# Chapter 2 Introduction to Jailbroken iOS

Compared with what we see on Apps’ UI, we are more interested in their low-level implementation, which is exactly the motivation of reverse engineering. But as we know, non-jailbroken iOS is a closed blackbox, it has not been exposed to the public until dev teams like evad3rs, PanguTeam and TaiG jailbroke it, then we’re able to take a peek under the hood.

## 2.1 iOS System Hierarchy

For non-jailbroken iOS, Apple provides very few APIs in the SDK to directly access the filesystem. By refering to the documents, App Store developers may have no idea of iOS system hierarchy at all.

Because of very limited permission, App Store Apps (hereafter referred to as StoreApps) cannot access most directories apart from their own. However, for jailbroken iOS, Cydia Apps can possess higher permission than StoreApps, which enables them to access the whole filesystem. For example, iFile from Cydia is a famous third-party file management App, as shown in figure 2-1.

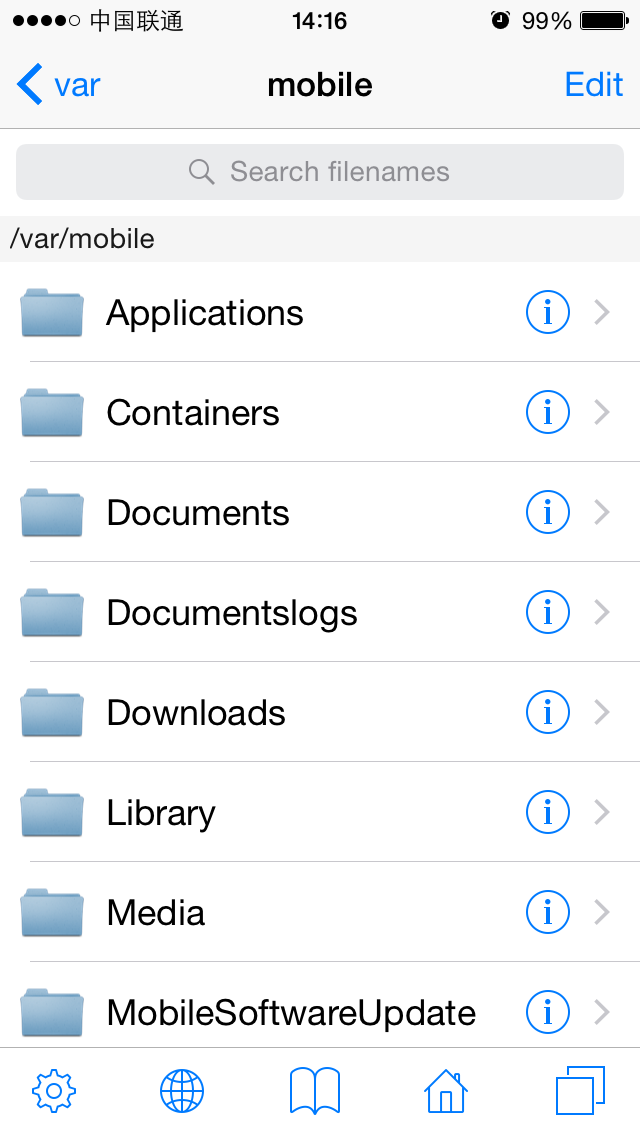


Figure 2- 1 iFile

With the help of AFC2, we can also access the whole iOS filesystem via software like iFunBox on PC, as shown in figure 2-2.

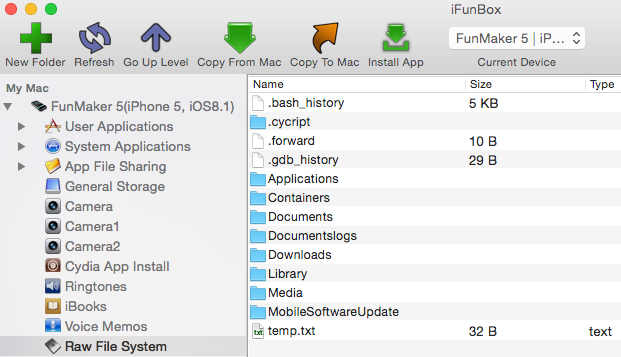


Figure 2- 2 iFunBox

Because our reverse engineering targets come right from iOS, being able to access the whole iOS filesystem is the prerequisite of our work.

### 2.1.1 iOS filesystem

iOS comes from OSX, which is based on UNIX. Although there are huge differences among them,they are somehow related to each other. We can get some knowledge of iOS filesystem from Filesystem Hierarchy Standard and hier(7).

Filesystem Hierarchy Standard (hereafter referred to as FHS) provides a standard for all \*NIX filesystems. The intention of FHS is to make the location of files and directories predictable for users. Evolving from FHS, OSX has its own standard, called hier(7). Common \*NIX filesystem is as follows.

* /

Root directory. All other files and directories expand from here.

* /bin

Short for “binary”. Binaries that provide basic user-level functions, like ls and ps are stored here.

