# Part III Theories

After you have learned the basic concepts of iOS reverse engineering from part 1 and then have tried tools mentioned in part 2 by yourself, you now are equipped with the fundamental knowledge of iOS reverse engineering. Once you’ve completed all previous examples in the book, you may be frustrated because you don't know what to do next. Actually, learning reverse engineering is a process of getting our hands dirty, but where and how to do that? Luckily, there are some good patterns for us to follow. In chapter 5 and 6, we will start from the perspective of Objective-C and ARM respectively, combine unique theories in iOS reverse engineering with tools we’ve mentioned before, then summarize a universal methodology of iOS reverse engineering. Let's get started!

# Chapter 5 Objective-C related iOS reverse engineering

Objective-C is a typical object-oriented programming language and most developers are surely proficient with its basic usage. Using Objective-C in the introductory phase of iOS reverse engineering can help us get a smooth transition from App development to reverse engineering. Fortunately, the file format used in iOS is Mach-O and it consists of enough raw data for us to restore the headers of binaries through class-dump or some other tools. With this information, we can start reverse engineering from the level of Objective-C, and writing tweaks is undoubtedly the most popular amusement at this stage. So let's start from writing tweaks.

## 5.1 How does a tweak work in Objective-C

When we were talking about Theos in chapter 3, we have introduced the concept of tweak already. From wikipedia, the definition of tweak is tools for fine-tuning or adjusting a complex system, usually an electronic device. In iOS, tweak refers to dylibs that can be used for enhancing the capabilities of other processes and they’re the most important part in jailbroken iOS.

Because of tweaks, jailbreak users can customize iOS based on their own preferences. Also, with tweak, developers are able to enrich the functionalities of other great software. All these facilities cannot be satisfied within the non-jailbroken iOS and AppStore. Almost all popular software in Cydia are various creative tweaks (A tweak icon is shown in figure 5-1), such as Activator, Barrel, SwipeSelection, etc. Generally speaking, the core of tweaks is a variety of hooks and most hooks target Objective-C methods. So how does a tweak work in Objective-C?

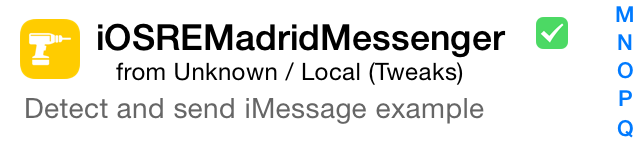


Figure 5- 1 Tweak icon

Objective-C is a typical object-oriented programming language; iOS consists of many small components and each component is an object. For example, every single icon, message and photo is an object. Besides these visible objects, there are also many objects working in the background, providing a variety of support for foreground objects. For instance, some objects are responsible for communicating with servers of Apple and some others are responsible for reading and writing files. One object can own other objects, such as an icon object has a label object, which displays the name of the App. In general, each object has its own significance. By combination of different objects, developers can implement different features. In Objective-C, we call the function of an object “method”. The behavior of method is called “implementation”. The relationship among objects, methods and implementation is where tweaks take effect.

If an object is provided with some certain function, we can send it a message like [object method] which lets the object perform its function, i.e. we can call the method of the object. So far, you may wonder that “object” and “method” are both nouns, where is the verb that used to perform the function? Good point, we lack a verb representing the implementation of “method”. So here, the word “implementation” can be the missing verb and it means that when we call the method, what does iOS do inside the method or in other words, what code is executed. In Objective-C, the relationship between method and its implementation is decided during run time instead of compile time.

During development, method in [object method] may not be a noun. Instead, it can be a verb. However, with only a brief [object method], we still don't know how this method works. OK, may be a little confusing. Let's take a look at the following examples.

1) When here comes a phone call, we may say that "Mom, answer the phone, please". When we want to translate this sentence into Objective-C, it will be [mom answerThePhone]. Here, the object is “mom” and the method is “answerThePhone”. The implementation could be “Mom stops cooking and goes to the sitting room to answer the phone”.

2) "snakeninny, come here and help me move out this item". This could be translated into [snakeninny moveOutTheItem]. The object here is “snakeninny” and method is “moveOutTheItem” while the implementation could be “snakeninny stops working and goes to the boss’ office to move a box downstairs”.

In the above examples, if there is no specific implementation, even we call a method of an object, the object still doesn't know what to do. So now, we can think implementation as the interpretation of method. Is it a little confusing? Don't worry. Let's draw an analogy between programming and dictionary. You can just imagine the method here to be a word in the dictionary and the implementation to be the meaning of that word. When you look up the dictionary, you always want to find what does a certain abstruse word mean. When it comes to programming, the implementation of a method does exactly the same as a word’s meaning in the dictionary. Easier to understand, right? Lets' move on.

As time goes on, the contents of dictionary have changed a lot and some old phrases have been given some new interpretations. For example, when talking along with Apple, which doesn't refer to the fruit, jailbreak is not considered a crime, and SpringBoard has nothing to do with a swimming pool. This phenomenon embodies in iOS especially. We can change the associated implementation of a method in order to change function of the object. As long as someone looks up a word in our modified dictionary, he or she will get the new meaning of the word. For example, in LowPowerBanner as shown in figure 5-2, the system will show a notification banner as a reminder to users when the device is in low battery. Interesting? It is because I have changed the implementation of low battery reminder from popup alerts to banners.



Figure 5- 2 LowPowerBanner

Another example is SMSNinja, as shown in figure 5-3. When you receive a spam message, SMSNinja puts the spam message into trash box automatically. This feature is achieved by changing the implementation of delegate method of receiving a message; I’ve added extra spam detecting function to the original method. This kind of approach is similar to changing the contents of dictionary and can be realized through the hook function provided by CydiaSubstrate. The usage of CydiaSubstrate has been explained in the last two chapters, so if you’ve already forgotten about it, you should go back and have a review.

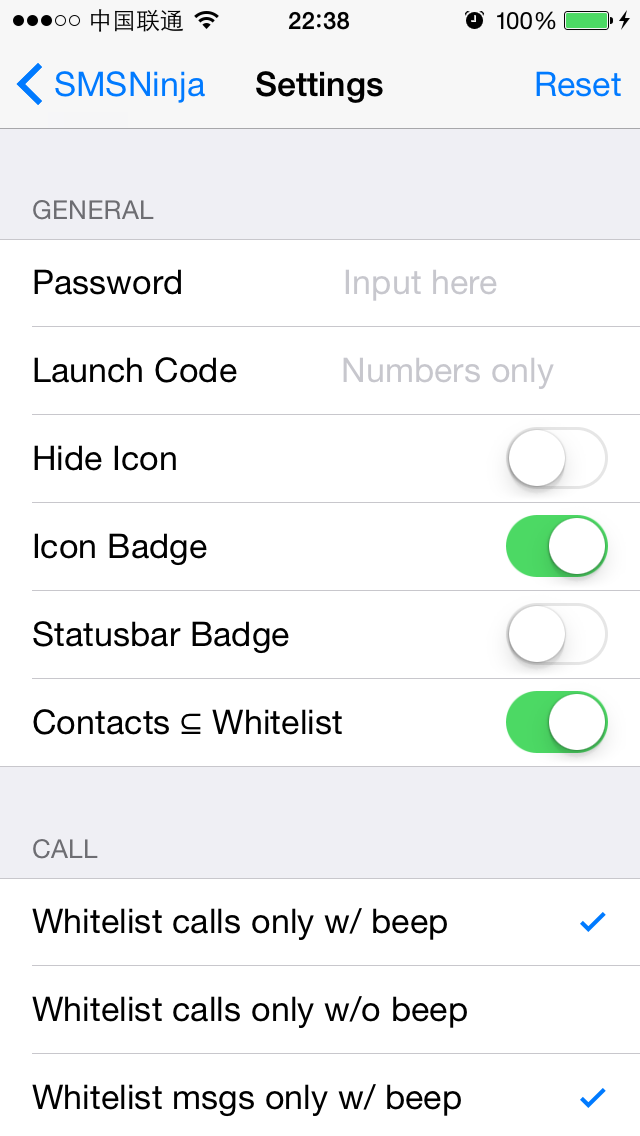


Figure 5- 3 SMSNinja

## 5.2 Methodology of writing a tweak

Not until understanding how tweaks work can we have a clear mind on what are our goals or what we are doing when we’re writing tweaks. Generally speaking, we use C, C++ and Objective-C to write a tweak. When we have an idea, how can we manage to turn it into a useful tweak? Actually, the pattern of writing a tweak is easy to follow and it will become clearer when you have deeper understanding with iOS and its programming language. In the following part, we will focus on a simple tweak example, start from the perspective of our most frequently used programming language Objective-C, to summarize theories of iOS reverse engineering from the level of Objective-C.

### 5.2.1 Look for inspiration

So far, some developers have already been able to write tweaks with knowledge introduced in the previous chapters, but most still don't know where to start. I know it’s uncomfortable when we don’t know where to use our abilities, so here are some tips to help you look for inspiration for your first tweak.

