SNAKENINNY, HANGCOM

Translated by Ziqi Wu, 0xBBC, tianqing and Fei Cheng

iOS App Reverse Engineering

SAMPLE

iosre.com

Table of Contents

The full table of contents is available at http://iosre.com/t/ios-app-reverse-engineering-table-of-contents/1078

Recommendation

In our lives, we pay very little attention to things that work. Everything we interact with hides a fractal of complexity—hundreds of smaller components, all of which serve a vital role, each disappearing into its destined form and function. Every day, millions of people take to the streets with phones in their hands, and every day hardware, firmware, and software blend into one contiguous mass of games, photographs, phone calls, and text messages.

It holds, then, that each component retains leverage over the others. Hardware owns firmware, firmware loads and reins in software, and software in turn directs hardware. If you could take control of one of them, could you influence a device to enact your own desires?

iOS App Reverse Engineering provides a unique view inside the software running on iOS™, the operating system that powers the Apple iPhone® and iPad®. Within, you will learn what makes up application code and how each component fits into the software ecosystem at large. You will explore the hidden second life your phone leads, wherein it is a full-fledged computer and software development platform and there is no practical limit to its functionality.

So, young developer, break free of restricted software and find out exactly what makes your phone tick!

Dustin L. Howett
iPhone Tweak Developer

Preface

I'm a man who loves traveling by myself. On every vacation in university, I spent about 7 to 10 days as a backpacker, traveling around China. Since it was self-guiding tours, no guide would come to help me arrange anything. As a result, before traveling, my friends and I had to prepare everything by ourselves, such as scheduling, confirming the routes and buying tickets. We also needed to put deep thought into our plans, and thought about their dangers.

It's a commonly held belief that traveling, especially backpacking, is a great way to expand one's horizons. What I see during my trips can make me more knowledgeable about the world around me. More importantly, before start traveling, I need to get everything prepared for this journey. My mind has arrived at the destination, even if my body is still at the starting point. This way of thinking is good for cultivating a holistic outlook as well as making us think about problems from a wider, longer term perspective.

Before pursuing my master degree in 2009, I thought deeply about what I wanted to study. My major was computer science. From the beginning of undergraduate year, most of my classmates engaged in the study of Windows. As a student who wasn't good at programming then, there were two alternatives for me to choose—one was to continue the study of Windows, and the other was to explore something else. If I chose the former, there were at least two benefits for me. Firstly, there were lots of documents for reference. The second one was that there were numerous people engaging in the study of Windows. When I met problems, I could consult and discuss with them. However, from the other side, there were also some disadvantages. More references possibly led to less creativity, and the more people engaged in studying Windows, the more competition I would face.

In a nutshell, if I engaged in Windows related work, I could start my career very easily. However, there was no guarantee that I could be outstanding among the researchers. If I chose to do something else, it might be very difficult at the beginning. But as long as I persist with my goal, I could make something different.

Fortunately, my mentor had the same idea. He recommended me to work on mobile development. At that time, there were very few people engaging in this area in China and I had no idea about smart phones. My mobile phone was an out of date Philips phone, so that it was very hard for me to start to develop applications. Despite the difficulties, I trusted my mentor and myself. Not only because I had only chosen him after careful research and recommendations by my senior fellow students, but also that we shared the same opinions. So I started to search online for mobile development related information. After learning only a few concepts about smart phones and mobile Internet, I faintly found that this industry was conductive to the theory that computers and Internet would become smaller, faster and more tightly related with our lives. Many things could be done in this area. So I chose to study iOS.

Everything was hard in the beginning. There were lots of differences between iOS and Windows. For example, iOS was an UNIX-like operating system, which was a complete, but

closed, ecosystem. Its main programming language Objective-C, and jailbreak, were all strange fields lacking of information at that point. So I learned by myself, week by week, in a hackintosh. And this lasted for almost a year. During this period of time, I read the book "Learn Objective-C on the Mac", input the code on the book into Xcode and checked the result by running the simulator. However, the code and the UI were hard to be associated with each other. Besides, I searched those half-UNIX concepts like backgrounding on Google and tried to understand them, but they were really hard to understand. When my classmates published their papers, I even wondered what I was doing during these several months. When they went out and party all night, I decided to code alone in the dormitory. When they had fallen asleep, I had to keep on working in the lab. Although these things made me feel lonely, they benefitted me a lot. I learnt a lot and became more informative during this period. As well, it made me become confident. The more knowledge I got, the less lonely I felt. A man can be excellent when he can bear the loneliness. What you pay will finally return and enrich yourself. After one-year of practice, in March 2011, the obscure code suddenly became understandable. The meaning of every word and the relationship of every sentence became clearer. All fragmented knowledge appeared to be organized in my head and the logic of the whole system became explicit.

So I sped up my research. In April 2011, I finished the prototype of my master thesis and got high praise from my mentor who didn't keep high expectation on my iOS research. Since then, I changed from a person who felt good to a man who was really good, which signified my pass of entry level of iOS research.

In the past few years, I made friends with the author of Theos, DHowett, consulted questions with the father of Activator, rpetrich and quarreled with the admin of TheBigBoss repo, Optimo. They were the people who solved most of my problems along the way. During the development of SMSNinja, I met Hangcom, the second author of this book. As research continues, I met a group of people who was doing excellent things but keeping low profile and finally I realized I'm not alone—We stand alone together.

Taking a look back at the past five years, I'm glad that I made the right choice. It's hard to imagine that you can publish a book related to Windows with only 5-years of research. However, this dream comes true with iOS. The fierce competition among Apple, Microsoft and Google and the feedback from market both prove that this industry will definitely play a leading role in the next 10 years. I feel very lucky that I can be a witness and participant. So, iOS fans, don't hesitate, come and join us, right now!

When received the invitation from Hangcom to write this book, I was a bit hesitant. Due to the large population of China, there were fierce competitions in all walks of life. I summarized all accumulated knowledge from countless failures and if I shared all of them in details, would it result in more competitors? Would my advantages be handed over to others? But throughout the history of jailbreak, from Cydia and CydiaSubstrate to Theos, all these pieces of software were open source and impressed me a lot. It was because these excellent engineers shared their "advantages" that we could absorb knowledge from and then gradually grew better. 'TweakWeek' led by rpetrich and 'OpenJailbreak' led by posixninja also shared their valuable core source code so that more fans could participate in building up the ecosystem of jailbroken iOS. They were the top developers in this area and their advantages didn't get reduced by sharing. I was a learner who benefitted a lot from this sharing chain. Moreover, I intended to continue my research. If I didn't stop, my advantage would stay and the only competitor was myself. I believed sharing would help a lot of developers who were stuck at the entry level where I used to be. And sharing could also combine all wisdom together to make science and technology serve people better. Meanwhile, I could make more friends. From this point of view, writing this book can be regarded as a long term thought, just like what I did as a backpacker.

Ok, What I said above is too serious for the preface. Let me say something about this book. The content of the book is suitable for the majority of iOS developers who are not satisfied with developing Apps. To be honest, this book is techinically better than my master thesis. And if you want to follow up, please focus on our official website http://bbs.iosre.com and our IRC channel #Theos on irc.saurik.com. Together, let us build the jailbreak community!

Here, I want to say thank you to my mother. Without her support, I cannot focus on my research and study. Thanks to my grandpa for the enlightenment of my English studying, having good command of the English language is essential for communicating internationally. Thanks to my mentor for his guidance that helped me grew fast during the three-year master career. Thanks to DHowett, rpetrich, Optimo and those who gave me much help as well as sharp criticism. They helped me grew fast and made me realized that I still had a lot to do. Thanks to britta, Cody d51, DHowett, Haifisch, Tyilo, uroboro and yrp for suggestions and review. Also, I would like to say thank you to my future girlfriend. It is the absence of you that makes me focus on my research. So, I will share half of this book's revenue with you :)

Career, family, friendship, love are life-long pursuits of ordinary people. However, most of us would fail to catch them all, we have to partly give up. If that offends someone, I would like to sincerely apologize for my behaviors and thank you for your forgiveness.

At last, I want to share a poem that I like very much. Despite regrets, life is amazing.

The Road Not Taken
Robert Frost, 1874 – 1963

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I--
I took the one less traveled by,
And that has made all the difference.

In memory of my Grandpa Hanmin Liu and Grandma Chaoyu Wu

snakeinny

Foreword

Why did I write this book?

Two years ago, I changed my job from network administrator to mobile development. It was the time that mobile development was booming in China. Many startups had sprung up and social networking Apps were very popular among investors. As long as you had a good idea, you could get venture capital at scale of millions, and high salary recruitment dazzles everyone.

At that time, I had already developed some difficult enterprise Apps and I wanted to try some cooler techniques rather than developing social Apps, which were too easy for me. By chance, I joined the company Security Manager, built the iOS team from scratch, and took the responsibility for developing iOS Apps for both App Store and Cydia.

In fact, the foundation of jailbreak development is iOS reverse engineering. However, I didn't have too much experience at that time. I was totally a newbie in this area. Fortunately, I could search and learn knowledge on Google. And for iOS developers, jailbreak development and reverse engineering were not completely separated. Although the information shared on the Internet was fragmented and sometimes duplicated, they could still be organized into a complete knowledge map as long as you paid much attention.

However, studying alone makes people feel lonely, especially when you encounter a problem that no one else has encountered. Every time I had to solve problems by myself, I felt that it would be very happy if there were some skillful people that I could communicate with. Although I could email my questions to those experts like Ryan Petrich, I thought it might be some disturbance for them if my questions were too easy for them. So I always tried to dig into the problems and solve it by myself before I decided to open my mouth.

This embarrassing period lasted for over half a year and it ended when I met another author of this book, snakeninny, in 2012. At that time, he was a master student who faced the pressure of graduation. However, he didn't write his master thesis. Instead, he focused on the underlying iOS research and made big progress. I once asked him why not choose to develop iOS Apps since there were already lots of people engaging in it and had made large amount of money. He said that compared with making money, he'd rather be a top developer in the world. Oh boy, how ambitious!

Most of time we solved problems independently. Although we just occasionally discussed with each other on the Internet, we still made some valuable collaborations. Before we started to write this book, we once cracked MOMO (a social App targeting Chinese) by reverse engineering and made a tweak that could show position of girls on the map. Of course, we were harmless developers and we submitted this bug to MOMO and they soon fixed it. This time, we cooperate again, summarize our knowledge into this book and present it to you.

During these years of research on jailbreak development and reverse engineering, the biggest payoff for me is that when I look at an iOS App, I always try to analyze it from underlying architecture and its performance. Both can directly reflect the skill level of its development team. Not only can reverse engineering experiences be applied to jailbreak development, but also they are suitable for App development. Of course, we must admit there are both positive and negative impacts on reverse engineering. However, we cannot deny the necessity of this area even if Apple doesn't advocate jailbreak development. If we blindly believe that the security issues exposed in this book don't actually exist, we're just lying to ourselves.

Every experienced developer understands that the more knowledge you know, the more likely you have to deal with underlying technologies. For example, what does sandbox do? Is it a pity that we only study the mechanism of runtime theoretically?

In the field of Android development, the underlying technologies are open source. However, for iOS, only the tip of the iceberg has been exposed. Although there are some iOS security related books such as *Hacking and Securing iOS Applications* and *iOS Hacker's Handbook*, they are too hard for most App developers to understand. Even those who already have some experience in reverse engineering, like us, have difficulties reading these books.

Since those books are too hard for most people, why not write a book consists of more junior stage details and examples? So concepts, tools, theories and practices make up the contents of this book in a serialized and methodological way. We illustrate our experience and knowledge from easy to hard accompanying with lots of examples, helping readers explore the internals of Apps step by step. We do not try to analyze only a piece of code snippets in depth like some tech blogs. Also, we don't want to puzzle you with how many similar solutions can we use to fix the same problem. What we want to do is to provide readers with a complete system of knowledge and a methodology of iOS reverse engineering. We believe that readers will gain a lot from this book.

Recently, more and more programming experts are joining the jailbreak development community. Although they keep low profile, their works, such as jailbreak tools, App assistants and Cydia tweaks, have great influence on iOS. Their technique level is far beyond mine. But I'm more eager to share knowledge in the hope of helping others.

Who are our target readers?

People of the following kinds may find this book useful.

- iOS enthusiasts.
- Senior iOS developers, who have good command of App development and have the desire to understand iOS better.
- Architects. During the process of reverse engineering, they can learn architectures of those excellent Apps so that they can improve their ability of architecture design.
- Reverse engineers in other systems who're also interested in iOS.

How to read this book?

There are four parts in this book. They are concepts, tools, theories and practices, respectively. The first three parts will introduce the background, knowledge and its associated

tools as well as theories. The fourth part consists of four examples so that readers will have a deeper understanding of previous knowledge in a practical way.

If the reader doesn't have any experience in iOS reverse engineering, we recommend you to start from the first part rather than jumping to the fourth part directly. Although practices are visually cool, hacking is tasteless if you don't know how everything is working under the hood.

Errata and Support

Due to our limited skills and writing schedule, it is inevitable that there are some errors or inaccuracies in the book. We plea for your correction and criticism. Also, readers can visit our official forum (http://bbs.iosre.com) and you will find iOS reverse engineers all over the world on it. Your questions will definitely get satisfied answers.

Because all authors, translators and the editor (snakeninny himself) are not native English speakers, this book may be linguistically ugly. But we promise that this book is techinically pretty. So if you think anything needs to be reworded, please get to us. Thank you!

Acknowledgements

In the first place, I want to say thank you to evad3rs, PanguTeam, TaiG, saurik and other top teams and experts.

Also thanks to Dustin Howett. His Theos is a powerful tool that helped me to step into iOS reverse engineering.

Thanks to Security Manager for providing me with a nice atmosphere for studying reverse engineering. Although I have left this company, I do wish it a better future.

Thanks to everyone who offers help to me. Thanks for your support and encouragement.

This book is dedicated to my dearest family, and many friends who love iOS development.

Hangcom

It's more fun to be a pirate than to join the Navy.

- Steve Jobs

Some of us like to play it safe and take each day as it comes. Some of us want to take that crazy walk on the wild side. So... For those of us who like living dangerously, this one's for you.

- Michael Jackson

I

Concepts

Software reverse engineering refers to the process of deducing the implementation and design details of a program or a system by analyzing the functions, structures or behaviors of it. When we are very interested in a certain software feature while not having the access to the source code, we can try to analyze it by reverse engineering.

For iOS developers, Apps on iOS are one of the most complex but fantastic virtual items as far as we know. They are elaborate, meticulous and creative. As developers, when you see an exquisite App, not only will you be amazed by its implementation, but also you will be curious about what kind of techniques are used in this App and what we can learn from it.

1 Introduction to iOS reverse engineering

Although the recipe of Coca-Cola is highly confidential, some other companies can still copy its taste. Although we don't have access to the source code of others' Apps, we can dig into their details by reverse engineering.

1.1 Prerequisites of iOS reverse engineering

iOS reverse engineering refers to the process of reverse analysis at software-level. If you want to have strong skills on iOS reverse engineering, you'd better be familiar with the hardware constitution of iOS and how iOS works. Also, you should have rich experiences in developing iOS Apps. If you can infer the project scale of an App after using it for a while, its related technologies, its MVC pattern, and which open source projects or frameworks it references, you can announce that you have a good ability on reverse engineering.

Sounds demanding? Aha, a bit. However, all above prerequisites are not fully necessary. As long as you can keep a strong curiosity and perseverance in iOS reverse engineering, you can also become a good iOS reverse engineer. The reason is that during the process of reverse engineering, your curiosity will drive you to study those classical Apps. And it is inevitable that you will encounter some problems that you can't fix immediately. As a result, it takes your perseverance to support you to overcome the difficulties one by one. Trust me, you will surely get your ability improved and feel the beauty of reverse engineering after putting lots of efforts on programming, debugging and analyzing the logic of software.

1.2 What does iOS reverse engineering do

Metaphorically speaking, we can regard iOS reverse engineering as a spear, which can break the seemingly safe protection of Apps. It is interesting and ridiculous to note that many companies that develop Apps are not aware of the existence of this spear and think their Apps are unbreakable.

For IM Apps like WeChat or WhatsApp, the core of this kind of Apps is the information they exchange. For software of banks, payment or e-commerce, the core is the monetary transaction data and customer information. All these core data have to be securely protected. So developers have to protect their Apps by combining anti-debugging, data encryption and code obfuscation together. The aim is to increase the difficulty of reverse engineering and prevent similar security issues from affecting user experience.

However, the technologies currently being used to protect Apps are not in the same dimension with those being used in iOS reverse engineering. For general App protections, they look like fortified castles. By applying the MVC architecture of Apps inside the castle with thick walls outside, we may feel that they are insurmountable, as shown in figure 1-1.

Figure 1-1 Strong fortress, taken from Assassin's Creed

But if we step onto another higher dimension and overlook into the castle where the App resides, you find that structure inside the castle is no longer a secret, as shown in figure 1-2.

Figure 1-2 Overlook the castle, taken from Assassin's Creed

All Objective-C interfaces, all properties, all exported functions, all global variables, even all logics are exposed in front of us, which means all protections have became useless. So if we are in this dimension, walls are no longer hindrances. What we should focus on is how can we find our targets inside the huge castle.

At this point, by using reverse engineering techniques, you can enter the low dimension castle from any high dimension places without damaging walls of the castle, which is definitely tricky while not laborious. By monitoring and even changing the logics of Apps, you can learn the core information and design details easily.

Sounds very incredible? But this is true. According to the experiences and achievements I've got from the study of iOS reverse engineering, I can say that reverse engineering can break the protection of most Apps, all their implementation and design details can be completely exposed.

The metaphor above is only my personal viewpoint. However, it vividly illustrates how powerful iOS reverse engineering is. In a nutshell, there are two major functions in iOS reverse engineering as below:

- Analyze the target App and get the core information. This can be concluded as security related reverse engineering.
- Learn from other Apps' features and then make use of them in our own Apps. This can be concluded as development related reverse engineering.

1.2.1 Security related iOS reverse engineering

Security related IT industry would generally make extensive use of reverse engineering. For example, reverse engineering plays the key roles in evaluating the security level of a financial App, finding solutions of killing viruses, and setting up a spam phone call firewall on iOS, etc.

1. Evaluate security level

Apps which consist of sensitive features like financial transactions will encrypt the data at first and then save the encrypted data locally or transfer them via network. If developers do not have strong awareness of security, it is very possible for them to save or send the sensitive information such as bank accounts and passwords without encryption, which is definitely a great security risk.

If a company with high reputation wants to release an App. In order to make the App qualified with the reputation as well as the trust from customers, the company will hire a security organization to evaluate this App before releasing it. In most cases, the security organization does not have access to the source code so that they cannot evaluate the security level via code review. Therefore the only way they can do is reverse engineering. They try to attack the App and then evaluate the security level based on the result.

2. Reverse engineering malware

iOS is the operating system of smart devices, it has no essential difference with computer operating systems. From the first generation, iOS is capable of browsing the Internet. However, the Internet is the best medium of malware. Ikee, exposed in 2009, is the first virus in iOS. It can infect those jailbroken iOS devices which have installed ssh but have not changed the default password "alpine". It can change the background image of the lockscreen to photo of a British singer. Another virus WireLurker appeared at the end of 2014, it can steal private information of users and spread on PC or Mac, bringing users disastrous harm.

For malware developers, by targeting system and software vulnerabilities through reverse engineering, they can penetrate into the target hosts, access to sensitive data and do whatever they want.

For anti-virus software developers, they can analyze samples of viruses through reverse engineering, observe the behaviors of viruses and then try to kill them in the infected hosts as well as summarize the methods to protect against viruses.

3. Detect software backdoors

A big advantage of open source software is its good security. Tens of thousands of developers review the code and modify the bug of open source software. As a result, the possibilities that there are backdoors inside the code are minimized, and the security related bugs would be fixed before they are disclosed. For closed source software, reverse engineering is one of the most frequently used methods to detect the backdoors in software. For example, we often install different kinds of Apps on jailbroken iPhones through third-party App Stores. All these Apps are not officially examined and reviewed by Apple so there could be unrevealed risks. Even worse, some developers will put backdoors inside their Apps on the purpose of stealing something from users. So reverse engineering is often involved in the process of detecting that kind of behaviors.

4. Remove software restriction

Selling Apps on AppStore or Cydia is one primary economic source for App developers. In the software world, piracy and anti-piracy will coexist forever. Many developers have already added protection in their software to prevent piracy. However, just like the war between spear and shield will never stop, no matter how good the protection of an App is, there will definitely be one day that the App is cracked. The endless emergency of pirated software makes it an impossible task for developers to prevent piracy. For example, the most famous share repository "xsellize" on Cydia is able to crack any App in just one day and it is notorious among the industry.

1.2.2 Development related iOS reverse engineering

For iOS developers, reverse engineering is one of the most practical techniques. For example, we can do reverse engineering on system APIs to use some private functions, which are not documented. Also, we can learn good architecture and design from those classical Apps through reverse engineering.

1. Reverse System APIs

The reason that Apps are able to run in the operating system and to provide users with a variety of functions is that these functions are already embedded in the operating system itself, what developers need to do is just reassembling them. As we all know, functions we used for developing Apps on AppStore are restricted by Apple's document and are under the strict regulation of Apple. For example, you cannot use undocumented functions like making phone calls or sending messages. However, if you're targeting Cydia Store, absence of private functions makes your App much less competitive. If you want to use undocumented functions, the most effective reference is from reversing iOS system APIs, then you can recreate the code of corresponding functions and apply it to your own Apps.

2. Learn from other Apps

The most popular scenario for reverse engineering is to learn from other Apps. For most Apps on AppStore, the implementations of them are not very difficult, their ingenious ideas and good business operation are the keys to success. So, if you just want to learn a function from another App, it is time-consuming and laborious to restore the code through reverse engineering; I'd suggest you write a similar App from scratch. However, reverse engineering plays a critical role in the situation when we don't know how a feature of an App is implemented. This is often seen in Cydia Apps with extensive use of private functions. For example, Audio Recorder, known as the first phone call recording App, is a closed source App. Yet it is very interesting for us to learn how it is implemented. Under this circumstance you can learn a little bit through iOS reverse engineering.

There are some classical Apps with neat code, reasonable architecture, and elegant implementation. Compared with developers of those Apps, we don't have profound technical background. So if we want to learn from those Apps while not having an idea of where to start, we can turn to reverse engineering. Through reverse engineering those Apps, we can extract the architecture design and apply it to our own projects so that we can enhance our Apps. For example, the stability and robustness of WhatsApp is so excellent that if we want to develop our own IM Apps, we can benefit a lot from learning the architecture and design of WhatsApp.

1.3 The process of iOS reverse engineering

When we want to reverse an App, how should we think? Where should we start? The purpose of this book is to guide the beginners into the field of iOS reverse engineering, and cultivate readers to think like reversers.

Generally speaking, reverse engineering can be regarded as a combination of analysis on two stages, which are system analysis and code analysis, respectively. In the phase of system analysis, we can find our targets by observing behavioral characteristics of program and organizations of files. During code analysis, we need to restore the core code and then ultimately achieve our goals.

1.3.1 System Analysis

At the stage of system analysis, we should run target Apps under different conditions, perform various operations, observe the behavioral characteristics and find out features that we are interested in, such as which option we choose leads to a popup alert? Which button makes a sound after pressing it? What is the output associated with our input, etc. Also, we can browse the filesystem, see the displayed images, find the configuration files' locations, inspect the information stored in databases and check whether the information is encrypted.

Take Sina Weibo as an example. When we look over its Documents folder, we can find some databases:

```
-rw-r--r-- 1 mobile mobile  210944 Oct 26 11:34 db_46100_1001482703473.dat
-rw-r--r-- 1 mobile mobile  106496 Nov 16 15:31 db_46500_1001607406324.dat
-rw-r--r-- 1 mobile mobile  630784 Nov 28 00:43 db_46500_3414827754.dat
-rw-r--r-- 1 mobile mobile 6078464 Dec  6 12:09 db_46600_1172536511.dat
……
```

Open them with SQLite tools, we can find some followers' information in it, as shown in figure 1-3.