* /boot

Stores all necessary files for booting up. This directory is empty on iOS.

* /dev

Short for “device”, stores BSD device files. Each file represents a block device or a character device. In general, block devices transfer data in block, while character devices transfer data in character.

* /sbin

Short for “system binaries”. Binaries that provide basic system-level functions, like netstat and reboot are stored here.

* /etc

Short for “Et Cetera”. This directory stores system scripts and configuration files like passwd and hosts. On iOS, this is a symbolic link to /private/etc.

* /lib

This directory stores system-level lib files, kernel files and device drivers. This directory is empty on iOS.

* /mnt

Short for “mount”, stores temporarily mounted filesystems. On iOS, this directory is empty.

* /private

Only contains 2 subdirectories, i.e. /private/etc and /private/var.

* /tmp

Temporary directory. On iOS, this directory is a symbolic link to /private/var/tmp.

* /usr

A directory containing most user-level tools and programs. /usr/bin is used for other basic functions which are not provided in /bin or /sbin, like nm and killall. /usr/include contains all standard C headers, and /usr/lib stores lib files.

* /var

Short for “variable”, stores files that frequently change, such as log files, user data and temporary files. /var/mobile/ is for mobile user and /var/root/ is for root user, these 2 subdirectories are our main focus.

Most directories listed above are rather low-level that they’re difficult to reverse engineer. As beginners, it’s better for us to start with something much easier. As App developers, most of our daily work is dealing with iOS specific directories. Reverse engineering becomes more approachable when it comes to these familiar directories:

* /Applications

Directory for all system Apps and Cydia Apps, excluding StoreApps, as shown in figure 2-3.

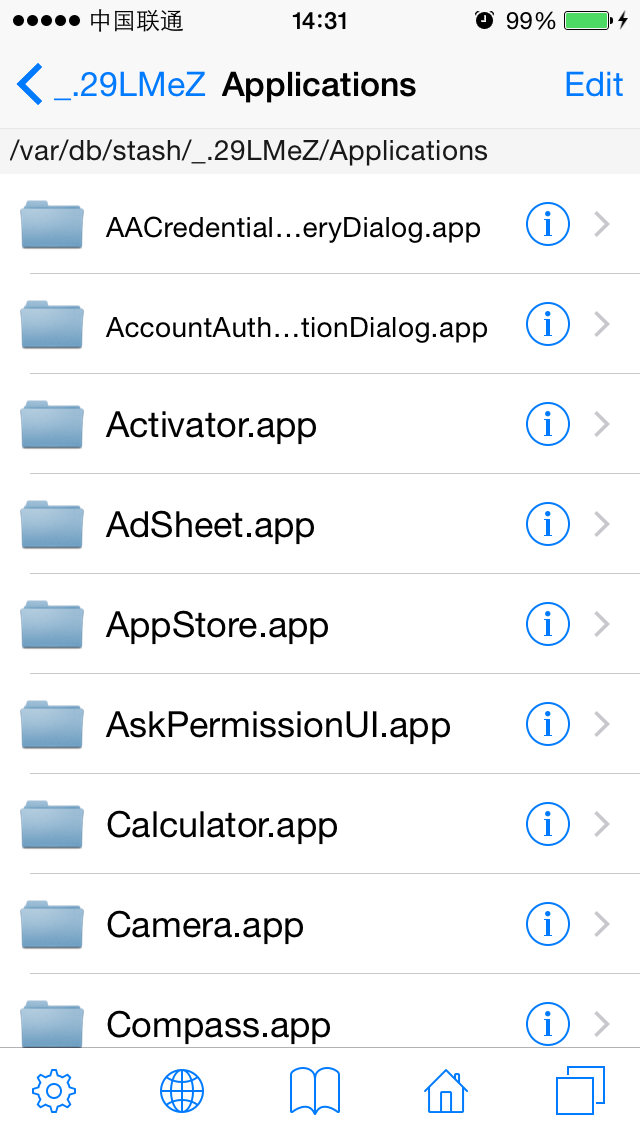


Figure 2- 3 /Applications

* /Developer

If you connect your device with Xcode and can see it in “Devices” category like figure 2-4 shows, a “/Developer” directory will be created automatically on device, as shown in figure 2-5. Inside this directory, there are some data files and tools for debugging.

