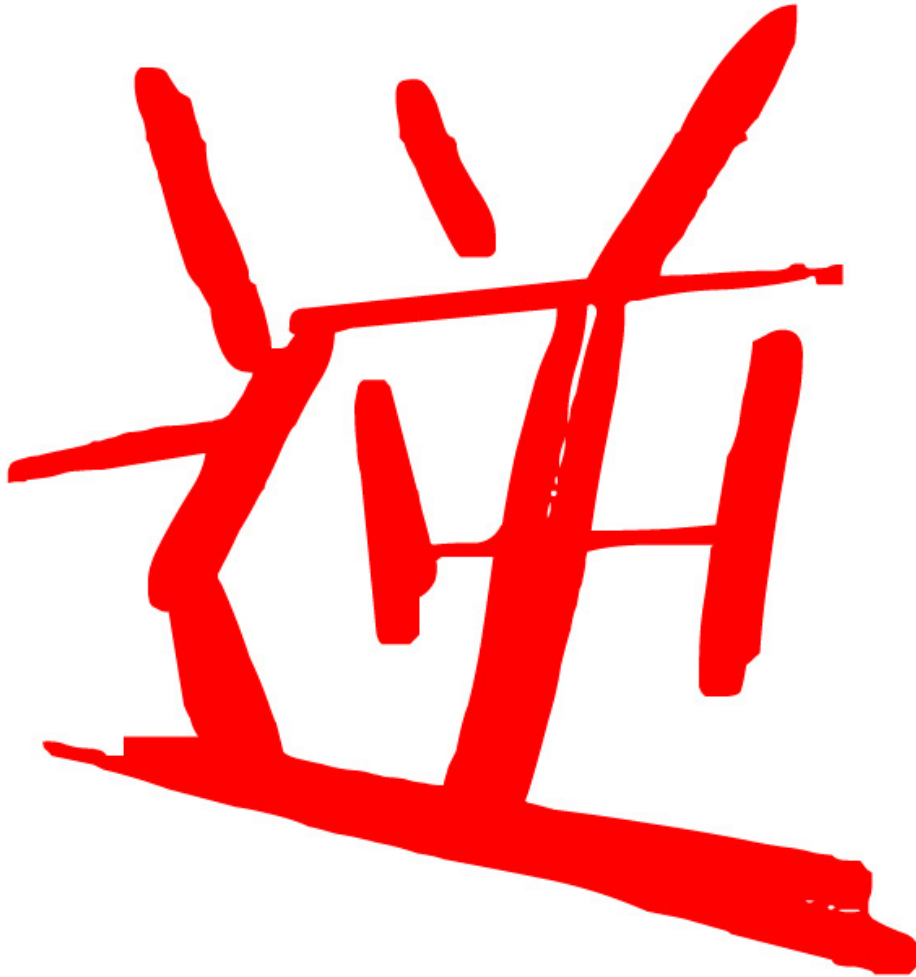


SHAKENINNY, HANGCOM

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



iOS App Reverse Engineering

Table of Contents

Recommendation	1
Preface.....	2
Foreword.....	7
Part 1 Concepts	12
Chapter 1 Introduction to iOS reverse engineering	13
1.1 Prerequisites of iOS reverse engineering.....	13
1.2 What does iOS reverse engineering do	13
1.2.1 Security related iOS reverse engineering.....	16
1.2.2 Development related iOS reverse engineering	17
1.3 The process of iOS reverse engineering.....	19
1.3.1 System Analysis	19
1.3.2 Code Analysis.....	20
1.4 Tools for iOS reverse engineering.....	20
1.4.1 Monitors	21
1.4.2 Disassemblers.....	21
1.4.3 Debuggers	23
1.4.4 Development kit	23
1.5 Conclusion	23
Chapter 2 Introduction to jailbroken iOS.....	24
2.1 iOS System Hierarchy.....	24
2.1.1 iOS filesystem	26
2.1.2 iOS file permission	32
2.2 iOS file types	33
2.2.1 Application.....	33
2.2.2 Dynamic Library.....	37
2.2.3 Daemon	38
2.3 Conclusion	39
Part 2 Tools.....	40
Chapter 3 OSX toolkit	41
3.1 class-dump.....	41
3.2 Theos	43
3.2.1 Introduction to Theos	43
3.2.2 Install and configure Theos.....	44
3.2.3 Use Theos	46
3.2.4 An example tweak	67
3.3 Reveal	70
3.4 IDA	76

3.4.1	<i>Introduction to IDA</i>	76
3.4.2	<i>Use IDA</i>	77
3.4.3	<i>An analysis example of IDA</i>	90
3.5	iFunBox	95
3.6	dyld_decache	96
3.7	Conclusion	97
Chapter 4	iOS toolkit	98
4.1	CydiaSubstrate	98
4.1.1	<i>MobileHooker</i>	98
4.1.2	<i>MobileLoader</i>	109
4.1.3	<i>Safe mode</i>	109
4.2	Cycript	111
4.3	LLDB and debugserver	115
4.3.1	<i>Introduction to LLDB</i>	115
4.3.2	<i>Introduction to debugserver</i>	116
4.3.3	<i>Configure debugserver</i>	116
4.3.4	<i>Process launching and attaching using debugserver</i>	118
4.3.5	<i>Use LLDB</i>	119
4.3.6	<i>Miscellaneous LLDB</i>	133
4.4	dumpdecrypted	134
4.5	OpenSSH	137
4.6	usbmuxd	138
4.7	iFile	140
4.8	MTerminal	141
4.9	syslogd to /var/log/syslog	142
4.10	Conclusion	142
Part 3	Theories	143
Chapter 5	Objective-C related iOS reverse engineering	144
5.1	How does a tweak work in Objective-C	144
5.2	Methodology of writing a tweak	147
5.2.1	<i>Look for inspiration</i>	147
5.2.2	<i>Locate target files</i>	150
5.2.3	<i>Locate target functions</i>	156
5.2.4	<i>Test private methods</i>	158
5.2.5	<i>Analyze method arguments</i>	160
5.2.6	<i>Limitations of class-dump</i>	162
5.3	An example tweak using the methodology	163
5.3.1	<i>Get inspiration</i>	164
5.3.2	<i>Locate files</i>	165
5.3.3	<i>Locate methods and functions</i>	172
5.3.4	<i>Test methods and functions</i>	174
5.3.5	<i>Write tweak</i>	175
5.4	Conclusion	176
Chapter 6	ARM related iOS reverse engineering	178
6.1	Introduction to ARM assembly	178
6.1.1	<i>Basic concepts</i>	179
6.1.2	<i>Interpretation of ARM/THUMB instructions</i>	184
6.1.3	<i>ARM calling conventions</i>	191
6.2	Advanced methodology of writing a tweak	193

6.2.1	<i>Cut into the target App and find the UI function</i>	195
6.2.2	<i>Locate the target function from the UI function</i>	207
6.3	Advanced LLDB usage	241
6.3.1	<i>Look for a function's caller</i>	241
6.3.2	<i>Change process execution flow</i>	247
6.4	Conclusion	249
Part 4 Practices		250
Chapter 7 Practice 1: Characount for Notes 8		251
7.1	Notes.....	251
7.2	Tweak prototyping.....	252
7.2.1	<i>Locate Notes' executable</i>	255
7.2.2	<i>class-dump MobileNotes' headers</i>	256
7.2.3	<i>Find the controller of note browsing view using Cycrypt</i>	257
7.2.4	<i>Get the current note object from NoteDisplayController</i>	258
7.2.5	<i>Find a method to monitor note text changes in real time</i>	261
7.3	Result interpretation	265
7.4	Tweak writing	266
7.4.1	<i>Create tweak project "CharacountforNotes8" using Theos</i>	266
7.4.2	<i>Compose CharacountForNotes8.h</i>	266
7.4.3	<i>Edit Tweak.xml</i>	267
7.4.4	<i>Edit Makefile and control files</i>	267
7.4.5	<i>Test</i>	268
7.5	Conclusion	272
Chapter 8 Practice 2: Mark user specific emails as read automatically		273
8.1	Mail	273
8.2	Tweak prototyping.....	274
8.2.1	<i>Locate and class-dump Mail's executable</i>	278
8.2.2	<i>Import headers into Xcode</i>	279
8.2.3	<i>Find the controller of "Mailboxes" view using Cycrypt</i>	280
8.2.4	<i>Find the delegate of "All Inboxes" view using Reveal and Cycrypt</i>	282
8.2.5	<i>Locate the refresh completion callback method in MailboxContentViewController</i>	284
8.2.6	<i>Get all emails from MessageMegaMall</i>	288
8.2.7	<i>Get sender address from MFLibraryMessage and mark email as read using MessageMegaMall</i>	290
8.3	Result interpretation	295
8.4	Tweak writing	296
8.4.1	<i>Create tweak project "iOSREMailMarker" using Theos</i>	296
8.4.2	<i>Compose iOSREMailMarker.h</i>	297
8.4.3	<i>Edit Tweak.xml</i>	297
8.4.4	<i>Edit Makefile and control files</i>	298
8.4.5	<i>Test</i>	299
8.5	Conclusion	301
Chapter 9 Practice 3: Save and share Sight in WeChat		302
9.1	WeChat	302
9.2	Tweak prototyping.....	304
9.2.1	<i>Observe Sight view and look for cut-in points</i>	304
9.2.2	<i>Get WeChat headers using class-dump</i>	305
9.2.3	<i>Import WeChat headers into Xcode</i>	306
9.2.4	<i>Locate the Sight view using Reveal</i>	307
9.2.5	<i>Find the long press action selector</i>	308

