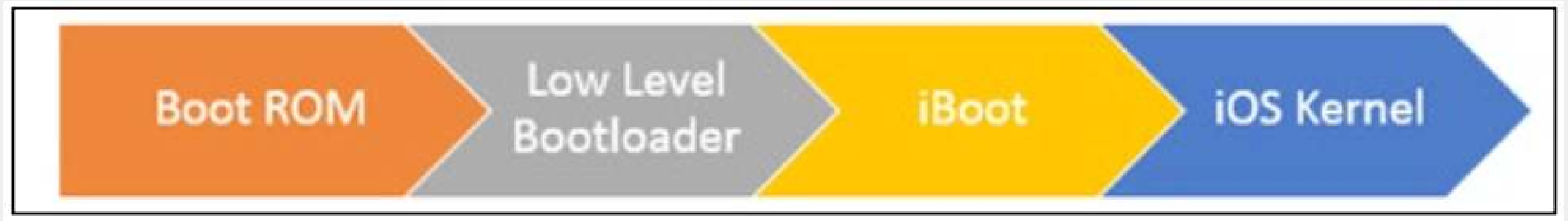
# System-level security

Apple designed the system-level security layer by authorizing system software on or before system updates and implementing a secure boot chain, Secure Enclave, and Touch ID.

**An introduction to the secure boot chain**

The mechanism that maintains the integrity of iOS from firmware initialization to loading the code into the iOS device is termed the secure boot chain or chain of trust. This chain ensures at all levels from hardware to software, making sure the code are trusted, tamperproof and run only on valid devices.

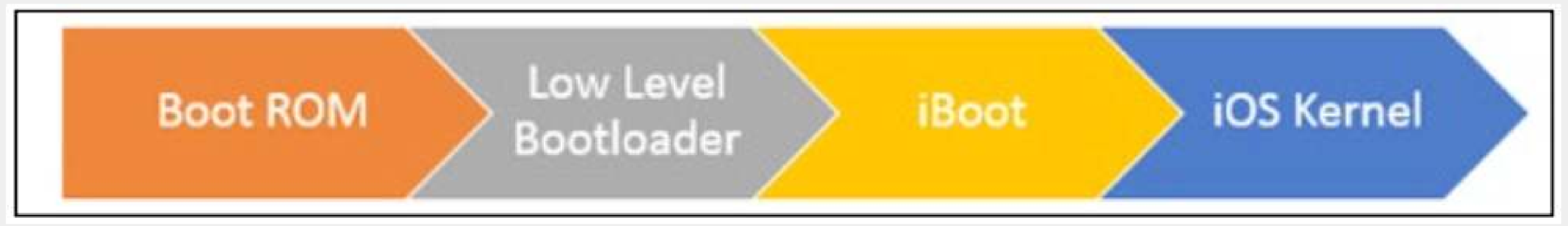
The following diagram shows the secure boot chain in an iOS device:



# System-level security

**An introduction to the secure boot chain**

The entire chain of trust is maintained since Apple signs every single step. In simple terms, when a device is booted, the processor executes the code from **Boot ROM**, which is also called the hardware root of trust, and it is essentially connected to the chip's fabrication, which includes Apple's root CA certificate. Before iOS loads, the **Low Level Bootloader (LLB)** needs to be signed by Apple. Once the LLB is passed and the verification is done, the next stage, **iBoot**, is loaded, and finally, the **iOS Kernel** is executed. iBoot normally acts like the second-level bootloader, which is responsible for verifying and loading the iOS Kernel into the device.



## System-level security

**System software authorization**
Normally, software update pushes in iOS are done through iTunes or over the air. The mechanism by which Apple prevents malicious users from downgrading the existing iOS version to a lower one is done through system software authorization.

# System-level security

**Secure Enclave**
To prevent kernel-level attacks, Secure Enclave was introduced at the hardware level to ensure that integrity is never compromised. This is independent from the application processor. Interestingly, the version of Secure Enclave used on the A7 or A8 Apple processors comes with unique IDs that are not known to Apple. Secure Enclave is also responsible for Touch ID sensors, fingerprint verification, and access approval.