#### 1) Use more, observe more

Play with your iPhone and take a look at every corner of iOS whenever you have spare time rather than waste your time on social networks. Although iOS consists of lots of amazing features, it still cannot meet the exact requirements of every single user. So the more you use, the more you know about iOS and you are more likely to find where in iOS the user experience is not that good, which turns out to be ideas. With huge base of iOS users, you will surely find some users who share the same ideas with you. In other words, if you have a problem to solve, regard it as a tweak inspiration. That's how Characount for Notes was born on iOS 6. At that time, I always saved the content of a tweet into a note. Since a tweet has a 140 characters limit, I’ve written a tweak to show the character count of per note as a reminder. There was an Arabic user who sent mail to me to express his appreciation of this tweak and asked me to add more features to make it work like MSWord. But I was not interested in this idea, I had to say sorry to him.

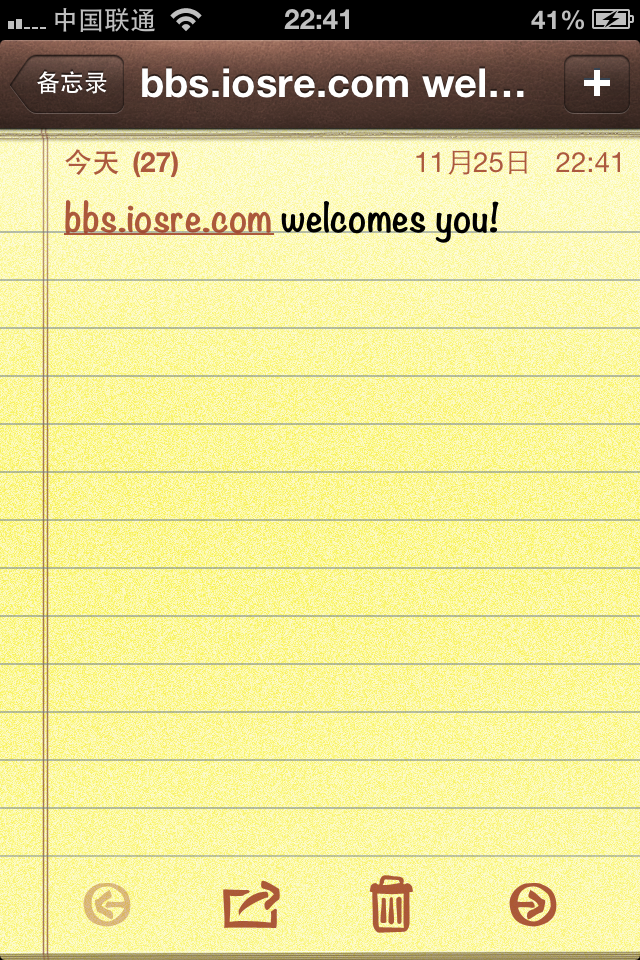


Figure 5- 4 Characount for Notes

#### 2) Listen to users' voice

Different people use iOS in different ways, which depends on their own requirements. If you don't have much inspiration, you can listen to the requirements of users. As long as there are requirements, there are potential users of your tweaks that meet these requirements.

If large projects have been done, you can write customized tweaks for minority. If you are not qualified to reverse low-level functions, you can start from simple functions of high level. After each release, listen to your users' feedbacks humbly and improve your tweaks with rapid iteration. Trust me, your effort will pay off. Take LowPowerBanner as an example, the idea of LowPowerBanner came from the suggestion of a user PrimeCode. I finished the first version of LowPowerBanner in less than 5 hours and it had no more than 50 lines of code. However, within 8 hours after the release, downloads had approached 30,000 (as shown in figure 5-5), the popularity of it was far beyond my expectation. Remember, users’ wisdom is inexhaustible. If you don't have any good ideas, listening to users would be surprisingly helpful!

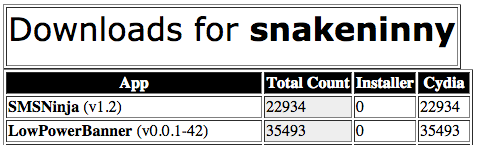


Figure 5-5 Downloads of LowPowerBanner 1.0

#### 3) Anatomize iOS

The greater your ability is, the more things you can do. Starting from writing small Apps, with more and more practices you will have deeper and deeper understanding of iOS. iOS is a closed operating system and only a tip of iceberg has been exposed to us. There are still far too many features that are worth to be further explored. Every time a new jailbreak comes out, someone will post the latest class-dump headers on the Internet. We can easily find the download link by searching “iOS private headers” on Google, which eliminates the trouble of class-dumping by ourselves. Objective-C methods follow a regular naming convention, making it possible for us to guess the meanings of most methods. For example, in SpringBoard.h:

- (void)reboot;

- (void)relaunchSpringBoard;

And in UIViewController.h:

- (void)attentionClassDumpUser:(id)arg1

yesItsUsAgain:(id)arg2

althoughSwizzlingAndOverridingPrivateMethodsIsFun:(id)arg3

itWasntMuchFunWhenYourAppStoppedWorking:(id)arg4

pleaseRefrainFromDoingSoInTheFutureOkayThanksBye:(id)arg5;

Browsing method names is an important source of inspiration as well as a shortcut for you to get familiar with low-level iOS. The more implementation details of iOS you master, the more powerful tweaks you can write. Audio Recorder, developed by limneos, is the best example. Even though the launch of iOS dates back to 2007, there is no feature like phone call recording until Audio Recorder’s born 7 years later. I'm sure that there are a lot of people who have the same idea and even have already tried to realize it by themselves. But why only limneos succeeds? It is because limneos has a deeper understanding of iOS than others. “Talk is cheap. Show me the code.”

### 5.2.2 Locate target files

After we know what functions we want to implement, we should start to look for the binaries that provide these functions. In general, the most frequently used methods to locate the binaries are as follows.

#### Fixed location

At this stage, our targets of reverse engineering are usually dylibs, bundles and daemons. Fortunately, the locations of these files are almost fixed in the filesystem.

1) CydiaSubstrate based dylibs are all stored in "/Library/MobileSubstrate/DynamicLibraries/". We can find them without effort.

2) Bundles can be divided into 2 categories, which are App and framework respectively. Bundles of AppStore Apps are stored in "/var/mobile/Containers/Bundle/Application/", bundles of system Apps are stored in “/Applications/”, and bundles of frameworks are stored in "/System/Library/Frameworks" and "/System/Library/PrivateFrameworks". For bundles of other types, you can discuss with us on http://bbs.iosre.com.

3) Configuration files of daemons, which are plist formatted, are all stored in "/System/Library/LaunchDaemons/", "/Library/LaunchDaemons" and "/Library/LaunchAgents/". The “ProgramArguments” fields in these files are the absolute paths of daemon exectuables, such as:

snakeninnys-MacBook:~ snakeninny$ plutil -p /Users/snakeninny/Desktop/com.apple.backboardd.plist

{

……

"ProgramArguments" => [

0 => "/usr/libexec/backboardd"

]

……

}

#### Locate by Cydia

Deb packages installed through command "dpkg -i" will be recorded by Cydia. You can locate these debs in Cydia’s "Expert" view under “Installed” category, as shown in figure 5-6.

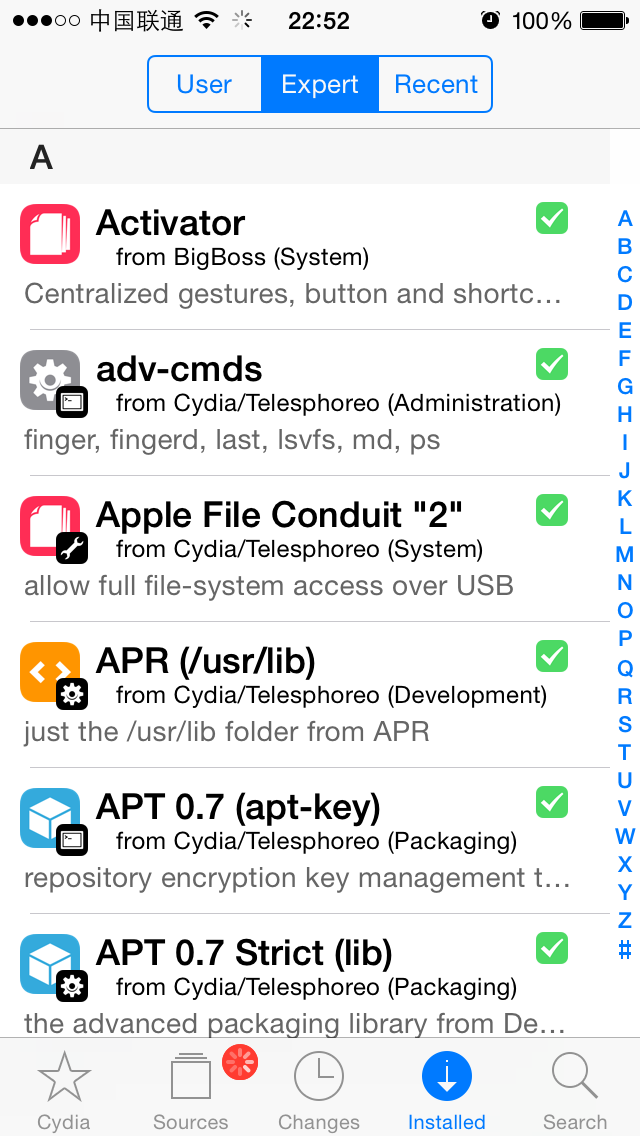


Figure 5-6 Expert view in Cydia

Then you can choose the target App and go to “Details” view, as shown in figure 5-7.