1807621622	天生歌姬A-Lin	http://tp3.sinaimg.cn/1807621622/50/5700356795/0
1497487043	焦波和俺爹俺娘	http://tp4.sinaimg.cn/1497487043/50/1297238551/1
2835121504	Angela侯湘婷	http://tp1.sinaimg.cn/2835121504/50/5664550946/0
1744390777	林凡Freya	http://tp2.sinaimg.cn/1744390777/50/40053380567/0
2875568950	EDC尤原庆	http://tp3.sinaimg.cn/2875568950/50/40001989049/1
1565668374	财上海	http://tp3.sinaimg.cn/1565668374/50/5703348848/1
3962782795	陳綺貞cheerego	http://tp4.sinaimg.cn/3962782795/50/40043044497/0
1283498527	许哲佩PeggyHsu	http://tp4.sinaimg.cn/1283498527/50/40054968787/0
1198922365	曹方Icy	http://tp2.sinaimg.cn/1198922365/50/5635147308/0
2270268414	Dawen王大文	http://tp3.sinaimg.cn/2270268414/50/5705504906/1
1787113000	Mrdadado黄玠	http://tp1.sinaimg.cn/1787113000/50/5602434900/1
1751505334	魏如萱waa	http://tp3.sinaimg.cn/1751505334/50/5711190523/0

Figure 1-3 Sina Weibo database

Such information provides us with clues for reverse engineering. Database file names, Sina Weibo user IDs, URLs of user information, all can be used as cut-in points for reverse engineering. Finding and organizing these clues, then tracking down to what we are interested in, is often the first step of iOS reverse engineering.

1.3.2 Code Analysis

After system analysis, we should do code analysis on the App binary. Through reverse engineering, we can deduce the design pattern, internal algorithms, and the implementation details of an App. However, this is a very complex process and can be regarded as an art of deconstruction and reconstruction. To improve your reverse engineering skill level into the state of art, you must have a thorough understanding on software development, hardware principles, and iOS itself. Analyzing the low-level instructions bit by bit is not easy and cannot be fully covered in one single book.

The purpose of this book is just to introduce tools and methodologies of reverse engineering to beginners. Technologies are evolving constantly, so we cannot cover all of them. For this reason, I've build up a forum, http://bbs.iosre.com, where we can discuss and exchange ideas with each other in real time.

1.4 Tools for iOS reverse engineering

After learning some concepts about iOS reverse engineering, it is time for us to put theory into practice with some useful tools. Compare with App development, tools used in reverse engineering are not as "smart" as those in App development. Most tasks have to be done manually, so being proficient with tools can greatly improve the efficiency of reverse engineering. Tools can be divided into 4 major categories; they are monitors, disassemblers, debuggers and development kit.

1.4.1 Monitors

In the field of iOS reverse engineering, tools used for sniffing, monitoring and recording targets' behaviors can all be concluded as monitors. These tools generally record and display certain operations performed by the target programs, such as UI changes, network activities and file accesses. Reveal, snoop-it, introspy, etc., are frequently used monitors.

Reveal, as shown in figure 1-4, is a tool to see the view hierarchy of an App in real-time.

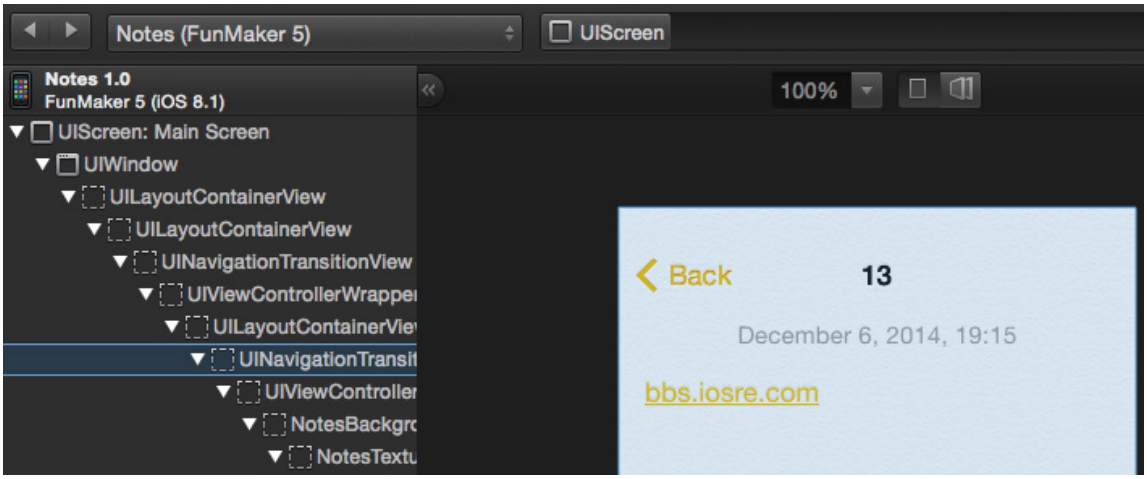


Figure 1- 4 Reveal

Reveal can assist us in locating what we are interested in an App so that we can quickly approach the code from the UI.

1.4.2 Disassemblers

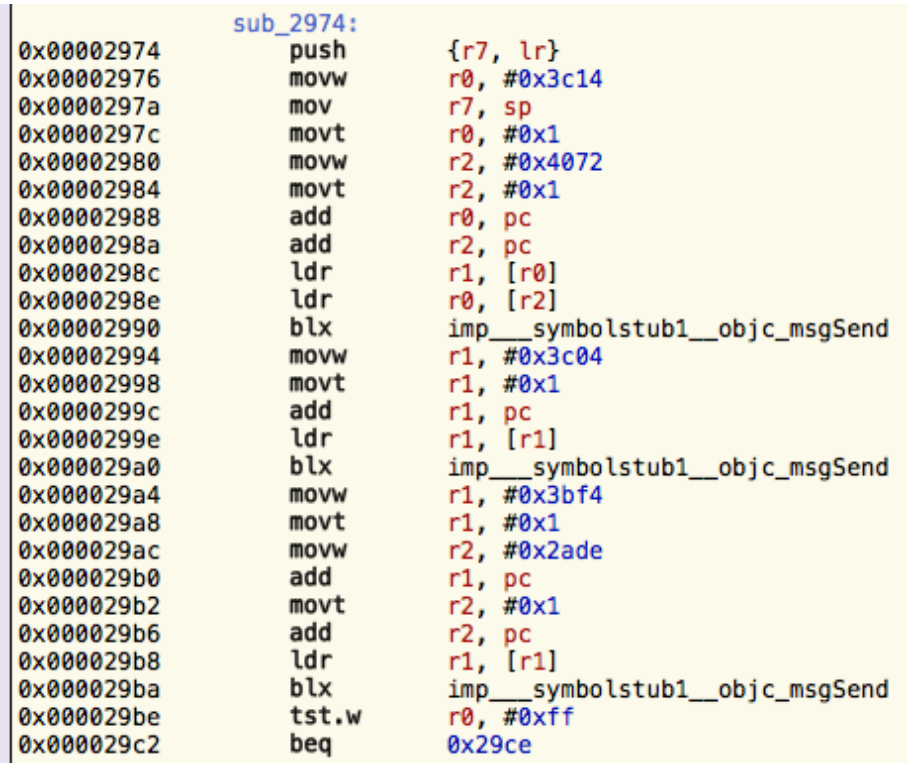
After approaching the code from the UI, we have to use disassembler to sort out the code. Disassemblers take binaries as input, and output assembly code after processing the files. IDA and Hopper are two major disassemblers in iOS reverse engineering.

As an evergreen disassembler, IDA is one of the most commonly used tools in reverse engineering. It supports Windows, Linux and OSX, as well as multiple processor architectures, as shown in figure 1-5.

```
sub_2974
PUSH            {R7,LR}
MOVW            R0, #(:lower16:(selRef_processInfo - 0x298C))
MOV             R7, SP
MOVT.W          R0, #(:upper16:(selRef_processInfo - 0x298C))
MOV             R2, #(classRef_NSProcessInfo - 0x298E) ; classRef_NSProcessInfo
ADD             R0, PC ; selRef_processInfo
ADD             R2, PC ; classRef_NSProcessInfo
LDR             R1, [R0] ; "processInfo"
LDR             R0, [R2] ; _OBJC_CLASS_$_NSProcessInfo
BLX             _objc_msgSend
MOV             R1, #(selRef_processName - 0x29A0) ; selRef_processName
ADD             R1, PC ; selRef_processName
LDR             R1, [R1] ; "processName"
BLX             _objc_msgSend
MOV             R1, #(selRef_isEqualToString_ - 0x29B4) ; selRef_isEqualToString_
MOVW            R2, #(:lower16:(cfstr_Springboard - 0x29BA)) ; "SpringBoard"
ADD             R1, PC ; selRef_isEqualToString_
MOVT.W          R2, #(:upper16:(cfstr_Springboard - 0x29BA)) ; "SpringBoard"
ADD             R2, PC  ; "SpringBoard"
LDR             R1, [R1] ; "isEqualToString:"
BLX             _objc_msgSend
TST.W           R0, #0xFF
BEQ             loc_29CE
```

Figure 1- 5 IDA

Hopper is a disassembler that came out in recent years, which mainly targets Apple family operating systems, as shown in figure 1-6.



```
                sub_2974:
0x00002974          push        {r7, lr}
0x00002976          movw        r0, #0x3c14
0x0000297a          mov         r7, sp
0x0000297c          movt        r0, #0x1
0x00002980          movw        r2, #0x4072
0x00002984          movt        r2, #0x1
0x00002988          add         r0, pc
0x0000298a          add         r2, pc
0x0000298c          ldr         r1, [r0]
0x0000298e          ldr         r0, [r2]
0x00002990          blx         imp___symbolstub1__objc_msgSend
0x00002994          movw        r1, #0x3c04
0x00002998          movt        r1, #0x1
0x0000299c          add         r1, pc
0x0000299e          ldr         r1, [r1]
0x000029a0          blx         imp___symbolstub1__objc_msgSend
0x000029a4          movw        r1, #0x3bf4
0x000029a8          movt        r1, #0x1
0x000029ac          movw        r2, #0x2ade
0x000029b0          add         r1, pc
0x000029b2          movt        r2, #0x1
0x000029b6          add         r2, pc
0x000029b8          ldr         r1, [r1]
0x000029ba          blx         imp___symbolstub1__objc_msgSend
0x000029be          tst.w       r0, #0xff
0x000029c2          beq         0x29ce
```

Figure 1- 6 Hopper

After disassembling binaries, we have to read the generated assembly code. This is the most challenging task as well as the most interesting part in iOS reverse engineering, which will be explained in detail in chapters 6 to 10. We will use IDA as the main disassembler in this book and you can reference the experience of Hopper on http://bbs.iosre.com.

1.4.3 Debuggers

iOS developers should be familiar with debuggers because we often need to debug our own code in Xcode. We can set a breakpoint on a line of code so that process will stop at that line and display the current status of the process in real time. We constantly use LLDB for debugging during both App development and reverse engineering. Figure 1-7 is an example of debugging in LLDB.

```
snakeninnys-MacBook:~ snakeninny$ lldb
(lldb) attach Finder
Process 303 stopped
Executable module set to "/System/Library/CoreServices/
Finder.app/Contents/MacOS/Finder".
Architecture set to: x86_64-apple-macosx.
(lldb) c
Process 303 resuming
```

Figure 1- 7 LLDB

1.4.4 Development kit

After finishing all the above steps, we can get results from analysis and start to code for now. For App developers, Xcode is the most frequently used development tool. However, if we transfer the battlefield from AppStore to jailbroken iOS, our development kit gets expanded. Not only is there an Xcode based iOSOpenDev, but also a command line based Theos. Judging

from my own experiences, Theos is the most exciting development tool. Before knowing Theos, I felt like I was restricted to the AppStore. Not until I mastered the usage of Theos did I break the restriction of AppStore and completely understood the real iOS. Theos is the major development tool in this book and we'll discuss about iOSOpenDev on our website.

1.5 Conclusion

In this chapter, we have introduced some concepts about iOS reverse engineering in order to provide readers with a general idea of what we'll be focusing on. More details and examples will be covered in the following chapters. Stay tuned with us!

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.

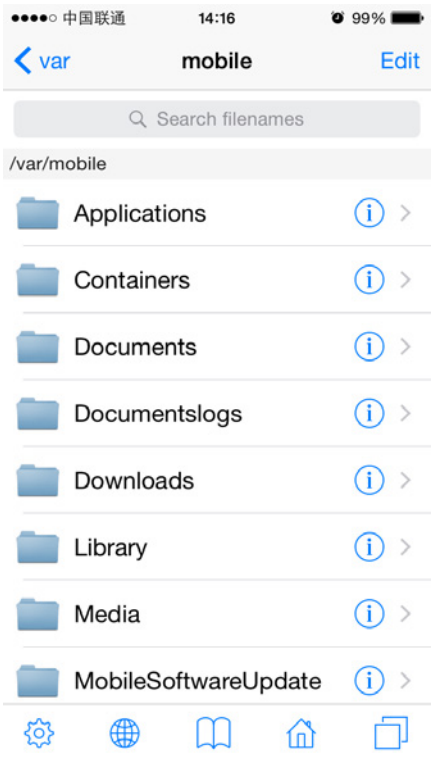


Figure 2- 1 iFile

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.

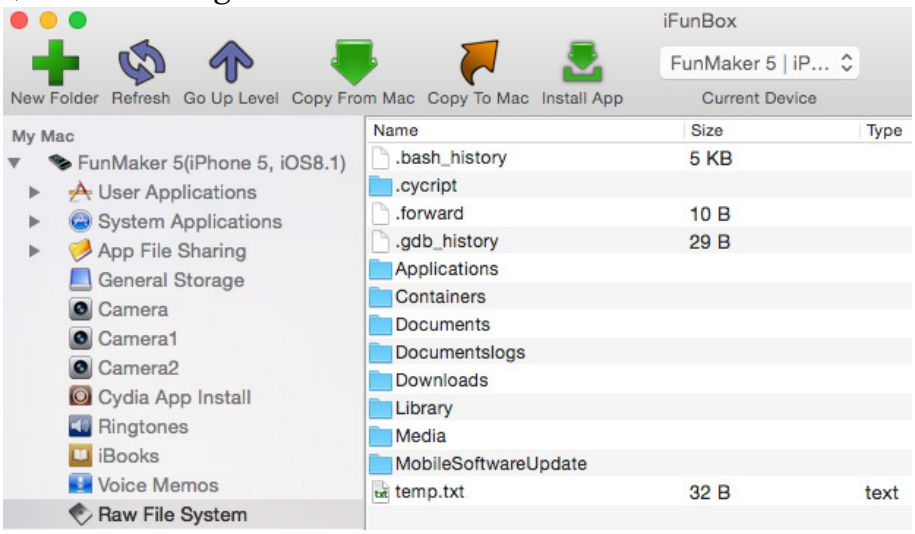


Figure 2- 2 iFunBox

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.

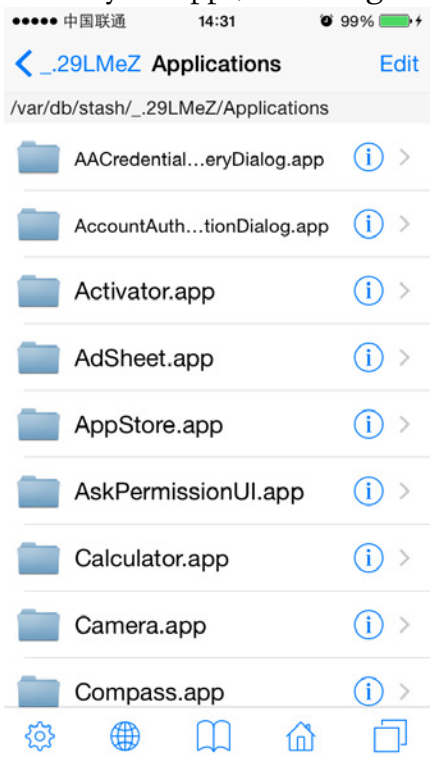


Figure 2- 3 /Applications

- /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.

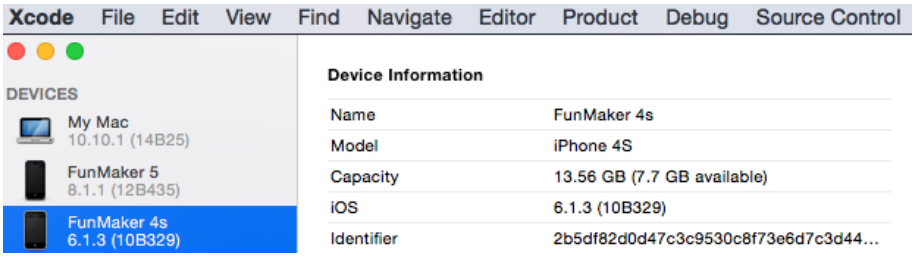


Figure 2- 4 Enable debugging on device

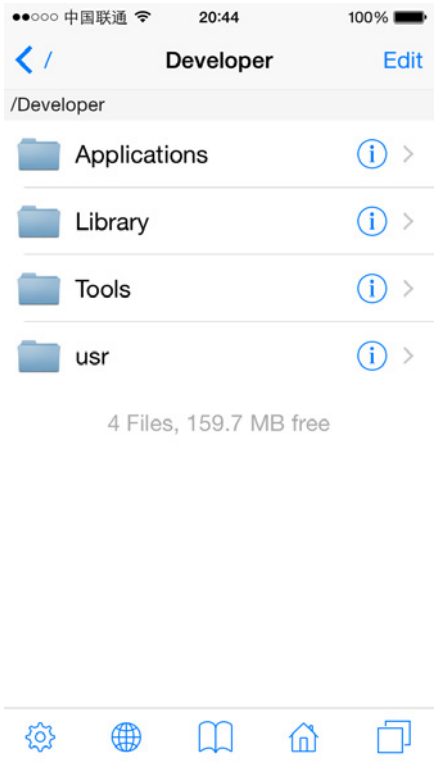


Figure 2- 5 /Developer

- /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.

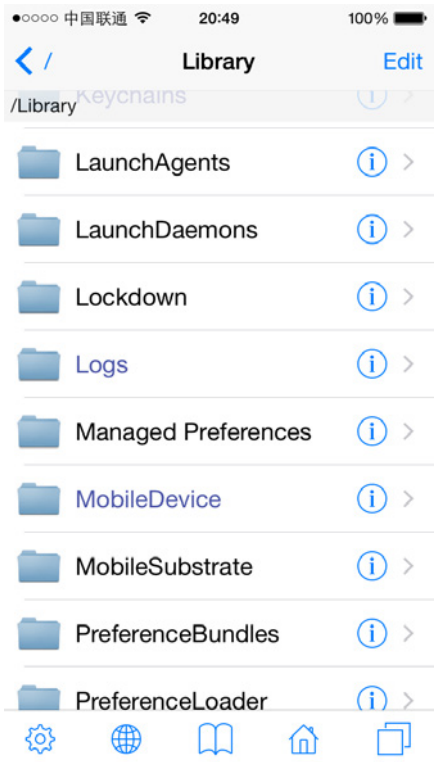


Figure 2- 6 /Library

- /System/Library

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

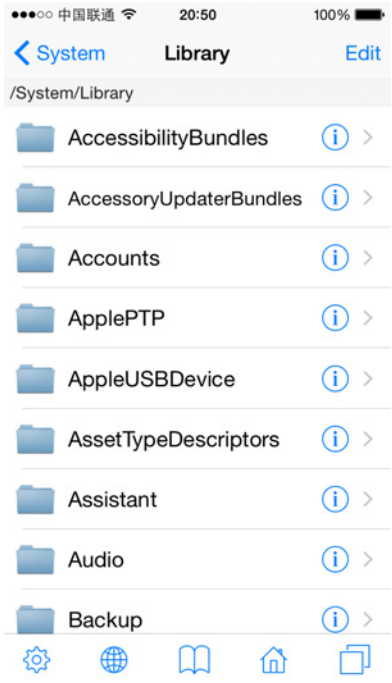


Figure2- 7 /System/Library

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.

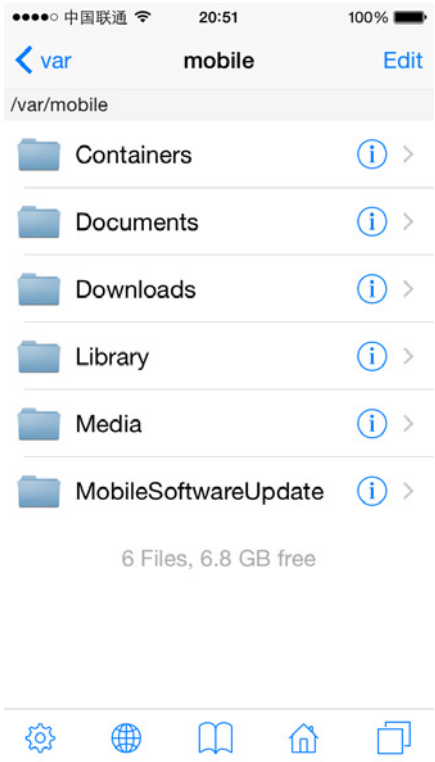


Figure 2- 8 /User

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.

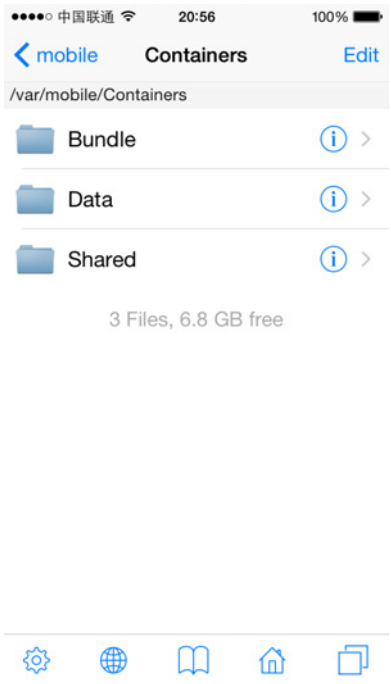


Figure 2- 9 /var/mobile/Containers

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 to 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.

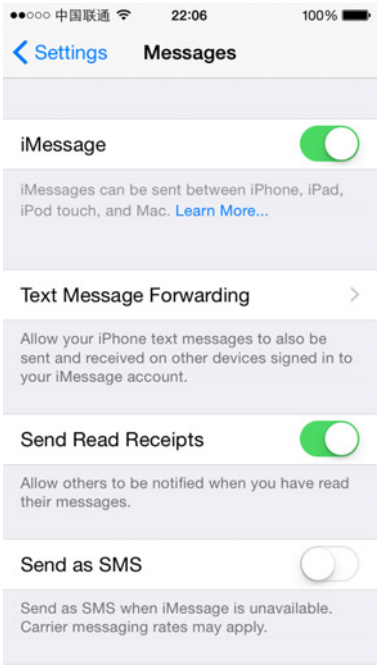


Figure 2- 10 PreferenceBundle

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.

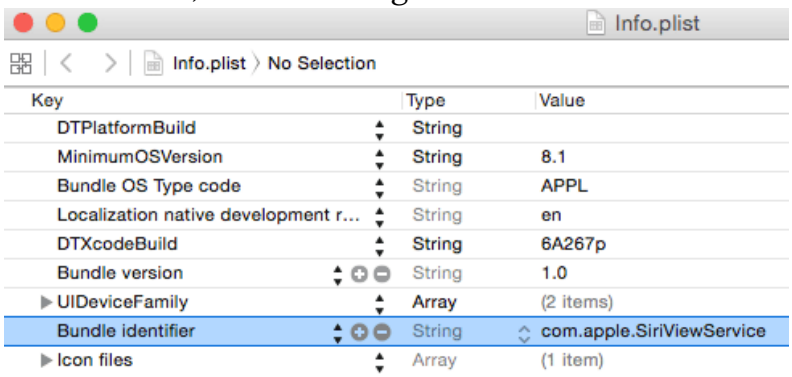


Figure 2- 11 Browse Info.plist in Xcode

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.pli
st | 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.