9.2.6	Find the controller of Sight view using Cycrypt.....	314
9.2.7	Find the Sight object in WCTimeLineViewController.....	316
9.2.8	Get a WCDataItem object from WCContentItemViewTemplateNewSight	321
9.2.9	Get target information from WCDataItem	324
9.3	Result interpretation	333
9.4	Tweak writing	333
9.4.1	Create tweak project “iOSREWVideoDownloader” using Theos.....	333
9.4.2	Compose iOSREWVideoDownloader.h.....	334
9.4.3	Edit Tweak.xml.....	335
9.4.4	Edit Makefile and control files	336
9.4.5	Test	337
9.5	Easter eggs.....	339
9.5.1	Find the Sight in UIMenuItem	339
9.5.2	Historical transition of WeChat’s headers count	340
9.6	Conclusion	343
Chapter 10	Practice 4: Detect And Send iMessages	345
10.1	iMessage	345
10.2	Detect if a number or email address supports iMessage	345
10.2.1	Observe MobileSMS and look for cut-in points.....	345
10.2.2	Find placeholder using Cycrypt.....	348
10.2.3	Find the 1st data source of placeholderText using IDA and LLDB.....	356
10.2.4	Find the Nth data source of placeholderText using IDA and LLDB.....	359
10.2.5	Restore the process of the original data source becoming placeholderText	390
10.3	Send iMessages.....	391
10.3.1	Observe MobileSMS and look for cut-in points.....	391
10.3.2	Find response method of “Send” button using Cycrypt	393
10.3.3	Find suspicious sending action in response method	394
10.4	Result Interpretation	422
10.5	Tweak writing	424
10.5.1	Create tweak project “iOSREMadridMessenger” using Theos.....	424
10.5.2	Compose iOSREMadridMessenger.h.....	425
10.5.3	Edit Tweak.xml.....	425
10.5.4	Edit Makefile and control files	426
10.5.5	Test with Cycrypt	427
10.6	Conclusion	427
	Jailbreaking for Developers, An Overview	429
	Evading the Sandbox	432
	Tweaking is the new-age hacking.....	434

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 conducive 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 technically 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, Codyd51, 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 royalty 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 technically 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

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.

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 become 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lcy	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.

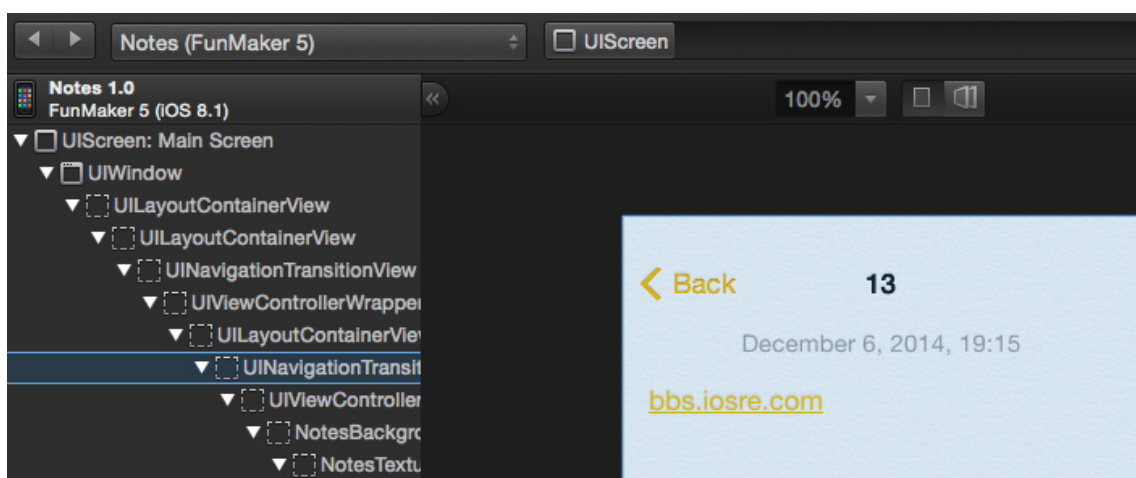


Figure 1- 4 Reveal

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!

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 referring 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.

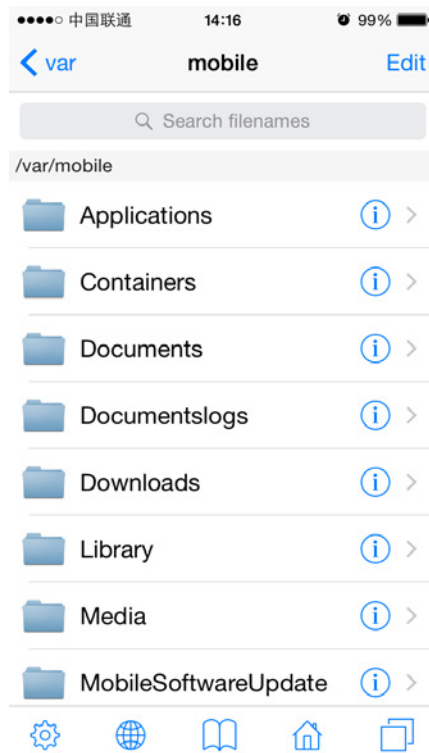


Figure 2- 1 iFile

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.

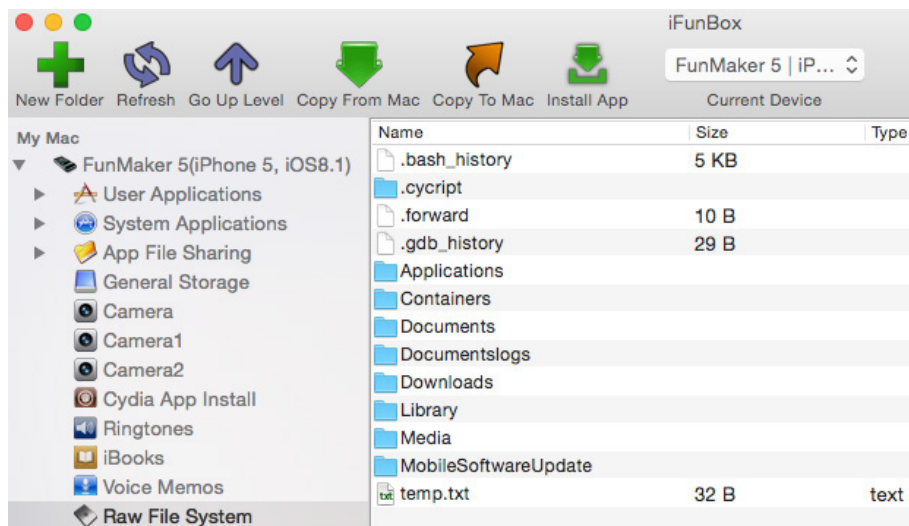


Figure 2- 2 iFunBox

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.

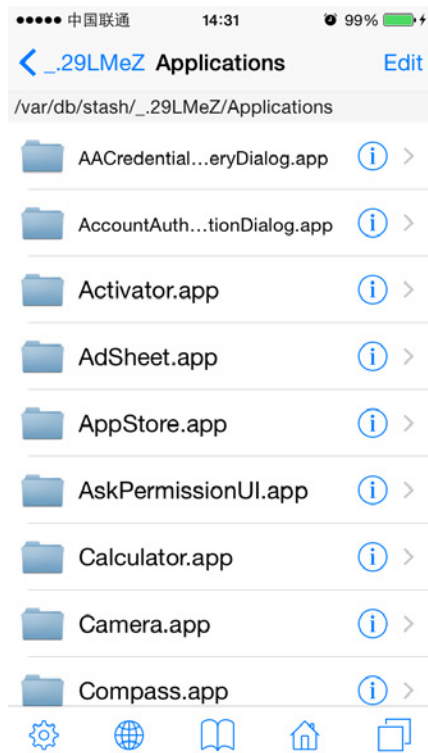


Figure 2- 3 /Applications

- /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.

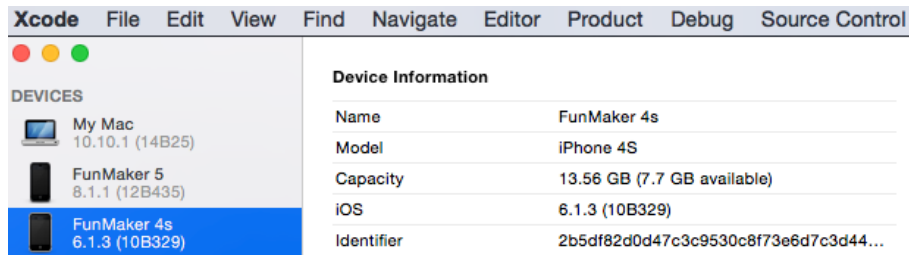


Figure 2- 4 Enable debugging on device

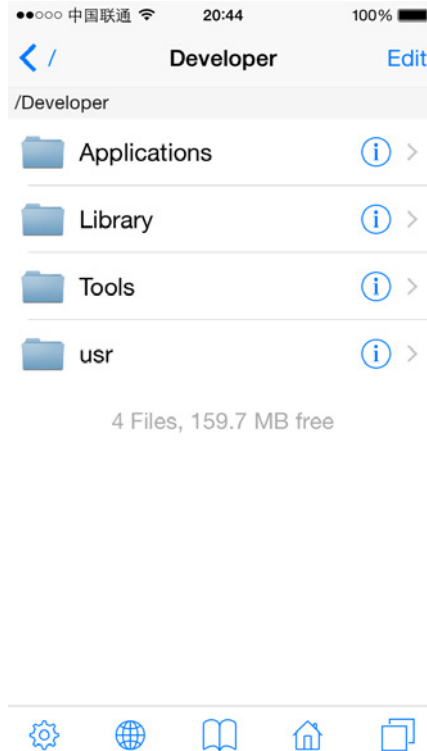


Figure 2- 5 /Developer

- /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.