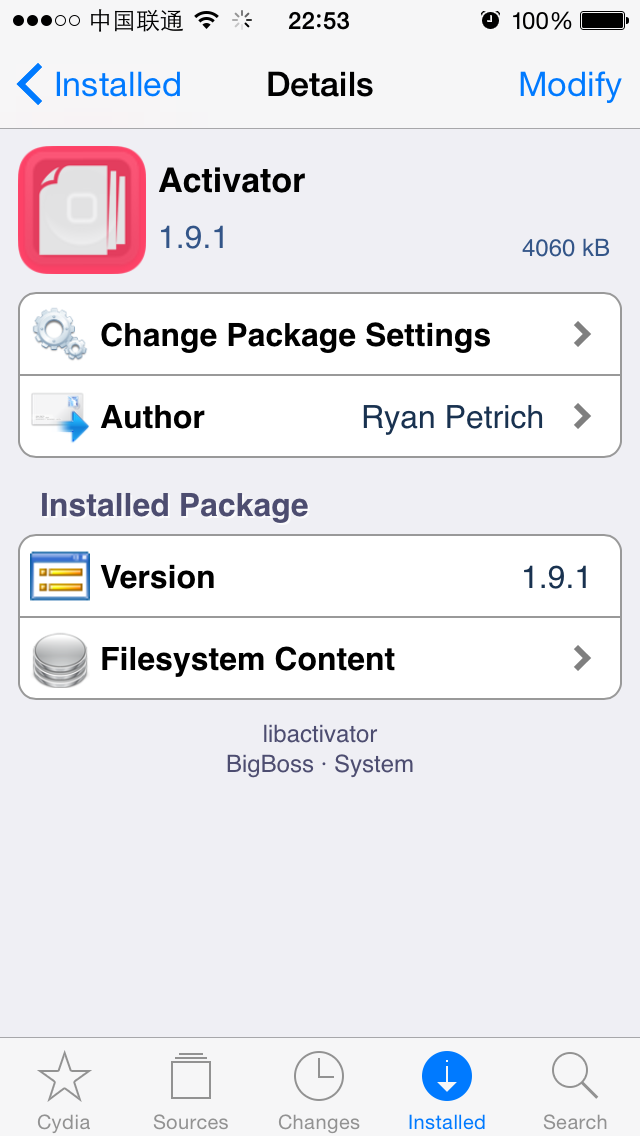


Figure 5-7 Details View

After that, choose "Filesystem Content" and you will see all files in the deb package, as shown in figure 5-8.



Figure 5- 8 Installed files

You can easily find each file’s location now.

#### 3. PreferenceBundle

PreferenceBundle resides in the Settings App and its functionality is somehow vague. It can be either used as a configuration of another process such as “DimInCall”, shown in figure 5-9.

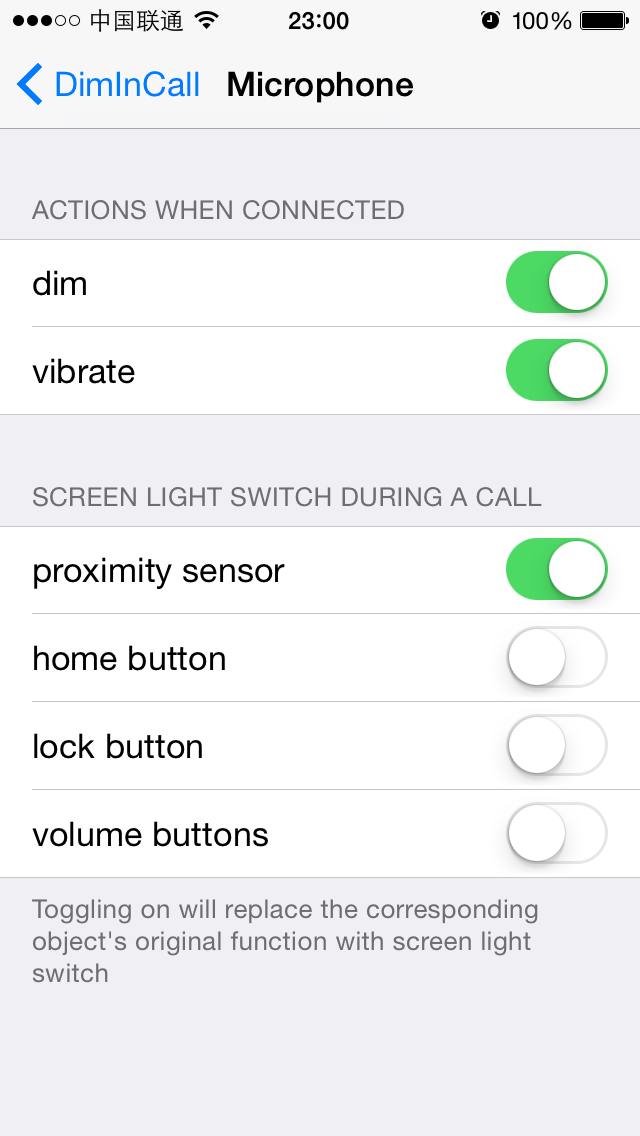


Figure 5- 9 DimInCall

Or it can perform some actual operations and function like an executable such as “WLAN”, shown in figure 5-10.

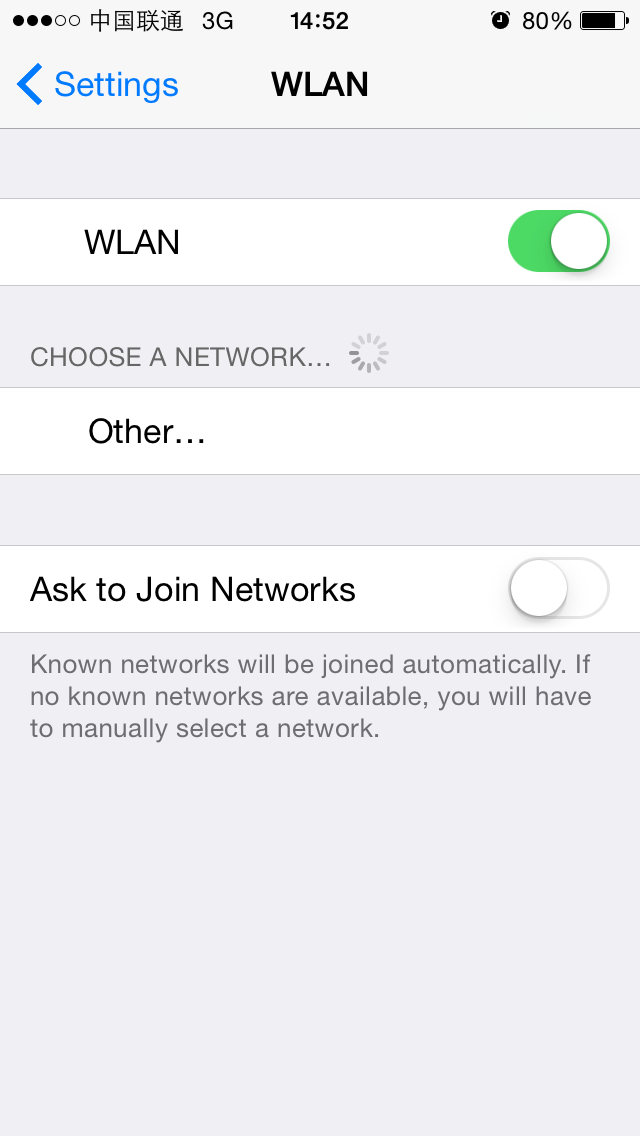


Figure 5- 10 WLAN

Our attention lies on actual operations of an App for sure. As a result, how to locate PreferenceBundle binaries that perform the actual operations is one topic for us to study. Third party PreferenceBundles that come from AppStore can be only used as configuration of their corresponding Apps, they don’t provide any actual functions, there’s no need to locate them. PreferenceBundles from Cydia are also not problems because the solution was already introduced in “locate by Cydia”. However, when it comes to the iOS built-in PreferenceBundles, the process of locating their binaries is a bit complicated.

The UI of a PreferenceBundle can be written programmatically or be constructed from a plist file with a fixed format (You can refer to http://iphonedevwiki.net/index.php/Preferences\_specifier\_plist for the format). When we try to reverse engineer a PreferenceBundle, if all control object types in the PreferenceBundle UI come from preferences specifier plist, such as the “About” view shown in figure 5-11, we should pay attention to distinguish whether it is written programmatically or constructed from plist.

For a built-in PreferenceBundle, if it is written programmatically, its actual function is very probably to be included in its binary, which can be located in "/System/Library/PreferenceBundles/". Otherwise, if it’s constructed from a preferences specifier plist, we have to analyze the relationship between the plist and its actual function, try to find a cut-in point and then locate the binary that provides the actual function. In a nutshell, the case of PreferenceBundle is comparatively complex and is inappropriate as a novice practice. If you find that you don't completely understand the content mentioned above, don't worry, we will present an example later in this chapter. Meanwhile, you can go to our website for more discussion on PreferenceBundle.