**Touch ID**
This is nothing but the fingerprinting technology added by Apple to its devices, with which users can protect their devices from unauthorized access. However, even if Touch ID is enabled, it is possible to unlock the device with a valid PIN or passcode.

## Data-level security

The biggest challenge that developers have to deal with is data storage on mobile devices. Data-level security is primarily aimed at protecting data that is not in transit. This is normally achieved by enforcing encryption techniques using hardware and software components and also through data-protection classes. You can set up the device in such a way that it can remotely wipe all the data if a predefined number of attempts has been made to unlock the device in the case of a stolen or lost device.

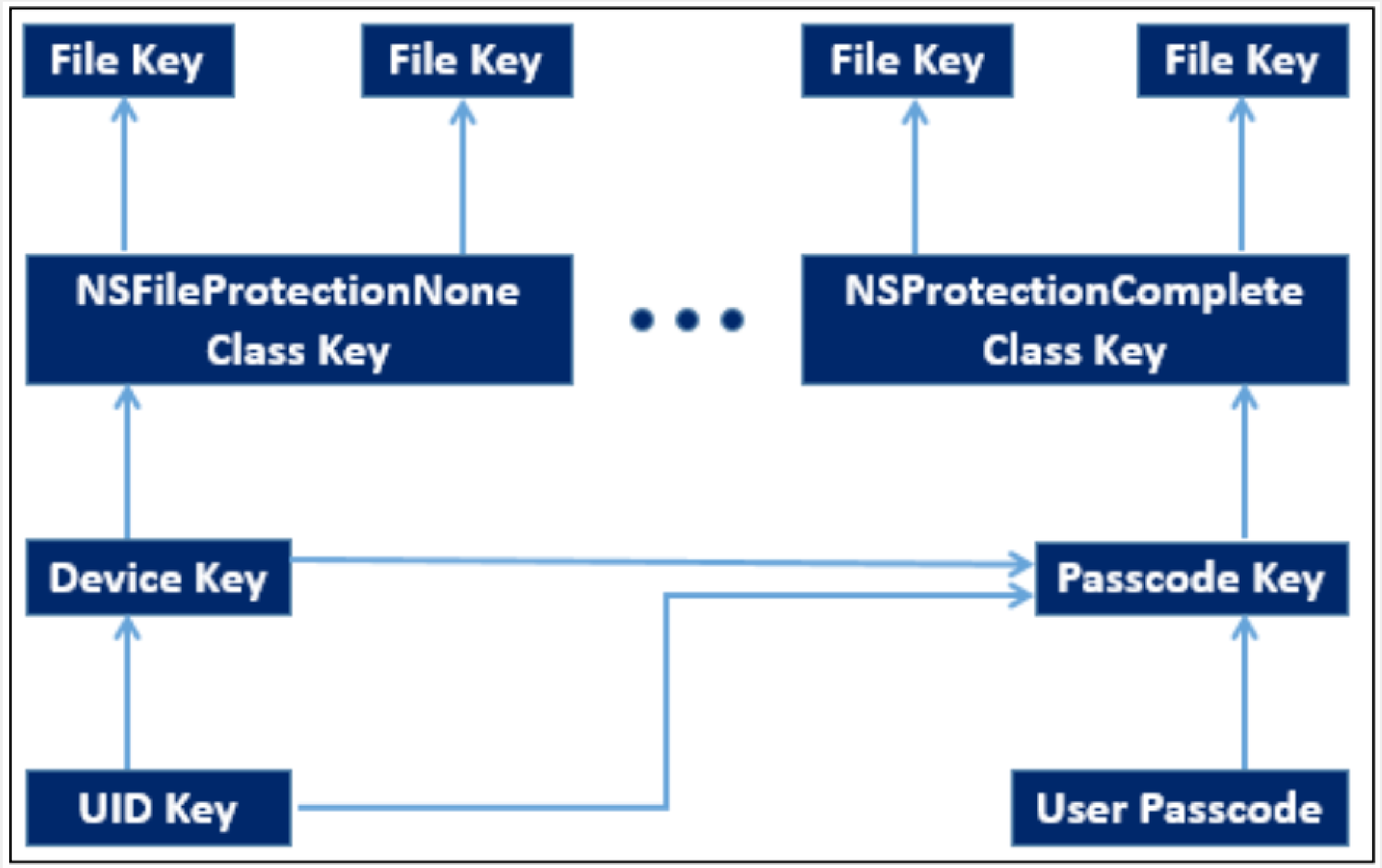
All the techniques involve encryption keys combined with device passcode or PIN.