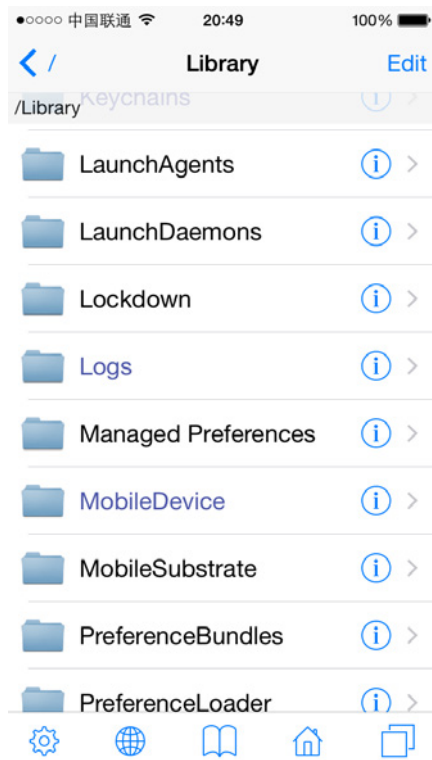


Figure 2- 6 /Library

- /System/Library

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

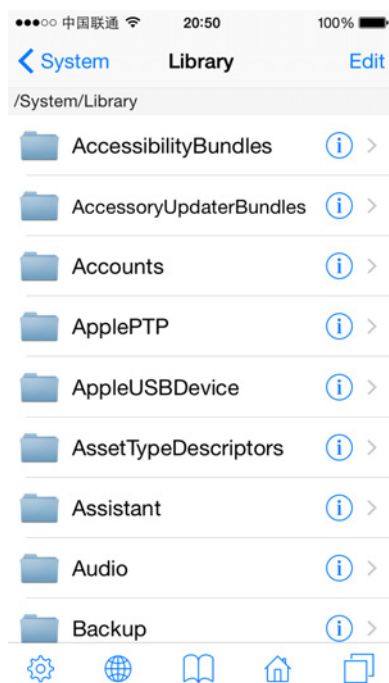


Figure2- 7 /System/Library

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.

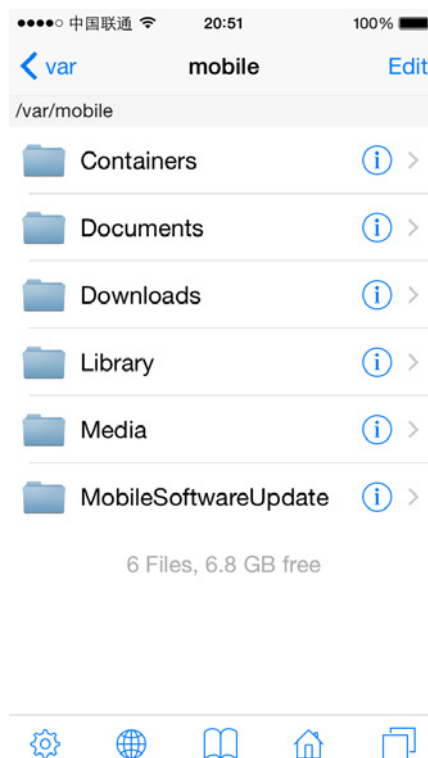


Figure 2- 8 /User

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.

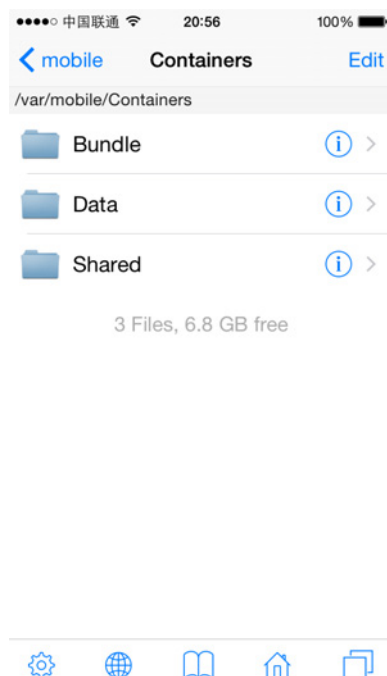


Figure 2- 9 `/var/mobile/Containers`

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 `dlib`) 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 NeXTSTEP. 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.

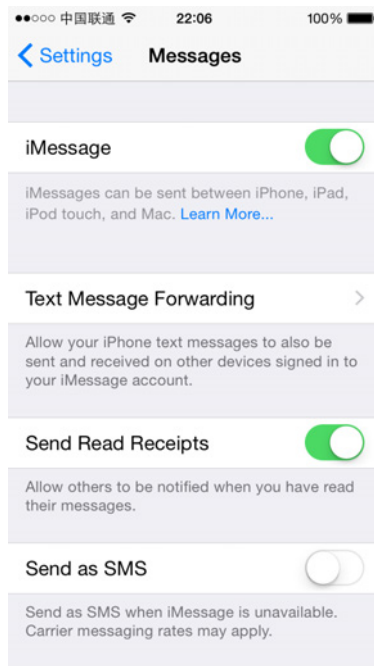


Figure 2- 10 PreferenceBundle

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.

Key	Type	Value
DTPlatformBuild	String	
MinimumOSVersion	String	8.1
Bundle OS Type code	String	APPL
Localization native development r...	String	en
DTXcodeBuild	String	6A267p
Bundle version	String	1.0
UIDeviceFamily	Array	(2 items)
Bundle identifier	String	com.apple.SiriViewService
Icon files	Array	(1 item)

Figure 2- 11 Browse Info.plist in Xcode

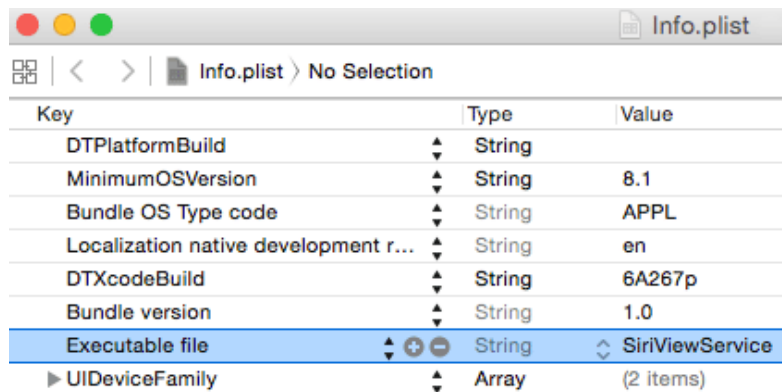
Or use a command line tool, plutil, to view its value.

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/SiriViewService.app/Info.plist |
grep CFBundleIdentifier
"CFBundleIdentifier" => "com.apple.SiriViewService"
```

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

- Executable

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



Key	Type	Value
DTPlatformBuild	String	
MinimumOSVersion	String	8.1
Bundle OS Type code	String	APPL
Localization native development r...	String	en
DTXcodeBuild	String	6A267p
Bundle version	String	1.0
Executable file	String	SiriViewService
UIDeviceFamily	Array	(2 items)

Figure 2- 12 Browse Info.plist in Xcode

Or with plutil:

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

- lproj directories

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

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/SiriViewService.app/en.lproj/Localiz
able.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/LaunchDaemons/com.
apple.imagent.plist
```

```

{
  "WorkingDirectory" => "/tmp"
  "Label" => "com.apple.imagent"
  "JetsamProperties" => {
    "JetsamMemoryLimit" => 3000
  }
  "EnvironmentVariables" => {
    "NSRunningFromLaunchd" => "1"
  }
  "POSIXSpawnType" => "Interactive"
  "MachServices" => {
    "com.apple.hsa-authentication-server" => 1
    "com.apple.imagent.embedded.auth" => 1
    "com.apple.incoming-call-filter-server" => 1
  }
  "UserName" => "mobile"
  "RunAtLoad" => 1
  "ProgramArguments" => [
    0 => "/System/Library/PrivateFrameworks/IMCore.framework/imagent.app/imagent"
  ]
  "KeepAlive" => {
    "SuccessfulExit" => 0
  }
}

```

Compared with Apps, daemons provide much much lower level functions, accompanying with much much greater difficulties reverse engineering them. If you don't know what you're doing for sure, don't even try to modify them! It may break your iOS, leading to booting failures, so you'd better stay away from daemons as reverse engineering newbies . After you get some 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>.

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.

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:

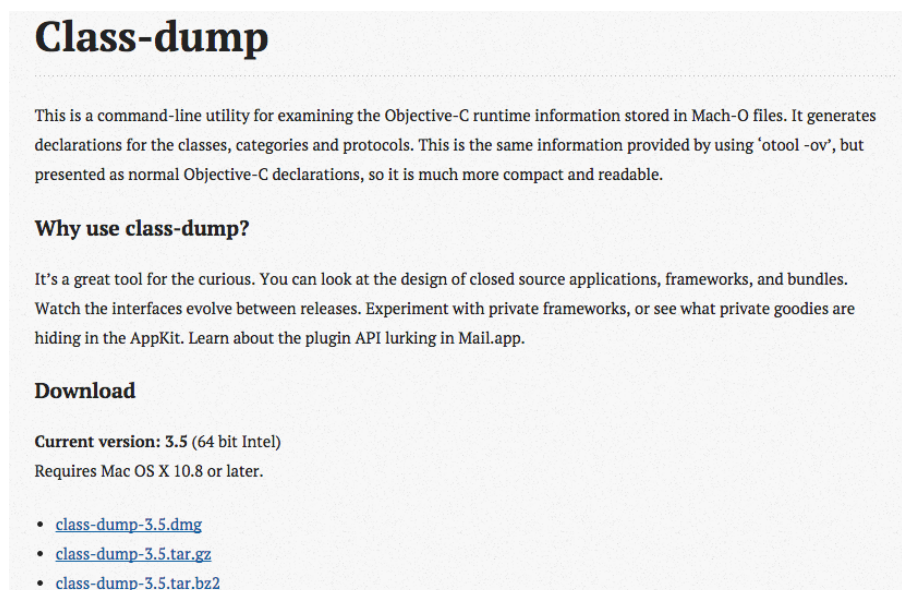
The image shows a screenshot of the class-dump project homepage. At the top, the title "Class-dump" is displayed in a large, bold, black font. Below the title, a paragraph describes the tool as a command-line utility for examining Objective-C runtime information in Mach-O files, generating declarations for classes, categories, and protocols. A section titled "Why use class-dump?" explains that it's a great tool for the curious, allowing users to look at the design of closed source applications, frameworks, and bundles. A "Download" section lists the current version as 3.5 (64 bit Intel) and provides three download links: class-dump-3.5.dmg, class-dump-3.5.tar.gz, and class-dump-3.5.tar.bz2. The page has a light gray background with a white content area.