Figure 5- 11 About

#### 4. grep

grep is a command line tool from UNIX and it is capable of searching files that match a given regular expression. Grep is a built-in command on OSX; on iOS, it is ported by Saurik and installed accompanying with Cydia by default. grep can quickly narrow down the search scope when we want to find the source of a string. For example, if we want to find which binaries call [IMDAccount initWithAccountID:defaults:service:], we can rely on grep after we sshed into iOS:

FunMaker-5:~ root# grep -r initWithAccountID:defaults:service: /System/Library/

Binary file /System/Library/Caches/com.apple.dyld/dyld\_shared\_cache\_armv7s matches

grep: /System/Library/Caches/com.apple.dyld/enable-dylibs-to-override-cache: No such file or directory

grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGCorePDF.dylib: No such file or directory

grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltin.dylib: No such file or directory

grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMaps.dylib: No such file or directory

grep: /System/Library/Frameworks/System.framework/System: No such file or directory

From the result, we can see that the method appears in dyld\_shared\_cache\_armv7s. Now, we can use grep again in the decached dyld\_shared\_cache\_armv7s:

snakeninnysiMac:~ snakeninny$ grep -r initWithAccountID:defaults:service: /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5

Binary file /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/dyld\_shared\_cache\_armv7s matches

grep: /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/System/Library/Caches/com.apple.xpc/sdk.dylib: Too many levels of symbolic links

grep: /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/System/Library/Frameworks/OpenGLES.framework/libLLVMContainer.dylib: Too many levels of symbolic links

Binary file /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/System/Library/PrivateFrameworks/IMDaemonCore.framework/IMDaemonCore matches

You can see that in the "/System/Library/" directory, [IMDAccount initWithAccountID:defaults:service:] appears in IMDaemonCore, so we can start our analysis from this binary.

### 5.2.3 Locate target functions

After we’ve located the target binaries, we can class-dump them and look for target methods in the headers. Locating target functions is relatively easy and can be done in two ways.

#### Use the bulit-in search function in OSX

It’s an undeniable fact that the bulit-in search function in OSX is the most powerful one among all operating systems I have ever used. It is so powerful that not only can we search file names, but also we’re able to search file contents. Further, its search speed is fast for both searching inside a folder or the entire disk. Taking advantage of this tool can help us locate target files in a pile of files very fast. For example, if we are interested in the proximity sensor on iPhone and want to take a look at what features are provided within those related methods, we can open the folder in which we save class-dump headers, then type “proximity” (case insensitive) in the search bar at top-right corner, as shown in figure 5-12.

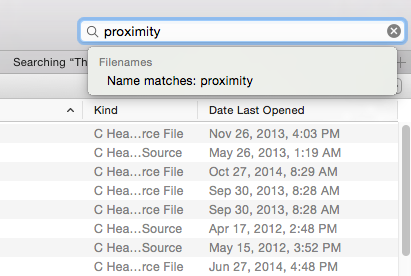


Figure 5- 12 Search in Finder

In default case, all text files containing the keyword “proximity” will be listed in Finder, as shown in 5-13.

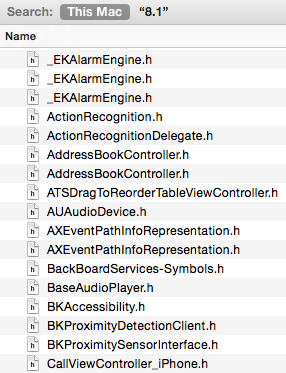


Figure 5-13 Search results in Finder

You can also narrow down the scope of your search by choosing recursively search the file name in current directory. The remaining task is to open the result files and locate the target methods inside.

#### grep

Yes, it's grep again! Since we have already mentioned that we can use grep to search strings in binaries, it's just a piece of cake for grep to deal with text files. Let's try grep with previous example:

snakeninnysiMac:~ snakeninny$ grep -r -i proximity /Users/snakeninny/Code/iOSPrivateHeaders/8.1

/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/CoreLocation/CDStructures.h: char proximityUUID[512];

/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/CoreLocation/CLBeacon.h: NSUUID \*\_proximityUUID;

……

/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/SpringBoard.h:- (\_Bool)proximityEventsEnabled;

/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/SpringBoard.h:- (void)\_proximityChanged:(id)arg1;

Although the results of grep are comprehensive, it looks a little messy. Here, I recommend using the built-in search function in OSX. After all, graphical interface looks more straightforward than command line.

### 5.2.4 Test private methods

In reverse engineering, most methods we are interested in are private. As a result, there are no documentations available for reference. If lucky enough, you can get some information from Google. However, it may indicate that your target methods have already been reversed by others, hence your tweak may not be unique. If there is nothing on Google, congratulations, you are probably the first one to come up with the tweak idea, but you have to test the private methods by yourself.

Testing Objective-C methods is much simpler than testing C/C++ functions, which can be done via either CydiaSubstrate or Cycript.

#### 1. CydiaSubstrate

When testing methods, we mainly use CydiaSubstrate to hook them in order to determine when they’re called. Suppose we think saveScreenShot: in SBScreenShooter.h is called during screenshot, we can write following code to verify it:

%hook SBScreenShotter

- (void)saveScreenshot:(BOOL)screenshot

{

%orig;

NSLog(@"iOSRE: saveScreenshot: is called");

}

%end

Set the tweak filter to "com.apple.springboard", package it into deb using Theos and install it on iOS, then respring. If you feel a bit rusty, don't worry, that's normal; what we care about is stability rather than speed. After lock screen appears, press the home button and lock button at the same time to take a screenshot and then ssh into iOS to view the syslog:

FunMaker-5:~ root# grep iOSRE: /var/log/syslog

Nov 24 16:22:06 FunMaker-5 SpringBoard[2765]: iOSRE: saveScreenshot: is called

You can see that our message is shown in syslog, which means saveScreenshot: is called during screenshot. Since the method name is so explicit, I think most of you still wonder can we really take a screenshot by calling this method?

In iOS reverse engineering, don't be afraid of your curiosity; try Cycript to satisfy your curiosity.

#### 2. Cycript

Before I get to know Cycript, I used Theos to test methods. For example, to test saveScreenshot:, I might write a tweak as follows:

%hook SpringBoard

- (void)\_menuButtonDown:(id)down

{

%orig;

SBScreenShotter \*shotter = [%c(SBScreenShotter) sharedInstance];

[shotter saveScreenshot:YES]; // For the argument here, I guess it’s YES; later we’ll see what happens if it’s NO

}

%end

After the tweak takes effect, press the home button and saveScreenShot: will be called. Then you can check whether there is a white flash on screen and whether there is a screenshot in your album. After that, uninstall the tweak in Cydia.

This approach looked pretty simple before I use Cycript. However, after I’ve achieved the same goal with Cycript, how regretful I was that I had wasted so much time.

The usage of Cycript has already been introduced in chapter 4. Since SBScreenShotter is a class in SpringBoard, we should inject Cycript into SpringBoard and call the method directly to test it out. Unlike tweaks, Cycript doesn’t ask for compilation and clearing up, which saves us great amount of time.

ssh to iOS and then execute the following commands:

FunMaker-5:~ root# cycript -p SpringBoard

cy# [[SBScreenShotter sharedInstance] saveScreenshot:YES]

Do you see a white flash on your screen with a shutter sound and a screenshot in your album, just like pressing home button and lock button together? OK, now we can make sure that calling this method manages to take a screenshot. To further satisfy our curiosity, press the up key on keyboard to repeat the last Cycript command and change YES to No. What is the execution result? We will disclose the details in next section.

### 5.2.5 Analyze method arguments

In the above example, in spite of clear arguments and obvious name meanings, we still don't know whether we should pass YES or NO to the argument, so we have to guess. By browsing the class-dump headers, we can see that most argument types are id, which is the generic type in Objective-C and is determined in runtime. As a consequence, we can't even make any guesses. Starting from getting inspiration, we have overcome so many difficulties to reach arguments analyzing. Should we give up only one step away from the final success? No. Absolutely no. We still have CydiaSubstrate and Theos.

Do you still remember how to judge when a method is called? Since we can print out a custom string, we can also print out arguments of a method. A very useful method, “description”, can represent the contents of an object as an NSString, and object\_getClassName is able to represent the class name of an object as a char\*. These two representations can be printed out by %@ and %s respectively and as a result, we will be given enough information for analyzing arguments. For the above screenshot example, whether the argument of saveScreenShot: is YES or NO just determines whether there is a white flash on screen. According to this clue, we can very soon locate the suspicious SBScreenFlash class, which contains a very interesting method flashColor:withCompletion:. We know that the flash can be enabled or not, is there also any possibilities for us to change the flash color? Let's write the following code to satisfy our curiosity.

%hook SBScreenFlash

- (void)flashColor:(id)arg1 withCompletion:(id)arg2

{

%orig;

NSLog(@"iOSRE: flashColor: %s, %@", object\_getClassName(arg1), arg1); // [arg1 description] can be replaced by arg1

}

%end

We present it here as an exercise for you to rewrite it as a tweak.