# Data-protection classes

Here is a list of important data-protection classes:

- **NSFileProtectionComplete**: It provides complete protection; to access the file, one must always enter the passcode or use Touch ID.
- **NSFileProtectionCompleteUnlessOpen**: It provides complete protection to the file unless it is open.
- **NSFileProtectionCompleteUntilFirstUserAuthentication**: It provides complete protection to the file until it is opened. This is the class most commonly deployed by third-party application developers.
- **NSFileProtectionNone**: It provides no protection, but still, files in iOS are encrypted by default.

# Data-protection classes



## Data-protection classes

Application developers can protect files or keychain items by using data-protection classes. This normally includes whether the class protects the files or keychain items. As illustrated in the preceding diagram, on the left, **NSfileProtectionNone** indicates that data can be accessed any time even if the device is locked. On the right, the **NSProtectionComplete** class is used, which means that data can only be accessed if the device is unlocked either by passcode or fingerprint.

# Keychain data protection

A keychain is engaged by Apple to perform basic-level password management. Similar to the previous file data protection classes, keychain data is also protected with classes:

- **kSecAttrAccessibleAfterFirstUnlock**: Keychains can be accessed while the device is locked but in the case of a reboot, it requires an unlock before allowing access to data
- **kSecAttrAccessibleWhenUnlocked**: All the keychain data will be accessible when the device is unlocked
- **kSecAttrAccessibleAlways**: All the data is accessible at any point of time
- **kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly**: This is similar to **kSecAttrAccessibleWhenUnlocked**
- **kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly**: This similar to **kSecAttrAccessibleAfterFirstUnlock**, but data migration between devices through backups is not possible
- **kSecAttrAccessibleAlwaysThisDeviceOnly**: This is similar to **kSecAttrAccessibleAlways**, but data migration is not possible through backups

## Network-level security

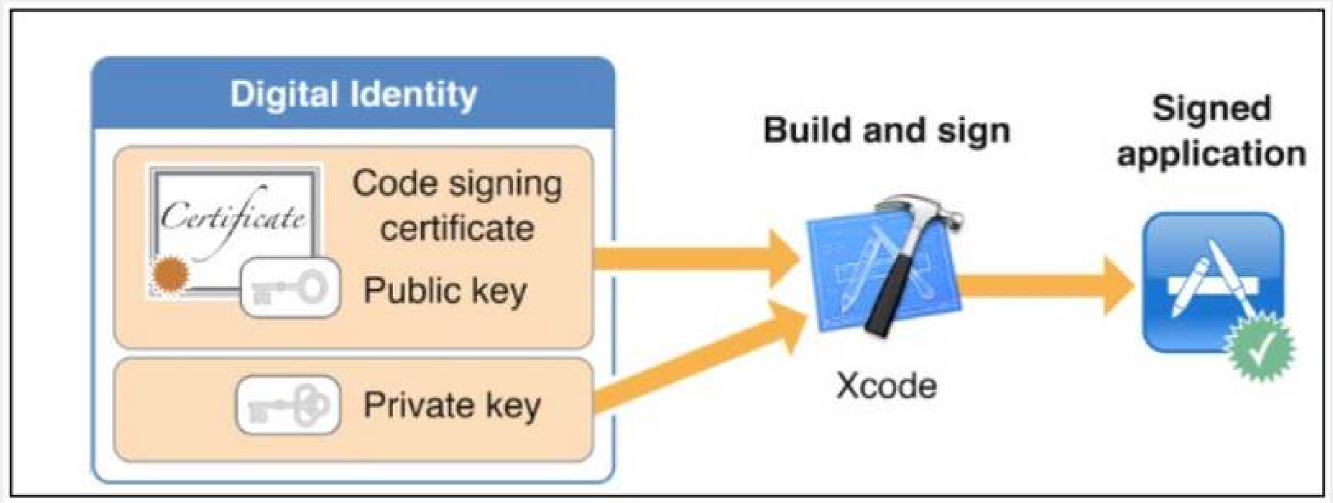
All data traversals over the network are protected using encryption technologies for VPN, applications, Wi-Fi, Bluetooth, Airdrop, and so on.
A majority of inbuilt applications, such as Mail and Safari, use Transport Layer Security by default (TLS version 1.0 to 1.2). Some important classes for a well-developed app include the **CFNetwork** class, which disallows SSLv3 connections. Also note the **NSURLConnection** and **NSURLSessionCFURL** APIs being used.
Apps that are compiled for iOS 9 automatically ensure that app transport security is enforced.