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 iPhone0S 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.

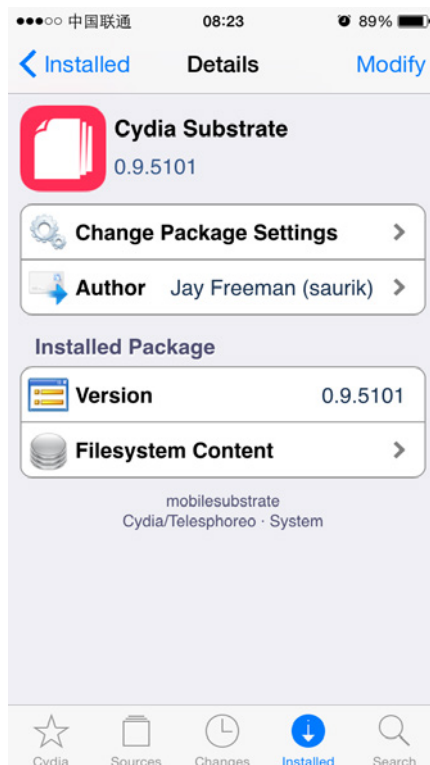


Figure 3- 2 CydiaSubstrate

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/cydyget
[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.xml  
-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.xml  
  
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.

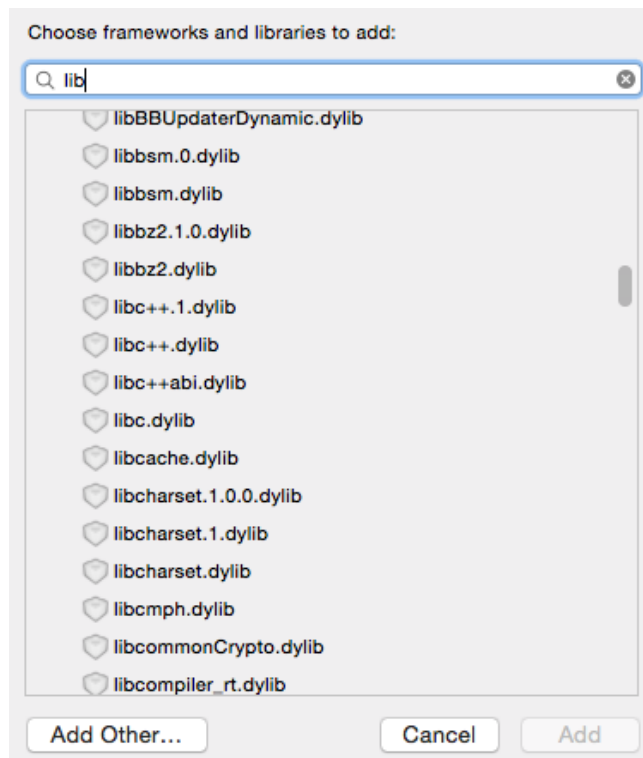


Figure 3- 3 Link Mach-O Objects

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;
}
%end

```

```

        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 NSStringFromClass, 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:

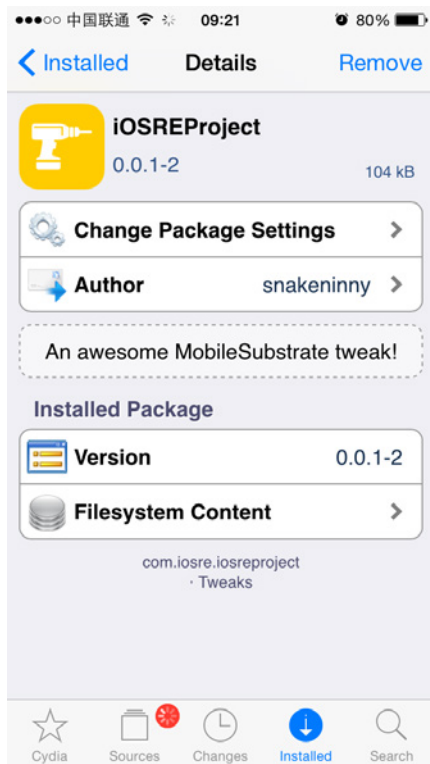


Figure 3- 5 Control informaton in Cydia

- 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:

Key	Type	Value
▼ Root	Dictionary	(1 item)
▶ Filter	Dictionary	(1 item)

Figure 3- 6 iOSREProject.plist

There’s a series of arrays under “Filter”, which can be categorized into 3 types.

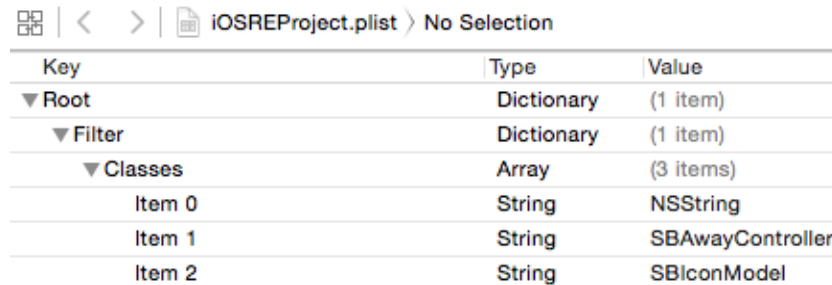
- ✧ “Bundles” specifies several bundles as the tweak’s targets, as shown in figure 3-7.

Key	Type	Value
▼ Root	Dictionary	(1 item)
▼ Filter	Dictionary	(1 item)
▼ Bundles	Array	(3 items)
Item 0	String	com.naken.smsninja
Item 1	String	com.apple.AddressBook
Item 2	String	com.apple.springboard

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.



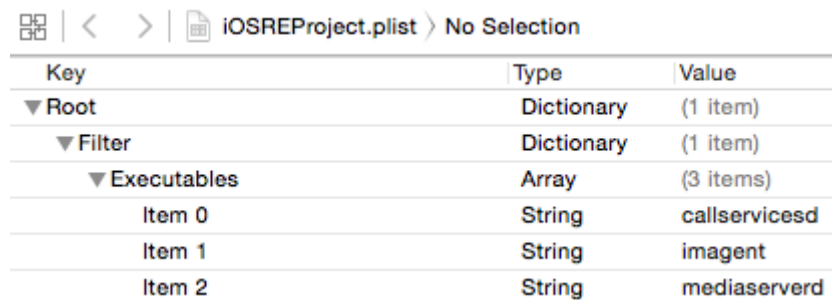
The screenshot shows a plist editor interface for 'iOSREProject.plist'. The 'Classes' section is expanded, showing an array of three string items. The first item is 'NSString', the second is 'SBAwayController', and the third is 'SBIconModel'.

Key	Type	Value
▼ Root	Dictionary	(1 item)
▼ Filter	Dictionary	(1 item)
▼ Classes	Array	(3 items)
Item 0	String	NSString
Item 1	String	SBAwayController
Item 2	String	SBIconModel

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.



The screenshot shows a plist editor interface for 'iOSREProject.plist'. The 'Executables' section is expanded, showing an array of three string items. The first item is 'callservicesd', the second is 'imagent', and the third is 'mediaserverd'.

Key	Type	Value
▼ Root	Dictionary	(1 item)
▼ Filter	Dictionary	(1 item)
▼ Executables	Array	(3 items)
Item 0	String	callservicesd
Item 1	String	imagent
Item 2	String	mediaserverd

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.

Key	Type	Value
▼ Root	Dictionary	(1 item)
▼ Filter	Dictionary	(4 items)
Mode	String	Any
▼ Bundles	Array	(1 item)
Item 0	String	com.apple.springboard
▼ Classes	Array	(1 item)
Item 0	String	TUCallServicesCallController
▼ Executables	Array	(1 item)
Item 0	String	callservicesd

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.xml...
Compiling Tweak.xml...
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.xml
-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.xml.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 “dpkg-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:

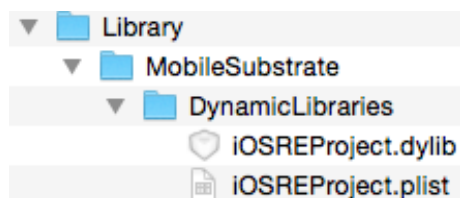


Figure 3-11 Library directory structure

If compared with the contents of deb package:

```

snakeninnysMac: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.xml
-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.

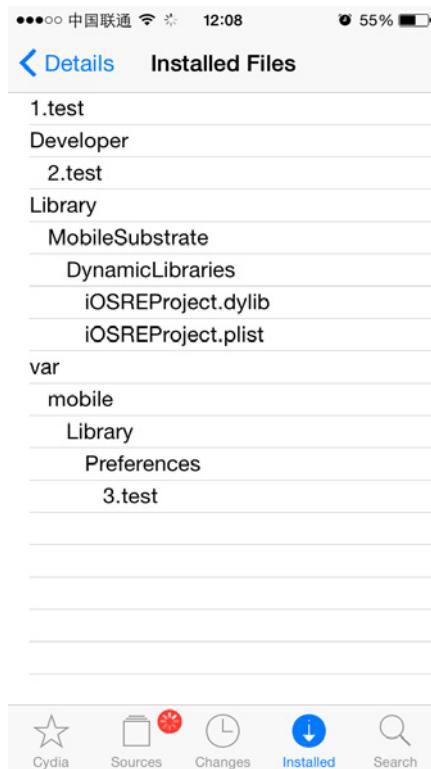


Figure 3-13 Installed files of iOSREProject

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.

- ✧ Install 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.

- ✧ Install 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
hXFscxBCVXgqXhwm4PUoUVBFWRrNeG6gVI3Ewm4dqwusoRcyCxZtm5bRiv4bXfkPjsRkWVvfrW3uT52Hhx4RqIuC
OxtWE7tZqc1vVap4HIzUu3mwBuxog7WiFbsbbaJY4AagNZmX83Wmvf8li5aYMsuKeNagdJHzJNtjm3vtuskK4jKz
BkNuj0M89TrV4iEmKtI4VEoEmHMYzWwMzExXbyX5NyEg5CRFmA46XeYCbcaY0L90GEXsWMMLA27tA1Vt1ndHrKN
xZttgAw31J90UDn0GLMbWW4M7FEqRWQsWXxfGpk0W7A1A54vaDXl1I5CD5nLAu4VkrjPIUBrdH501fqQ3qGkPayh
sym3g0VZeYgU4JAMeFc3
```