After you install the tweak, respring once and take a screenshot. Then ssh to iOS to check the syslog again, you should find information as follows:

FunMaker-5:~ root# grep iOSRE: /var/log/syslog

Nov 24 16:40:33 FunMaker-5 SpringBoard[2926]: iOSRE: flashColor: UICachedDeviceWhiteColor, UIDeviceWhiteColorSpace 1 1

It can be seen that flash color is an object of type UICachedDeviceWhiteColor and its description is "UIDevice WhiteColorSpace 1 1". According to the Objective-C naming conventions, UICachedDeviceWhiteColor is a class in UIKit, but we cannot find it in the document, meaning it is a private class. Class-dump UIKit and then open UICachedDeviceWhiteColor.h:

@interface UICachedDeviceWhiteColor : UIDeviceWhiteColor

{

}

- (void)\_forceDealloc;

- (void)dealloc;

- (id)copy;

- (id)copyWithZone:(struct \_NSZone \*)arg1;

- (id)autorelease;

- (BOOL)retainWeakReference;

- (BOOL)allowsWeakReference;

- (unsigned int)retainCount;

- (id)retain;

- (oneway void)release;

@end

It inherits from UIDeviceWhiteColor, so let's continue with UIDeviceWhiteColor.h:

@interface UIDeviceWhiteColor : UIColor

{

float whiteComponent;

float alphaComponent;

struct CGColor \*cachedColor;

long cachedColorOnceToken;

}

- (BOOL)getHue:(float \*)arg1 saturation:(float \*)arg2 brightness:(float \*)arg3 alpha:(float \*)arg4;

- (BOOL)getRed:(float \*)arg1 green:(float \*)arg2 blue:(float \*)arg3 alpha:(float \*)arg4;

- (BOOL)getWhite:(float \*)arg1 alpha:(float \*)arg2;

- (float)alphaComponent;

- (struct CGColor \*)CGColor;

- (unsigned int)hash;

- (BOOL)isEqual:(id)arg1;

- (id)description;

- (id)colorSpaceName;

- (void)setStroke;

- (void)setFill;

- (void)set;

- (id)colorWithAlphaComponent:(float)arg1;

- (struct CGColor \*)\_createCGColorWithAlpha:(float)arg1;

- (id)copyWithZone:(struct \_NSZone \*)arg1;

- (void)dealloc;

- (id)initWithCGColor:(struct CGColor \*)arg1;

- (id)initWithWhite:(float)arg1 alpha:(float)arg2;

@end

UIDeviceWhiteColor inherits from UIColor. Since UIColor is a public class, stop our analysis at this level is enough for us to get the result. For other id type arguments, we can apply the same solution.

After we have known the effect of calling a method and analyzed its arguments, we can write our own documents. I suggest you make some simple notes on the analysis results of private methods so that you can recall it quickly next time you use the same private method.

Next, let’s use Cycript to test this method and see what effect it is when we pass [UIColor magentaColor] as the argument.

FunMaker-5:~ root# cycript -p SpringBoard

cy# [[SBScreenFlash mainScreenFlasher] flashColor:[UIColor magentaColor] withCompletion:nil]

A magenta flash scatters on the screen and it is much cooler than the original white flash. Check the album and we don't find a new screenshot. Therefore we guess that this method is just for flashing the screen without actually performing the screenshot operation. Aha, a new tweak inspiration arises, we can hook flashColor:withCompletion: and pass it a custom color to enrich the screen flash with more colors. Also, we present it as an exercise and ask you to write a tweak.

All above methodologies are summary of my 5-year experience. Because there is no official documentations for iOS reverse engineering, my personal experiences will inevitably be biased and impossible to cover everything. So you are welcome to http://bbs.iosre.com for further discussions if you have any questions.

### 5.2.6 Limitations of class-dump

By analyzing class-dump headers, we’ve found what we are interested in. In section 5.2.4, we’ve seen the effect by passing two contrary arguments to [SBScreenShotter saveScreenShot:].

In section 5.2.5, we’ve analyzed the 1st argument of flashColor:withCompletion: in SBScreenFlash. From the effect of flashColor:withCompletion:, we guess that it should happen inside saveScreenShot:. But if we just take class-dump headers and the private methods’ effects as references, we can only know the execution order of saveScreenShot: and flashColor:withCompletion:. Neither can we know anything about implementation details and their relationship, nor can we verify our guesses.

So far, we should celebrate for a while since we have just finished a tweak. Starting from the idea, to target binaries, to interested methods and eventually to the tweak, all reverse engineering on the level of Objective-C follows this methodology; the only differences lie in implementation details. Even if you haven’t worked on jailbreak development at all, you can still master this methodology, it’s nothing harder than App development. However, lower the threshold is, fiercer the competition is. After you have mastered methodologies of iOS reverse engineering on the level of Objective-C and wanted to advance to a higher level, you will find class-dump is not enough.

With a finished tweak, we still need to realize that we don't fully understand the knowledge related to this tweak, and class-dump headers is insufficient to satisfy our requirements to master all knowledge. It’s like we are in a forest, class-dump just provide us with a shelter while it is not able to help us go out. To find the exit, we further need a map and a compass, which are IDA and LLDB. But these two tools are two high mountains in front of us. Most rookie reverse engineers failed to climb over them and gave up in the half way. For those who’ve successfully conquered the mountains of IDA and LLDB, they’ve finally enjoyed a magnificent vista just like a dream has come true. A dream you dream alone is only a dream. A dream we dream together is reality. Let’s stay together to climb over the mountains!

## 5.3 An example tweak using the methodology

Before overcoming mountains, we'd better consolidate the knowledge learned so far. So in this section, we will focus on a practical example, which covers all theories mentioned above, in the hope of offering you a smoother transition to chapter 6. The content of this practice is a real example that fully covers the development process of my iOS 6 tweak, "Speaker SBSettings Toggle". (As shown in figure 5-14). At that moment, I didn't know how to use IDA and LLDB, so all clues were from class-dump headers and guesses. This is a stage most of you will experience when learning iOS reverse engineering, therefore could be a very valuable reference.



Figure 5- 14 Speaker SBSettings Toggle

Notice: The following steps no longer work on iOS 8. However, the thinking pattern is good to know.

### 5.3.1 Get inspiration

At the end of March 2012, I received an email from Shoghian, an Iranian-Canaidan. In the mail, he shared an idea that iOS users could switch between microphone and speaker during a phone call while few people knew the speaker could be turned on by default. This feature was very useful for those who were cooking, driving or not convenient to hold the phone during a call. However, such a useful feature was hidden in "Settings" → "General" → "Accessibility" → "Incoming Calls", which was a four-level menu (as shown in figure 5-15) so the set up was very cumbersome. Various toggles in SBSettings are aimed to solve problems like this. So I planned to rewrite it as a toggle to make this good feature handier.

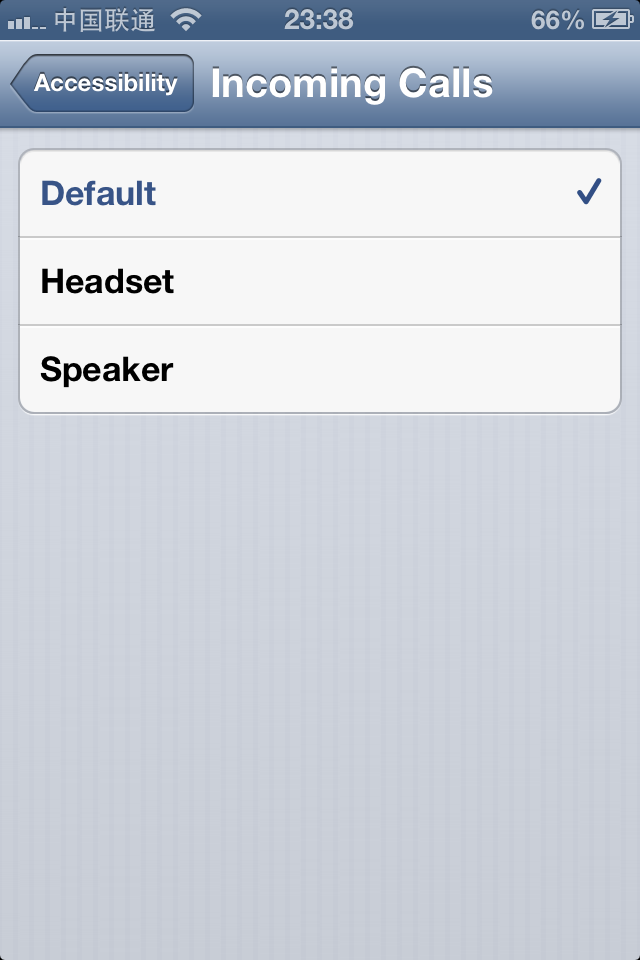


Figure 5- 15 Incoming Calls

### 5.3.2 Locate files

Since this feature was inside Settings App, my first reaction was to look for suspicious files under "/Applications/Preferences.app" and "/System/Library/PreferenceBundles/". What I’ve done are roughly described as follows.

#### Change the system language to English

Because the iOS filesystem was in English, I had set the system language to English before analyzing, so that I was more likely to find correspondence between keywords from filesystem and keywords displayed on UI.