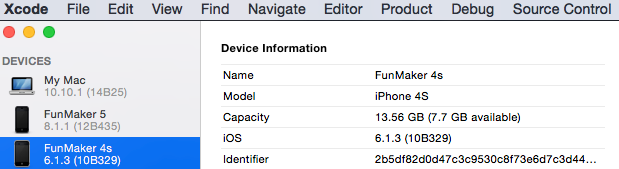


Figure 2- 4 Enable debugging on device

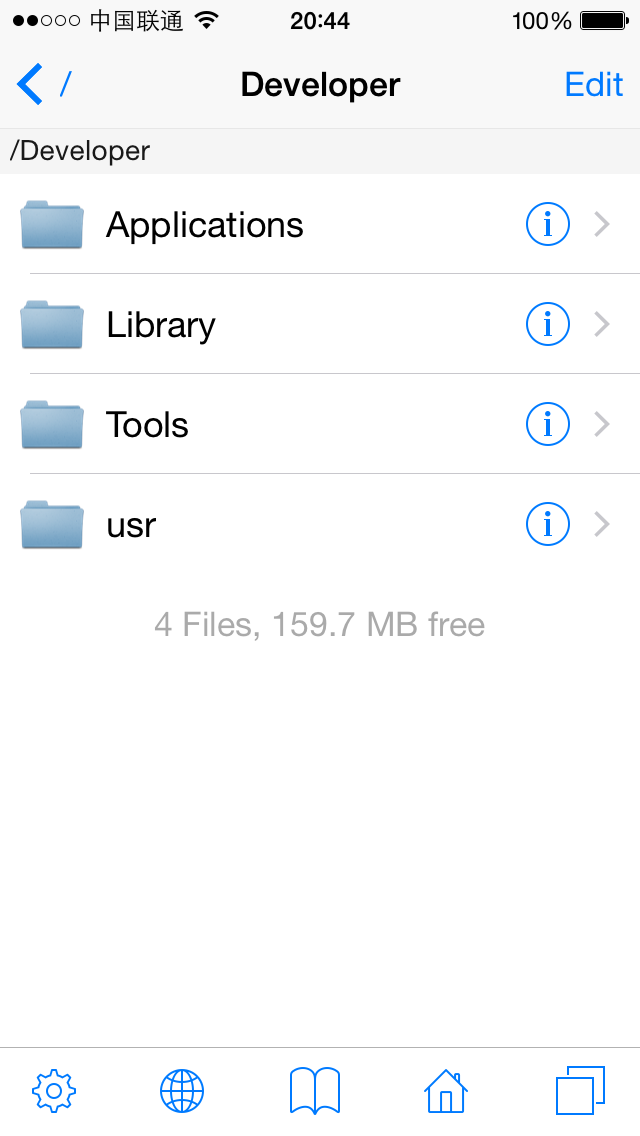


Figure 2- 5 /Developer

* /Library

This directory contains some system-supported data as shown in figure 2-6. One subdirectory of it named MobileSubstrate is where all CydiaSubstrate (formerly known as MobileSubstrate) based tweaks are.

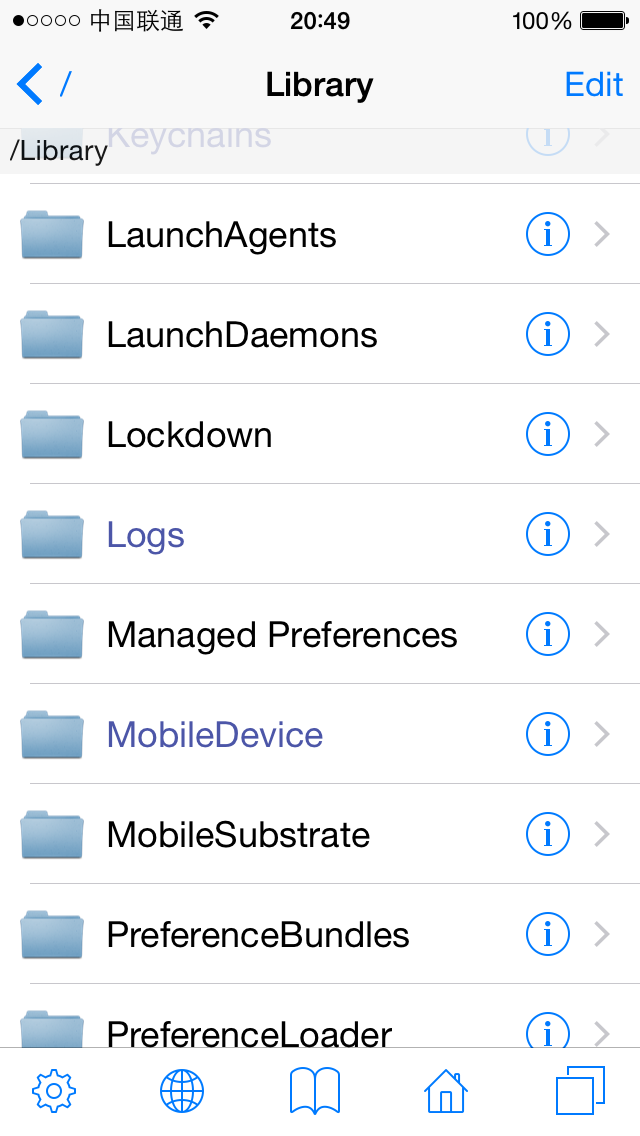


Figure 2- 6 /Library

* /System/Library

One of the most important directories on iOS, stores lots of system components, as shown in figure 2-7.