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/cydyget
[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.xml

The edited Tweak.xml 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:

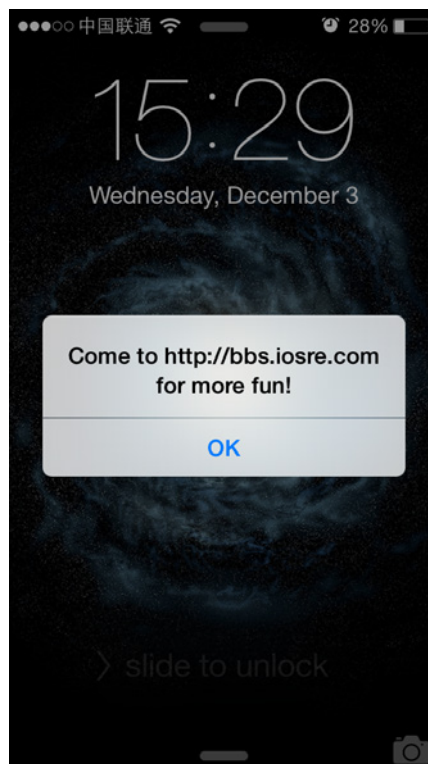


Figure 3- 14 Our first tweak

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 become 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.

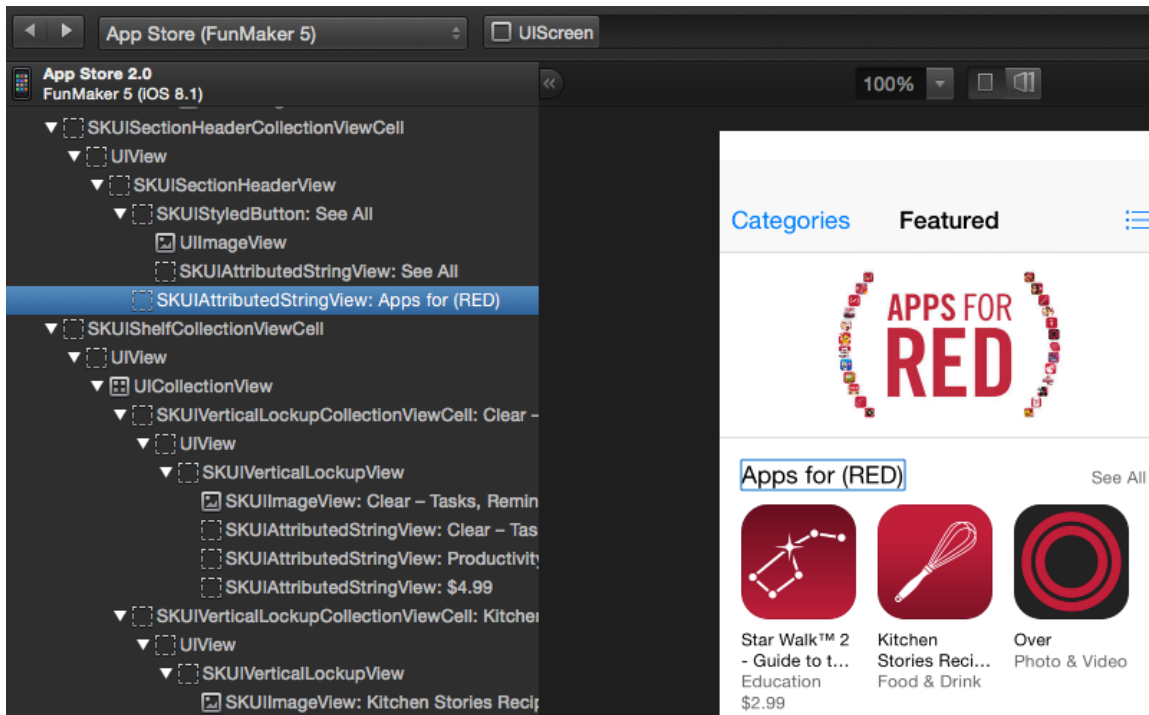


Figure 3-16 See the view hierarchy of AppStore

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.

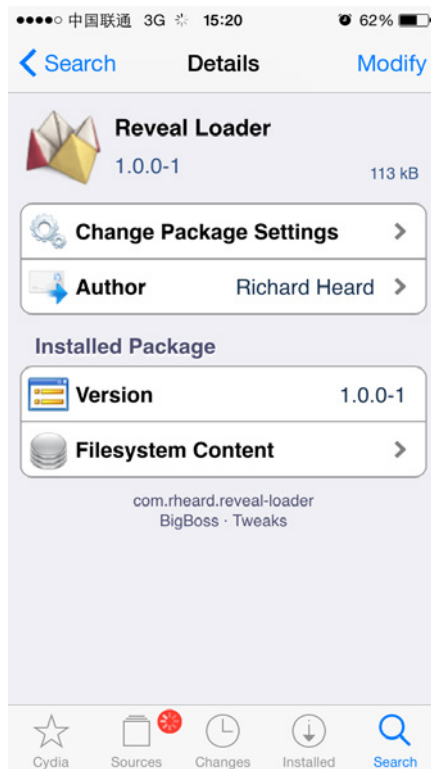


Figure 3-17 Reveal Loader

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.

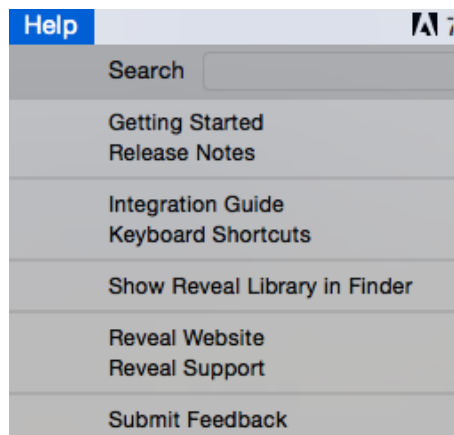


Figure 3- 18 Show Reveal Library in Finder

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

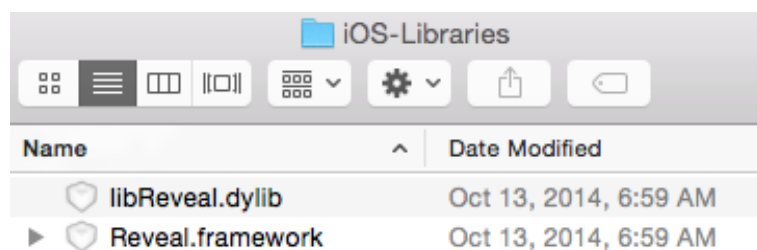


Figure 3- 19 libReveal.dylib

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.

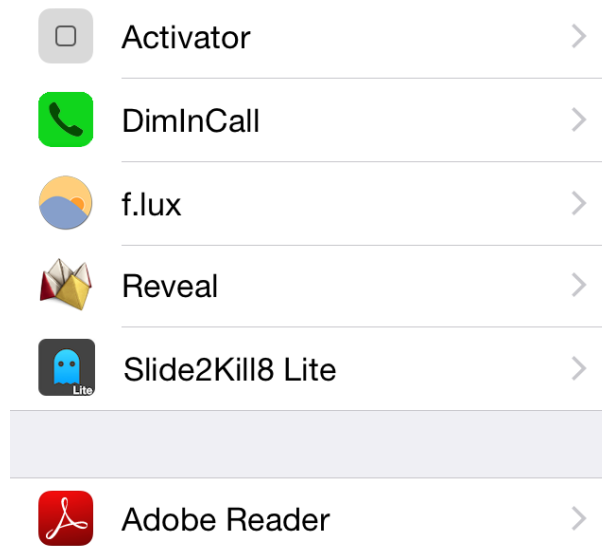


Figure 3- 20 Reveal Loader

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

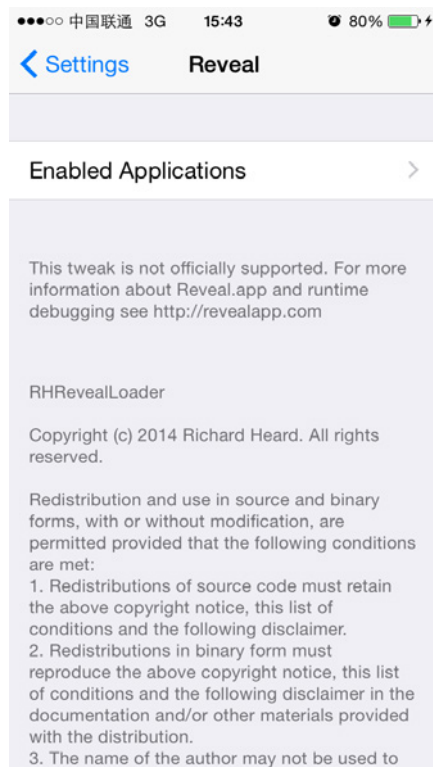


Figure 3-21 Declaration of Reveal Loader

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.