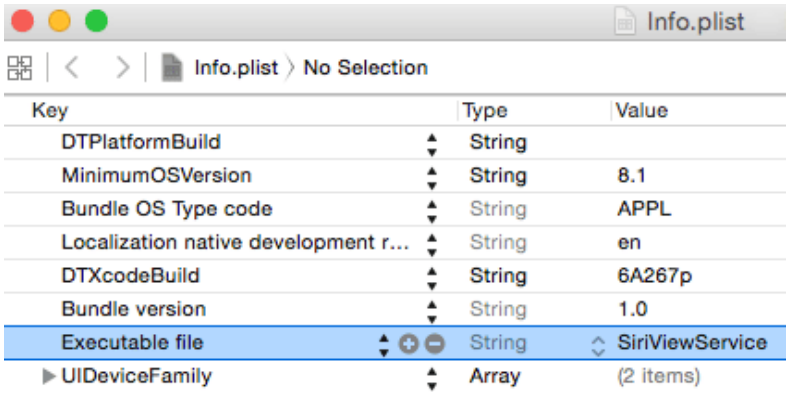


Figure 2- 12 Browse Info.plist in Xcode

Or with plutil:

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/SiriViewService.app/Info.pli
st | 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:

- 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?
- 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?
- 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 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 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/LaunchDaemo
ns/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 experiences 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.

II

Tools

In the 1st part, we've introduced the basic concepts of iOS reverse engineering. In this part, we will introduce the toolkit of iOS reverse engineering.

Compared with App development, the main feature of iOS reverse engineering is it's more "mixed". When you are writing Apps, most work can be done within Xcode, since it is the product of Apple, it's convenient to download, install and use. As for some other tools and plugins, they are just some kind of icing on the cake, thus useful but non-essential.

But, in iOS reverse engineering, we have to face so many complicated tools. Let me make an example, there are two dinner tables in front of you, on the first table there's simply a pair of chopsticks, it's named Xcode; the other one is full of knives and forks, in which some of the big shots are Theos, Reveal, IDA and etc…

Unlike Xcode, there is no tight connection among those reverse engineering tools; they are separated from each other, so we need to integrate them manually. We cannot cover all reverse engineering tools in this part, but I think you will have the ability to find and use proper tools according to the situation you face when you finish reading this book. You can also share your findings with us on http://bbs.iosre.com.

Because the tools to be introduced are quite disordered, we split this part to two chapters, one is for OSX tools, the other is for iOS. The device used in this part is iPhone 5 with iOS 8.1.

3 OSX toolkit

Tools used for iOS reverse engineering have different functions, and they play different roles. These tools mainly help us develop and debug on OSX. Because of the small screen size of iOS devices, they are not suitable for development or debug.

In this chapter, 4 major tools will be introduced, they're class-dump, Theos, Reveal and IDA. Other tools are assistants for them.

3.1 class-dump

class-dump, as the name indicates, is a tool used for dumping the class information of the specified object. It makes use of the runtime mechanism of Objective-C language to extract the headers information stored in Mach-O files, and then generates .h files.

class-dump is simple to use. Firstly, you need to download the latest version from http://stevenygard.com/projects/class-dump, as figure 3-1 shows:

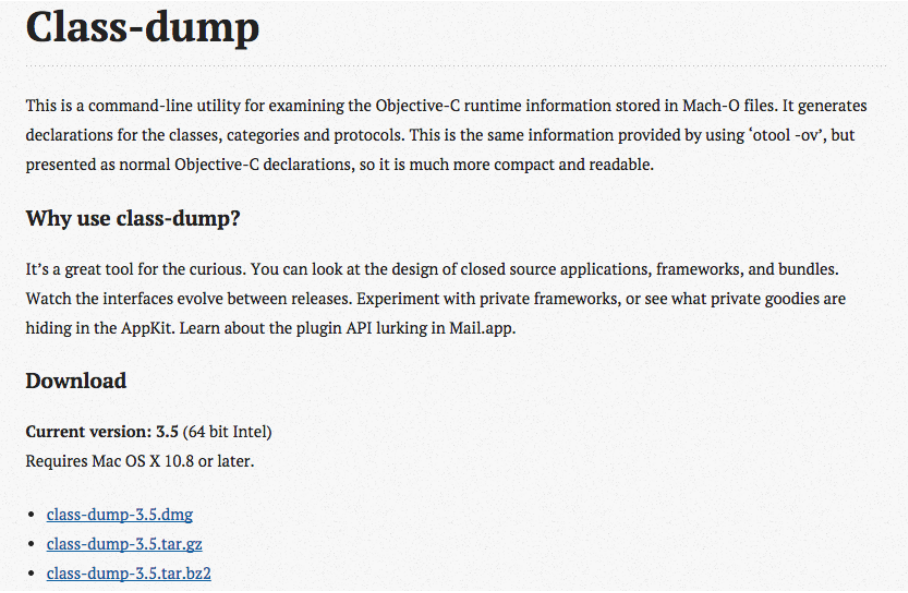


Figure 3-1 Homepage of class-dump

After downloading and decompressing class-dump-3.5.dmg, copy the class-dump executable

to "/usr/bin", and run "sudo chmod 777 /usr/bin/class-dump" in Terminal to grant it execute permission. Run class-dump, you will see its usage:

```
snakeninnysiMac:~ snakeninny$ class-dump
class-dump 3.5 (64 bit)
Usage: class-dump [options] <mach-o-file>

  where options are:
        -a             show instance variable offsets
        -A             show implementation addresses
        --arch <arch>  choose a specific architecture from a universal binary
(ppc, ppc64, i386, x86_64, armv6, armv7, armv7s, arm64)
        -C <regex>     only display classes matching regular expression
        -f <str>       find string in method name
        -H             generate header files in current directory, or directory
specified with -o
        -I             sort classes, categories, and protocols by inheritance
(overrides -s)
        -o <dir>       output directory used for -H
        -r             recursively expand frameworks and fixed VM shared
libraries
        -s             sort classes and categories by name
        -S             sort methods by name
        -t             suppress header in output, for testing
        --list-arches  list the arches in the file, then exit
        --sdk-ios      specify iOS SDK version (will look in
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS<version>.sdk
        --sdk-mac      specify Mac OS X version (will look in
/Developer/SDKs/MacOSX<version>.sdk
        --sdk-root     specify the full SDK root path (or use --sdk-ios/--sdk-mac
for a shortcut)
```

The targets of class-dump are Mach-O binaries, such as library files of frameworks and executables of Apps. Now, I will show you an example of how to use class-dump.

1. Locate the executable of an App

First, copy the target App to OSX, as I placed it under "/Users/snakeninny". Then go to App's directory in Terminal, and use plutil, the Xcode built-in tool to inspect the "CFBundleExecutable" field in Info.plist:

```
snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/SMSNinja.app/
snakeninnysiMac:SMSNinja.app snakeninny$
snakeninnysiMac:SMSNinja.app snakeninny$ plutil -p Info.plist | grep
CFBundleExecutable
  "CFBundleExecutable" => "SMSNinja"
```

"SMSNinja" in the current directory is the executable of the target App.

2. class-dump the executable

class-dump SMSNinja headers to the directory of "/path/to/headers/SMSNinja/", and sort them by name, as follows:

```
snakeninnysiMac:SMSNinja.app snakeninny$ class-dump -S -s -H SMSNinja -o
/path/to/headers/SMSNinja/
```

Repeat this on your own App, and compare the original headers with class-dump headers, aren't they very similar? You will see all the methods are nearly the same except that some arguments' types have been changed to id and their names are missing. With "-S" and "-s" options, the headers are even more readable.

class-dumping our own Apps doesn't make much sense; since class-dump works on closed-source Apps of our own, it can also be used to analyze others' Apps.

From the dumped headers, we can take a peek at the architecture of an App; information under the skin is the cornerstone of iOS reverse engineering. Now that App sizes have become bigger and bigger, more and more third-party libraries are integrated into our own projects, class-dump often produces hundreds and thousands of headers. It'd be a great practice analyzing them one by one manually, but that's overwhelming workload. In the following chapters, we will show you several ways to lighten our workload and focus on the core problems.

It's worth mentioning that, Apps downloaded from AppStore are encrypted by Apple, executables are "shelled" like walnuts, protecting class-dump from working, class-dump will fail in this situation. To enable it again, we need other tools to crack the shell at first, and I'll leave this to the next chapter. To learn more about class-dump, please refer to http://bbs.iosre.com.

3.2 Theos

3.2.1 Introduction to Theos

Theos is a jailbreak development tool written and shared on GitHub by a friend, Dustin Howett (@DHowett). Compared with other jailbreak development tools, Theos' greatest feature is simplicity: It's simple to download, install, compile and publish; the built-in Logos syntax is simple to understand. It greatly reduces our work besides coding.

Additionally, iOSOpenDev, which runs as a plugin of Xcode is another frequently used tool in jailbreak development, developers who are familiar with Xcode may feel more interested in this tool, which is more integrated than Theos. But, reverse engineering deals with low-level

knowledge a lot, most of the work can't be done automatically by tools, it'd be better for you to get used to a less integrated environment. Therefore I strongly recommend Theos, when you use it to finish one practice after another, you will definitely gain a deeper understanding of iOS reverse engineering.

3.2.2 Install and configure Theos

1. Install Xcode and Command Line Tools

Most iOS developers have already installed Xcode, which contains Command Line Tools. For those who don't have Xcode yet, please download it from Mac AppStore for free. If two or more Xcodes have been installed already, one Xcode should be specified as "active" by "xcode-select", Theos will use this Xcode by default. For example, if 3 Xcodes have been installed on your Mac, namely Xcode1.app, Xcode2.app and Xcode3.app, and you want to specify Xcode3 as active, please use the following command:

```
snakeninnys-MacBook:~ snakeninny$ sudo xcode-select -s
/Applications/Xcode3.app/Contents/Developer
```

2. Download Theos

Download Theos from GitHub using the following commands:

```
snakeninnysiMac:~ snakeninny$ export THEOS=/opt/theos
snakeninnysiMac:~ snakeninny$ sudo git clone git://github.com/DHowett/theos.git
$THEOS
Password:
Cloning into '/opt/theos'...
remote: Counting objects: 4116, done.
remote: Total 4116 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (4116/4116), 913.55 KiB | 15.00 KiB/s, done.
Resolving deltas: 100% (2063/2063), done.
Checking connectivity... done
```

3. Configure ldid

ldid is a tool to sign iOS executables; it replaces codesign from Xcode in jailbreak development. Download it from http://joedj.net/ldid to "/opt/theos/bin/", then grant it execute permission using the following command:

```
snakeninnysiMac:~ snakeninny$ sudo chmod 777 /opt/theos/bin/ldid
```

4. Configure CydiaSubstrate

First run the auto-config script in Theos:

```
snakeninnysiMac:~ snakeninny$ sudo /opt/theos/bin/bootstrap.sh substrate
Password:
Bootstrapping CydiaSubstrate...
 Compiling iPhoneOS CydiaSubstrate stub... default target?
 failed, what?
 Compiling native CydiaSubstrate stub...
 Generating substrate.h header...
```

Here we'll meet a bug that Theos cannot generate a working libsubstrate.dylib, which requires our manual fixes. Piece of cake: first search and install CydiaSubstrate in Cydia, as shown in figure 3-2.

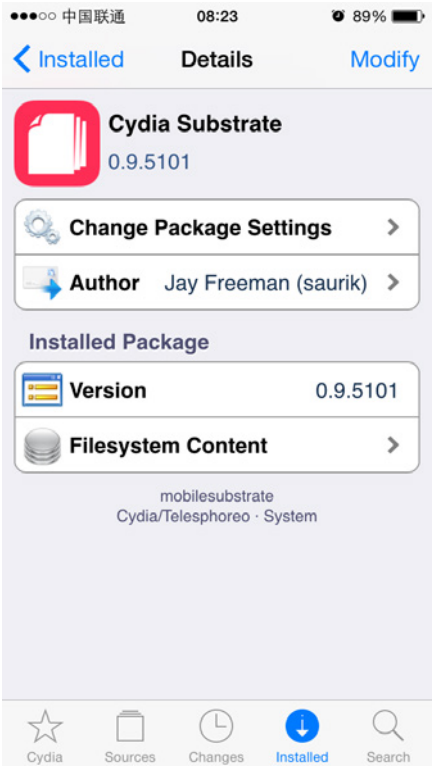


Figure 3- 2 CydiaSubstrate

Then copy "/Library/Frameworks/CydiaSubstrate.framework/CydiaSubstrate" on iOS to somewhere on OSX such as the desktop using iFunBox or scp. Rename it libsubstrate.dylib and copy it to "/opt/theos/lib/libsubstrate.dylib" to replace the invalid file.

5. Configure dpkg-deb

The standard installation package format in jailbreak development is deb, which can be manipulated by dpkg-deb. Theos uses dpkg-deb to pack projects to debs.

Download dm.pl from

https://raw.githubusercontent.com/DHowett/dm.pl/master/dm.pl, rename it dpkg-deb and move it to "/opt/theos/bin/", then grant it execute permission using the following command:

```
snakeninnysiMac:~ snakeninny$ sudo chmod 777 /opt/theos/bin/dpkg-deb
```

6. Configure Theos NIC templates

It is convenient for us to create various Theos projects because Theos NIC templates have 5 different Theos project templates. You can also get 5 extra templates from https://github.com/DHowett/theos-nic-templates/archive/master.zip and put the 5 extracted .tar files under "/opt/theos/templates/iphone/". Some default values of NIC can be customized, please refer to http://iphonedevwiki.net/index.php/NIC#How_to_set_default_values.

3.2.3 Use Theos

1. Create Theos project

1) Change Theos' working directory to whatever you want (like mine is "/User/snakeninny/Code"), and then enter "/opt/theos/bin/nic.pl" to start NIC (New Instance Creator), as follows:

```
snakeninnysiMac:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
```

There are 10 templates available, among which 1, 4, 6, 8, 9 are Theos embedded, and 2, 3, 5, 7, 10 are downloaded in the previous section. At the beginning stage of iOS reverse engineering, we'll be writing tweaks most of the time, usage of other templates can be discussed on http://bbs.iosre.com.

2) Chose "9" to create a tweak project:

```
Choose a Template (required): 9
```

3) Enter the name of the tweak project:

```
Project Name (required): iOSREProject
```
4) Enter a bundle identifier as the name of the deb package:

```
Package Name [com.yourcompany.iosreproject]: com.iosre.iosreproject
```
5) Enter the name of the tweak author:

```
Author/Maintainer Name [snakeninny]: snakeninny
```
6) Enter "MobileSubstrate Bundle filter", i.e. bundle identifier of the tweak target:

```
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.springboard
```
7) Enter the name of the process to be killed after tweak installation:

```
[iphone/tweak] List of applications to terminate upon installation (space-
separated, '-' for none) [SpringBoard]: SpringBoard
Instantiating iphone/tweak in iosreproject/...
Done.
```
After these 7 simple steps, a folder named iosreproject is created in the current directory, which contains the tweak project we just created.

2. Customize project files

It's convenient to create a tweak project with Theos, but the project is so rough that it needs further polish, more information is required. Anyway, let's take a look at our project folder:

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 40
-rw-r--r--  1 snakeninny  staff   184 Dec  3 09:05 Makefile
-rw-r--r--  1 snakeninny  staff  1045 Dec  3 09:05 Tweak.xm
-rw-r--r--  1 snakeninny  staff   223 Dec  3 09:05 control
-rw-r--r--  1 snakeninny  staff    57 Dec  3 09:05 iOSREProject.plist
lrwxr-xr-x  1 snakeninny  staff    11 Dec  3 09:05 theos -> /opt/theos
```
There are only 4 files except one symbolic link pointing to Theos. To be honest, when I first created a tweak project with Theos as a newbie, the simplicity of this project actually attracted me instead of freaking me out, which surprised me. Less is more, Theos does an amazing job in good user experience.

4 files are enough to build a roughcast house, yet more decoration is needed to make it flawless. We're going to extend the 4 files for now.

- Makefile

The project files, frameworks and libraries are all specified in Makefile, making the whole compile process automatic. The Makefile of iOSREProject is shown as follows:

```
include theos/makefiles/common.mk

TWEAK_NAME = iOSREProject
iOSREProject_FILES = Tweak.xm
```

```
include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 SpringBoard"
```

Let's do a brief introduction line by line.

```
include theos/makefiles/common.mk
```

This is a fixed writing pattern, don't make changes.

```
TWEAK_NAME = iOSREProject
```

The tweak name, i.e. the "Project name" in NIC when we create a Theos project. It corresponds to the "Name" field of the control file, please don't change it.

```
iOSREProject_FILES = Tweak.xm
```

Source files of the tweak project, excluding headers; multiple files should be separated by spaces, like:

```
iOSREProject_FILES = Tweak.xm Hook.xm New.x ObjC.m ObjC++.mm
```

It can be changed on demand.

```
include $(THEOS_MAKE_PATH)/tweak.mk
```

According to different types of Theos projects, different .mk files will be included. In the beginning stage of iOS reverse engineering, 3 types of projects are commonly created, they are Application, Tweak and Tool, whose corresponding files are application.mk, tweak.mk and tool.mk respectively. It can be changed on demand.

```
after-install::
    install.exec "killall -9 SpringBoard"
```

I guess you know what's the purpose of these two lines of code from the literal meaning, which is to kill SpringBoard after the tweak is installed during development, and to let CydiaSubstrate load the proper dylibs into SpringBoard when it relaunches.

The content of Makefile seems easy, right? But it's too easy to be enough for a real tweak project. How do we specify the SDK version? How do we import frameworks? How do we link libs? These questions remain to be answered. Don't worry, the bread will have of, the milk will also have of.

✧ Specify CPU architectures

```
export ARCHS = armv7 arm64
```

Different CPU architectures should be separated by spaces in the above configuration. Note, Apps with arm64 instructions are not compatible with armv7/armv7s dylibs, they have to link dylibs of arm64. In the vast majority of cases, just leave it as "armv7 arm64".

✧ Specify the SDK version

```
export TARGET = iphone:compiler:Base SDK:Deployment Target
```
For example:

```
export TARGET = iphone:clang:8.1:8.0
```
It specifies the base SDK version of this project to 8.1, as well deployment target to iOS 8.0. We can also specify "Base SDK" to "latest" to indicate that the project will be compiled with the latest SDK of Xcode, like:

```
export TARGET = iphone:clang:latest:8.0
```

✧ Import frameworks

```
iOSREProject_FRAMEWORKS = framework name
```
For example:

```
iOSREProject_FRAMEWORKS = UIKit CoreTelephony CoreAudio
```
There is nothing to explain. However, as tweak developers, how to import private frameworks attracts us more for sure. It's not much difference to importing documented frameworks:

```
iOSREProject_PRIVATE_FRAMEWORKS = private framework name
```
For example:

```
iOSREProject_PRIVATE_FRAMEWORKS = AppSupport ChatKit IMCore
```
Although it seems to be only one inserted word " PRIVATE ", there's more to notice. Importing private frameworks is not allowed in AppStore development, most of us are not familiar with them. Private frameworks change a lot in each iOS version, so before importing them, please make sure of the existence of the imported frameworks. For example, if you want your tweak to be compatible with both iOS 7 and iOS 8, then Makefile could be written as follows:

```
export ARCHS = armv7 arm64
export TARGET = iphone:clang:latest:7.0

include theos/makefiles/common.mk

TWEAK_NAME = iOSREProject
iOSREProject_FILES = Tweak.xm
iOSREProject_PRIVATE_FRAMEWORK = BaseBoard
include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 SpringBoard"
```
This tweak can be compiled and linked successfully without any error. However, BaseBoard.framework only exists in SDK of iOS 8 and above, so this tweak would fail to work on iOS 7 because of the lack of specified frameworks. In this case, "weak linking" or dyld series functions like dlopen(), dlsym() and dlclose() can solve this problem.

✧ Link Mach-O Objects

`iOSREProject_LDFLAGS = -lx`

Theos use GNU Linker to link Mach-O objects, including .dylib, .a and .o files. Input "man ld" in Terminal and locate to "-lx", it is described as follows:

"-lx This option tells the linker to search for libx.dylib or libx.a in the library search path. If string x is of the form y.o, then that file is searched for in the same places, but without prepending `lib' or appending `.a' or `.dylib' to the filename."

As shown in figure 3-3, all Mach-O objects are named in the formats of "libx.dylib" and "y.o", who're fully compatible with GNU Linker.

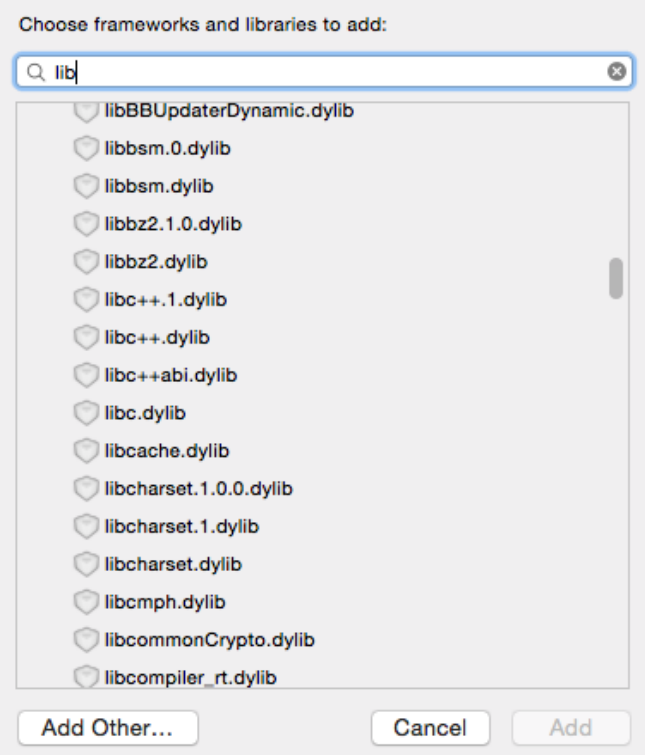


Figure 3- 3 Link Mach-O Objects

So, linking Mach-O objects becomes convenient. For example, if you want to link libsqlite3.0.dylib, libz.dylib and dylib1.o, you can do it like this:

`iOSREProject_LDFLAGS = -lz –lsqlite3.0 –dylib1.o`

There is still one more field to introduce later, but without it Makefile is good to work for now. For more Makefile introductions, you can refer to http://www.gnu.org/software/make/manual/html_node/Makefiles.html.

- Tweak.xm

The default source file of a tweak project created by Theos is Tweak.xm. "x" in "xm"

indicates that this file supports Logos syntax; if this file is suffixed with an only "x", it means Tweak.x will be processed by Logos, then preprocessed and compiled as objective-c; if the suffix is "xm", Tweak.xm will be processed by Logos, then preprocessed and compiled as objective-c++, just like the differences between "m" and "mm" files. There are 2 more suffixes as "xi" and "xmi", you can refer to http://iphonedevwiki.net/index.php/Logos#File_Extensions_for_Logos for details.

The default content of Tweak.xm is as follows:

```
/* How to Hook with Logos
Hooks are written with syntax similar to that of an Objective-C @implementation.
You don't need to #include <substrate.h>, it will be done automatically, as will
the generation of a class list and an automatic constructor.

%hook ClassName

// Hooking a class method
+ (id)sharedInstance {
    return %orig;
}

// Hooking an instance method with an argument.
- (void)messageName:(int)argument {
    %log; // Write a message about this call, including its class, name and
arguments, to the system log.

    %orig; // Call through to the original function with its original arguments.
    %orig(nil); // Call through to the original function with a custom argument.