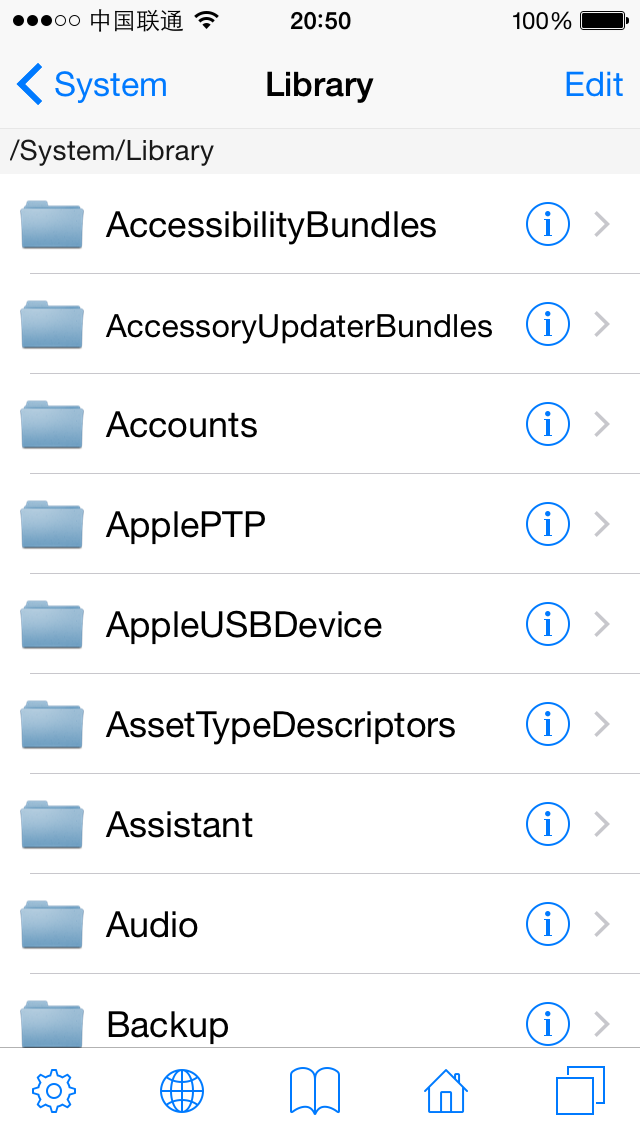


Figure2- 7 /System/Library

Under this directory, we beginners should mainly focus on these subdirectories:

* /System/Library/Frameworks and /System/Library/PrivateFrameworks:Stores most iOS frameworks. Documented APIs are only a tiny part of them, while countless private APIs are hidden in those frameworks.
* /System/Library/CoreServices/SpringBoard.app:iOS' graphical user interface, as is explorer to Windows. It is the most important intermediate between users and iOS.

More directories under “/System” deserve our attention. For more advanced contents, please visit <http://bbs.iosre.com>.

* /User

User directory, it’s a symbolic link to /var/mobile, as shown in figure 2-8.

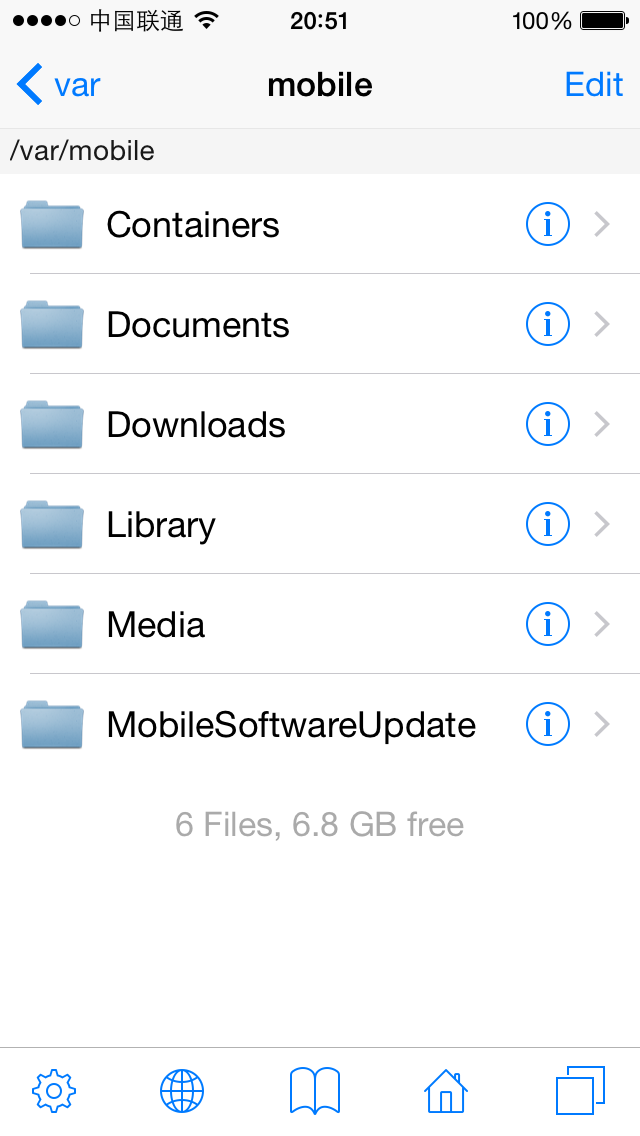


Figure 2- 8 /User

This directory contains large numbers of user data, such as:

* Photos are stored in /var/mobile/Media/DCIM;
* Recording files are stored in /var/mobile/Media/Recordings;
* SMS/iMessage databases are stored in /var/mobile/Library/SMS;
* Email data is stored in /var/mobile/Library/Mail.