#### Discover keyword "Accessibility"

After I had changed the system language, the four-level menu has been translated from“设置”→“通用”→“ 辅助功能”→“ 来电使用”(in Chinese) to "Settings" → "General" → "Accessibility" → "Incoming calls". The keyword "Accessibility" caught my attention. The reason was that without combining the context, "Accessibility" was too generic to contain "Incoming Calls". So I sshed to iOS and greped the whole filesystem with keyword “Accessibility”. The result was as follows:

FunMaker-4s:~ root# grep -r Accessibility /

grep: /Applications/Activator.app/Default-568h@2x~iphone.png: No such file or directory

grep: /Applications/Activator.app/Default.png: No such file or directory

grep: /Applications/Activator.app/Default~iphone.png: No such file or directory

grep: /Applications/Activator.app/LaunchImage-700-568h@2x.png: No such file or directory

Binary file /Applications/Activator.app/en.lproj/Localizable.strings matches

grep: /Applications/Activator.app/iOS7-Default-Landscape@2x.png: No such file or directory

grep: /Applications/Activator.app/iOS7-Default-Portrait@2x.png: No such file or directory

Binary file /Applications/AdSheet.app/AdSheet matches

Binary file /Applications/Compass.app/Compass matches

……

Despite so many outputs, files shown below with suffix "strings" were very attractive to me:

Binary file /Applications/Preferences.app/English.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/English.lproj/General~iphone.strings matches

Binary file /Applications/Preferences.app/General-Simulator.plist matches

Binary file /Applications/Preferences.app/General.plist matches

Binary file /Applications/Preferences.app/Preferences matches

Binary file /Applications/Preferences.app/en\_GB.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/en\_GB.lproj/General~iphone.strings matches

If nothing went wrong, they were localization files for Apps, which should contain the code name of “Accessibility”. It was very convenient for us to inspect localization files with plutil. So let's take a look at "/Applications/Preferences.app/English.lproj/General~iphone.strings" first.

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/General\~iphone.strings

{

"Videos..." => "• Videos..."

"Wallpaper" => "Wallpaper"

"TV\_OUT" => "TV Out"

"SOUND\_EFFECTS" => "Sound Effects"

"d\_MINUTES" => "%@ Minutes"

……

"ACCESSIBILITY" => "Accessibility"

"Multitasking\_Gestures" => "Multitasking Gestures"

……

}

From "ACCESSIBILITY" => "Accessibility " we could confirm that "ACCESSIBILITY" was the code name.

### 3. Discover General.plist

With new clues, I re-greped the filesystem with keyword "ACCESSIBILITY":

FunMaker-4s:~ root# grep -r ACCESSIBILITY /

grep: /Applications/Activator.app/Default-568h@2x~iphone.png: No such file or directory

grep: /Applications/Activator.app/Default.png: No such file or directory

grep: /Applications/Activator.app/Default~iphone.png: No such file or directory

grep: /Applications/Activator.app/LaunchImage-700-568h@2x.png: No such file or directory

grep: /Applications/Activator.app/iOS7-Default-Landscape@2x.png: No such file or directory

grep: /Applications/Activator.app/iOS7-Default-Portrait@2x.png: No such file or directory

Binary file /Applications/Preferences.app/Dutch.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/Dutch.lproj/General~iphone.strings matches

Binary file /Applications/Preferences.app/English.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/English.lproj/General~iphone.strings matches

Binary file /Applications/Preferences.app/French.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/French.lproj/General~iphone.strings matches

Binary file /Applications/Preferences.app/General-Simulator.plist matches

Binary file /Applications/Preferences.app/General.plist matches

Binary file /Applications/Preferences.app/German.lproj/General-Simulator.strings matches

Binary file /Applications/Preferences.app/German.lproj/General~iphone.strings matches

……

The result was almost the same as the previous. And "/Applications/Preferences.app/General.plist", which I didn't pay attention to a moment ago, was the most conspicuous one. In section 5.2.2, we’ve particularly mentioned the concept of PreferenceBundle. Here, General.plist was not only a plist file, but also contained the keyword. So let's see what’s inside.

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/General.plist

{

"title" => "General"

"items" => [

0 => {

"cell" => "PSGroupCell"

}

1 => {

"detail" => "AboutController"

"cell" => "PSLinkCell"

"label" => "About"

}

2 => {

"cell" => "PSLinkCell"

"id" => "SOFTWARE\_UPDATE\_LINK"

"detail" => "SoftwareUpdatePrefController"

"label" => "SOFTWARE\_UPDATE"

"cellClass" => "PSBadgedTableCell"

}

……

24 => {

"detail" => "PSInternationalController"

"cell" => "PSLinkCell"

"label" => "INTERNATIONAL"

}

25 => {

"cell" => "PSLinkCell"

"bundle" => "AccessibilitySettings"

"label" => "ACCESSIBILITY"

"requiredCapabilities" => [

0 => "accessibility"

]

"isController" => 1

}

26 => {

"cell" => "PSGroupCell"

}

……

]

}

#### Discover AccessibilitySetting.bundle

As expected, this file was a standard preferences specifier plist and the capitalized "ACCESSIBILITY" was in the 25th item. Compared with preferences specifier plist, I had locked my target in AccessibilitySettings bundle. From the name of AccessibilitySettings, I guessed that this bundle assumed the responsibility for all features in Accessibility. According to the fixed file location theory in section 5.2.2, AccessibilitySettings must be under "/System/Library/PreferenceBundles/" and we could locate it easily.

Took a look inside "/System/Library/PreferenceBundles/AccessibilitySetting.bundle":

FunMaker-4s:~ root# ls -la /System/Library/PreferenceBundles/AccessibilitySettings.bundle

total 240

drwxr-xr-x 37 root wheel 2414 Mar 10 2013 .

drwxr-xr-x 40 root wheel 1360 Jan 14 2014 ..

-rw-r--r-- 1 root wheel 2146 Mar 10 2013 Accessibility.plist

-rwxr-xr-x 1 root wheel 438800 Mar 10 2013 AccessibilitySettings

-rw-r--r-- 1 root wheel 238 Dec 22 2012 BluetoothDeviceConfig.plist

-rw-r--r-- 1 root wheel 252 Mar 10 2013 BrailleStatusCellSettings.plist

-rw-r--r-- 1 root wheel 4484 Dec 22 2012 ColorWellRound@2x.png

-rw-r--r-- 1 root wheel 916 Dec 22 2012 ColorWellSquare@2x.png

drwxr-xr-x 2 root wheel 646 Feb 7 2013 Dutch.lproj

drwxr-xr-x 2 root wheel 646 Dec 22 2012 English.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 French.lproj

drwxr-xr-x 2 root wheel 646 Dec 22 2012 German.lproj

-rw-r--r-- 1 root wheel 703 Mar 10 2013 GuidedAccessSettings.plist

-rw-r--r-- 1 root wheel 807 Mar 10 2013 HandSettings.plist

-rw-r--r-- 1 root wheel 652 Mar 10 2013 HearingAidDetailSettings.plist

-rw-r--r-- 1 root wheel 507 Mar 10 2013 HearingAidSettings.plist

-rw-r--r-- 1 root wheel 383 Dec 22 2012 HomeClickSettings.plist

-rw-r--r-- 1 root wheel 447 Dec 22 2012 IconPlay@2x.png

-rw-r--r-- 1 root wheel 1113 Dec 22 2012 IconRecord@2x.png

-rw-r--r-- 1 root wheel 170 Dec 22 2012 IconStop@2x.png

-rw-r--r-- 1 root wheel 907 Mar 10 2013 Info.plist

drwxr-xr-x 2 root wheel 646 Feb 7 2013 Italian.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 Japanese.lproj

-rw-r--r-- 1 root wheel 364 Dec 22 2012 LargeFontsSettings.plist

-rw-r--r-- 1 root wheel 217 Mar 10 2013 NavigateImagesSettings.plist

-rw-r--r-- 1 root wheel 1030 Dec 22 2012 QuickSpeakSettings.plist

-rw-r--r-- 1 root wheel 346 Dec 22 2012 RegionNamesNonLocalized.strings

drwxr-xr-x 2 root wheel 646 Feb 7 2013 Spanish.lproj

-rw-r--r-- 1 root wheel 394 Dec 22 2012 SpeakerLoad1@2x.png

-rw-r--r-- 1 root wheel 622 Mar 10 2013 TripleClickSettings.plist

-rw-r--r-- 1 root wheel 467 Dec 22 2012 VoiceOverBrailleOptions.plist

-rw-r--r-- 1 root wheel 2477 Mar 10 2013 VoiceOverSettings.plist

-rw-r--r-- 1 root wheel 540 Mar 10 2013 VoiceOverTypingFeedback.plist

-rw-r--r-- 1 root wheel 480 Dec 22 2012 ZoomSettings.plist

drwxr-xr-x 2 root wheel 102 Dec 22 2012 \_CodeSignature

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ar.lproj

-rw-r--r-- 1 root wheel 8371 Dec 22 2012 bottombar@2x~iphone.png

-rw-r--r-- 1 root wheel 2701 Dec 22 2012 bottombarblue@2x~iphone.png

-rw-r--r-- 1 root wheel 2487 Dec 22 2012 bottombarblue\_pressed@2x~iphone.png

-rw-r--r-- 1 root wheel 2618 Dec 22 2012 bottombarred@2x~iphone.png

-rw-r--r-- 1 root wheel 2426 Dec 22 2012 bottombarred\_pressed@2x~iphone.png

-rw-r--r-- 1 root wheel 2191 Dec 22 2012 bottombarwhite@2x~iphone.png

-rw-r--r-- 1 root wheel 2357 Dec 22 2012 bottombarwhite\_pressed@2x~iphone.png

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ca.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 cs.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 da.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 el.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 en\_GB.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 fi.lproj

-rw-r--r-- 1 root wheel 955 Dec 22 2012 hare@2x.png

drwxr-xr-x 2 root wheel 646 Feb 7 2013 he.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 hr.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 hu.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 id.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ko.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ms.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 no.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 pl.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 pt.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 pt\_PT.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ro.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 ru.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 sk.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 sv.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 th.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 tr.lproj

-rw-r--r-- 1 root wheel 998 Dec 22 2012 turtle@2x.png

drwxr-xr-x 2 root wheel 646 Feb 7 2013 uk.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 vi.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 zh\_CN.lproj

drwxr-xr-x 2 root wheel 646 Feb 7 2013 zh\_TW.lproj

Here, words like GuidedAccess, HomeClick and HearingAid corresponded with contents we saw in "Accessibility" (as shown in figure 5-16), which confirmed my speculation.