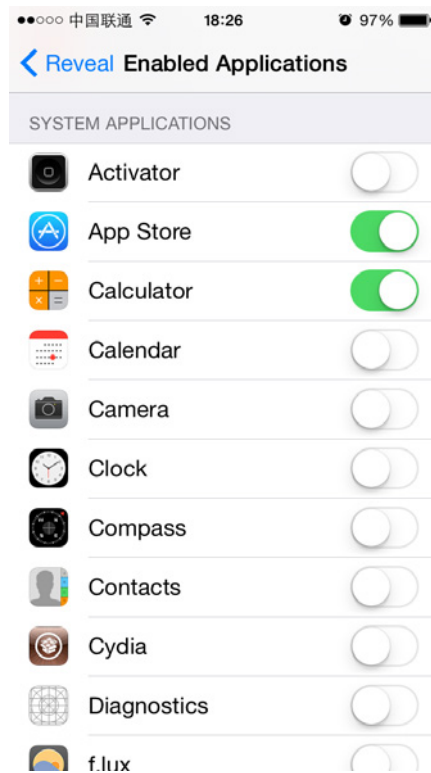


Figure 3-22 configuration of Reveal Loader

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.

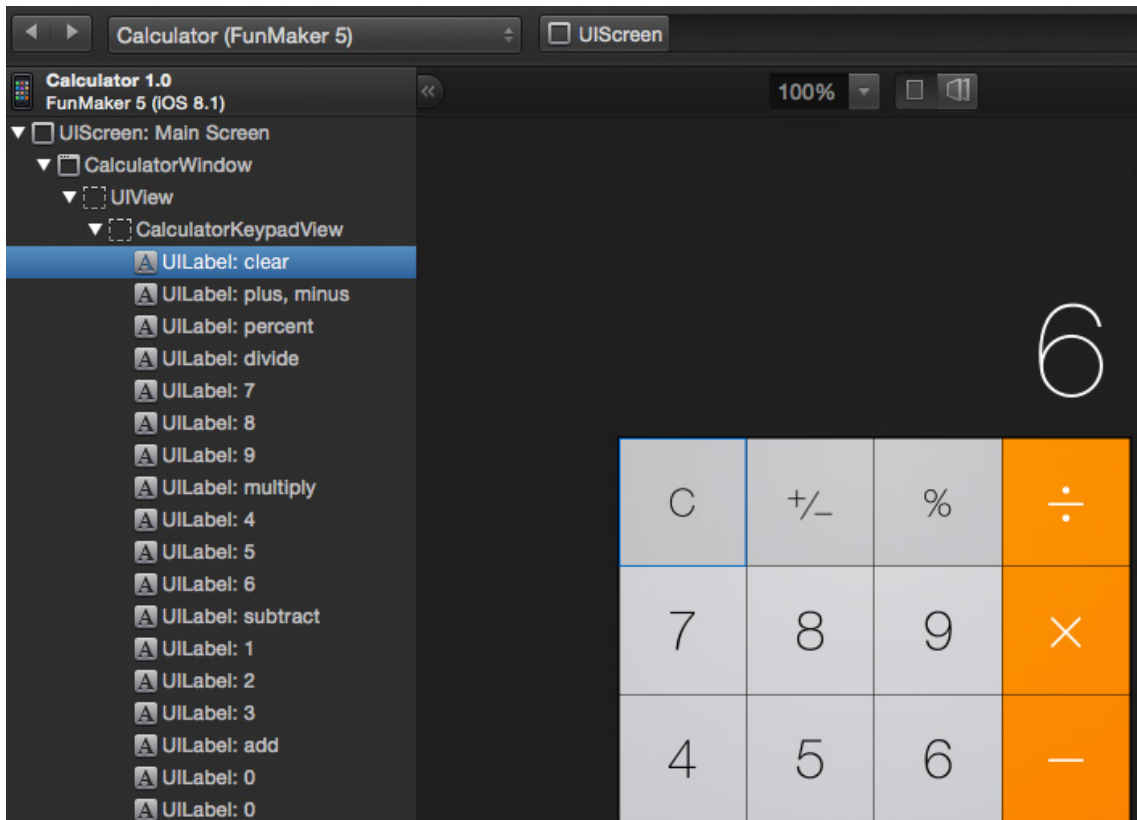


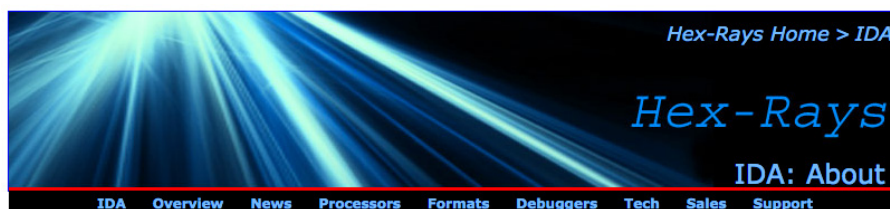
Figure 3-23 View hierarchy of Calculator

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 Cycrypt 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.



IDA: About

What is IDA all about?

IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger that offers so many features it is hard to describe them all. Just grab an [evaluation version](#) if you want a test drive.

An [executive summary](#) is provided for the non-technical user.

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.

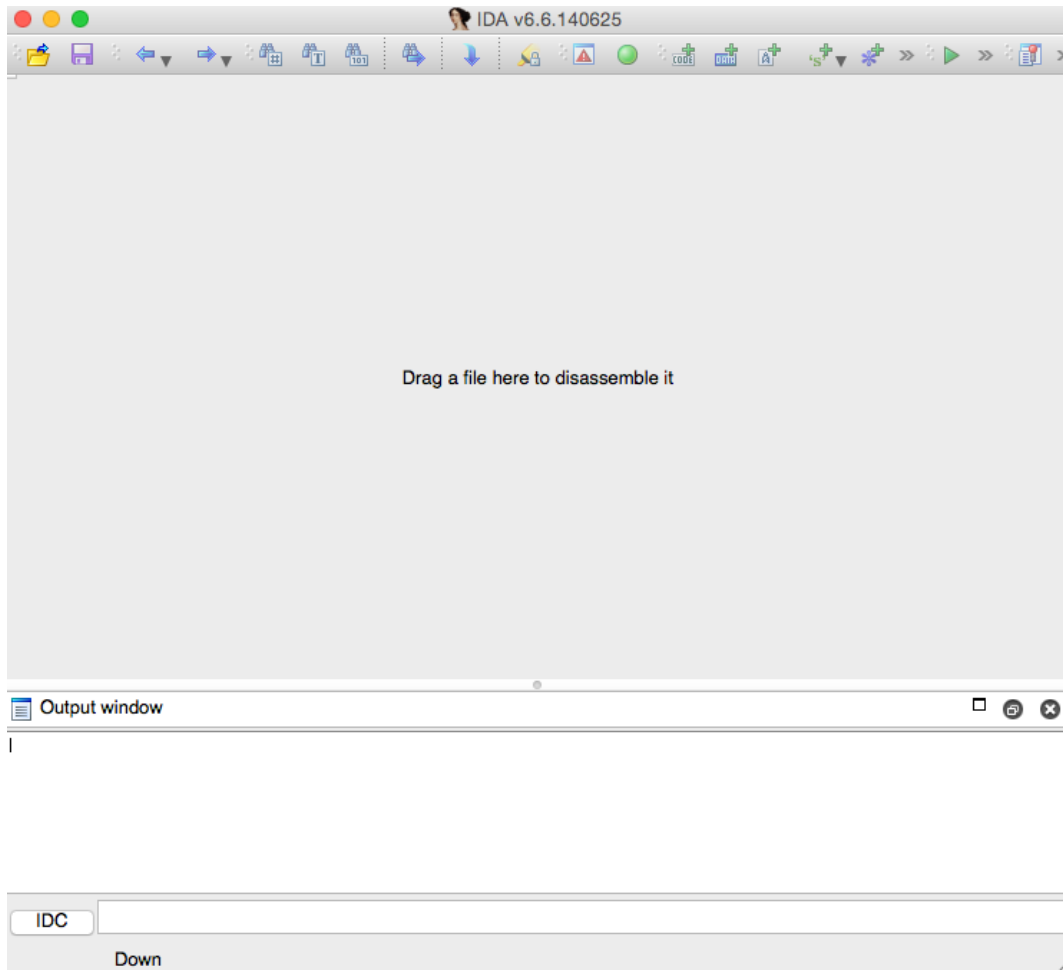


Figure 3-26 Main screen of IDA

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.

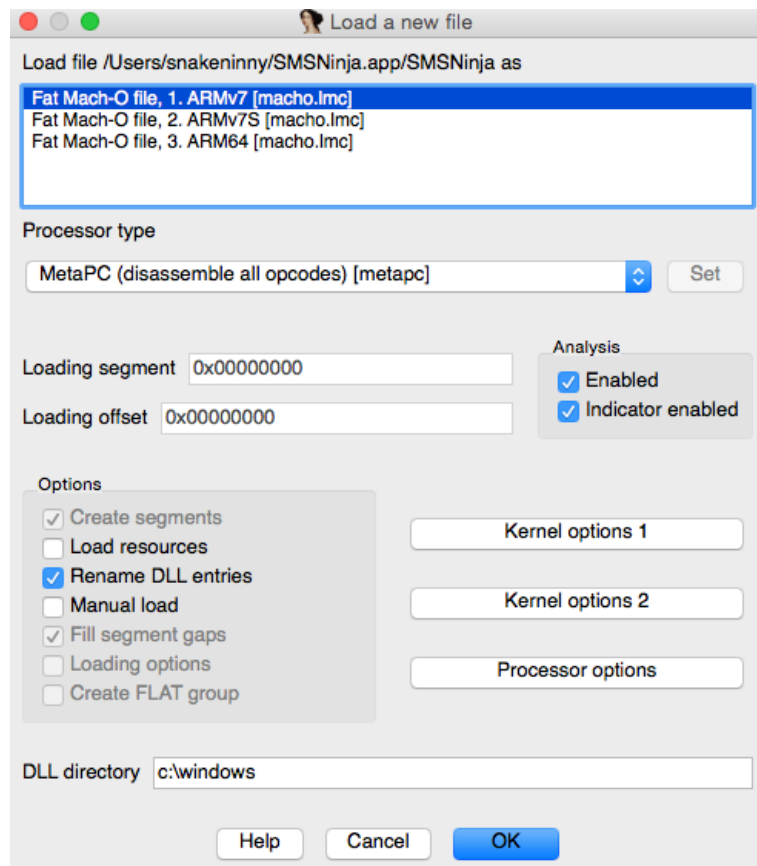


Figure 3-27 Initial configurations

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.