# Application-level security

Apple's close watch on app security allows plenty of layered approaches to protecting apps, using code signing, isolation mechanisms, and ASLR and stack-level protection.

**Application code signing**

The iOS app code-signing mechanism is similar to the one we saw in Android. However, iOS will not allow any application that is not signed by App Store. Each and every app installation will run through code signature checks during runtime.



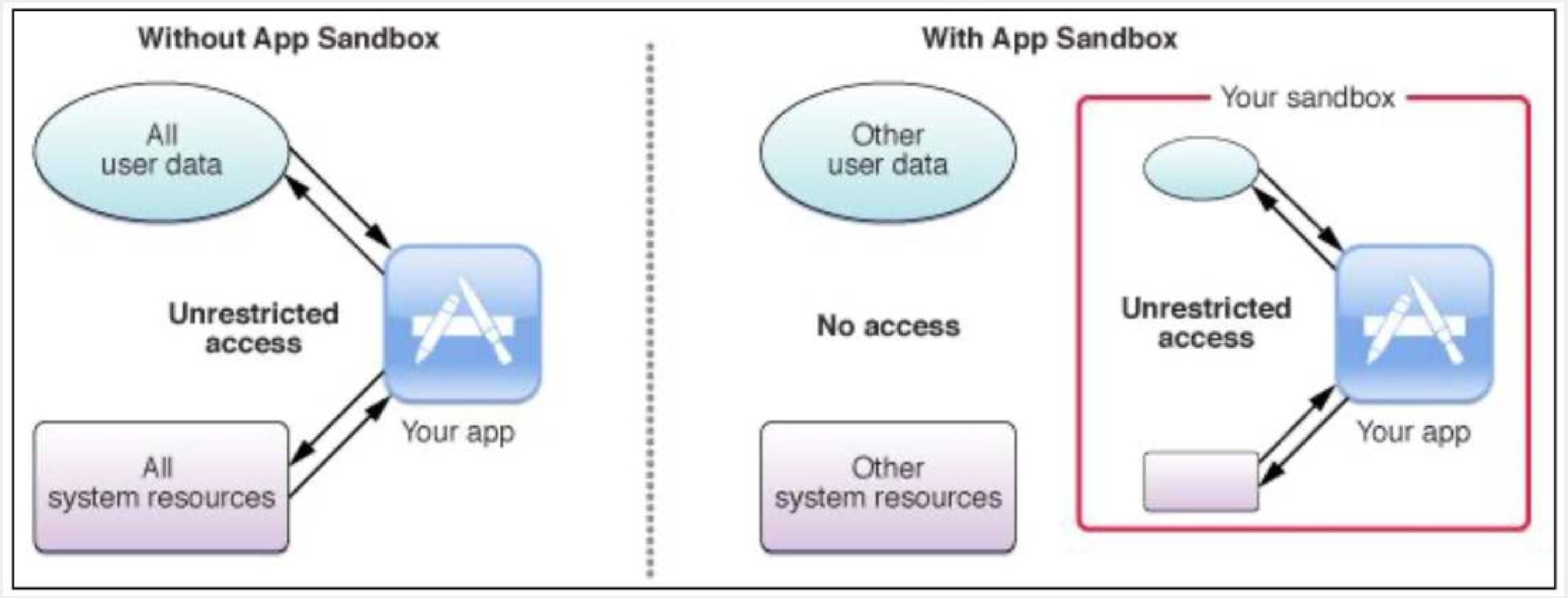
# Application-level security

**Application code signing**

The purpose of app signing is to verify whether the application that is being installed and run on your device originated from the company or person that it claims to have. However, app signing in iOS involves digital identification, which includes a developer-signed public key with a private key. Once the code is signed with the keys, it is eligible to be installed on the device. Only signed applications can be installed on a device Apple issues a set of credentials that can be used by the developers called code sign identity.

# The iOS app sandbox

The sandboxing techniques used in Android and iOS are pretty much similar.
iOS apps always run in a sandbox during installation time, and the sandbox is exclusively controlled by iOS in order to limit the app's access to various resources, such as files, hardware, preferences, and so on. By design the entire app is installed in its own sandbox directory, which would be the home for that particular app and its data. Apps can have unrestricted access without the sandbox mechanism, which is a possibility if the device is jailbroken.



## iOS isolation

The iOS operating system isolates each and every app on the system. Apps are not allowed to view or modify each other's data, business logic, and so on. Isolation prevents one app from knowing whether any other app is present on the system or whether apps can access the iOS operating system kernel until the device is jailbroken. This ensures a high degree of separation between the app and operating system.

iOS provides two types of isolation:

- **Process isolation**
- **Filesystem isolation**

## Process isolation

In process isolation, it is not possible for a random app to read another's memory region. Inter-app communication is restricted; there are no IPCs available for any process to communicate with another process.
All apps run in their own sandboxes. Apps are isolated not only from other apps but also from the operating system. By default, all apps on a device which is not jailbroken will be running as user mobile; the XNU kernel (similar to the Android Linux kernel) has a sandbox extension that separates the entire app using its own unique directory on the filesystem.