Another major subdirectory is /var/mobile/Containers, which holds StoreApps. It is noteworthy that bundles containing Apps’ executables reside in /var/mobile/Containers/Bundle, while Apps’ data files reside in /var/mobile/Containers/Data, as shown in figure 2-9.

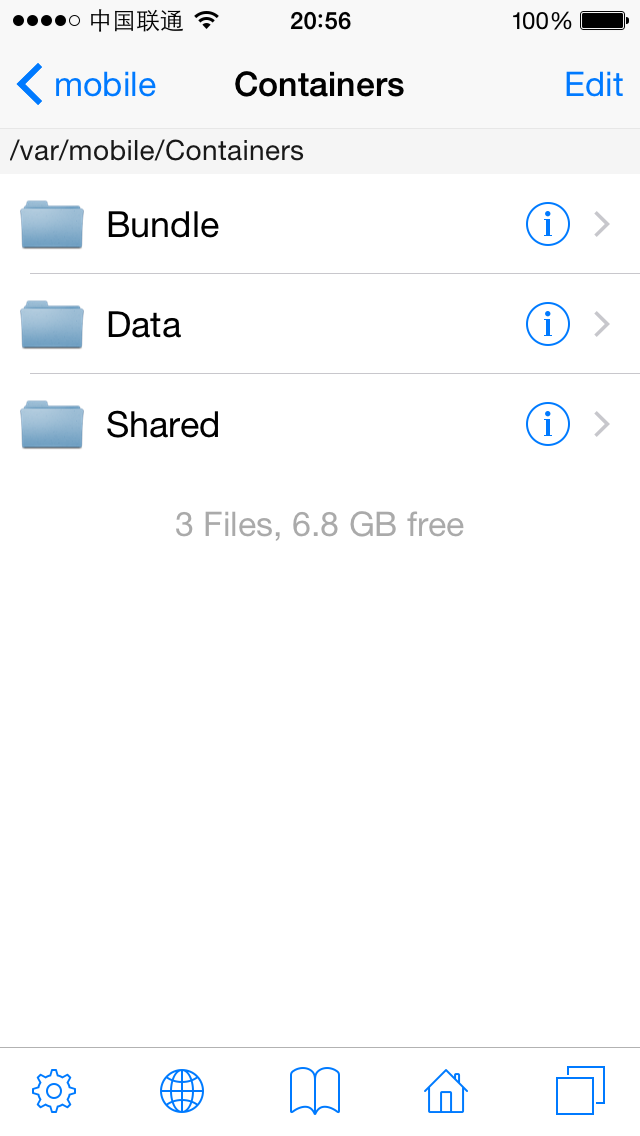


Figure 2- 9 /var/mobile/Containers

It’s helpful to have a preliminary knowledge of iOS filesystem when we discover some interesting functions and want to further locate their origins. What we’ve introduced above is only a small part of iOS filesystem. For more details, please visit <http://bbs.iosre.com>, or just type "man hier" in OSX terminal.

### 2.1.2 iOS file permission

iOS is a multi-user system. “User” is an abstract concept, it means the ownership and accessibility in system. For example, while root user can call “reboot” command to reboot iOS, mobile user cannot. “group” is a way to organize users. One group can contain more than one user, and one user can belong to more than one group.

Every file on iOS belongs to a user and a group, or to say, this user and this group own this file. And each file has its own permission, indicating what operations can the owner, the (owner) group and others perform on this file. iOS uses 3 bits to represent a file’s permission, which are r (read), w (write) and x (execute) respectively. There are 3 possible relationships between a user and a file:

* This user is the owner of this file.
* This user is not the owner of this file, but he is a member of the (owner) group.
* This user is neither the owner nor a member of the (owner) group.

So we need 3 \* 3 bits to represent a file’s permission in all situations. If a bit is set 1, it means the corresponding permission is granted. For instance, 111101101 represents rwxr-xr-x, in other words, the owner has r, w and x permission, but the (owner) group and other users only have r and x permission. Binary number 111101101 equals to octal number 755, which is another common representation form of permission.