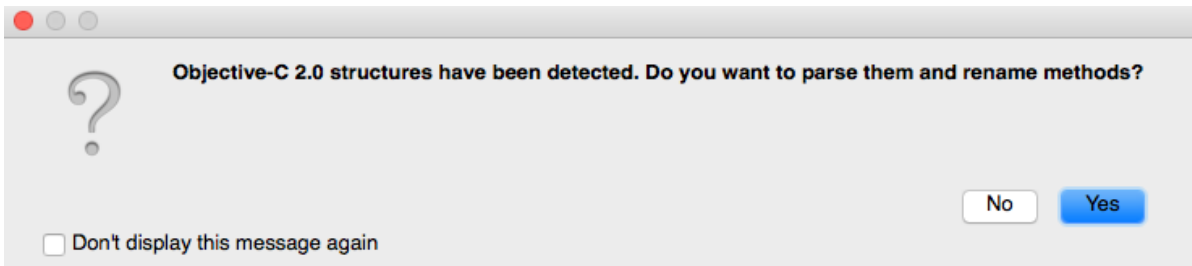


Figure 3-28 IDA launch option

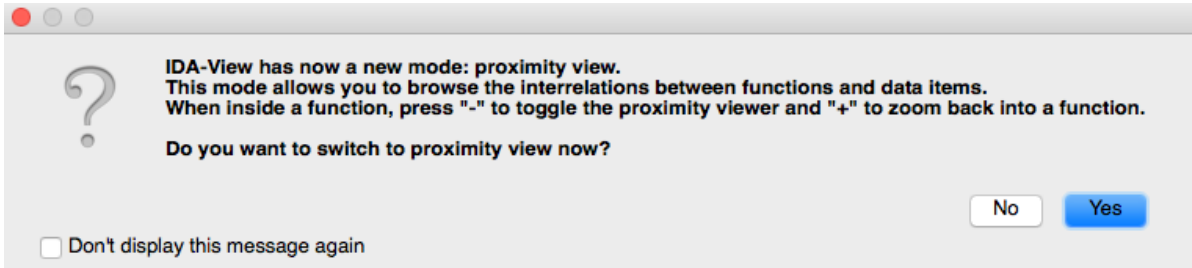


Figure 3-29 IDA launch option

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.

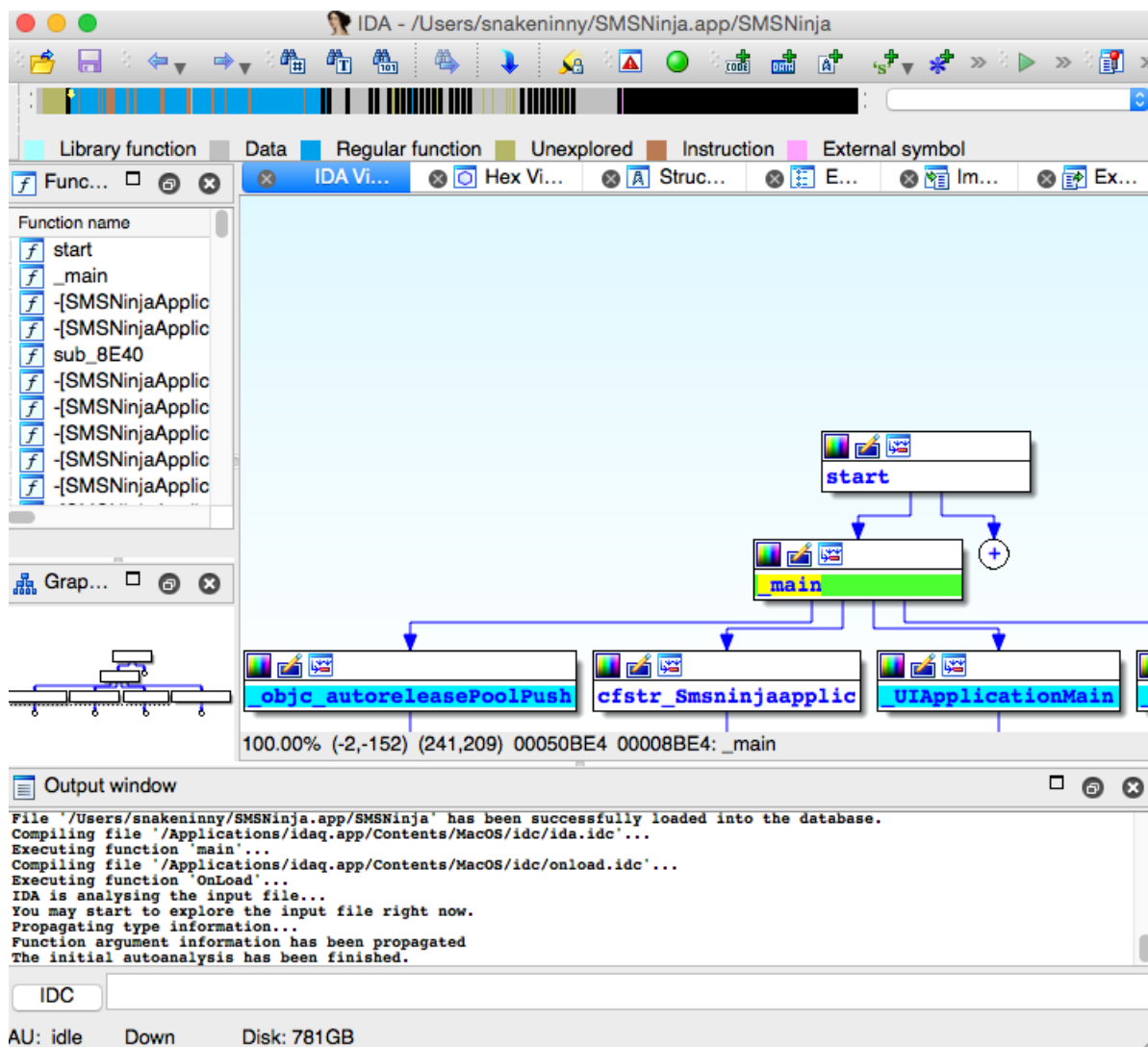


Figure 3-30 IDA main screen

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.

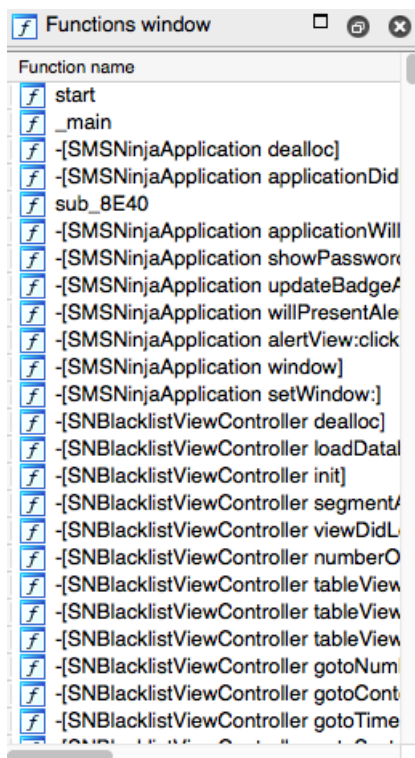


Figure 3-31 Functions window

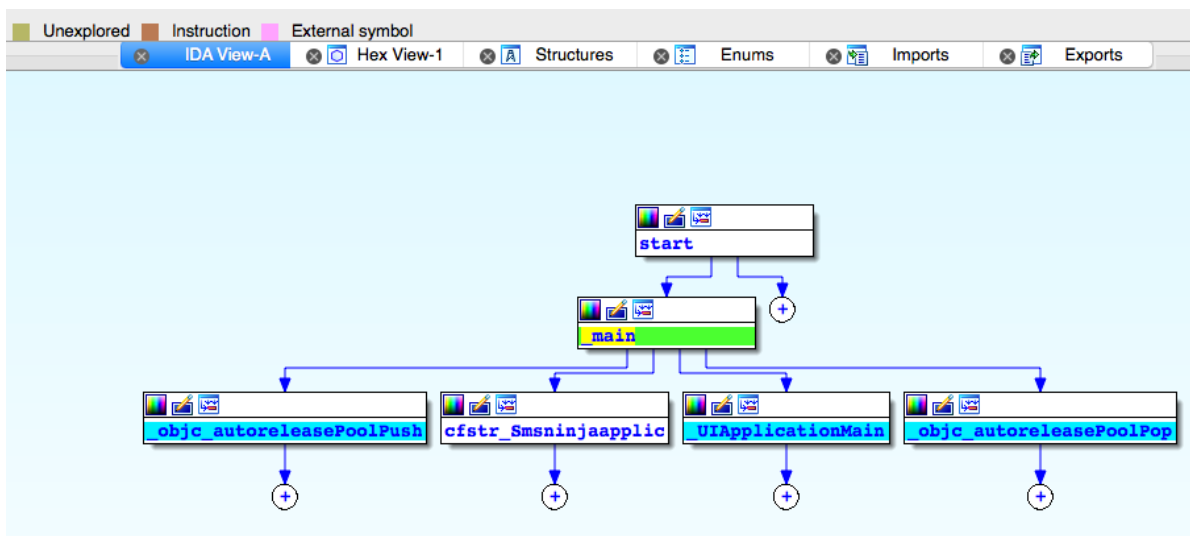


Figure 3-32 Main window

- 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.