## Filesystem isolation

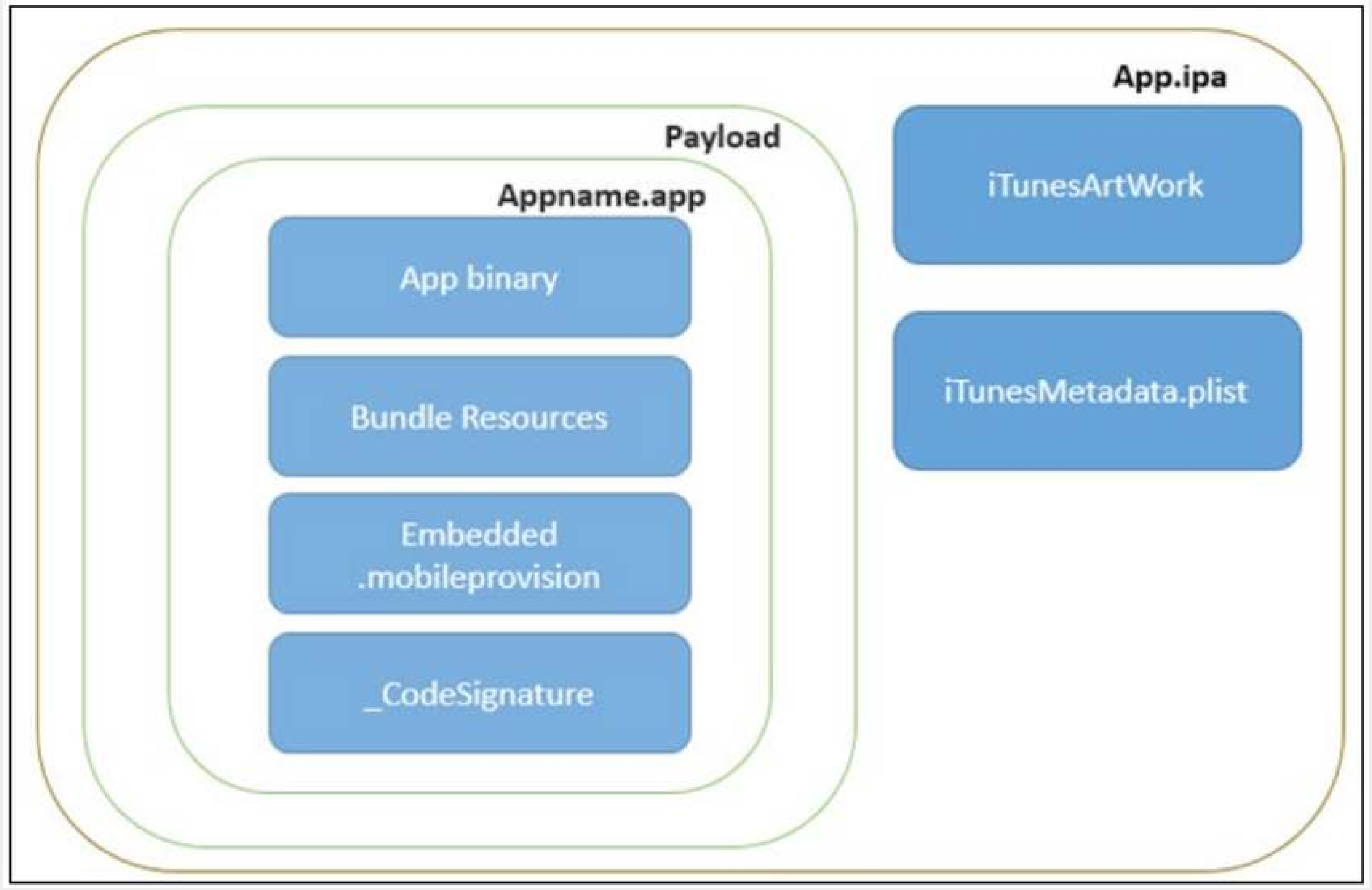
In filesystem isolation, if you have an app that actually saves a particular file onto the disk, any other app on the device cannot even know whether your app exists.
There are some stipulations around this: although there is a certain part of the iOS filesystem that is publicly readable, it is strictly read-only. This means no changes or modifications can be made, and there is no communication channel; however, it is still readable.

## Hardware-level security

iOS has very tight integration between hardware and software protection. All the devices built upon the Apple A8 or A7 processors provide cryptographic support. These devices use the AES (short for American Encryption Standard) 256 cryptographic engine and are built into a Direct Memory Access (DMA) path between the flash and main system memory. All devices are provided with a UID along with a device Group ID (GID), both of which are compiled at the processor level. A person testing the firmware will only be able to see the encryption and decryption of these techniques and will not have direct access.

## The iOS application structure

Now that we have understood the iOS security model and its permissions, we will see how all the compiled application code, resources, and application metadata required to define a complete application are zipped and signed with the developer's certificate and finally issued as an iOS app store package (iPA). The structural representation of an iOS application would typically be as shown in this diagram:



## The iOS application structure

When an iPA file is opened with any archiving software such as 7-Zip, WinRAR, and so on, you can see the following:

- **Payload**: This folder contains all the application data
    - **Application.app**: This folder contains all the following along with static images and other resources
        - **App binary**: This is the binary executable
        - **Bundle Resources**: All the resources required by the app binary are stored here
        - **Embedded.mobileprovision**: This file is the original provisioning file packaged with the application, and it helps the developers re-sign an iOS application without requiring Xcode
        - **CodeSignature**: This is responsible for verifying that every single byte within the .app file is exactly the same as when the application was signed by the developer

## The iOS application structure

- **iTunesArtwork**: This is an optional file, which is used by iTunesConnect when displaying your app's logo in the Store
- **iTunesMetadata.plist**: Contains the relevant application metadata, including details such as the developer's name, bundle identifier, and copyright information

## Jailbreaking

After looking at the security model, you might think that it takes somewhat more effort than Android to break into iOS apps. However, there are tech communities that are coming up with new ways of circumventing the security features implemented by iOS. Jailbreaking is one of the techniques used to remove the limitations imposed by the operating system on devices, through the use of software exploits.
Similar to Android rooting, jailbreaking your iPhone will also void your warranty and support from Apple, so do not use your personal device for testing purposes.