    // If you use %orig(), you MUST supply all arguments (except for self and
_cmd, the automatically generated ones.)
}

// Hooking an instance method with no arguments.
- (id)noArguments {
    %log;
    id awesome = %orig;
    [awesome doSomethingElse];

    return awesome;
}

// Always make sure you clean up after yourself; Not doing so could have grave
consequences!
%end
*/
```

These are the basic Logos syntax, including 3 preprocessor directives: %hook, %log and %orig. The next 3 examples show how to use them.

✧ %hook

%hook specifies the class to be hooked, must end with %end, for example:

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    NSLog(@"You've pressed home button.");
    %orig; // call the original _menuButtonDown:
}
%end
```

This snippet is to hook [SpringBoard _menuButtonDown:], write something to syslog before executing the original method.

✧ %log

This directive is used inside %hook to write the method arguments to syslog. We can also append anything else with the format of %log([(<type>)<expr>, …]), for example:

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    %log((NSString *)@"iOSRE", (NSString *)@"Debug");
    %orig; // call the original _menuButtonDown:
}
%end
```

The output is as follows:

```
Dec  3 10:57:44 FunMaker-5 SpringBoard[786]: -[<SpringBoard: 0x150eb800>
_menuButtonDown:++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++
    Timestamp:           75607608282
    Total Latency:       20266 us
    SenderID:            0x0000000100000190
    BuiltIn:             1
    AttributeDataLength: 16
    AttributeData:       01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ValueType:           Absolute
    EventType:           Keyboard
    UsagePage:           12
    Usage:               64
    Down:                1
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ]: iOSRE, Debug
```

✧ %orig

%orig is also used inside %hook; it executes the original hooked method, for example:

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    NSLog(@"You've pressed home button.");
    %orig; // call the original _menuButtonDown:
```

```
}
%end
```

If %orig is removed, the original method will not be executed, for example:

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    NSLog(@"You've pressed home button but it's not functioning.");
}
%end
```

It can also be used to replace arguments of the original method, for example:

```
%hook SBLockScreenDateViewController
- (void)setCustomSubtitleText:(id)arg1 withColor:(id)arg2
{
    %orig(@"iOS 8 App Reverse Engineering", arg2);
}
%end
```

The lock screen looks like figure 3-4 with the new argument:

Figure 3- 4 Hack the lock screen

Besides %hook, %log and %orig, there are other common preprocessor directives such as %group, %init, %ctor, %new and %c.

✧ %group

This directive is used to group %hook directives for better code management and conditional initialization (We'll talk about this soon). %group must end with %end, one %group

can contain multiple %hooks, all %hooks that do not belong to user-specific groups will be grouped into %group _ungrouped. For example:

```
%group iOS7Hook
%hook iOS7Class
- (id)iOS7Method
{
    id result = %orig;
    NSLog(@"This class & method only exist in iOS 7.");
    return result;
}
%end
%end // iOS7Hook

%group iOS8Hook
%hook iOS8Class
- (id)iOS8Method
{
    id result = %orig;
    NSLog(@"This class & method only exist in iOS 8.");
    return result;
}
%end
%end // iOS8Hook

%hook SpringBoard
-(void)powerDown
{
    %orig;
}
%end
```

Inside %group iOS7Hook, it hooks [iOS7Class iOS7Method]; inside %group iOS8Hook, it hooks [iOS8Class iOS8Method]; and inside % group _ungrouped, it hooks [SpringBoard powerDown]. Can you get it?

Notice, %group will only work with %init.

✧ %init

This directive is used for %group initialization; it must be called inside %hook or %ctor. If a group name is specified, it will initialize %group SpecifiedGroupName, or it will initialize %group _ungrouped, for example:

```
#ifndef kCFCoreFoundationVersionNumber_iOS_8_0
#define kCFCoreFoundationVersionNumber_iOS_8_0 1140.10
#endif

%hook SpringBoard
- (void)applicationDidFinishLaunching:(id)application
{
    %orig;
```

```
    %init; // Equals to %init(_ungrouped)
    if (kCFCoreFoundationVersionNumber >= kCFCoreFoundationVersionNumber_iOS_7_0
&& kCFCoreFoundationVersionNumber <
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS7Hook);
    if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_8_0) init(iOS8Hook);
}
%end
```

Please remember, a %group will only take effect with a corresponding %init.

✧ %ctor

The constructor of a tweak, it is the first function to be called in the tweak. If we don't define a constructor explicitly, Theos will create one for us automatically, and call %init(_ungrouped) inside it.

```
%hook SpringBoard
- (void)reboot
{
    NSLog(@"If rebooting doesn't work then I'm screwed.");
    %orig;
}
%end
```

The above code works fine, because Theos has called %init implicitly like this:

```
%ctor
{
    %init(_ungrouped);
}
```

However,

```
%hook SpringBoard
- (void)reboot
{
    NSLog(@"If rebooting doesn't work then I'm screwed.");
    %orig;
}
%end

%ctor
{
    // Need to call %init explicitly!
}
```

This %hook never works, because we've defined %ctor explicitly without calling %init explicitly, there lacks a %group(_ungrouped). Generally, %ctor is used to call %init and MSHookFunction, for example:

```
#ifndef kCFCoreFoundationVersionNumber_iOS_8_0
#define kCFCoreFoundationVersionNumber_iOS_8_0 1140.10
#endif
```

```
%ctor
{
    %init;
    if (kCFCoreFoundationVersionNumber >= kCFCoreFoundationVersionNumber_iOS_7_0
&& kCFCoreFoundationVersionNumber <
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS7Hook);
    if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS8Hook);
    MSHookFunction((void *)&AudioServicesPlaySystemSound,
         (void *)&replaced_AudioServicesPlaySystemSound,
         (void **)&original_AudioServicesPlaySystemSound);
}
```

Attention, %ctor doesn't end with %end.

✧ %new

%new is used inside %hook to add a new method to an existing class; it's the same as class_addMethod, for example:

```
%hook SpringBoard
%new
- (void)namespaceNewMethod
{
    NSLog(@"We've added a new method to SpringBoard.");
}
%end
```

Some of you may wonder, category in Objective-C can already add new methods to classes, why do we still need %new? The difference between category and %new is that the former is static while the latter is dynamic. Well, does static adding or dynamic adding matter? Yes, especially when the class to be added is from a certain executable, it matters. For example, the above code adds a new method to SpringBoard. If we use category, the code should look like this:

```
@interface SpringBoard (iOSRE)
- (void)namespaceNewMethod;
@end

@implementation SpringBoard (iOSRE)
- (void)namespaceNewMethod
{
    NSLog(@"We've added a new method to SpringBoard.");
}
@end
```

We will get "error: cannot find interface declaration for 'SpringBoard'" when trying to compile the above code, which indicates that the compiler cannot find the definition of SpringBoard. We can compose a SpringBoard class to cheat the compiler:

```
@interface SpringBoard : NSObject
```

```
@end

@interface SpringBoard (iOSRE)
- (void)namespaceNewMethod;
@end

@implementation SpringBoard (iOSRE)
- (void)namespaceNewMethod
{
    NSLog(@"We've added a new method to SpringBoard.");
}
@end
```

Recompile it, we'll still get the following error:

```
Undefined symbols for architecture armv7:
  "_OBJC_CLASS_$_SpringBoard", referenced from:
      l_OBJC_$_CATEGORY_SpringBoard_$_iOSRE in Tweak.xm.b1748661.o
ld: symbol(s) not found for architecture armv7
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

ld cannot find the definition of SpringBoard. Normally, when there's "symbol(s) not found", most of you may think, if this is because I forget to import any framework? But, SpringBoard is a class of SpringBoard.app rather than a framework, how do we import an executable? I bet you know %new's usage right now.

✧ %c

This directive is equal to objc_getClass or NSClassFromString, it is used in %hook or %ctor to dynamically get a class by name.

Other Logos preprocessor directives including %subclass and %config are seldom used, at least I myself have never used them before. Nonetheless, if you're interested in them, you can refer to http://iphonedevwiki.net/index.php/Logos, or go to http://bbs.iosre.com for discussion.

- control

The contents of control file are basic information of the current deb package; they will be packed into the deb package. The contents of iOSREProject's control file are shown as follows:

```
Package: com.iosre.iosreproject
Name: iOSREProject
Depends: mobilesubstrate
Version: 0.0.1
Architecture: iphoneos-arm
Description: An awesome MobileSubstrate tweak!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
```

Let me give a brief introduction of this file.

- Package field is the name of the deb package, it has the similar naming convention to bundle identifier, i.e. reverse DNS format. It can be changed on demand.
- Name field is used to describe the name of the project; it also can be changed.
- Depends field is used to describe the dependency of this deb package. Dependency means the basic condition to run this tweak, if the current environment does not meet the condition described in depends field, this tweak cannot run properly. For example, the following code means the tweak must run on iOS 8.0 or later with CydiaSubstrate installed.

```
Depends: mobilesubstrate, firmware (>= 8.0)
```

- Version field is used to describe the version of the deb package; it can be changed on demand.
- Architecture field is used to describe the target device architecture, do not change it.
- Description field is used to give a brief introduction of the deb package; it can be changed on demand.
- Maintainer field is used to describe the maintainer of the deb package, say, all deb packages on TheBigBoss are maintained by BigBoss instead of the author. This field can be changed on demand.
- Author field is used to describe the author of the tweak, which is different from the maintainer. It can be changed on demand.
- Section field is used to describe the program type of the deb package, don't change it.

There are still some other fields in control file, but the above fields are enough for Theos projects. For more information, please refer to the official site of debian, http://www.debian.org/doc/debian-policy/ch-controlfields.html, or control files in other deb packages. It's worth mentioning that Theos will further edit control file when packaging:

```
Package: com.iosre.iosreproject
Name: iOSREProject
Depends: mobilesubstrate
Architecture: iphoneos-arm
Description: An awesome MobileSubstrate tweak!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Version: 0.0.1-1
Installed-Size: 104
```

During editing, Theos changes the Version field to indicate packaging times; adds an Installed-Size field to indicate the size of the package. This size may not be exactly the same to the actual size, but don't change it.

The information of control file will show in Cydia directly, as shown in figure 3-5:

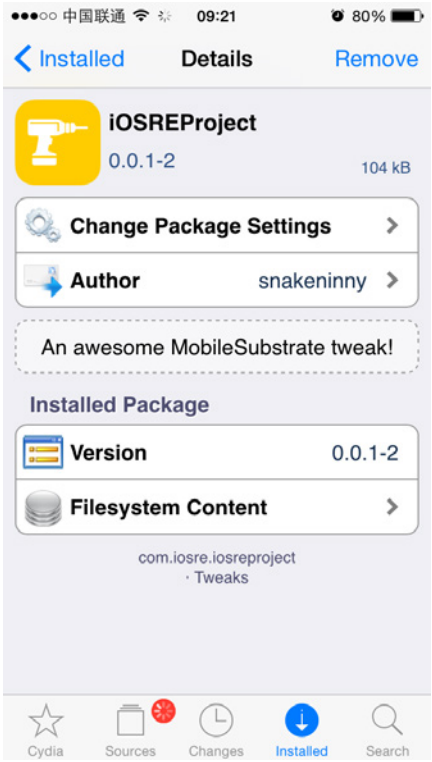


Figure 3- 5 Control informaton in Cydia

- iOSREProject.plist

This plist file has the similar function to Info.plist of an App, which records some configuration information. Specifically in a tweak, it describes the functioning scope of the tweak. It can be edited with plutil or Xcode.

iOSREProject.plist consists of a "Root" dictionary, which has a key named "Filter", as shown in figure 3-6:

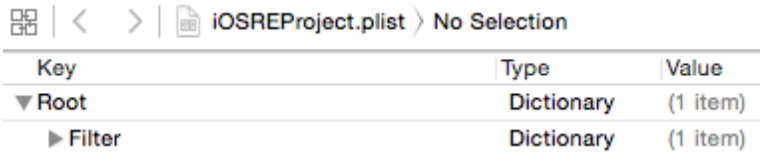


Figure 3- 6 iOSREProject.plist

There's a series of arrays under "Filter", which can be categorized into 3 types.

✧ "Bundles" specifies several bundles as the tweak's targets, as shown in figure 3-7.

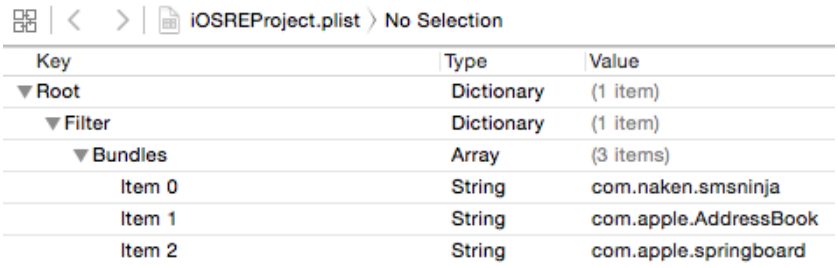


Figure 3- 7 Bundles

According to figure 3-7, this tweak targets 3 bundles, i.e. SMSNinja, AddressBook.framework and SpringBoard.

✧ "Classes" specifies several classes as the tweak's targets, as shown in figure 3-8.

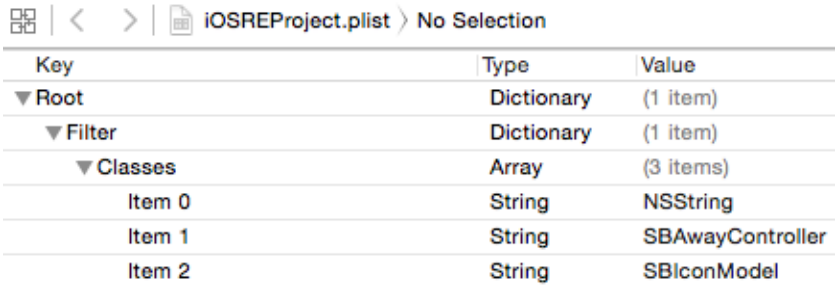


Figure 3- 8 Classes

According to figure 3-8, this tweak targets 3 classes, i.e. NSString, SBAwayController and SBIconModel.

✧ "Executables" specifies several executables as the tweak's targets, as shown in figure 3-9.

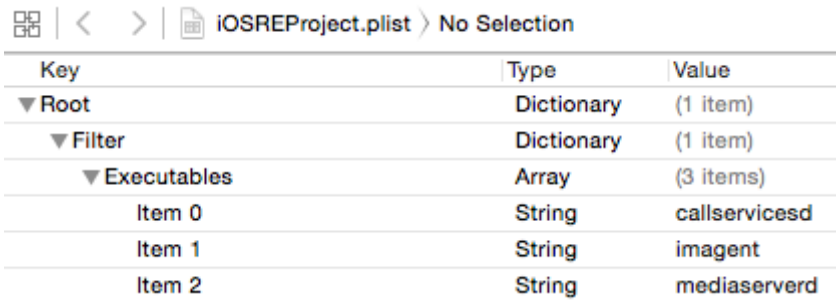


Figure 3- 9 Executables

According to figure 3-9, this tweak targets 3 executables, i.e. callservicesd, imagent and mediaserverd.

These 3 types can be used together, as shown in figure 3-10.

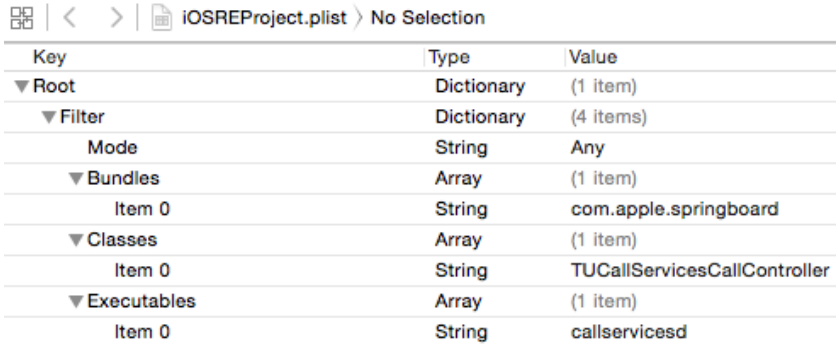


Figure 3- 10 A Mix-targeted tweak

Attention, when there're different kinds of arrays in "Filter", we have to add an extra "Mode : Any" key-value pair.

3. Compile + Package + Install

We've installed Theos, created our first tweak project via NIC, and gone over all project files. In the end, we must compile the tweak and install it on iOS to start experiencing "safe mode" again and again. Are you excited?

- Compile

"make" command is used to compile Theos project. Just run "make" under our Theos project directory:

```
snakeninnysiMac:iosreproject snakeninny$ make
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
```

From the output, we know Theos has finished preprocessing, compiling, linking, stripping and signing. After that, an "obj" folder appears in the current folder.

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 32
-rw-r--r--  1 snakeninny  staff  262 Dec  3 09:20 Makefile
-rw-r--r--  1 snakeninny  staff    0 Dec  3 11:28 Tweak.xm
-rw-r--r--  1 snakeninny  staff  223 Dec  3 09:05 control
-rw-r--r--@ 1 snakeninny  staff  175 Dec  3 09:48 iOSREProject.plist
drwxr-xr-x  5 snakeninny  staff  170 Dec  3 11:28 obj
lrwxr-xr-x  1 snakeninny  staff   11 Dec  3 09:05 theos -> /opt/theos
```

There is a .dylib file in it:

```
snakeninnysiMac:iosreproject snakeninny$ ls -l ./obj
total 272
-rw-r--r--  1 snakeninny  staff  33192 Dec  3 11:28 Tweak.xm.b1748661.o
-rwxr-xr-x  1 snakeninny  staff  98784 Dec  3 11:28 iOSREProject.dylib
```

It's the core of our tweak.

- Package

Theos uses "make package" command to pack Theos projects. In fact, "make package" executes "make" and "dpkb-deb" in sequence to finish its job.

```
snakeninnysiMac:iosreproject snakeninny$ make package
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
Making stage for tweak iOSREProject...
dm.pl: building package `com.iosre.iosreproject' in
`./com.iosre.iosreproject_0.0.1-7_iphoneos-arm.deb'.
```

"make package" has created a "com.iosre.iosreproject_0.0.1-7_iphoneos-arm.deb" file, which is ready to be published.

There is another important function of "make package" command. After executing this command, besides "obj" folder, another "_" folder is also created as shown below.

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 40
-rw-r--r--  1 snakeninny  staff   262 Dec  3 09:20 Makefile
-rw-r--r--  1 snakeninny  staff     0 Dec  3 11:28 Tweak.xm
drwxr-xr-x  4 snakeninny  staff   136 Dec  3 11:35 _
-rw-r--r--  1 snakeninny  staff  2396 Dec  3 11:35 com.iosre.iosreproject_0.0.1-
7_iphoneos-arm.deb
-rw-r--r--  1 snakeninny  staff   223 Dec  3 09:05 control
-rw-r--r--@ 1 snakeninny  staff   175 Dec  3 09:48 iOSREProject.plist
drwxr-xr-x  5 snakeninny  staff   170 Dec  3 11:35 obj
lrwxr-xr-x  1 snakeninny  staff    11 Dec  3 09:05 theos -> /opt/theos
```

What's this folder for? Open it, we can see 2 subfolders in it, namely "DEBIAN" and

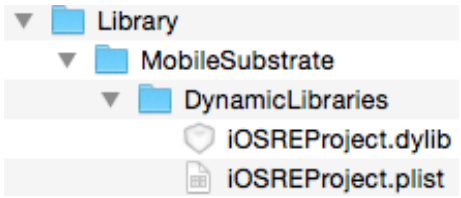
“Library”:

```
snakeninnysiMac:iosreproject snakeninny$ ls -l _
total 0
drwxr-xr-x  3 snakeninny  staff  102 Dec  3 11:35 DEBIAN
drwxr-xr-x  3 snakeninny  staff  102 Dec  3 11:35 Library
```

There is only an edited control file in “DEBIAN”.

```
snakeninnysiMac:iosreproject snakeninny$ ls -l _/DEBIAN
total 8
-rw-r--r--  1 snakeninny  staff  245 Dec  3 11:35 control
```

The structure of “Library” directory is shown in figure 3-11:



Fire 3- 11 Library directory structure

If compared with the contents of deb package:

```
snakeninnysiMac:iosreproject snakeninny$ dpkg -c com.iosre.iosreproject_0.0.1-
7_iphoneos-arm.deb
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35 ./
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35 ./Library/
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35 ./Library/MobileSubstrate/
drwxr-xr-x snakeninny/staff  0 2014-12-03
11:35 ./Library/MobileSubstrate/DynamicLibraries/
-rwxr-xr-x snakeninny/staff 98784 2014-12-03
11:35 ./Library/MobileSubstrate/DynamicLibraries/iOSREProject.dylib
-rw-r--r-- snakeninny/staff   175 2014-12-03
11:35 ./Library/MobileSubstrate/DynamicLibraries/iOSREProject.plist
```

And the files of iOSREProject seen in Cydia, as shown in figure 3-12.

Figure 3-13 iOSREProject files

We can see that they have the same directory structures, and you may have already guessed that this deb package is simply a combination of "DEBIAN" which contains debian information, and "Library" which contains the actual files. In fact, we can also create a "layout" folder under the current project directory before packaging and installing the project on iOS. In this way, all files in "layout" will be extracted to the same positions of iOS filesystem ("layout" mentioned here acts as root directory, i.e. "/" on iOS), enhancing the functionality of deb packages lot. Let's take an example to see the magic of "layout".

Go back to iOSREProject, input "make clean" and "rm *.deb" in Terminal to restore the project to the original state:

```
snakeninnysiMac:iosreproject snakeninny$ make clean
rm -rf ./obj
rm -rf "/Users/snakeninny/Code/iosreproject/_"
snakeninnysiMac:iosreproject snakeninny$ rm *.deb
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 32
-rw-r--r--  1 snakeninny  staff  262 Dec  3 09:20 Makefile
-rw-r--r--  1 snakeninny  staff    0 Dec  3 11:28 Tweak.xm
-rw-r--r--  1 snakeninny  staff  223 Dec  3 09:05 control
-rw-r--r--@ 1 snakeninny  staff  175 Dec  3 09:48 iOSREProject.plist
lrwxr-xr-x  1 snakeninny  staff   11 Dec  3 09:05 theos -> /opt/theos
```

Then create a new "layout" folder:

```
snakeninnysiMac:iosreproject snakeninny$ mkdir layout
```

And put some random empty files under "layout":

```
snakeninnysiMac:iosreproject snakeninny$ touch ./layout/1.test
snakeninnysiMac:iosreproject snakeninny$ mkdir ./layout/Developer
snakeninnysiMac:iosreproject snakeninny$ touch ./layout/Developer/2.test
snakeninnysiMac:iosreproject snakeninny$ mkdir -
p ./layout/var/mobile/Library/Preferences
snakeninnysiMac:iosreproject
snakeninny$ touch ./layout/var/mobile/Library/Preferences/3.test
```

At last, run "make package" to pack, then copy the deb package to iOS, and install it via iFile. Now you can inspect files of iOSREProject in Cydia, as shown in figure 3-13.

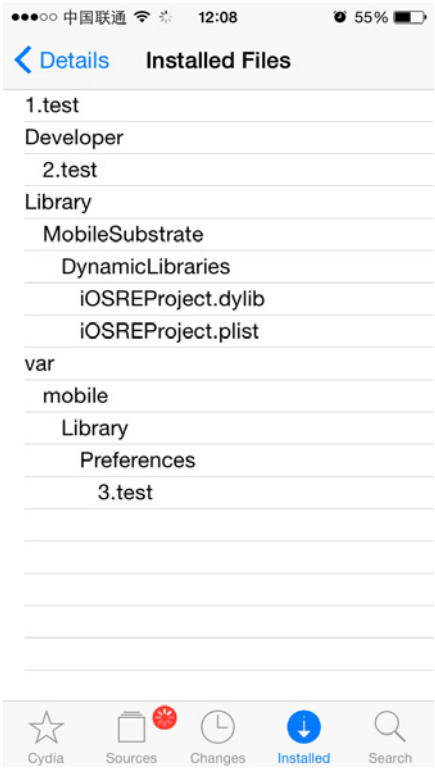


Figure 3-13 Installed files of iOSREProject

As we can see, all the files except "DEBIAN" are extracted to the same positions of iOS filesystem, all necessary subfolders are also created automatically. There are still many things about deb package we didn't mention, please refer to http://www.debian.org/doc/debian-policy for more information.