Figure 5- 16 Matching keywords

#### Discover keyword "ACCESSIBILITY\_DEFAULT\_HEADSET"

In virtue of the powerful tool, grep, I searched ”Incoming” in this bundle:

FunMaker-4s:~ root# grep -r Incoming /System/Library/PreferenceBundles/AccessibilitySettings.bundle

Binary file /System/Library/PreferenceBundles/AccessibilitySettings.bundle/English.lproj/Accessibility~iphone.strings matches

Binary file /System/Library/PreferenceBundles/AccessibilitySettings.bundle/en\_GB.lproj/Accessibility~iphone.strings matches

The search result was very similar to the one at the beginning of this section. Open "/System/Library/PreferenceBundles/ AccessibilitySettings.bundle/English.lproj/Accessibility~iphone.strings" and see what’s inside.

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/Accessibility\~iphone.strings

{

"HAC\_MODE\_POWER\_REDUCTION\_N90" => "Hearing Aid Mode improves performance with some hearing aids, but may reduce cellular reception."

"LEFT\_RIGHT\_BALANCE\_SPOKEN" => "Left-Right Stereo Balance"

"QUICKSPEAK\_TITLE" => "Speak Selection"

"LeftStereoBalanceIdentifier" => "L"

"ACCESSIBILITY\_DEFAULT\_HEADSET" => "Incoming Calls"

"HEADSET" => "Headset"

"CANCEL" => "Cancel"

"ON" => "On"

"CUSTOM\_VIBRATIONS" => "Custom Vibrations"

"CONFIRM\_INVERT\_COLORS\_REMOVAL" => "Are you sure you want to disable inverted colors?"

"SPEAK\_AUTOCORRECTIONS" => "Speak Auto-text"

"DEFAULT\_HEADSET\_FOOTER" => "Choose route for incoming calls."

"HEARING\_AID\_COMPLIANCE\_INSTRUCTIONS" => "Improves compatibility with hearing aids in some circumstances. May reduce 2G cellular coverage."

"DEFAULT\_HEADSET" => "Default to headset"

"ROOT\_LEVEL\_TITLE" => "Accessibility"

"HEARING\_AID\_COMPLIANCE" => "Hearing Aid Mode"

"CUSTOM\_VIBES\_INSTRUCTIONS" => "Assign unique vibration patterns to people in Contacts. Change the default pattern for everyone in Sounds settings."

"VOICEOVERTOUCH\_TEXT" => "VoiceOver is for users with

blindness or vision disabilities."

"IMPORTANT" => "Important"

"COGNITIVE\_HEADING" => "Learning"

"HAC\_MODE\_EQUALIZATION\_N94" => "Hearing Aid Mode improves audio quality with some hearing aids."

"SAVE" => "Save"

"HOME\_CLICK\_TITLE" => "Home-click Speed"

"AIR\_TOUCH\_TITLE" => "AssistiveTouch"

"CONFIRM\_ZOT\_REMOVAL" => "Are you sure you want to disable Zoom?"

"VOICEOVER\_TITLE" => "VoiceOver"

"OFF" => "Off"

"GUIDED\_ACCESS\_TITLE" => "Guided Access"

"ZOOMTOUCH\_TEXT" => "Zoom is for users with low-vision acuity."

"INVERT\_COLORS" => "Invert Colors"

"ACCESSIBILITY\_SPEAK\_AUTOCORRECTIONS" => "Speak Auto-text"

"LEFT\_RIGHT\_BALANCE\_DETAILS" => "Adjust the audio volume balance between left and right channels."

"MONO\_AUDIO" => "Mono Audio"

"CONTRAST" => "Contrast"

"ZOOM\_TITLE" => "Zoom"

"TRIPLE\_CLICK\_HEADING" => "Triple-click"

"OK" => "OK"

"SPEAKER" => "Speaker"

"AUTO\_CORRECT\_TEXT" => "Automatically speak auto-corrections

and auto-capitalizations."

"HEARING" => "Hearing"

"LARGE\_FONT" => "Large Text"

"CONFIRM\_VOT\_USAGE" => "VoiceOver"

"CONFIRM\_VOT\_REMOVAL" => "Are you sure you want to disable VoiceOver?"

"HEARING\_AID\_TITLE" => "Hearing Aids"

"FLASH\_LED" => "LED Flash for Alerts"

"VISION" => "Vision"

"CONFIRM\_ZOOM\_USAGE" => "Zoom"

"DEFAULT" => "Default"

"MOBILITY\_HEADING" => "Physical & Motor"

"TRIPLE\_CLICK\_TITLE" => "Triple-click Home"

"RightStereoBalanceIdentifier" => "R"

}

"ACCESSIBILITY\_DEFAULT\_HEADSET" => "Incoming Calls" gave me a very clear hint to continue the search.

#### Locate Accessibility.plist

As you think, I’ve searched “ACCESSIBILITY\_DEFAULT\_HEADSET”:

FunMaker-4s:~ root# grep -r ACCESSIBILITY\_DEFAULT\_HEADSET /System/Library/PreferenceBundles/AccessibilitySettings.bundle

Binary file /System/Library/PreferenceBundles/AccessibilitySettings.bundle/Accessibility.plist matches

Binary file /System/Library/PreferenceBundles/AccessibilitySettings.bundle/Dutch.lproj/Accessibility~iphone.strings matches

……

All were localization files except one plist file. So that should be what I was look for. Its contents is as follows:

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/Accessibility.plist

{

"title" => "ROOT\_LEVEL\_TITLE"

"items" => [

0 => {

"label" => "VISION"

"cell" => "PSGroupCell"

"footerText" => "AUTO\_CORRECT\_TEXT"

}

1 => {

"cell" => "PSLinkListCell"

"label" => "VOICEOVER\_TITLE"

"detail" => "VoiceOverController"

"get" => "voiceOverTouchEnabled:"

}

2 => {

"cell" => "PSLinkListCell"

"label" => "ZOOM\_TITLE"

"detail" => "ZoomController"

"get" => "zoomTouchEnabled:"

}

……

18 => {

"cell" => "PSLinkListCell"

"label" => "HOME\_CLICK\_TITLE"

"detail" => "HomeClickController"

"get" => "homeClickSpeed:"

}

19 => {

"detail" => "PSListItemsController"

"set" => "accessibilitySetPreference:specifier:"

"validValues" => [

0 => 0

1 => 1

2 => 2

]

"get" => "accessibilityPreferenceForSpecifier:"

"validTitles" => [

0 => "DEFAULT"

1 => "HEADSET"

2 => "SPEAKER"

]

"requiredCapabilities" => [

0 => "telephony"

]

"cell" => "PSLinkListCell"

"label" => "ACCESSIBILITY\_DEFAULT\_HEADSET"

"key" => "DefaultRouteForCall"

}

]

}

It was another standard preferences specifier plist and I knew that the getter and setter for “Incoming Calls” were accessibilitySetPreference:specifier: and accessibilityPreferenceForSpecifier:. So it was time to move on to the next step.

### 5.3.3 Locate methods and functions

According to preferences specifier plist, when selecting a row in "Incoming calls", its setter, i.e. accessibilitySetPreference:specifier: would get called. However, a problem came up that this method was in AccessibilitySettings.bundle, I didn't know how to load this bundle into memory at that time and as a result, I wasn’t able to call the method. What's even worse, I didn't know how to use IDA and LLDB while there was nothing helpful in class-dump headers. I felt this problem was far beyond my ability and couldn't get solved in a short time. So I’ve sent a complaint email to Shoghian frustratingly, as shown in figure 5-17.



Figure 5- 17 A complaint email to Shoghian

I was stuck on this problem for nearly half a month. During that period, I was always thinking, what could iOS do inside the setter? Since preferences specifier plist used PostNotification to notify changes of configuration files to other processes, and the configuration of AccessibilitySettings was associated with MobilePhone, which happened to be the mode of inter-process communication. Would accessibilitySetPreference:specifier: change the configuration file and post a notification? To verify my guesses, I made use of LibNotifyWatch by limneos to observe if there were any related notifications through manually changing the configuration of "Incoming Calls". Unexpectedly, it really made me a lucky hit.