Actually, besides r, w, x permission, there are 3 more special permission, i.e. SUID, SGID and sticky. They are not used in most cases, so they don’t take extra permission bits, but instead reside in x permission’s bit. As beginners, there are slim chances that we will have to deal with these special permission, so don't worry if you don’t fully understand this. For those of you who are interested, <http://thegeekdiary.com/what-is-suid-sgid-and-sticky-bit/> is good to read.

## 2.2 iOS file types

Rookie reverse engineers’ main targets are Application, Dynamic Library (hereafter referred to as dylib) and Daemon binaries. The more we know them, the smoother our reverse engineering will be. These 3 kinds of binaries play different roles on iOS, hence have different file hierarchies and permission.

### 2.2.1 Application

Application, namely App, is our most familiar iOS component. Although most iOS developers deal with Apps everyday, our main focus on App is different in iOS reverse engineering. Knowing the following concepts is a prerequisite for reverse engineering.

#### 1. bundle

The concept of bundle originates from NeXTSETP. Bundle is indeed not a single file but a well-organized directory conforming to some standards. It contains the executable binary and all running necessities. Apps and frameworks are packed as bundles. PreferenceBundles (as shown in figure 2-10), which are common in jailbroken iOS, can be seen as a kind of Settings dependent App, which is also a bundle.

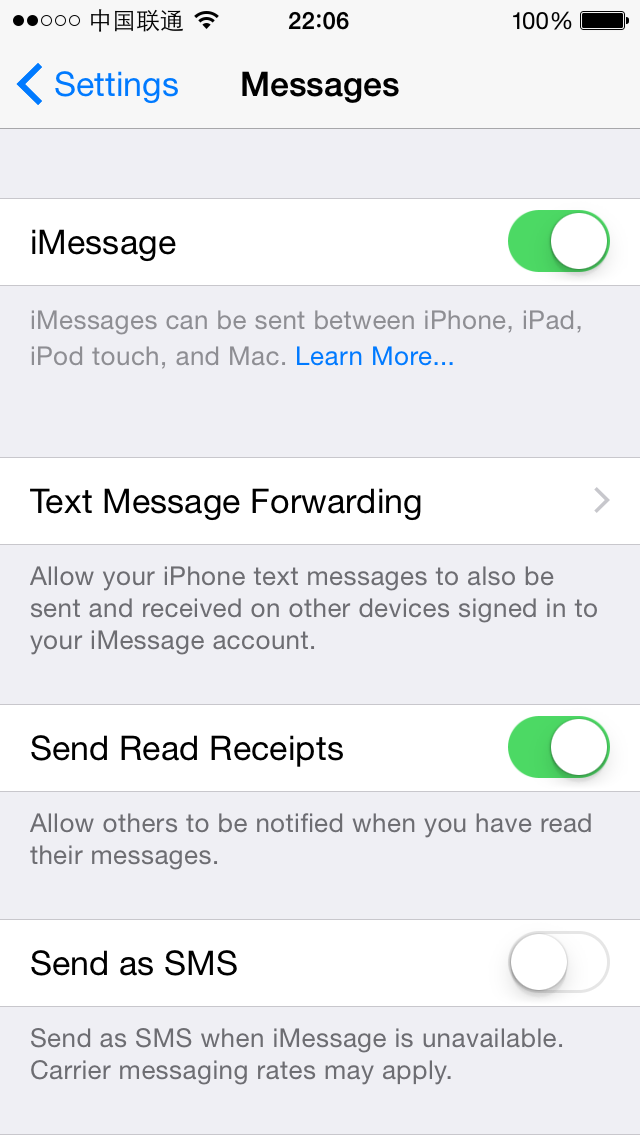


Figure 2- 10 PreferenceBundle

Frameworks are bundles too, but they contain dylibs instead of executables. Relatively speaking, frameworks are more important than Apps, because most parts of an App work by calling APIs in frameworks. When you target a bundle in reverse engineering, most of the work can be done inside the bundle, saving you significant time and energy.

#### 2. App directory hierarchy

Being familiar with App’s directory hierarchy is a key factor of our reverse engineering efficiency. There are 3 important components in an App’s directory:

* Info.plist

Info.plist records an App’s basic information, such as its bundle identifier, executable name, icon file name and so forth. Among these, bundle identifier is the key configuration value of a tweak, which will be discussed later in CydiaSubstrate section. We can look up the bundle identifier in Info.plist with Xcode, as shown in figure 2-11.

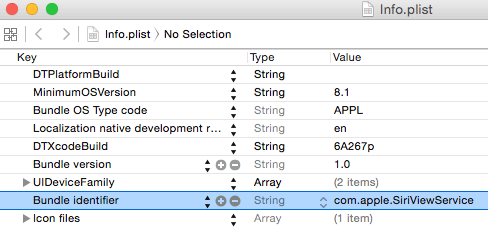


Figure 2- 11 Browse Info.plist in Xcode

Or use a command line tool, plutil, to view its value.

snakeninnysiMac:~ snakeninny$ plutil -p /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/SiriViewService.app/Info.plist | grep CFBundleIdentifier

"CFBundleIdentifier" => "com.apple.SiriViewService"

In this book, we mainly use plutil to browse plist files.