- Installation

Last but not least, we need to install this deb package on iOS. There are several ways to install, but installation through GUI and installation through command line are two of the most typical installation methods. Most of you may think the GUI way is easier, well, let's take a look at it first.

✧ Installation through GUI

This method is quite easy: First copy the deb package to iOS via iFunBox, then install it via iFile, and reboot iOS. All steps are operated on GUI, but there are too many interactions between human and device, we have to switch between PC and iPhone, which leads to inconvenience, hence is not suitable for tweak development.

✧ Installation through command line.

This method makes use of very simple ssh commands, which requires OpenSSH to be installed on jailbroken iOS. If you have no idea about what we are talking, go through the "OpenSSH" section in chapter 4 quickly to get some help. Let's see how to install through command line now.

First, add your iOS IP to the first line of Makefile:

```
export THEOS_DEVICE_IP = iOSIP
export ARCHS = armv7 arm64
export TARGET = iphone:clang:latest:8.0
```

Then enter "make package install" to compile, package and install in one click:

```
snakeninnysiMac:iosreproject snakeninny$ make package install
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
Making stage for tweak iOSREProject...
dm.pl: building package `com.iosre.iosreproject:iphoneos-arm' in
`./com.iosre.iosreproject_0.0.1-15_iphoneos-arm.deb'
install.exec "cat > /tmp/_theos_install.deb; dpkg -i /tmp/_theos_install.deb &&
rm /tmp/_theos_install.deb" < "./com.iosre.iosreproject_0.0.1-15_iphoneos-
arm.deb"
root@iOSIP's password:
Selecting previously deselected package com.iosre.iosreproject.
(Reading database ... 2864 files and directories currently installed.)
Unpacking com.iosre.iosreproject (from /tmp/_theos_install.deb) ...
Setting up com.iosre.iosreproject (0.0.1-15) ...
install.exec "killall -9 SpringBoard"
root@iOSIP's password:
```

Among the above information, Theos has asked for the root password twice. Although it seems safe, it's inconvenient. Fortunately, we can skip the input of password over and over by configuring the authorized_keys on iOS, as follows:

✧ Remove the entry of iOSIP in "/Users/snakeninny/.ssh/known_hosts".

Assume that your iOS IP address is iOSIP. Edit "/Users/snakeninny/.ssh/known_hosts",

and locate the entry of iOSIP:

```
iOSIP ssh-rsa
hXFscxBCVXgqXhwm4PUoUVBFWRrNeG6gVI3Ewm4dqwusoRcyCxZtm5bRiv4bXfkPjsRkWVVfrW3uT52Hh
x4RqIuCOxtWE7tZqc1vVap4HIzUu3mwBuxog7WiFbsbbaJY4AagNZmX83Wmvf8li5aYMsuKeNagdJHzJN
tjM3vtuskK4jKzBkNuj0M89TrV4iEmKtI4VEoEmHMYzWwMzExXbyX5NyEg5CRFmA46XeYCbcaY0L90GEx
XsWMMLA27tA1Vt1ndHrKNxZttgAw31J90UDnOGlMbWW4M7FEqRWQsWXxfGPk0W7AlA54vaDXllI5CD5nL
Au4VkRjPIUBrdH5O1fqQ3qGkPayhsym3g0VZeYgU4JAMeFc3
```

Delete this entry.

✧ Generate authorized_keys.

Execute the following commands in Terminal:

```
snakeninnysiMac:~ snakeninny$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/snakeninny/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/snakeninny/.ssh/id_rsa.
Your public key has been saved in /Users/snakeninny/.ssh/id_rsa.pub.
……
snakeninnysiMac:~ snakeninny$ cp /Users/snakeninny/.ssh/id_rsa.pub
~/authorized_keys
```

authorized_keys will be generated under users home directory.

✧ Configure iOS

Execute the following commands in Terminal:

```
FunMaker-5:~ root# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/var/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/root/.ssh/id_rsa.
Your public key has been saved in /var/root/.ssh/id_rsa.pub.
……
FunMaker-5:~ root# logout
Connection to iOSIP closed.
snakeninnysiMac:iosreproject snakeninny$ scp ~/authorized_keys
root@iOSIP:/var/root/.ssh
The authenticity of host 'iOSIP (iOSIP)' can't be established.
RSA key fingerprint is 75:98:9a:05:a3:27:2d:23:08:d3:ee:f4:d1:28:ba:1a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'iOSIP' (RSA) to the list of known hosts.
root@iOSIP's password:
authorized_keys
100%  408     0.4KB/s   00:00
```

ssh into iOS again to see if any passwords are required. Now, “make package install” becomes “one time configuration, one click installation”, yay!

✧ Clean

Theos provides a convenient command "make clean" to clean our project. It indeed excutes "rm -rf ./obj" and "rm -rf "/Users/snakeninny/Code/iosre/_"" in turn, thereby removes folders generated by "make" and "make package". Of course, you can further use "rm *.deb" to remove all deb packages generated by "make package".

3.2.4 An example tweak

The previous sections have introduced Theos almost thoroughly, although not all contents are covered, it is way enough for beginners. I have already talked so much about Theos without writing a single line of code, but we're not done yet.

Now, I will take a simplest tweak to explain everything we've introduced. After installing this tweak, a UIAlertView will popup after each respring.

1. Create tweak project "iOSREGreetings" using Theos

Use the following commands to create iOSREGreetings project:

```
snakeninnysiMac:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREGreetings
Package Name [com.yourcompany.iosregreetings]: com.iosre.iosregreetings
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.springboard
[iphone/tweak] List of applications to terminate upon installation (space-
separated, '-' for none) [SpringBoard]:
Instantiating iphone/tweak in iosregreetings/...
Done.
```

2. Edit Tweak.xm

The edited Tweak.xm looks like this:

```
%hook SpringBoard
```

```
- (void)applicationDidFinishLaunching:(id)application
{
    %orig;

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Come to
http://bbs.iosre.com for more fun!" message:nil delegate:self
cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alert show];
    [alert release];
}

%end
```

3. Edit Makefile and control

The edited Makefile looks like this:

```
export THEOS_DEVICE_IP = iOSIP
export ARCHS = armv7 arm64
export TARGET = iphone:clang:latest:8.0

include theos/makefiles/common.mk

TWEAK_NAME = iOSREGreetings
iOSREGreetings_FILES = Tweak.xm
iOSREGreetings_FRAMEWORKS = UIKit

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 SpringBoard"
```

The edited control looks like this:

```
Package: com.iosre.iosregreetings
Name: iOSREGreetings
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Greetings from iOSRE!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

This tweak is rather simple. When [SpringBoard applicationDidFinishLaunching:] is called, SpringBoard finishes launching. We hook this method, carry out the original implementation via %orig, then display a custom UIAlertView. With this tweak, every time we relaunch SpringBoard, a UIAlertView pops up. Can you get it?

If you're OK with this section so far, please enter "make package install" in Terminal. When the lock screen shows, you will see the magic as shown in figure 3-14:

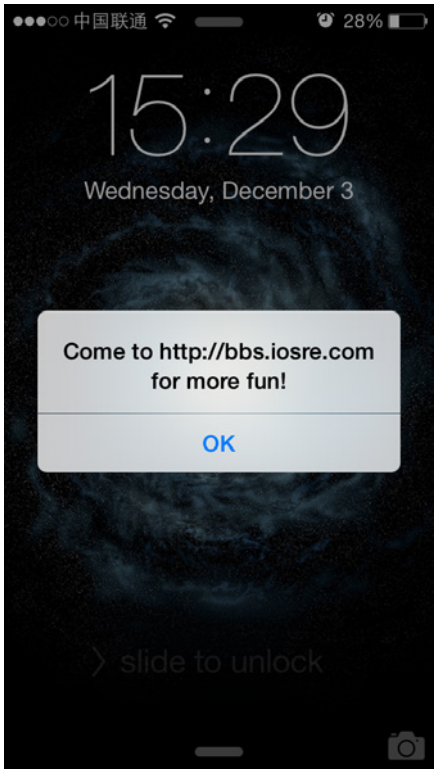


Figure 3- 14 Our first tweak

Yes, with just some tiny modifications, the behaviors of Apps are changed. Now, iOS is opening its long closed door to us… The common scenarios of Theos and Logos are mostly covered in this section, and for a more thorough introduction, please refer to http://iphonedevwiki.net/index.php/Theos and http://iphonedevwiki.net/index.php/Logos.

Because of Theos, it has never been easier to modify a closed-source App. As we have already mentioned, with the increase of App sizes, class-dump produces a greater amount of headers. It has became much more difficult to locate our targets than pure coding. Facing thousands lines of code, if there are no other tools to aid our analysis, reverse engineering would be a nightmare. Now, it's showtime of these auxiliary analysis tools.

3.3 Reveal

Figure 3- 15 Reveal

Reveal, as shown in figure 3-15, is a UI analysis tool by ITTY BITTY, enabling us to see the view hierarchy of an App intuitively. The official purpose of Reveal is to "See your application's view hierarchy at runtime with advanced 2D and 3D visualisations", but as reverse engineers, seeing our own Apps' view hierarchies is obviously not enough, we should be able to see other Apps' view hierarchies. Figure 3-16 shows the effect of seeing AppStore's view hierarchy using Reveal.

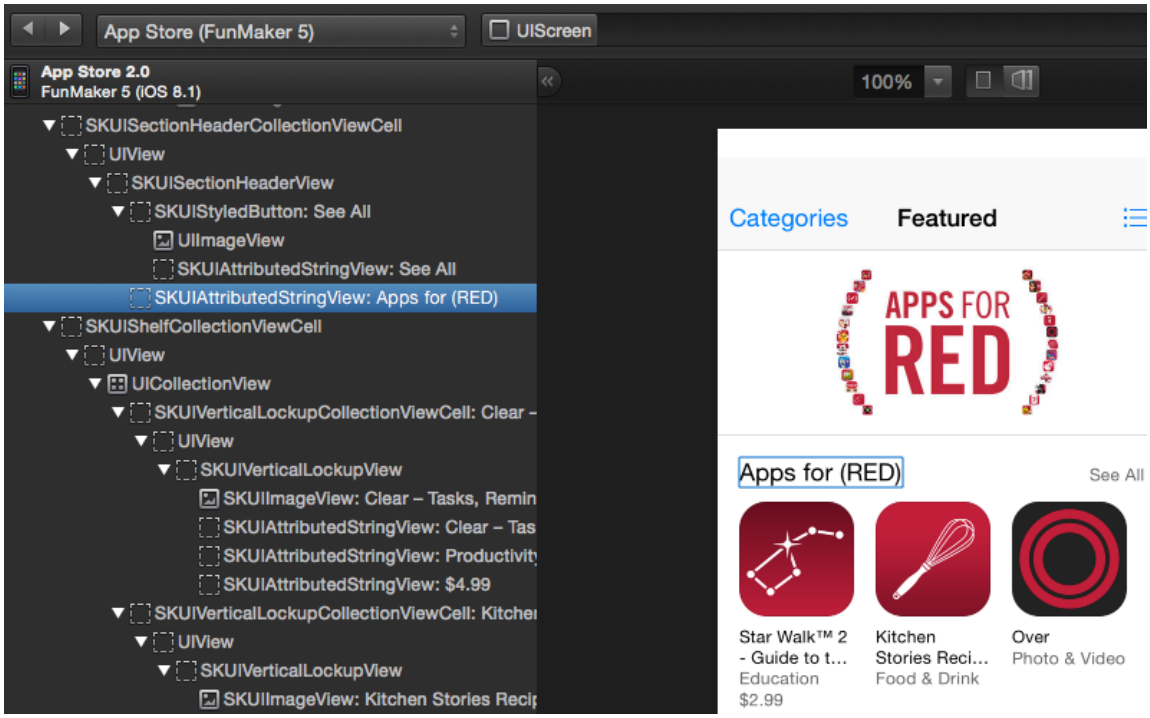


Figure 3-16 See the view hierarchy of AppStore

On the left side of Reveal, the UI layout of AppStore is presented as a tree, when choosing a control object, the corresponding UI element will be marked on the right side of Reveal in real time. At the same time, Reveal also parses the class name of this control object, as shown in figure 3-16, the class name of the selected object is SKUIAttributedStringView. To analyze the view hierarchies of other's Apps, we need to make some configurations in Reveal.

1. Install Reveal Loader

Search and install Reveal Loader in Cydia, as shown in figure 3-17.

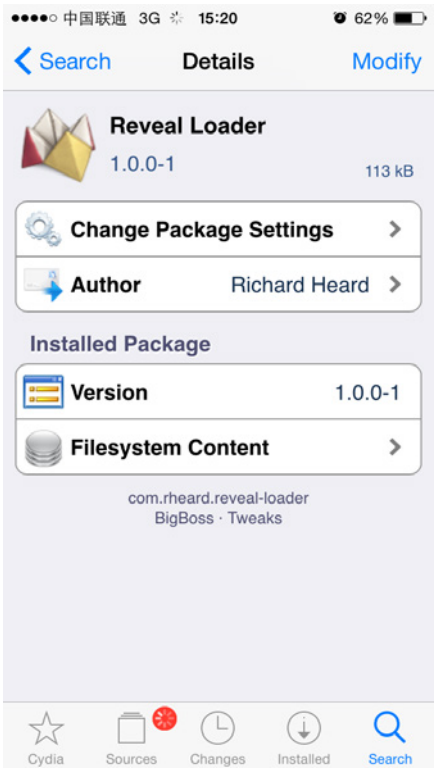


Figure 3-17 Reveal Loader

Remarkably, when installing Reveal Loader, it will download a necessary file libReveal.dylib from Reveal's official website automatically. If the network condition is not good, this file may not be downloaded successfully, and Reveal Loader is not fault tolerant to download failures. As a result, Cydia may stuck for a long time and stop responding. Therefore, after download completes, you'd better check whether there is a "RHRevealLoader" folder under the iOS directory "/Library/".

```
FunMaker-5:~ root# ls -l /Library/ | grep RHRevealLoader
drwxr-xr-x  2 root   admin  102 Dec  6 11:10 RHRevealLoader
```

If not, create one manually:

```
FunMaker-5:~ root# mkdir /Library/RHRevealLoader
```

Then open Reveal, click "Help" menu, choose "Show Reveal Library in Finder", as shown in figure 3-18.

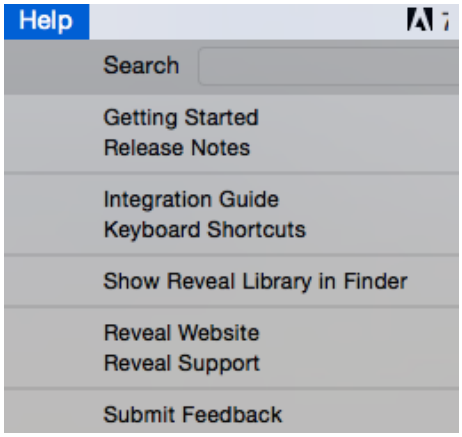


Figure 3- 18 Show Reveal Library in Finder

Then Finder will pop out just like figure 3-19.

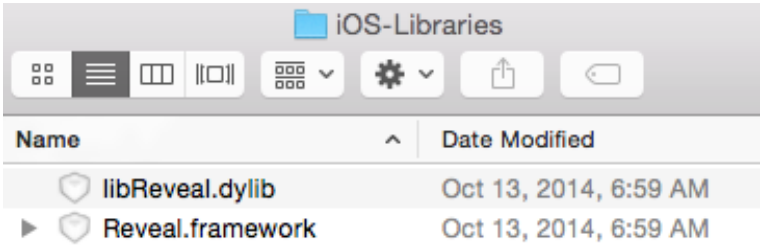


Figure 3- 19 libReveal.dylib

Copy libReveal.dylib to the RHRevealLoader folder through scp or iFunBox:

```
FunMaker-5:~ root# ls -l /Library/RHRevealLoader
total 3836
-rw-r--r-- 1 root admin 3927408 Dec  6 11:10 libReveal.dylib
```

By now, the installation of Reveal Loader completes.

2. Configure Reveal Loader

The configuration of Reveal Loader is inside the stock Settings App with the name “Reveal”, as shown in figure 3-20.

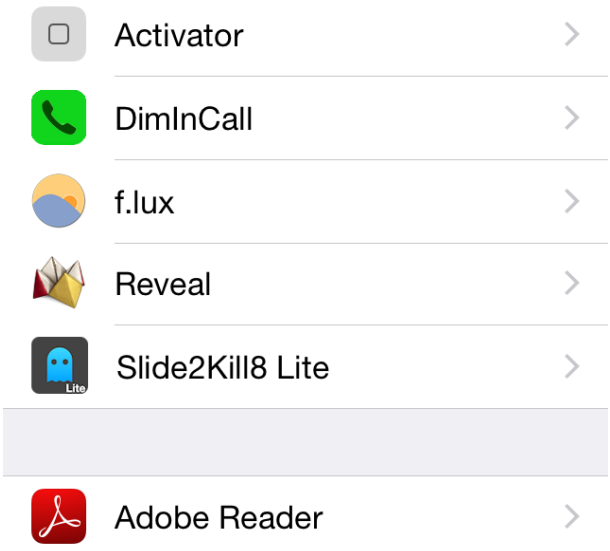


Figure 3- 20 Reveal Loader

Click "Reveal", some declaration appears as shown in figure 3-21.

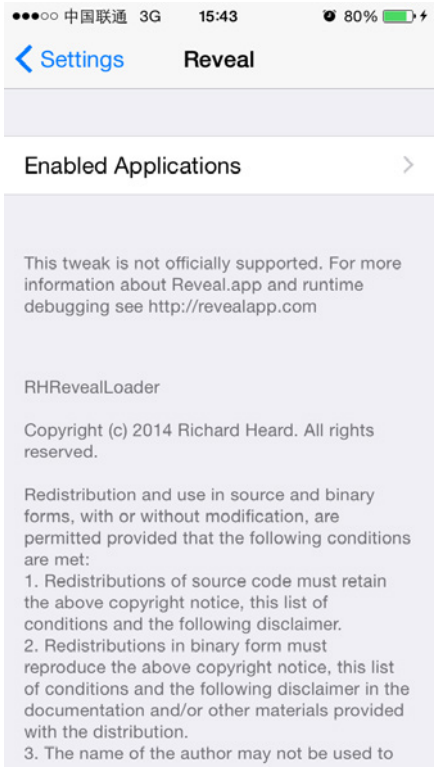


Figure 3-21 Declaration of Reveal Loader

Click "Enabled Applications" to enter the configuration view. Turn on the switch of the App you want to analyze. Here we've turned on AppStore and Calculator's switches, as shown in figure 3-22.

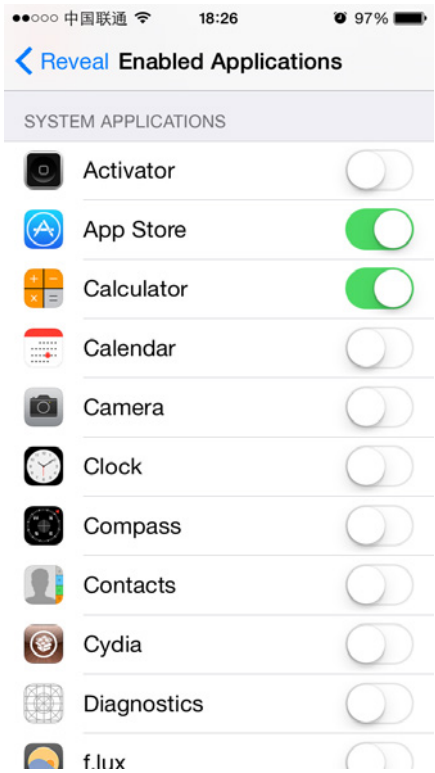


Figure 3-22 configuration of Reveal Loader

That's it. The configuration of Reveal Loader is simple and straightforward, isn't it?

3. Use Reveal to see the view hierarchy of the target App

Everything is ready, now it's the showtime of Reveal. First, one thing should be confirmed that OSX and iOS must be in the same LAN, then launch Reveal and relaunch the target App, i.e. if the target App is running, you need to terminate it first and run it again. The target App can be chosen from top left corner of Reveal. Wait a moment, Reveal will display the view hierarchy of the target App, as shown in figure 3-23.

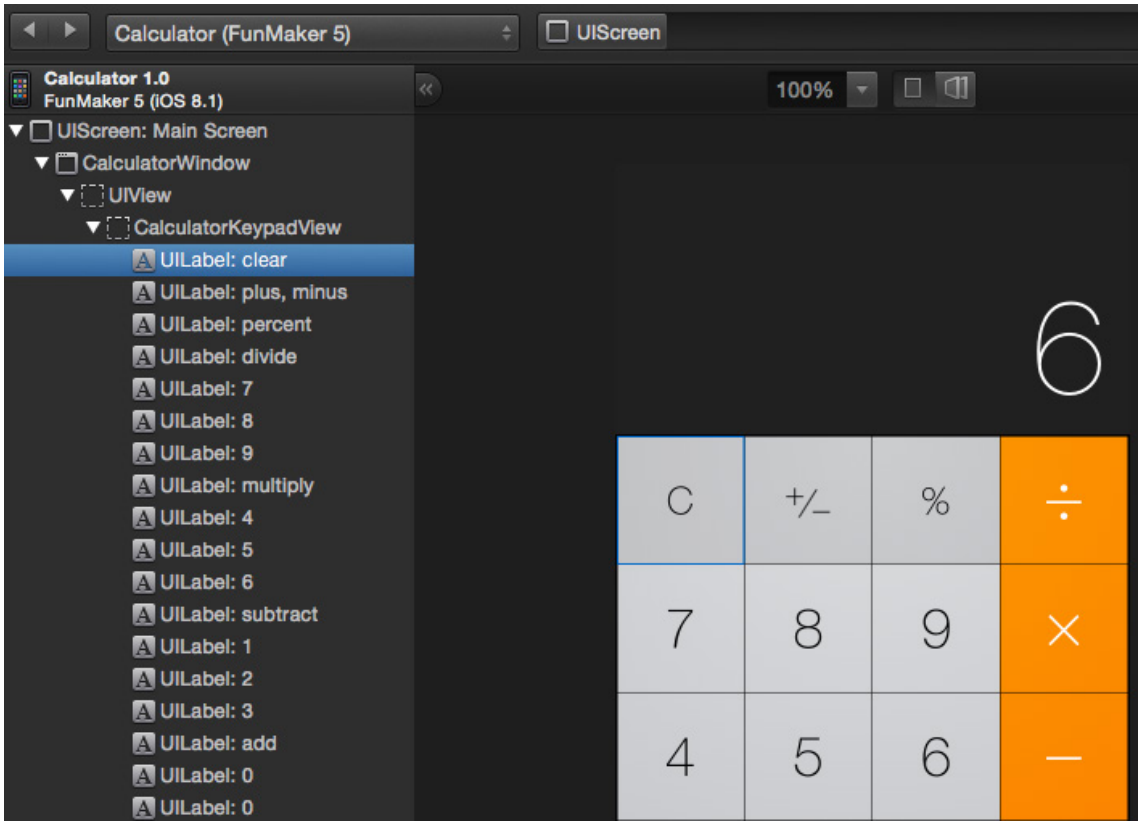


Figure 3-23 View hierarchy of Calculator

Reveal is not complicate and quite user-friendly. But in iOS reverse engineering, analysis on UI is not enough, Apps' inner implementations under the hood are our final goals. From part 3 of this book, we will use recursiveDescription function, which is the "command line" version of Reveal, together with Cycript to find the corresponding code snippets of UI, then you will know the real power of iOS reverse engineering.

3.4 IDA

3.4.1 Introduction to IDA

Even if you've never done any iOS reverse engineering before, you may have heard of IDA (The Interactive Disassembler), as shown in figure 3-24. For reverse engineers, IDA is so well-known that most of our daily work are tightly related to it. If class-dump can help us get the dots out of an App, then IDA can connect the dots to form a plane.

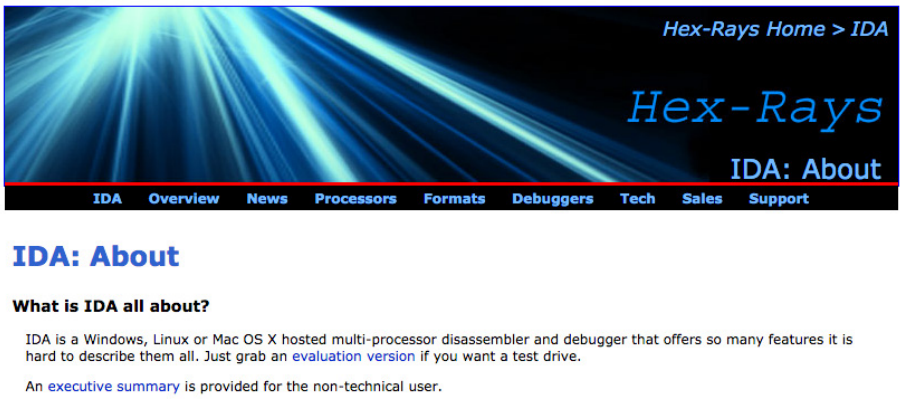


Figure 3-24 Official website of IDA

Generally speaking, IDA is a multi-processor disassembler and debugger fully supporting Windows, Linux and Mac OS X. It is so powerful that even the official site cannot give a complete function list.

To be honest, IDA is quite expensive for personal users. But the author is kind enough to offer a free evaluation version, which works well enough for beginners. It is convenient to download and install IDA, you can refer to https://www.hex-rays.com/products/ida/index.shtml for details.

3.4.2 Use IDA

IDA will shortly display an "About" window after launch, as shown in figure 3-25.

Figure 3- 25 IDA launch window

You can click "OK" or wait for a few seconds to close the window, after that you will see the main screen of IDA, as shown in figure 3-26.

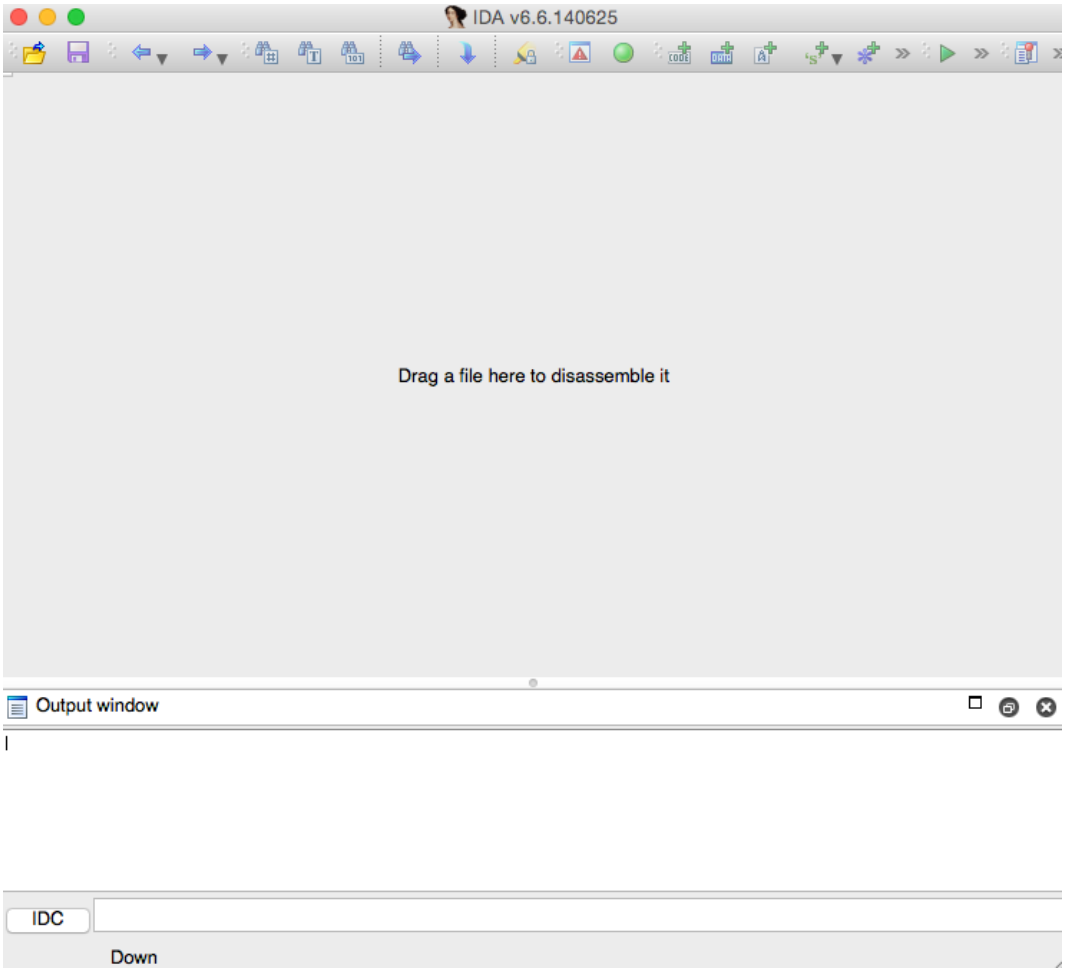


Figure 3-26 Main screen of IDA

In this screen, you don't have to search for "Open File" in the menu and locate the file to be disassembled folder by folder, but just drag the target file to the gray zone with the placeholder "Drag a file here to disassemble it". After opening the file, there is still something to be configured, as shown in figure 3-27.

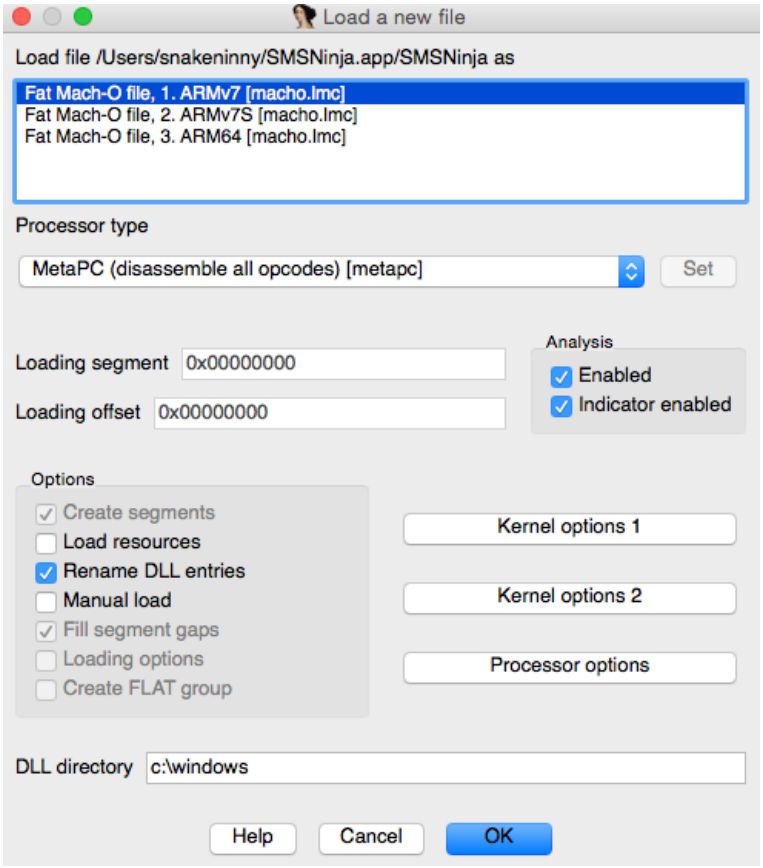


Figure 3-27 Initial configurations

There's one thing to mention: For a fat binary (which refers to the binary that contains different instruction sets for the purpose of being compatible with different CPU architectures), the white frame in figure 3-27 will list several Mach-O files. I suggest you read table 4-1 to find the ARM type of your device. For example, my iPhone 5 corresponds to ARMv7S. If the ARM type of your device is not in the white frame, you should choose the backward compatible one, i.e. for ARMv7S devices, choose ARMv7S if there is ARMv7S in the list, otherwise choose ARMv7. This selection method handles 99% of all cases, if you happen to be the 1%, please come to http://bbs.iosre.com, we'll solve the problem together.

Here, I've chosen ARMv7S, then click "OK". Several windows will popup, just click "YES" or "OK" to close them, as shown in figure 3-28 and 3-29.

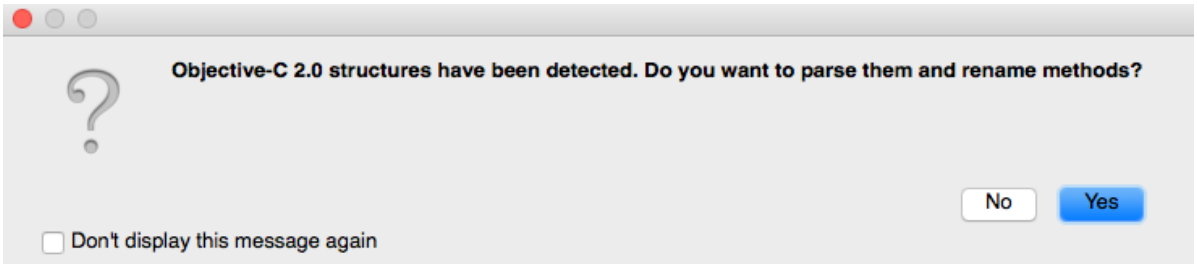


Figure 3-28 IDA launch option

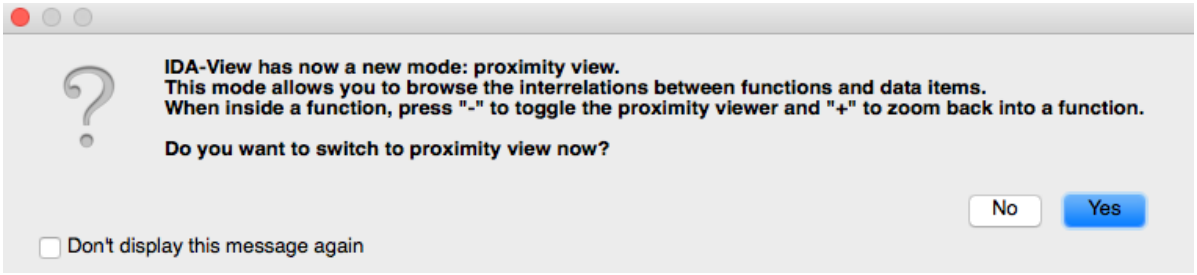


Figure 3-29 IDA launch option

Since we cannot save our configurations in the evaluation version of IDA, checking the box "Don't display this message again" doesn't work at all, it will still show in the next launch.

After clicking all the "OK" and "YES" buttons, the dazzling main screen shows up as in figure 3-30.

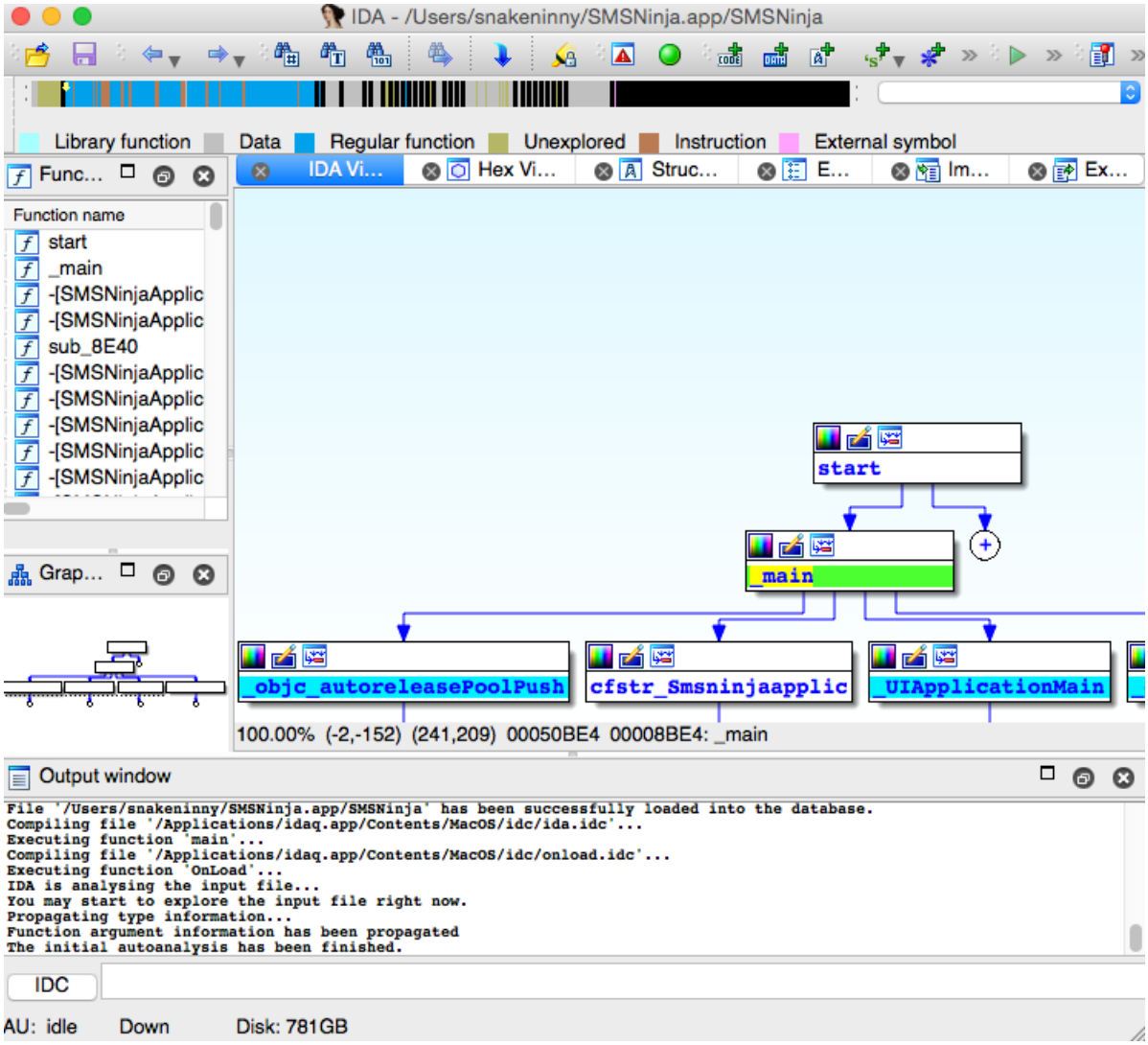


Figure 3-30 IDA main screen

When entering the screen in figure 3-30, you will see the progress bar at the top loading, the output window at the bottom printing the analysis progress. When the main color of the progress bar changes to blue, and the output window shows the message "The initial autoanalysis has been finished", it indicates the initial analysis is completed.

At the beginning stage, IDA is mainly used for static analysis, the output window is quite useless, we can close it for now.

Now that there are two major windows, on the left is "Functions window" as shown in figure 3-31, on the right is "Main window" as shown in figure 3-32. Now, let's take a look at them one by one.

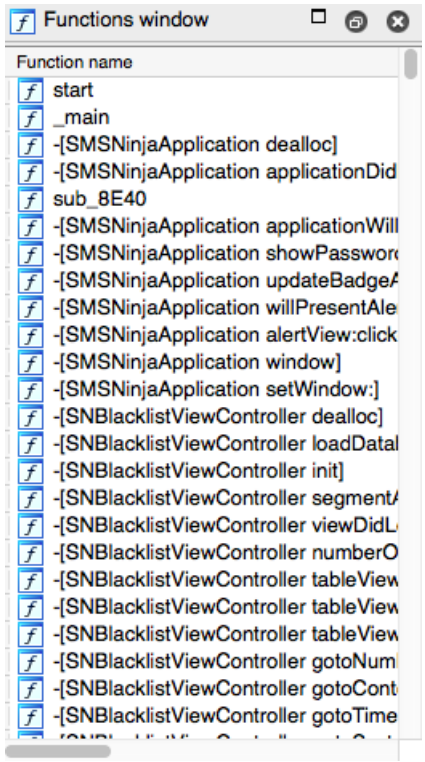


Figure 3-31 Functions window

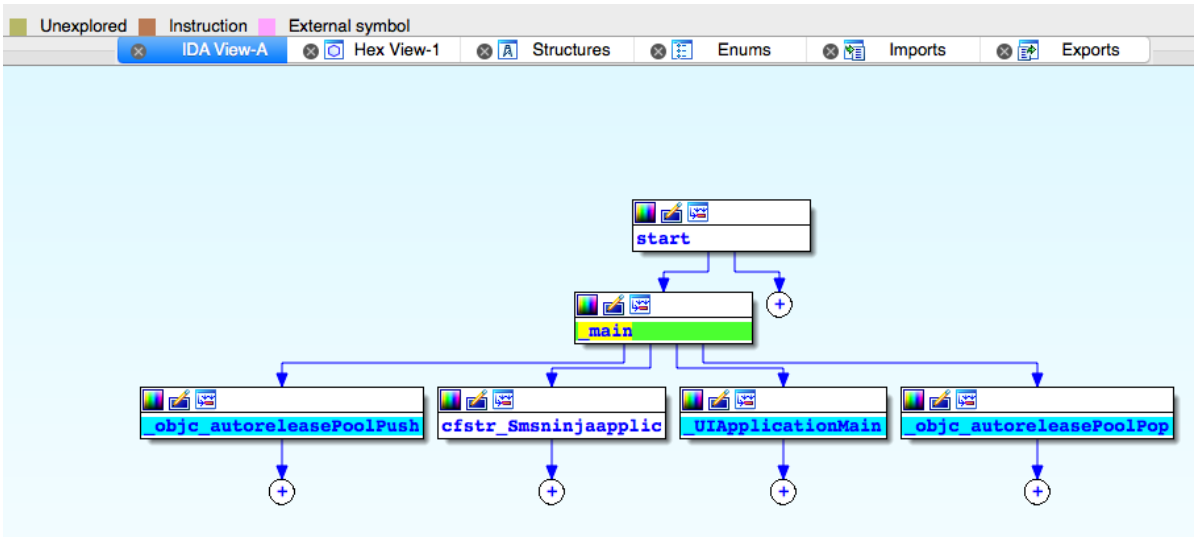


Figure 3-32 Main window

- Functions window

As its name indicates, this window shows all functions (More accurately, Objective-C functions should be called methods, but we're referring them to functions hereafter), double click one function name, the main window will show its implementation. When click "Search" menu of Function Window, a submenu will show up as figure 3-33.

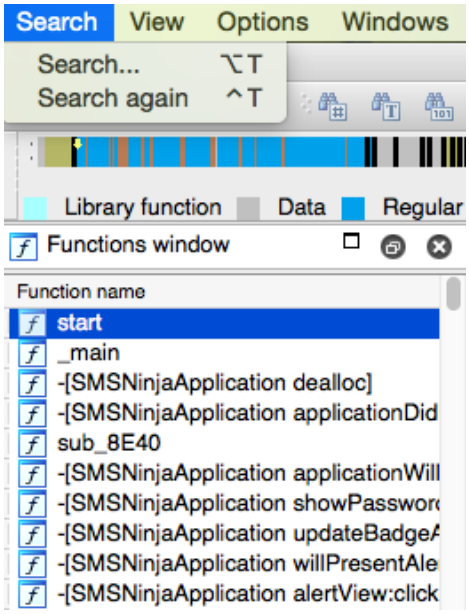


Figure 3-33 Search functions

Choose "Search…", then type in what you want to search as shown in figure 3-34, to search for your specified string in all function names. When the string appears in several function names, you can click "Search again" to go through all of them. Of course, all above operations can be done by shortcuts.

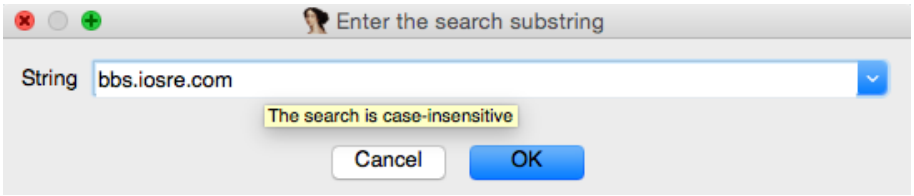


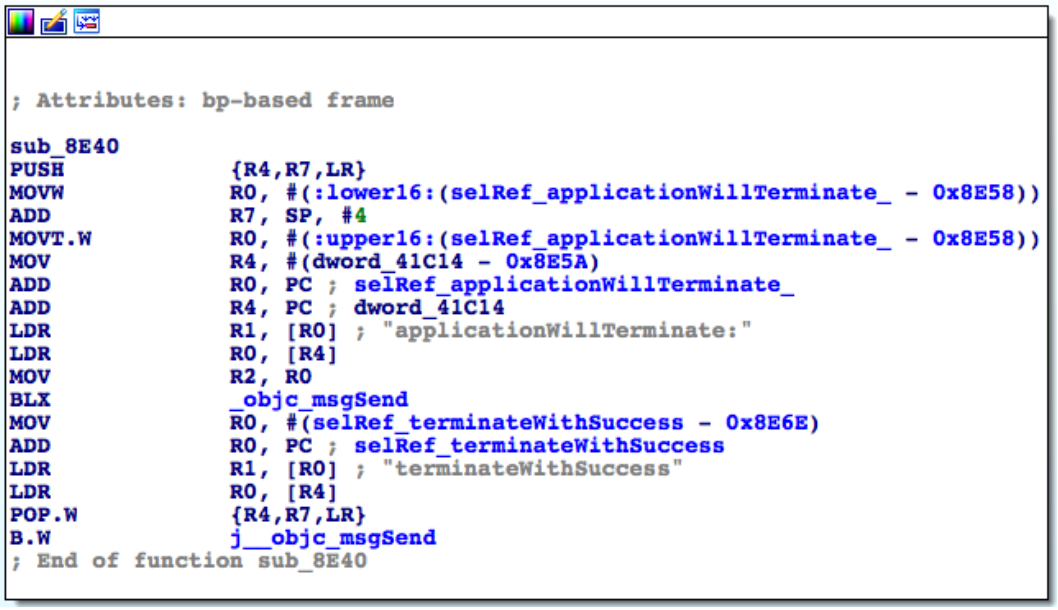
Figure 3-34 Search functions

The method names in functions window are the same as names exported by class-dump. Besides Objective-C methods, IDA lists all subroutines that we cannot get with class-dump. All class-dump contents are method names of Objective-C, it's easy to learn and read for beginners; the names of subroutines are just combinations of "sub_" and addresses, they don't have any literal meaning, hence are hard to learn and read, freaking many rookies out. However, low-level iOS is implemented in C and C++, which generate subroutines rather than Objective-C methods. In this situation, class-dump is entirely defeated, our only choices are tools like IDA. If we want to go deeper into iOS, we must get familiar with IDA.

- Main window

Most iOS developers who have never used IDA before are shocked by the "delirious" contents presented by main window. It seems a real mess for all beginners; some of them may

close IDA immediately, and never open it again. This perplexed feeling is similar to the first time when you write code. In fact, it is like every project needs a main function, in iOS reverse engineering, we also need to specify the entry function that we are interested in. Double click this entry function in function window, main window will show the function body, then select main window and press space key, the main window will become much clearer and more readable as shown in figure 3-35.