## Why jailbreak a device?

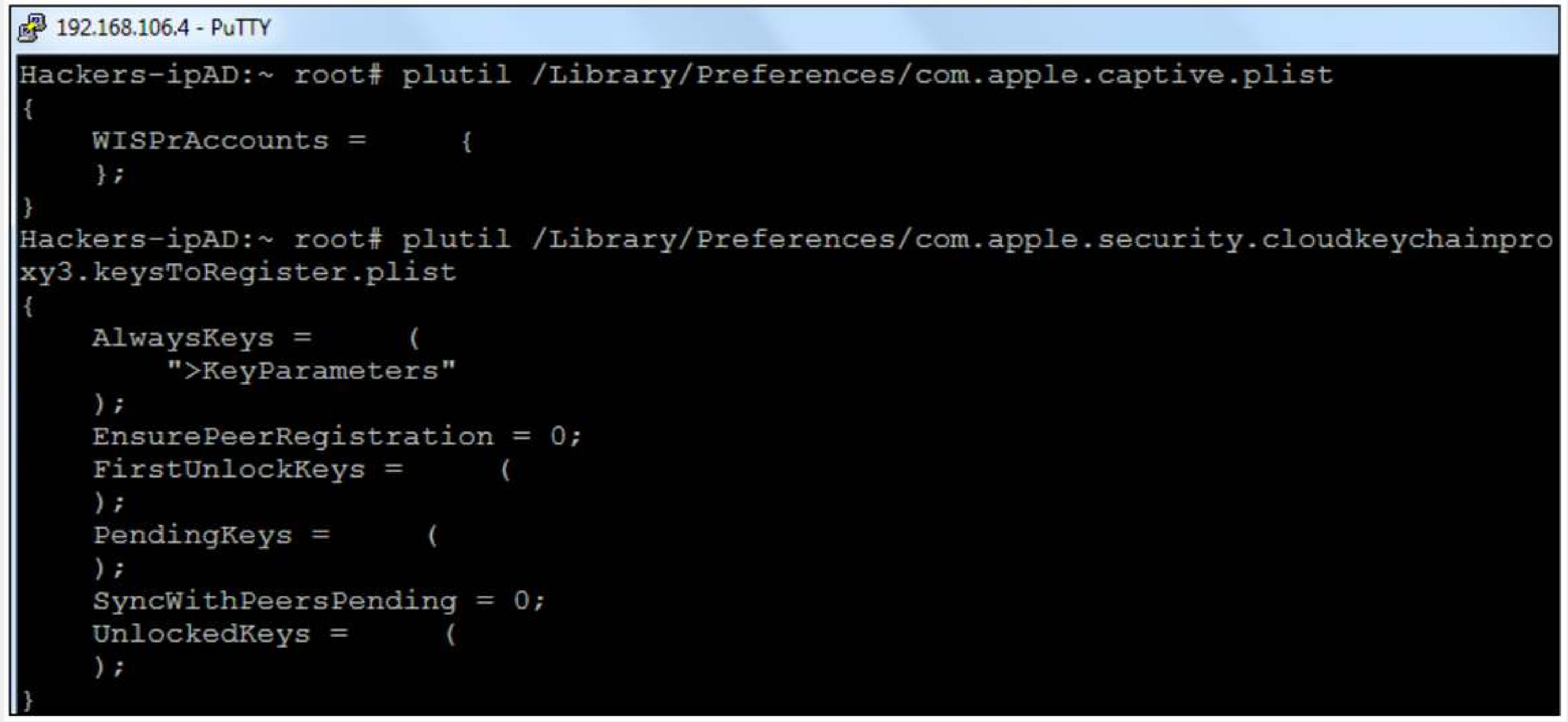
- You can change and customize the iOS interface
- You have full access to the iOS filesystem and device, which even allows you to remove built-in apps
- You can install custom apps or apps from non-traditional stores
- You can download other content for free (e-books, videos, music, and so on)
- There are big bounty programs