* Executable

Executable is the core of an App, as well our ultimate reverse engineering target, without doubt. We can locate the executable of an App with Xcode, as shown in figure 2-12.

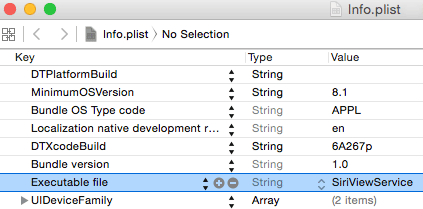


Figure 2- 12 Browse Info.plist in Xcode

Or with plutil:

snakeninnysiMac:~ snakeninny$ plutil -p /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/SiriViewService.app/Info.plist | grep CFBundleExecutable

"CFBundleExecutable" => "SiriViewService"

* lproj directories

Localized strings are saved in lproj directories. They are important clues of iOS reverse engineering. plutil tool can also parse those .string files.

snakeninnysiMac:~ snakeninny$ plutil -p /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/SiriViewService.app/en.lproj/Localizable.strings

{

"ASSISTANT\_INITIAL\_QUERY\_IPAD" => "What can I help you with?"

"ASSISTANT\_BOREALIS\_EDUCATION\_SUBHEADER\_IPAD" => "Just say “Hey Siri” to learn more."

"ASSISTANT\_FIRST\_UNLOCK\_SUBTITLE\_FORMAT" => "Your passcode is required when %@ restarts"

……

You will see how we make use of .strings in reverse engineering in chapter 5.

#### 3. System App VS. StoreApp

/Applications contains system Apps and Cydia Apps (We treat Cydia Apps as system Apps), and /var/mobile/Containers/Bundle/Application is where StoreApps reside. Although all of them are categorized as Apps, they are different in some ways:

* Directory hierarchy

Both system Apps and StoreApps share the similar bundle hierarchy, including Info.plist files, executables and lproj directories, etc. But the path of their data directory is different, for StoreApps, their data directories are under /var/mobile/Containers/Data, while for system Apps running as mobile, their data directories are under /var/mobile; for system Apps running as root, their data directories are under /var/root.

* Installation package and permission

In most cases, Cydia Apps’ installation packages are .deb formatted while StoreApps’ are .ipa formatted. .deb files come from Debian, and are later ported to iOS. Cydia Apps’ owner and (owner) group are usually root and admin, which enables them to run as root. .ipa is the official App format, whose owner and (owner) group are both mobile, which means they can only run as mobile.

* Sandbox

Broadly speaking, sandbox is a kind of access restriction mechanism, we can see it as a form of permission. Entitlements are also a part of sandbox. Sandbox is one of the core components of iOS security, which possesses a rather complicated implementation, and we’re not going to discuss it in details. Generally, sandbox restricts an App’s file access scope inside the App itself. Most of the time, an App has no idea of the existence of other Apps, not to mention accessing them. What’s more, sandbox restricts an App’s function. For example, an App has to ask for sandbox’s permission to take iCloud related operations.

Sandbox is not suitable to be beginners’ target, it’d be enough for us to know its existence. In iOS reverse engineering, jailbreak has already removed most security protections of iOS, and reduced sandbox’s constraints in some degree, so we are likely to ignore the existence of sandbox, hence leading to some strange phenomena such as a tweak cannot write to a file, or calls a function but it’s not functioning as expected. If you can make sure your code is 100% correct, then you should recheck if the problem is because of your misunderstanding of tweak’s permission or sandbox issues. Concepts about Apps cannot be fully described in this book, so if you have any questions, feel free to raise it on <http://bbs.iosre.com>.

### 2.2.2 Dynamic Library

Most of our developers’ daily work is writing Apps, and I guess just a few of you have ever written dylibs, so the concept of dylib is strange to most of you. In fact, you’re dealing with dylibs a lot: the frameworks and lib files we import in Xcode are all dylibs. We can verify this with ‘file’ command:

snakeninnysiMac:~ snakeninny$ file /Users/snakeninny/Code/iOSSystemBinaries/8.1.1\_iPhone5/System/Library/Frameworks/UIKit.framework/UIKit

/Users/snakeninny/Code/iOSSystemBinaries/8.1.1\_iPhone5/System/Library/Frameworks/UIKit.framework/UIKit: Mach-O dynamically linked shared library arm

If we shift our attention to jailbroken iOS, all the tweaks in Cydia work as dylibs. It is those tweaks’ existence that makes it possible for us to customize our iPhones. In reverse engineering, we’ll be dealing with all kinds of dylibs a lot, so it’d be good for us to know some basic concepts.

On iOS, libs are divided into two types, i.e. static and dynamic. Static libs will be integrated into an App’s executable during compilation, therefore increases the App’s size. Now that we have a bigger executable, iOS needs to load more data into memory during App launching, so the result is that, not surprisingly, App’s launch time increased, too. Dylibs are relatively “smart”, it doesn’t affect executable’s size, and iOS will load a dylib into memory only when an App needs it right away, then the dylib becomes part of the App.