```
; Attributes: bp-based frame

sub_8E40
PUSH            {R4,R7,LR}
MOVW            R0, #(:lower16:(selRef_applicationWillTerminate_ - 0x8E58))
ADD             R7, SP, #4
MOVT.W          R0, #(:upper16:(selRef_applicationWillTerminate_ - 0x8E58))
MOV             R4, #(dword_41C14 - 0x8E5A)
ADD             R0, PC ; selRef_applicationWillTerminate_
ADD             R4, PC ; dword_41C14
LDR             R1, [R0] ; "applicationWillTerminate:"
LDR             R0, [R4]
MOV             R2, R0
BLX             _objc_msgSend
MOV             R0, #(selRef_terminateWithSuccess - 0x8E6E)
ADD             R0, PC ; selRef_terminateWithSuccess
LDR             R1, [R0] ; "terminateWithSuccess"
LDR             R0, [R4]
POP.W           {R4,R7,LR}
B.W             j__objc_msgSend
; End of function sub_8E40
```

Figure 3- 35 Graph view

There are 2 display modes in main window, i.e. graph view and text view, which can be switched by space key. Graph view focuses on the logics; you can use control button and mouse wheel on it to zoom in and out. Graph view provides intuitive visualization of the relationship among different subroutines. Execution flows of different subroutines are presented by lines with arrows. When there's a conditional branch, subroutine that meets the condition will be connected with green line, otherwise with red line; for an unconditional branch, the next subroutine will be connected with blue line. For example, in figure 3-36, when the execution flow comes to the end of loc_1C758, it judges whether R0 is equal to 0, if R0 != 0, the condition of BNE is satisfied, it will branch to the right, otherwise it will branch to the left. This is one difficult point of IDA; it will be explained again and again in the following examples.

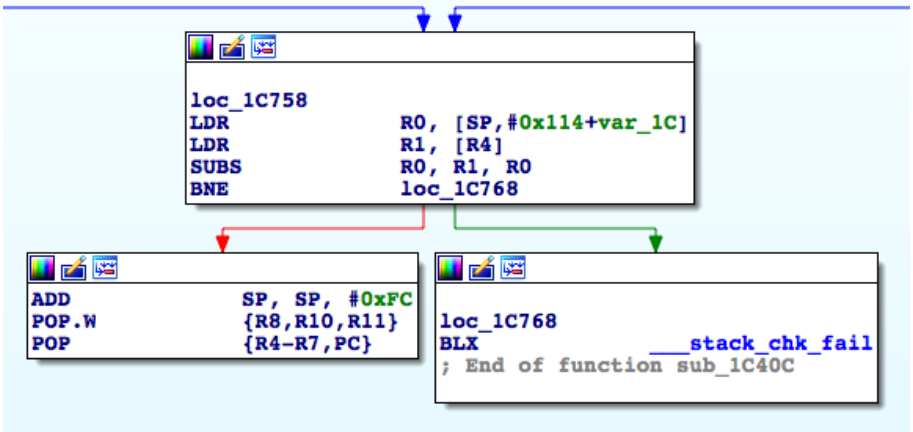


Figure 3- 36 Branches in IDA

Careful readers may have noticed that the fonts of IDA are colorful. In fact, different colors have different meanings, as shown in figure 3-37.

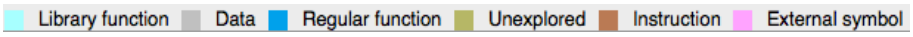


Figure 3-37 Color indication bar

When we choose a symbol, all the same symbols will be highlighted in yellow, making it convenient for us to track this symbol, as shown in figure 3-38.

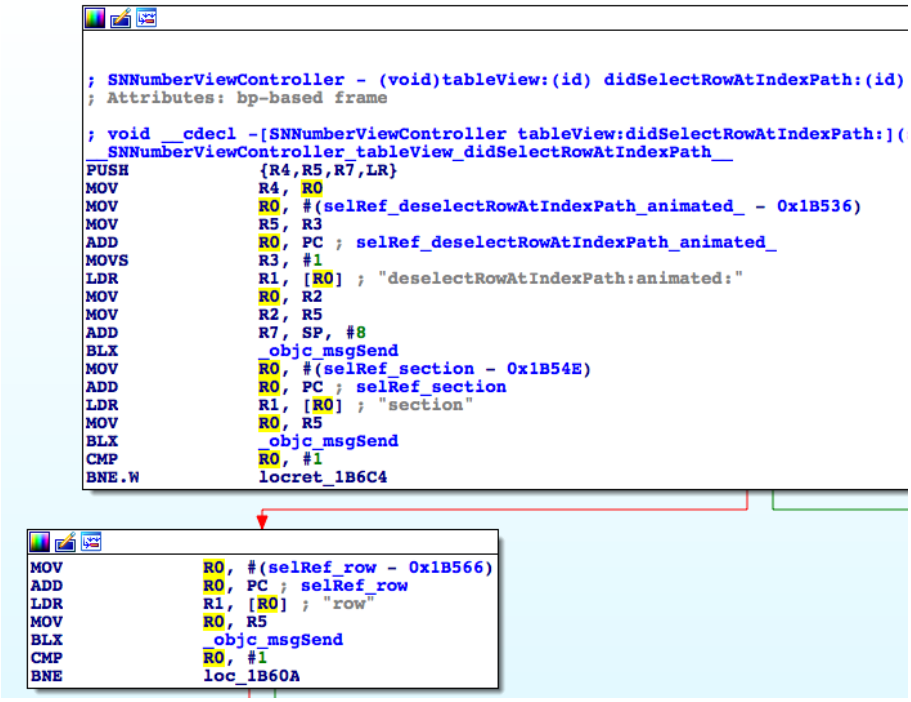


Figure 3-38 Symbol highlight

Double click a symbol to see its implementation as shown in figure 3-35. Right click a symbol to display a menu shown in figure 3-39.

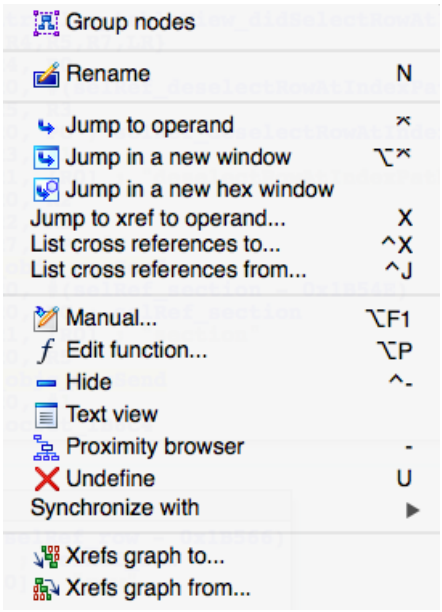


Figure 3-39 Right click on a symbol

Among the menu options, there is a very frequently used function "Jump to xref to operand…" with the shortcut X (meaning "cross"), click this option, all information explicitly cross referenced to this symbol will be displayed as shown in figure 3-40.

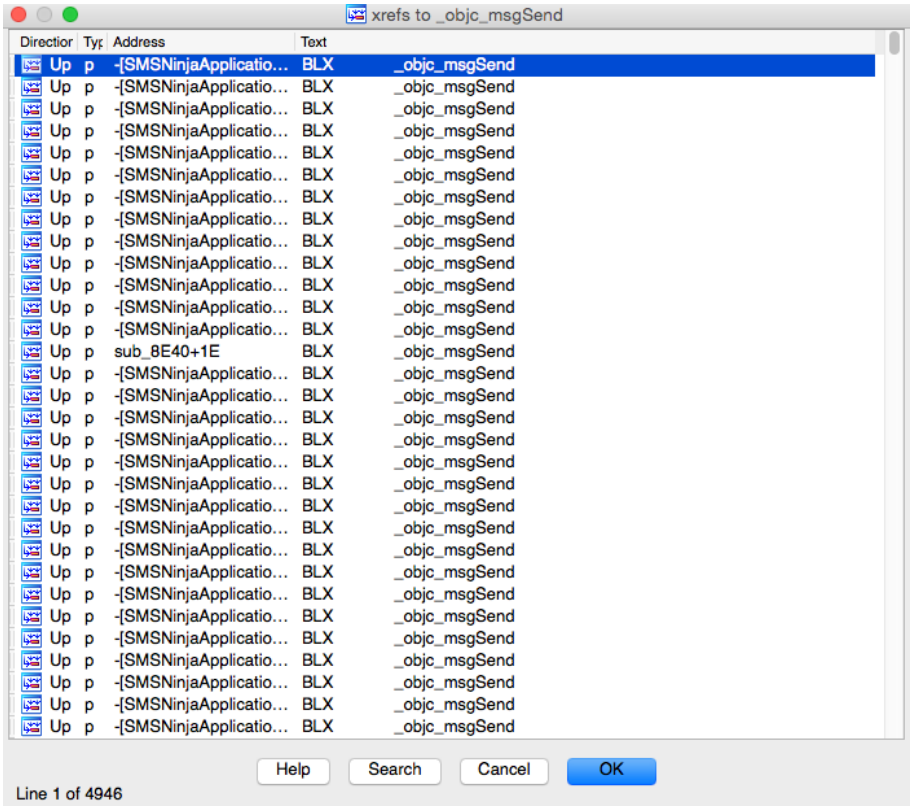


Figure 3- 40 Jump to xref to operand...

If you think this way is not straightforward and clear enough, yet prefer graph view, you can choose option "Xrefs graph to…". However, if this symbol is cross-referenced too much, the

graph view becomes a mess, just like figure 3-41 shows.

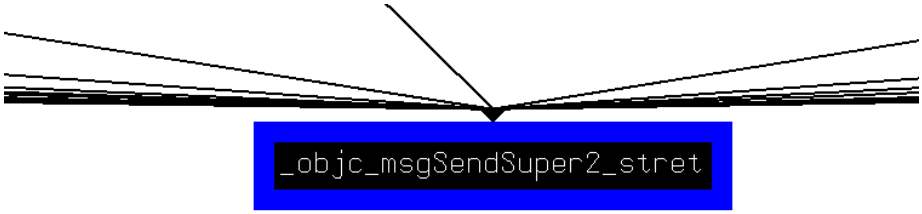


Figure 3-41 Xrefs graph to…

In figure 3-41, the irregular patterns in black are constructed by lines; lines are melting together on both sides. So we know the symbol _objc_msgSendSuper2_stret is cross-referenced many times.

Relatively, if we choose "Xrefs graph from..." , it will show all symbols cross referenced by the symbol you choose, as shown in figure 3-42.

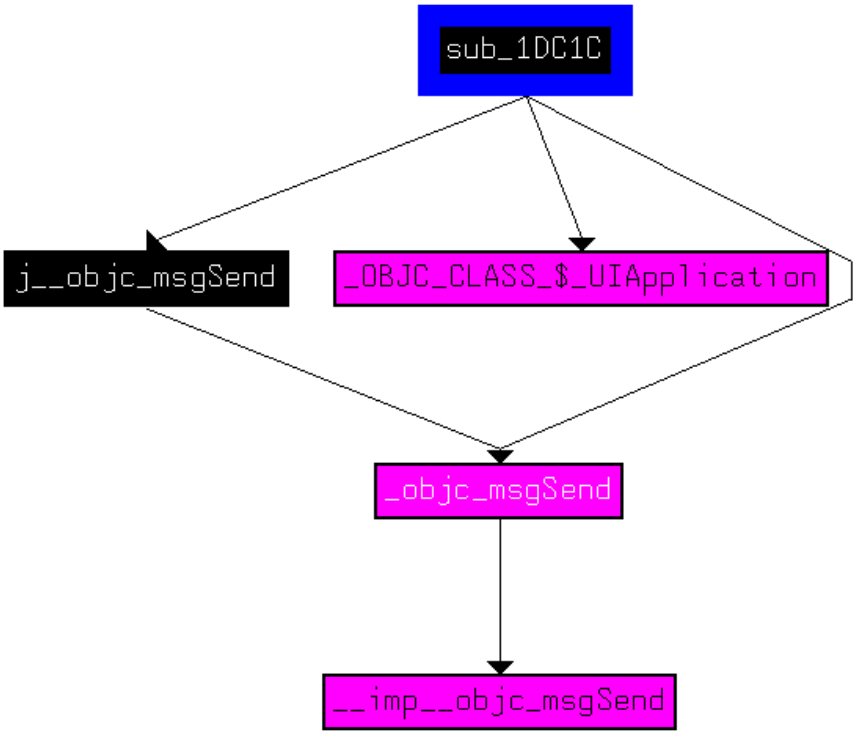


Figure 3-42 Xrefs graph from...

From figure 3-42 we know that sub_1DC1C is a subroutine, it cross-references j__objc_msgSend, _OBJC_CLASS_$_UIApplication and _objc_msgSend explicitly, and _objc_msgSend further cross-references __imp__objc_msgSend explicitly. Double click _objc_msgSend in main window, then double click __imp__objc_msgSend, you will see it is from libobjc.A.dylib, as shown in figure 3-43.

```
Imports from /usr/lib/libobjc.A.dylib
================================================================================
Segment type: Externs
                IMPORT ___objc_personality_v0
                                        ; DATA XREF: -[SNBlacklistViewController acti
                                        ; __text:0000D3DC↑o ...
                IMPORT __objc_empty_cache
                                        ; DATA XREF: __objc_data:_OBJC_CLASS_$_SMSNin
                                        ; __objc_data:_OBJC_METACLASS_$_SMSNinjaAppli
                IMPORT __imp__objc_autoreleasePoolPop
                                        ; CODE XREF: _objc_autoreleasePoolPop↑j
                                        ; DATA XREF: _objc_autoreleasePoolPop↑o ...
                IMPORT __imp__objc_autoreleasePoolPush
                                        ; CODE XREF: _objc_autoreleasePoolPush↑j
                                        ; DATA XREF: _objc_autoreleasePoolPush↑o ...
                IMPORT __imp__objc_enumerationMutation
                                        ; CODE XREF: _objc_enumerationMutation↑j
                                        ; DATA XREF: _objc_enumerationMutation↑o ...
                IMPORT __imp__objc_getClass ; CODE XREF: _objc_getClass↑j
                                        ; DATA XREF: _objc_getClass↑o ...
                IMPORT __imp__objc_msgSend ; CODE XREF: _objc_msgSend↑j
                                        ; DATA XREF: _objc_msgSend↑o ...
                IMPORT __imp__objc_msgSendSuper2
                                        ; CODE XREF: _objc_msgSendSuper2↑j
                                        ; DATA XREF: _objc_msgSendSuper2↑o ...
                IMPORT __imp__objc_msgSend_stret
                                        ; CODE XREF: _objc_msgSend_stret↑j
                                        ; DATA XREF: _objc_msgSend_stret↑o ...
                IMPORT __imp__objc_setProperty
                                        ; CODE XREF: _objc_setProperty↑j
                                        ; DATA XREF: _objc_setProperty↑o ...
```

Figure 3-43 Tracking the source of external symbols

In most cases, when we discover an interesting symbol, we want to find every related clue. One clumsy but effective way is to select main window and click "Search" on the menu bar. A submenu is shown like figure 3-44.

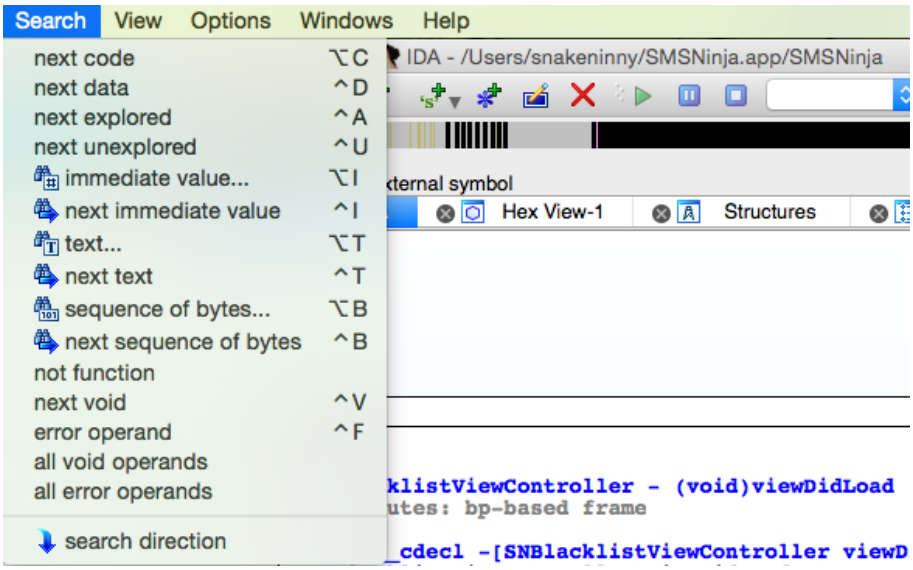


Figure 3-44 Search in Main window

Choose "text…", a window will popup, as shown in figure 3-45.

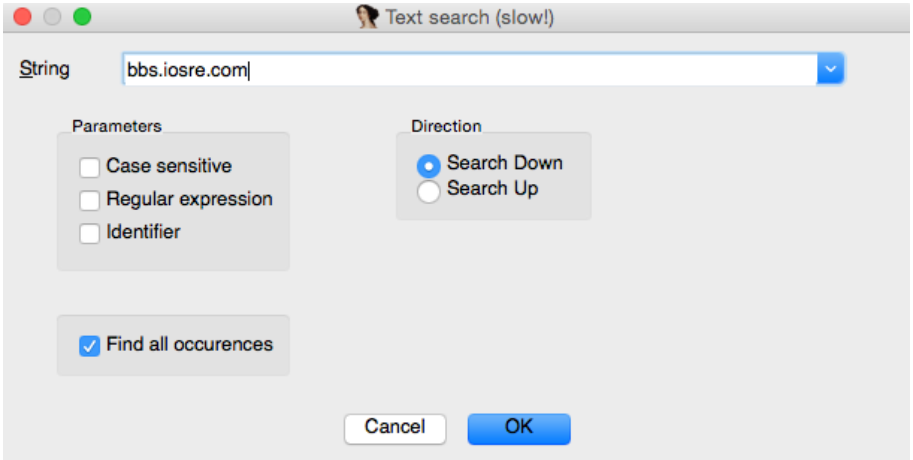


Figure 3-45 Text search

There're other searching options available, you can check them out according to your situations. Then check "Find all occurences" and click "OK". IDA will search the whole binary and show all the matching strings.

Graph view provides us with so many features; I've only introduced some common ones, proficiency in them ensures deeper research. Graph view is simple and clear, it's easy to see the logics between different subroutines. As newbies, we mostly use graph view. When using LLDB for debugging, we'll switch to text view to get the address of a symbol listed on the left side, as shown in figure 3-46.

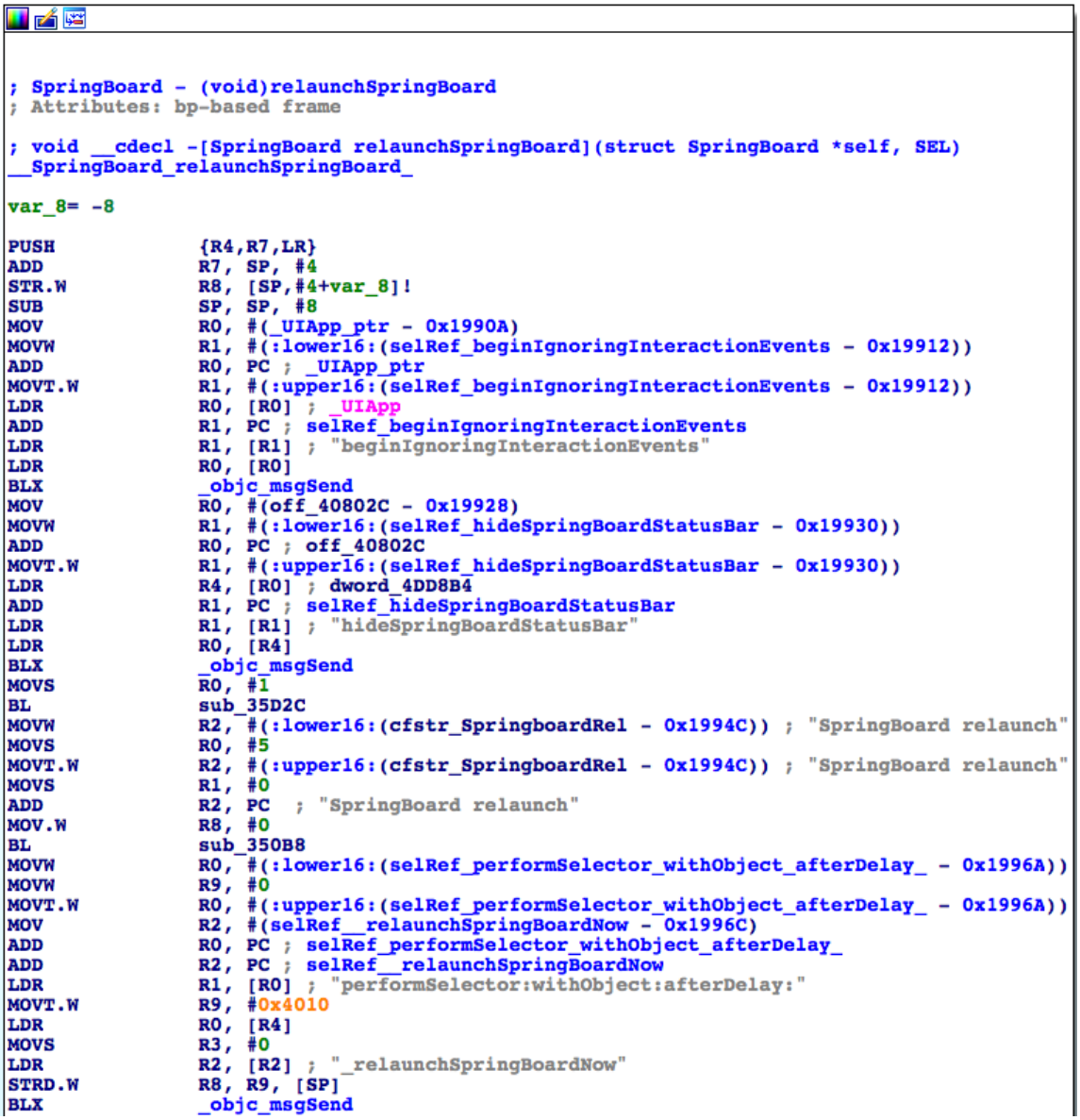
```
__text:00009D94    MOV     R1, #(classRef_SNBlacklistViewController_0 - 0x9DA2)
__text:00009D9C    STR     R0, [SP,#0x34+var_20]
__text:00009D9E    ADD     R1, PC ; classRef_SNBlacklistViewController_0
__text:00009DA0    LDR     R0, [R1] ; _OBJC_CLASS_$_SNBlacklistViewController
__text:00009DA2    MOV     R1, #(selRef_viewDidLoad - 0x9DAE)
__text:00009DAA    ADD     R1, PC ; selRef_viewDidLoad
__text:00009DAC    LDR     R1, [R1] ; "viewDidLoad"
__text:00009DAE    STR     R0, [SP,#0x34+var_1C]
__text:00009DB0    ADD     R0, SP, #0x34+var_20
__text:00009DB2    BLX     _objc_msgSendSuper2
__text:00009DB6    MOV     R0, #(selRef_alloc - 0x9DCA)
__text:00009DBE    MOV     R2, #(classRef_UISegmentedControl - 0x9DCC)
__text:00009DC6    ADD     R0, PC ; selRef_alloc
__text:00009DC8    ADD     R2, PC ; classRef_UISegmentedControl
__text:00009DCA    LDR     R1, [R0] ; "alloc"
__text:00009DCC    LDR     R0, [R2] ; _OBJC_CLASS_$_UISegmentedControl
__text:00009DCE    BLX     _objc_msgSend
```

Figure 3-46 Text view

It should be noted that one bug of IDA will cause the incomplete display of a subroutine at the end of its graph view (For example, one subroutine has 100 lines of instructions but only displays 80 lines). When you are suspicious about instructions in graph view, just switch to text view to see whether some code is missing. This bug occurs by very little chance, if you happen to encounter it unfortunately, welcome to http://bbs.iosre.com for discussion and solution.

3.4.3 An analysis example of IDA

Having introduced so many features of IDA, now I will use a simple example to show the real power of IDA. Jailbreak users know, Cydia will suggest us "Restart SpringBoard" when a tweak finishes installation. How does Cydia perform a respring? Please go through section 3.5 quickly and copy "/System/Library/CoreServices/SpringBoard.app/SpringBoard" from iOS to OSX using iFunBox, then open it with IDA. When the initial analysis is finished, search "relaunchSpringBoard" in function window, double click it to jump to its function body, as shown in figure 3-47.