## Types of jailbreaks

- **Untethered jailbreaks**
    - An untethered jailbreak is the preferred type of jailbreak, since it allows the device to run all apps and tweaks even after rebooting, with no consequences.
- **Tethered jailbreaks**
    - A tethered jailbreak is the least desired jailbreak of all; it requires you to plug your device in to the computer to start it up because the device needs some code from a program on the computer that will let it boot up. The reason it needs this code is because the device checks for unsigned software running on it and it will not let itself boot up without the code on the computer.
- **Semi-tethered jailbreaks**
    - A semi-tethered jailbreak allows you to boot the device without plugging it in to the computer, but you will not be able to use the jailbroken add-ons and tweaks until you boot up the system using a program such as Redsnow.

# Property lists

Property lists are nothing but XML files that are used to store application data. These files use the .plist extension and are often used to store the settings information of a third-party application. The **NSDefaults** class is used in property lists; typically, these are stored in the **/Library/Preferences** folder in the iOS filesystem.
Property lists can be accessed using the **plutil utility**, as shown in this screenshot:



```
Hackers-ipAD:~ root# plutil /Library/Preferences/com.apple.captive.plist
{
    WISPrAccounts =     {
    };
}
Hackers-ipAD:~ root# plutil /Library/Preferences/com.apple.security.cloudkeychainpro
xy3.keysToRegister.plist
{
    AlwaysKeys =     (
        ">KeyParameters"
    );
    EnsurePeerRegistration = 0;
    FirstUnlockKeys =     (
    );
    PendingKeys =     (
    );
    SyncWithPeersPending = 0;
    UnlockedKeys =     (
    );
}
```

## Exploring the iOS filesystem

Although a majority of our filesystem exploration will be interesting only when the device is jailbroken, it is also possible to access the filesystem on non-jailbroken devices and explore the files that are available. This is possible only when the device is paired with a PC. The versions of iOS 7 and later introduced a new feature that when a device is plugged in to a PC for pairing, the user is prompted to either trust the computer or not; earlier versions allowed pairing without issuing any alerts.
Some important file locations are summarized here:

- **/Applications**: All the system applications are stored in this location
- **/var/mobile/Applications**: Third-party applications are stored here; this has been replaced by the Containers folder in iOS 8 and later versions (**/private/var/mobile/Containers/Bundle/Applications**)
- **/private/var/mobile/Library/Voicemail**: This contains voicemail details
- **/private/var/mobile/Library/SMS**: This has SMS data

# Exploring the iOS filesystem

- **/private/var/mobile/Media/DCIM**: This contains photos
- **/private/var/mobile/Media/Videos**: Videos are stored here
- **/var/mobile/Library/AddressBook/AddressBook .sqlitedb**: This is the contacts database
- **/private/var/mobile/Library/Notes**: This contains notes information; sometimes, this includes passwords and usernames in plaintext
- **/private/var/mobile/Library/CallHistory**: This has the call history backup
- **/private/var/mobile/Library/Mail**: This contains the entire mail history
- **/private/var/mobile/Library/Calendar/**: This has calendar information

# References

- **Darwin operating system** - https://en.wikipedia.org/wiki/Darwin_(operating_system)
- **Cocoa application layer** - https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html
- **Media layer** - https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/MediaLayer/MediaLayer.html
- **Core services layer** - https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreServicesLayer/CoreServicesLayer.html
- **Plutil utility** - https://scriptingosx.com/2016/11/editing-property-lists/

BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

# End of the session