It’s worth mentioning that, although dylibs exist everywhere on iOS, and they are the main targets of reverse engineering, they are not executables. They cannot run individually, but only serve other processes. In other words, they live in and become a part of other processes. Thus, dylibs’ permission depends on the processes they live in, the same dylib’s permission is different when it lives in a system App or a StoreApp. For instance, suppose you write an Instagram tweak to save your favorite pictures locally, if the destination path is this App’s documents directory under /var/mobile/Containers/Data, there won’t be a problem because Instagram is a StoreApp, it can write to its own documents. But if the destination path is /var/mobile/Documents, then when you save pictures happily and want to review them wistfully, you’ll find nothing under /var/mobile/Documents. All the tweak operations are banned by sandbox.

### 2.2.3 Daemon

Since your first day doing iOS development, Apple has been telling you "There is no real backgrounding on iOS and your App can only operate with strict limitations." If you are a pure App Store developer, following Apple's rules and announcements can make the review process much easier! However, since you're reading this book you likely want to learn reverse engineering and this means straying into undocumented territory. Stay calm and follow me:

1. When I’m browsing reddit or reading tweets on my iPhone, suddenly a phone call comes in. All operations are interrupted immediately, and iOS presents the call to me. If there is no real backgrounding on iOS, how can iOS handle this call in real time?
2. For those who receive spam iMessages a lot, firewalls like SMSNinja are saviors. If a firewall fails to stay in the background, how could it filter every single iMessages instantaneously?
3. Backgrounder is a famous tweak on iOS 5. With the help of this tweak, we can enable real backgrounding for Apps! Thanks to this tweak, we don’t have to worry about missing WhatsApp messages because of unreliable push notifications any more. If there is no real backgrounding, how could Backgrounder even exist?

All these phenomena indicate that real backgrounding does exist on iOS. Does that mean Apple lied to us? I don’t think so. For a StoreApp, when user presses the home button, this App enters background, most functions will be paused. In other words, for App Store developers, you’d better view iOS as a system without real backgrounding, because the only thing Apple allows you to do doesn’t support real backgrounding. But iOS originates from OSX, and like all \*NIX systems, OSX has daemons (The same thing is called Service on Microsoft Windows). Jailbreak opens the whole iOS to us, thus reveals all daemons.

Daemons are born to run in the background, providing all kinds of services. For example, imagent guarantees the correct sending and receiving of iMessages, mediaserverd handles almost all audios and videos, and syslogd is used to record system logs. Each daemon is consists of two parts, one executable and one plist file. The root process on iOS is launchd, which is also a daemon, checks all plist files under /System/Library/LaunchDaemons and /Library/LaunchDaemons after each reboot, then run the corresponding executable to launch the daemon. A daemons’ plist file plays a similar role as an App’s Info.plist file, it records the daemon’s basic information, as shown in the following:

snakeninnys-MacBook:~ snakeninny$ plutil -p /Users/snakeninny/Code/iOSSystemBinaries/8.1.1\_iPhone5/System/Library/LaunchDaemons/com.apple.imagent.plist

{

"WorkingDirectory" => "/tmp"

"Label" => "com.apple.imagent"

"JetsamProperties" => {

"JetsamMemoryLimit" => 3000

}

"EnvironmentVariables" => {

"NSRunningFromLaunchd" => "1"

}

"POSIXSpawnType" => "Interactive"

"MachServices" => {

"com.apple.hsa-authentication-server" => 1

"com.apple.imagent.embedded.auth" => 1

"com.apple.incoming-call-filter-server" => 1

}

"UserName" => "mobile"

"RunAtLoad" => 1

"ProgramArguments" => [

0 => "/System/Library/PrivateFrameworks/IMCore.framework/imagent.app/imagent"

]

"KeepAlive" => {

"SuccessfulExit" => 0

}

}

Compared with Apps, daemons provide much much lower level functions, accompanying with much much greater difficulties reverse engineering them. If you don’t know what you’re doing for sure, don’t even try to modify them! It may break your iOS, leading to booting failures, so you’d better stay away from daemons as reverse engineering newbies . After you get some experience reverse engineering Apps, it’d be OK for you to challenge daemons. After all, it takes more time and energy to reverse a daemon, but great rewards pay off later. The community acknowledged “first iPhone call recording App”, i.e. Audio Recorder, is accomplished by reversing mediaserverd.

## 2.3 Conclusion

This chapter simply introduces iOS system hierarchy and file types, which are not necessities for App Store developers, who don’t even have an official way to learn about the concepts. This chapter’s intention is to introduce you the very important yet undocumented system level knowledge, which is essential in iOS reverse engineering.

In fact, every section in this chapter can be extended into another full chapter, but as beginners, knowing what we’re talking about and what to google when you encounter problems during iOS reverse engineering is enough. If you have anything to say, welcome to <http://bbs.iosre.com>.