```
; SpringBoard - (void)relaunchSpringBoard
; Attributes: bp-based frame

; void __cdecl -[SpringBoard relaunchSpringBoard](struct SpringBoard *self, SEL)
__SpringBoard_relaunchSpringBoard_

var_8= -8

PUSH            {R4,R7,LR}
ADD             R7, SP, #4
STR.W           R8, [SP,#4+var_8]!
SUB             SP, SP, #8
MOV             R0, #(_UIApp_ptr - 0x1990A)
MOVW            R1, #(:lower16:(selRef_beginIgnoringInteractionEvents - 0x19912))
ADD             R0, PC ; _UIApp_ptr
MOVT.W          R1, #(:upper16:(selRef_beginIgnoringInteractionEvents - 0x19912))
LDR             R0, [R0] ; _UIApp
ADD             R1, PC ; selRef_beginIgnoringInteractionEvents
LDR             R1, [R1] ; "beginIgnoringInteractionEvents"
LDR             R0, [R0]
BLX             _objc_msgSend
MOV             R0, #(off_40802C - 0x19928)
MOVW            R1, #(:lower16:(selRef_hideSpringBoardStatusBar - 0x19930))
ADD             R0, PC ; off_40802C
MOVT.W          R1, #(:upper16:(selRef_hideSpringBoardStatusBar - 0x19930))
LDR             R4, [R0] ; dword_4DD8B4
ADD             R1, PC ; selRef_hideSpringBoardStatusBar
LDR             R1, [R1] ; "hideSpringBoardStatusBar"
LDR             R0, [R4]
BLX             _objc_msgSend
MOVS            R0, #1
BL              sub_35D2C
MOVW            R2, #(:lower16:(cfstr_SpringboardRel - 0x1994C)) ; "SpringBoard relaunch"
MOVS            R0, #5
MOVT.W          R2, #(:upper16:(cfstr_SpringboardRel - 0x1994C)) ; "SpringBoard relaunch"
MOVS            R1, #0
ADD             R2, PC  ; "SpringBoard relaunch"
MOV.W           R8, #0
BL              sub_350B8
MOVW            R0, #(:lower16:(selRef_performSelector_withObject_afterDelay_ - 0x1996A))
MOVW            R9, #0
MOVT.W          R0, #(:upper16:(selRef_performSelector_withObject_afterDelay_ - 0x1996A))
MOV             R2, #(selRef__relaunchSpringBoardNow - 0x1996C)
ADD             R0, PC ; selRef_performSelector_withObject_afterDelay_
ADD             R2, PC ; selRef__relaunchSpringBoardNow
LDR             R1, [R0] ; "performSelector:withObject:afterDelay:"
MOVT.W          R9, #0x4010
LDR             R0, [R4]
MOVS            R3, #0
LDR             R2, [R2] ; "_relaunchSpringBoardNow"
STRD.W          R8, R9, [SP]
BLX             _objc_msgSend
```

Figure 3- 47 [SpringBoard relaunchSpringBoard]

As we can see in figure 3-47, this method's implementation is simple and clear. According to the execution flow from top to bottom, firstly it calls beginIgnoringInteractionEvents to ignore

all user interaction events; secondly, it calls hideSpringBoardStatusBar to hide the status bar in SpringBoard, then it executes two subroutines, they are sub_35D2C and sub_350B8. Now, double click sub_35D2C to jump to its implementation, as shown in figure 3-48.

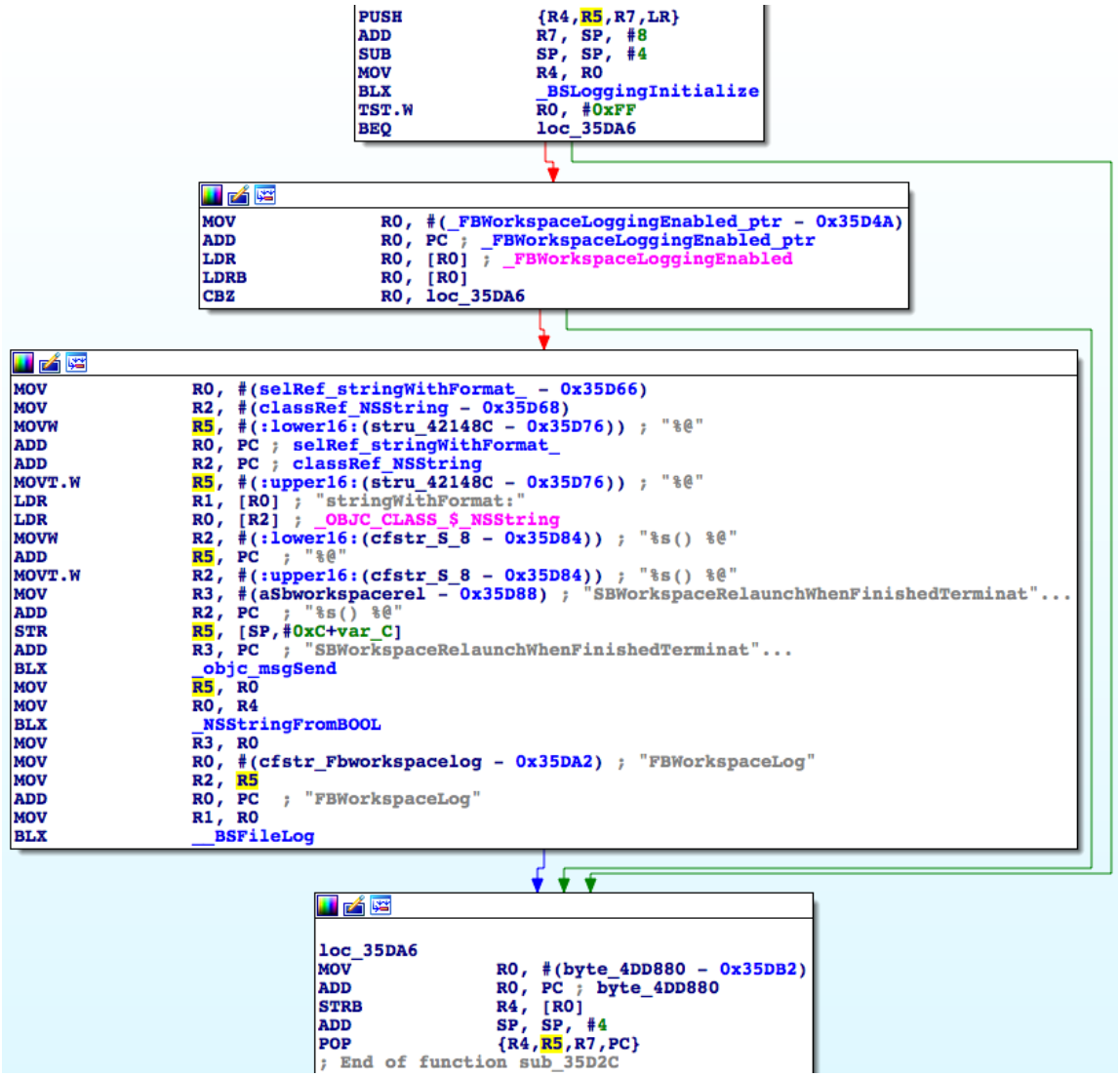


Figure 3- 48 sub_35D2C

In figure 3-48, "log" appears a lot: First "initialize", then check whether something is "enabled", at last "log" something. From those keywords, we can guess that this subroutine is used for logging respring related operations, it has nothing to do with the essential function of respring. Click the blue back button of IDA menu bar (as shown in figure 3-49), or just press ESC, to go back to the implementation of "relaunchSpringBoard" and continue our analysis.

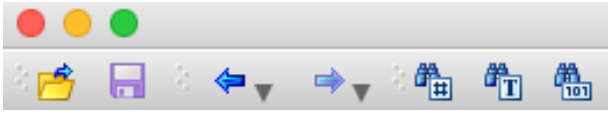


Figure 3-49 Back button

Double click sub_350B8 to jump to figure 3-50.

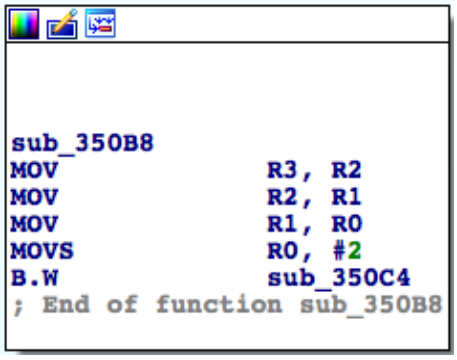
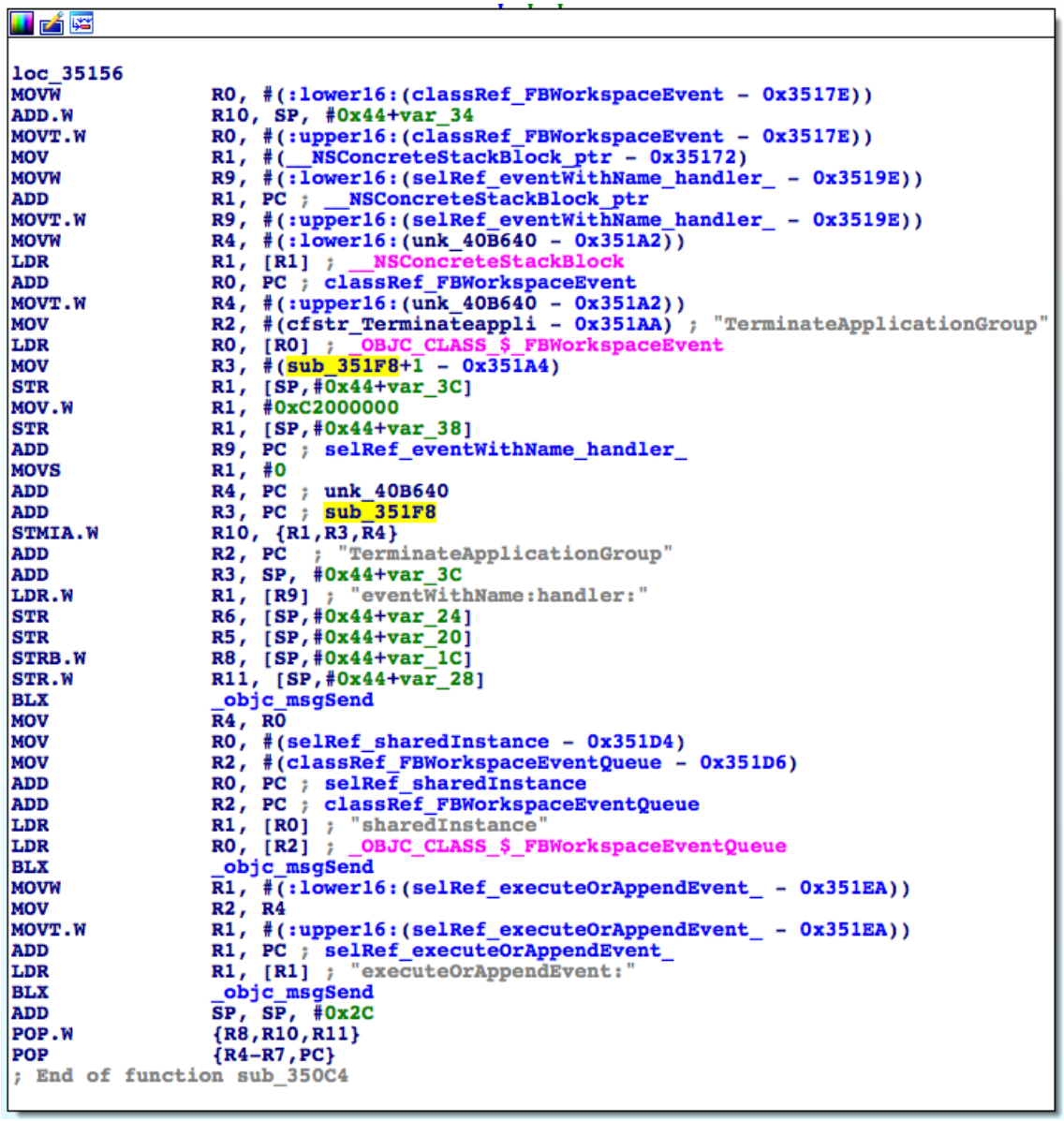


Figure 3- 50 sub_350B8

We know from figure 3-50 that this subroutine is just preparing for calling sub_350C4. Double click sub_350C4 to jump to its implementation, you will find the top half of sub_350C4 looks very similar to sub_35D2C as shown in figure 3-48, which only does some logging job. But what's different is that sub_350C4 additionally does something essential, as shown in figure 3-51.



```
loc_35156
MOVW            R0, #(:lower16:(classRef_FBWorkspaceEvent - 0x3517E))
ADD.W           R10, SP, #0x44+var_34
MOVT.W          R0, #(:upper16:(classRef_FBWorkspaceEvent - 0x3517E))
MOV             R1, #(__NSConcreteStackBlock_ptr - 0x35172)
MOVW            R9, #(:lower16:(selRef_eventWithName_handler_ - 0x3519E))
ADD             R1, PC ; __NSConcreteStackBlock_ptr
MOVT.W          R9, #(:upper16:(selRef_eventWithName_handler_ - 0x3519E))
MOVW            R4, #(:lower16:(unk_40B640 - 0x351A2))
LDR             R1, [R1] ; __NSConcreteStackBlock
ADD             R0, PC ; classRef_FBWorkspaceEvent
MOVT.W          R4, #(:upper16:(unk_40B640 - 0x351A2))
MOV             R2, #(cfstr_Terminateappli - 0x351AA) ; "TerminateApplicationGroup"
LDR             R0, [R0] ; _OBJC_CLASS_$_FBWorkspaceEvent
MOV             R3, #(sub_351F8+1 - 0x351A4)
STR             R1, [SP,#0x44+var_3C]
MOV.W           R1, #0xC2000000
STR             R1, [SP,#0x44+var_38]
ADD             R9, PC ; selRef_eventWithName_handler_
MOVS            R1, #0
ADD             R4, PC ; unk_40B640
ADD             R3, PC ; sub_351F8
STMIA.W         R10, {R1,R3,R4}
ADD             R2, PC  ; "TerminateApplicationGroup"
ADD             R3, SP, #0x44+var_3C
LDR.W           R1, [R9] ; "eventWithName:handler:"
STR             R6, [SP,#0x44+var_24]
STR             R5, [SP,#0x44+var_20]
STRB.W          R8, [SP,#0x44+var_1C]
STR.W           R11, [SP,#0x44+var_28]
BLX             _objc_msgSend
MOV             R4, R0
MOV             R0, #(selRef_sharedInstance - 0x351D4)
MOV             R2, #(classRef_FBWorkspaceEventQueue - 0x351D6)
ADD             R0, PC ; selRef_sharedInstance
ADD             R2, PC ; classRef_FBWorkspaceEventQueue
LDR             R1, [R0] ; "sharedInstance"
LDR             R0, [R2] ; _OBJC_CLASS_$_FBWorkspaceEventQueue
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef_executeOrAppendEvent_ - 0x351EA))
MOV             R2, R4
MOVT.W          R1, #(:upper16:(selRef_executeOrAppendEvent_ - 0x351EA))
ADD             R1, PC ; selRef_executeOrAppendEvent_
LDR             R1, [R1] ; "executeOrAppendEvent:"
BLX             _objc_msgSend
ADD             SP, SP, #0x2C
POP.W           {R8,R10,R11}
POP             {R4-R7,PC}
; End of function sub_350C4
```

Figure 3-51 sub_350C4

Now that we know little about assembly language, but from the literal meaning of these keywords, it can be concluded that the function of this subroutine is to generate an event named "TerminateApplicationGroup", specify sub_351F8 to be the handler of it, and then append this event to a queue for sequential execution, thus close all Apps by this way. This makes sense: Before a mall closes, we need to close all its shops; before respring, we need to close all Apps. Let's go to sub_351F8 to see its implementation, as shown in figure 3-52.

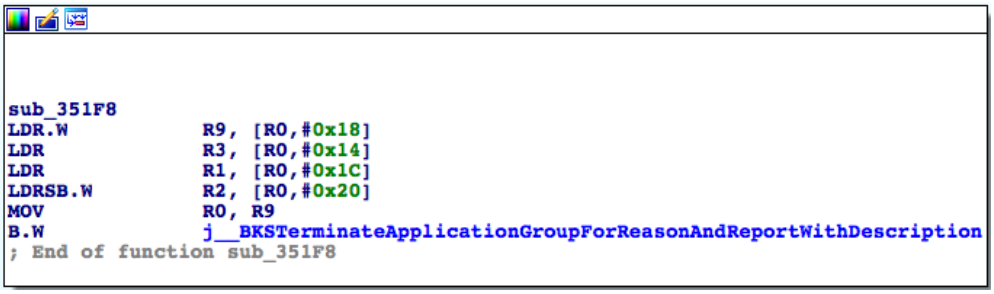


Figure 3-52 sub_351F8

We can infer from the name of BKSTerminateApplicationGroupForReasonAndReportWithDescription that sub_351F8 acts as a terminator, which just proves our analysis of sub_350C4. Go back to the function body of relaunchSpringBoard, our analysis comes to the end: _relaunchSpringBoardNow is called to finish respring.

Neither do we need to read assembly code nor be familiar with calling conventions, we've finished this reverse engineering task from scratch, right? However, we should not take much credits, kudos to IDA! In most cases, IDA plays the same role to the above example; you only need to be patient reading every line of code, it won't be long before you feel the beauty of reverse engineering.

The usage of IDA is much much more complicated than I have introduced in this book, if you have any questions about it, please discuss with us on http://bbs.iosre.com, or take The IDA Pro Book as reference.

3.5 iFunBox

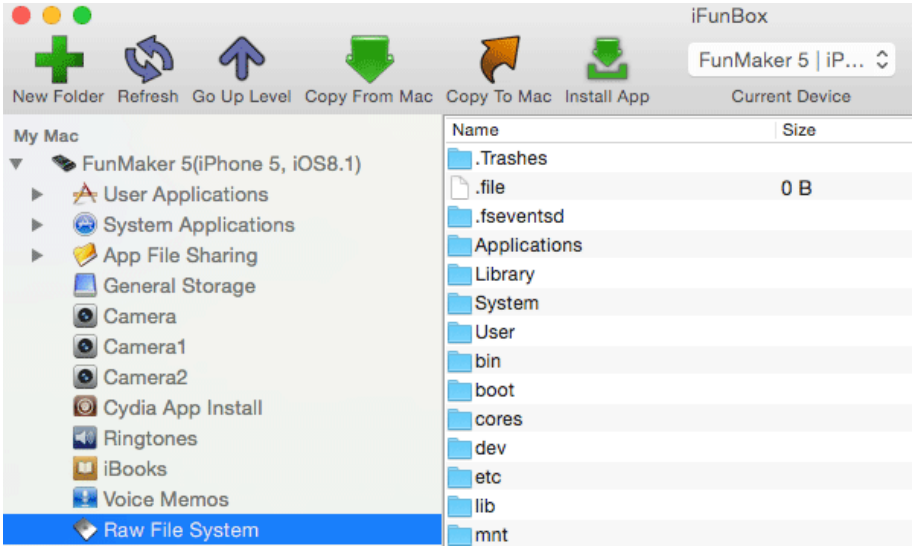


Figure 3-53 iFunBox

iFunBox (as shown in figure 3-53) is an evergreen iOS file management tool on Windows/OSX. In this book, we mainly make use of its file transfer feature. One thing to mention is that we must install "Apple File Conduit 2" (or AFC2 for short, as shown in figure 3-54) on iOS to browse the entire iOS file system, which is the prerequisite of the following operations in this book.

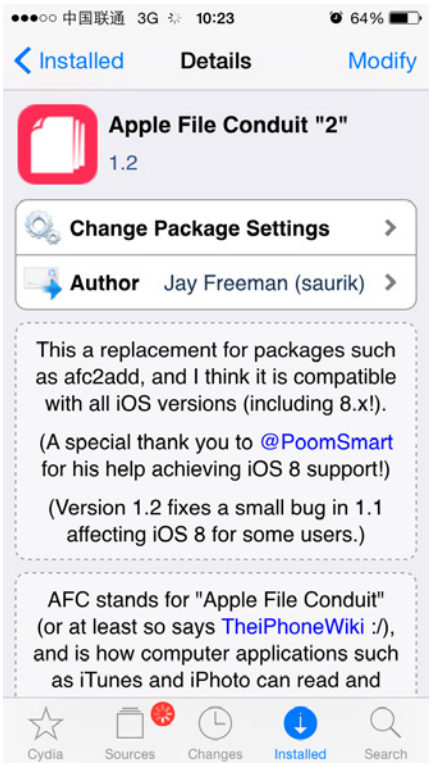


Figure 3-54 Apple File Conduit 2

3.6 dyld_decache

After installing iFunBox and AFC2, most of you would be eager to start browsing the iOS filesystem to explore the secrets hidden in iOS. But soon you'll discover that there are no library files under "/System/Library/Frameworks/" or "/System/Library/PrivateFrameworks/". What's going on?

From iOS 3.1, many library files including frameworks are combined into a big cache, which is located in "/System/Library/Caches/com.apple.dyld/ dyld_shared_cache_armx" (i.e. dyld_shared_cache_armv7, dyld_shared_cache_armv7s or dyld_shared_cache_arm64). We can use dyld_decache by KennyTM to extract the separate binaries from this cache, which guarantees that the files we analyze are right from iOS, avoiding the possibility that static and dynamic analysis targets mismatch each other. More about this cache, please refer to DHowett's blog at http://blog.howett.net/2009/09/cache-or-check/.

Before using dyld_decache, please use iFunBox (not scp) to copy "/System/Library/Caches/com.apple.dyld/dyld_shared_cache_armx" from iOS to OSX, then download dyld_decache from https://github.com/downloads/kennytm/Miscellaneous/dyld_decache[v0.1c].bz2 and grant execute permission to the decompressed executable:

```
snakeninnysiMac:~ snakeninny$ chmod +x /path/to/dyld_decache\[v0.1c\]
```

Then extract binaries from the cache:

```
snakeninnysiMac:~ snakeninny$ /path/to/dyld_decache\[v0.1c\] -o
/where/to/store/decached/binaries/ /path/to/dyld_shared_cache_armx
  0/877: Dumping
'/System/Library/AccessibilityBundles/AXSpeechImplementation.bundle/AXSpeechImple
mentation'...
  1/877: Dumping
'/System/Library/AccessibilityBundles/AccessibilitySettingsLoader.bundle/Accessib
ilitySettingsLoader'...
  2/877: Dumping
'/System/Library/AccessibilityBundles/AccountsUI.axbundle/AccountsUI'...
......
```

All the binaries are extracted into "/where/to/store/decached/binaries/". After that, binaries to be reversed are scattered on both iOS and OSX, which leads to inconvenience. So we suggest you copy iOS filesystem to OSX with scp, a tool to be introduced in the next chapter.

3.7 Conclusion

This chapter focuses on 4 tools, which are class-dump, Theos, Reveal and IDA. Familiarity with them is the prerequisite of iOS reverse engineering.

Please follow **@iOSAppRE** and **http://bbs.iosre.com** for more information!

The following chapters are coming soon!

Please stay tuned!

Thank you!