FunMaker-4s:~ root# grep LibNotifyWatch: /var/log/syslog

Nov 26 00:09:20 FunMaker-4s Preferences[6488]: LibNotifyWatch: <CFNotificationCenter 0x1e875600 [0x39b4b100]> postNotificationName:UIViewAnimationDidCommitNotification object:UIViewAnimationState userInfo:{

Nov 26 00:09:20 FunMaker-4s Preferences[6488]: LibNotifyWatch: <CFNotificationCenter 0x1e875600 [0x39b4b100]> postNotificationName:UIViewAnimationDidStopNotification object:<UIViewAnimationState: 0x1ea74f20> userInfo:{

……

Nov 26 00:09:21 FunMaker-4s Preferences[6488]: LibNotifyWatch: CFNotificationCenterPostNotification center=<CFNotificationCenter 0x1dd86bd0 [0x39b4b100]> name=com.apple.accessibility.defaultrouteforcall userInfo=(null) deliverImmediately=1

Nov 26 00:09:21 FunMaker-4s Preferences[6488]: LibNotifyWatch: notify\_post com.apple.accessibility.defaultrouteforcall

……

I’ve found two notifications named "com.apple.accessibility.defaultrouteforcall". Combining them with previous mentioned deductions, there was no need to further explain. After finding the most suspicious notification, I still had one more question: Where was the configuration file?

In chapter 2, I have mentioned that there were plenty of user data in "/var/mobile/". All App related data were in "/var/mobile/Containers"; all media files were in "/var/mobile/Media/"; and in "/var/mobile/Library/", we can easily find the directory "/var/mobile/library/Preferences/" then further locate "com.apple.Accessibility.plist", whose contents are as follows:

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/com.apple.Accessibility.plist

{

……

"DefaultRouteForCallPreference" => 2

"VOTQuickNavEnabled" => 1

"CurrentRotorTypeWeb" => 3

"PunctuationKey" => 2

……

"ScreenCurtain" => 0

"VoiceOverTouchEnabled" => 0

"AssistiveTouchEnabled" => 0

}

Change the configuration of "Incoming Calls" then observe the variation of DefaultRouteForCallPreference, we can easily conclude that 0 corresponds to default, 1 corresponds to headset, 2 corresponds to speaker, which totally matches the contents of Accessibility.plist.

### 5.3.4 Test methods and functions

After a long period of deduction, I have eventually got a feasible solution. With only a few lines of code, I can modify the configuration file and post a notification, and we’re done. Does it really work? When I was writing the following code, I felt both nervous and exciting. (At that time I didn't know how to use Cycript, so I wrote a test tweak instead).

%hook SpringBoard

- (void)menuButtonDown:(id)down

{

%orig;

NSMutableDictionary \*dictionary = [NSMutableDictionary dictionaryWithContentsOfFile:@"/var/mobile/Library/Preferences/com.apple. Accessibility.plist"];

[dictionary setObject:[NSNumber numberWithInt:2] forKey:@"DefaultRouteForCallPreference"];

[dictionary writeToFile:@"/var/mobile/Library/Preferences/com.apple. Accessibility.plist" atomically:YES];

notify\_post("com.apple.accessibility.defaultrouteforcall");

}

%end

After compiling, installing and respring, I pressed home button with my eyes closed, and then checked "Settings" → "General" → "Accessibility" → "Incoming Calls" with excitement. Aha, "Speaker" was chosen. I’ve made it!

### 5.3.5 Write tweak

Since the core function has been verified, writing code was a piece of cake. Following SBSettings toggle spec (<http://thebigboss.org/guides-iphone-ipod-ipad/sbsettings-toggle-spec>), the contents of Tweak.xm is as follows.

#import <notify.h>

#define ACCESSBILITY @"/var/mobile/Library/Preferences/com.apple.Accessibility. plist"

// Required

extern "C" BOOL isCapable() {

if (kCFCoreFoundationVersionNumber >= kCFCoreFoundationVersionNumber\_iOS\_5\_0 && [[[UIDevice currentDevice] model] isEqualToString:@"iPhone"])

return YES;

return NO;

}

// Required

extern "C" BOOL isEnabled() {

NSMutableDictionary \*dictionary = [[NSMutableDictionary alloc] initWithCont entsOfFile:ACCESSBILITY];

BOOL result = [[dictionary objectForKey:@"DefaultRouteForCallPreference"] intValue] == 0 ? NO : YES;

[dictionary release];

return result;

}

// Optional

// Faster isEnabled. Remove this if it's not necessary. Keep it if isEnabled() is expensive and you can make it faster here.

extern "C" BOOL getStateFast() {

return isEnabled();

}

// Required

extern "C" void setState(BOOL enabled) {

NSMutableDictionary \*dictionary = [[NSMutableDictionary alloc] initWithCont entsOfFile:ACCESSBILITY];

[dictionary setObject:[NSNumber numberWithInt:(enabled ? 2 : 0)] forKey:@"D efaultRouteForCallPreference"];

[dictionary writeToFile:ACCESSBILITY atomically:YES]; [dictionary release];

notify\_post("com.apple.accessibility.defaultrouteforcall");

}

// Required

// How long the toggle takes to toggle， in seconds.

extern "C" float getDelayTime() {

return 0.6f;

}

Because the inspiration of this tweak came from Shoghian, I’ve signed his name as the coauthor, as shown in figure 5-18. He was very happy and hence we made friends with each other. Speaker SBSettings Toggle is my third public tweak on Cydia, with very simple functions and no advertising, it still accumulated nearly 10,000 downloads, (as shown in figure 5-19), which was a happy ending. More importantly, it was unexpectedly exhausting writing this tweak. My target looked so simple until I really got my hands dirty, which gave me a warning that actions spoke louder than words, I still had a long way to go. Not until the similar situations happened again and again in later days then I finally realized that class-dump was only a supporting role in iOS reverse engineering, and it indirectly encouraged me to dig into IDA and LLDB, which helped me step onto a new stage in iOS reverse engineering.

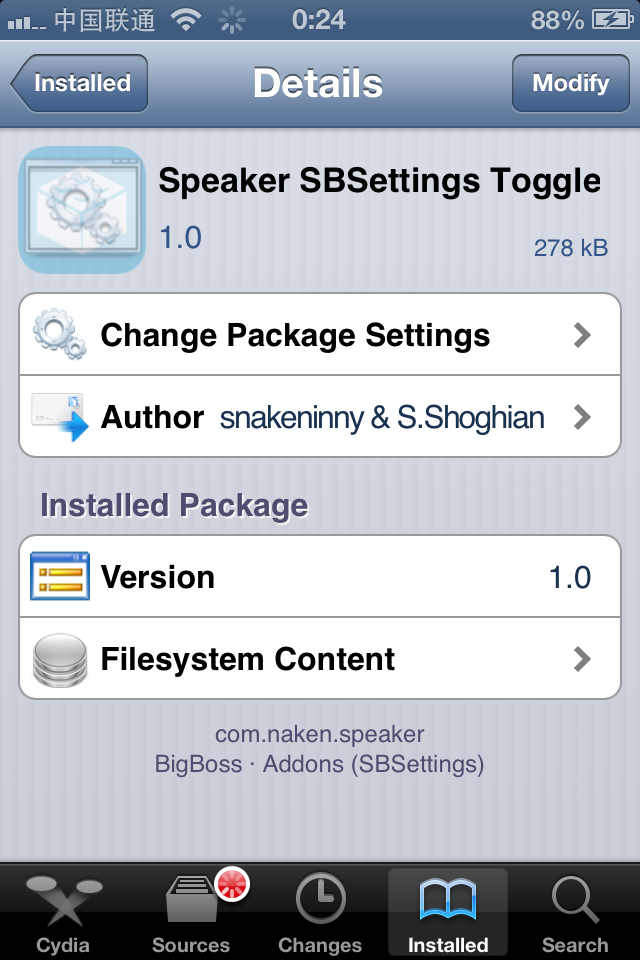


Figure 5- 18 Shoghian is the coauthor

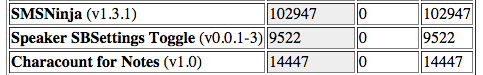


Figure 5- 19 Neary 10,000 downloads

## 5.4 Conclusion

In this chapter, we’ve comprehensively introduced how a tweak works as well as the thought and process of writing a tweak accompanying with practical examples, I believe these contents can help beginners learn iOS reverse engineering better. iOS reverse engineering in Objective-C level is the first hurdle of this book; without knowing IDA and LLDB, we are not able to go very deep into iOS reverse engineering, and our thinking logic is somehow disordered. I think you can feel from the example that our ability at this stage is not adequate to conduct elegant reverse engineering on binaries, so we have to guess a lot when we encounter problems. Although the code we wrote just now was far cry from the official implementation, it worked at least. The only reason is that Objective-C method names are very readable and meaningful so that we can achieve our goals by guessing the functions of class-dump headers, then tes them with Cycript and Theos. Although the methodology in this chapter is kind of “dirty”, it offers a totally different view from App development, which refreshes our mind and broadens our horizon.

As beginners of iOS reverse engineering, our main purpose is to get familiar with jailbreak environment and knowledge points in previous chapters. Also, we need to master the usage of a variety of tools and deliberately cultivate our thinking pattern on reverse engineering. If you have a lot of free time, I strongly recommend you to browse all class-dump headers and test the private methods you are interested in, which will greatly enhance your familiarity with low-level iOS and help you yield twice the result with half the effort after you learn IDA and LLDB. As long as we try to think reversely and practice more, we can surely summarize effective methodologies of ourselves, which helps us step onto a higher level both on iOS reverse engineering and App development.