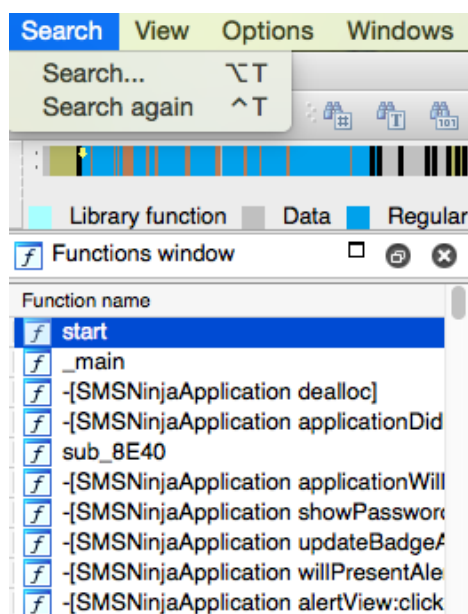


Figure 3-33 Search functions

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.

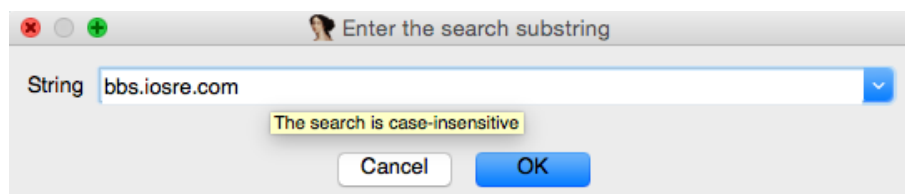


Figure 3-34 Search functions

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.

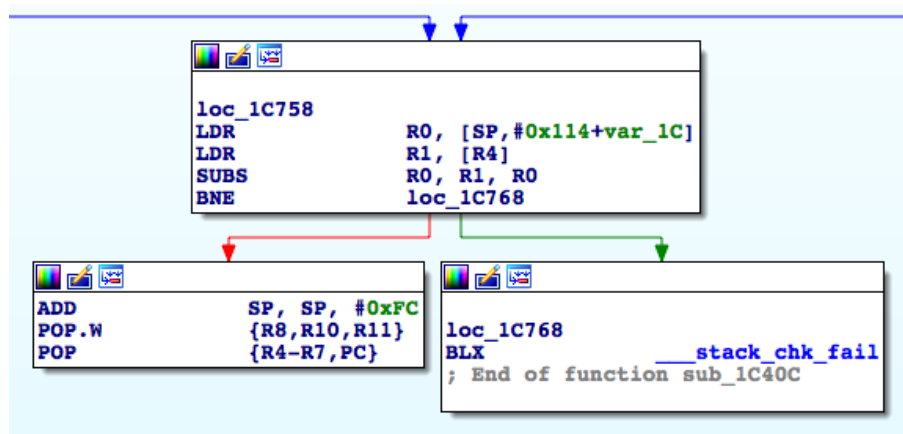


Figure 3- 36 Branches in IDA

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.

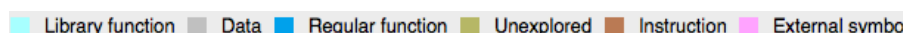


Figure 3-37 Color indication bar

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.

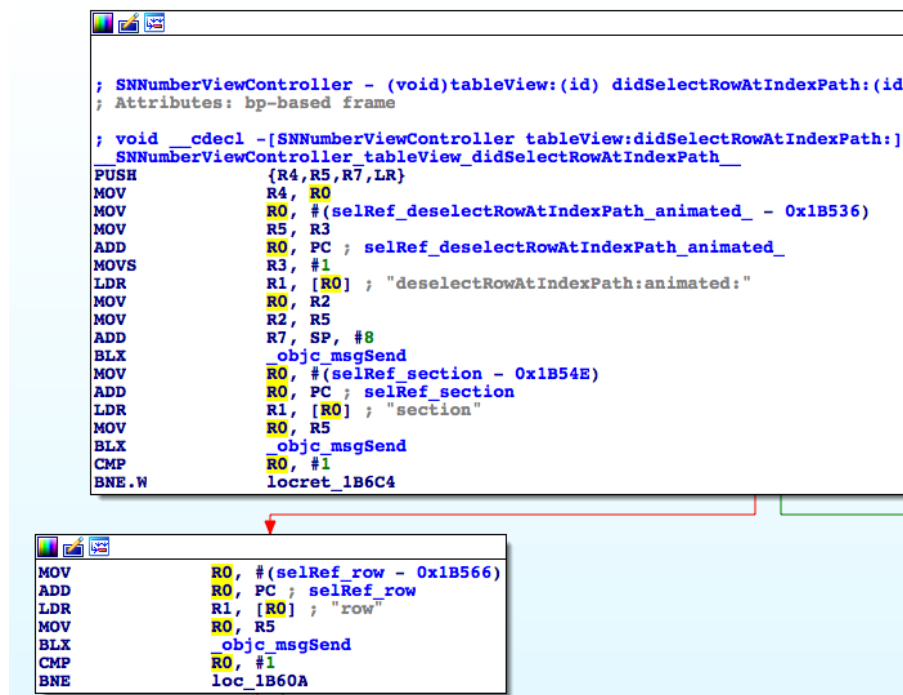


Figure 3-38 Symbol highlight

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.

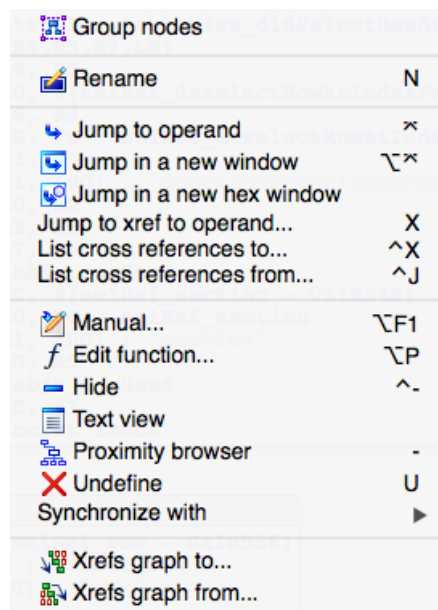


Figure 3-39 Right click on a symbol

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.

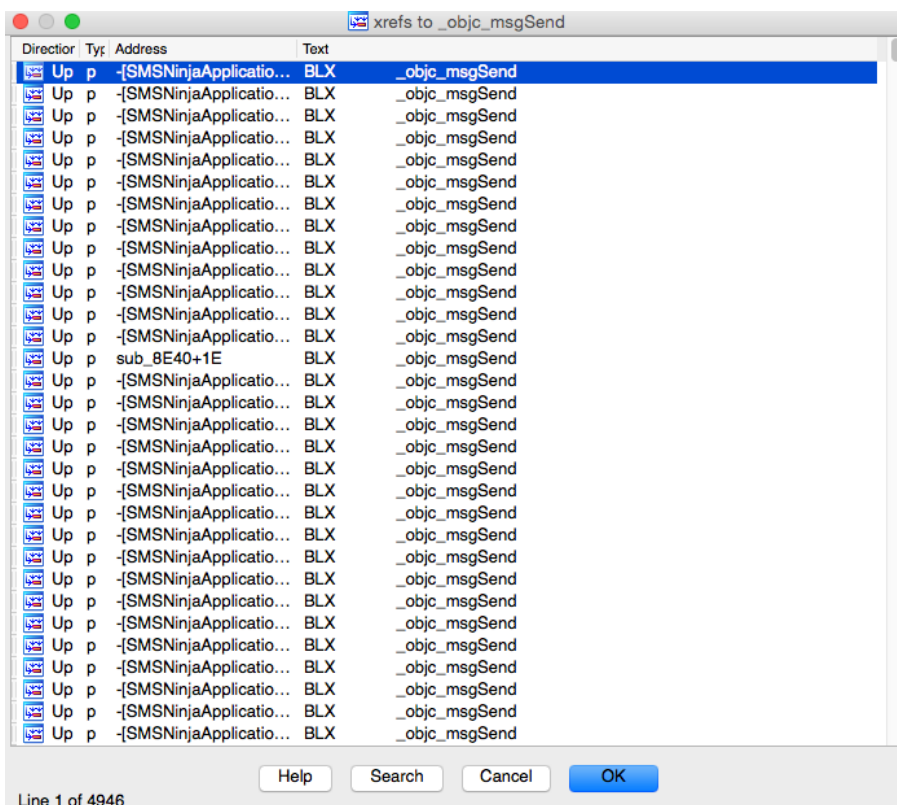


Figure 3- 40 Jump to xref to operand...

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.

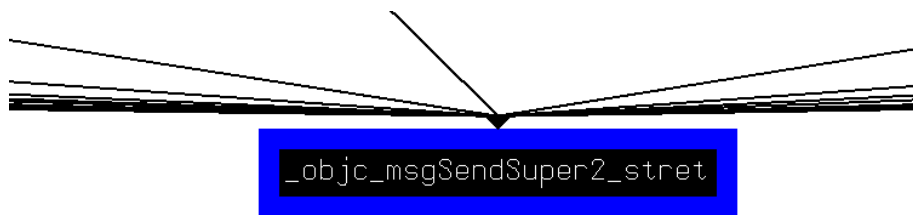


Figure 3-41 Xrefs graph to...

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.

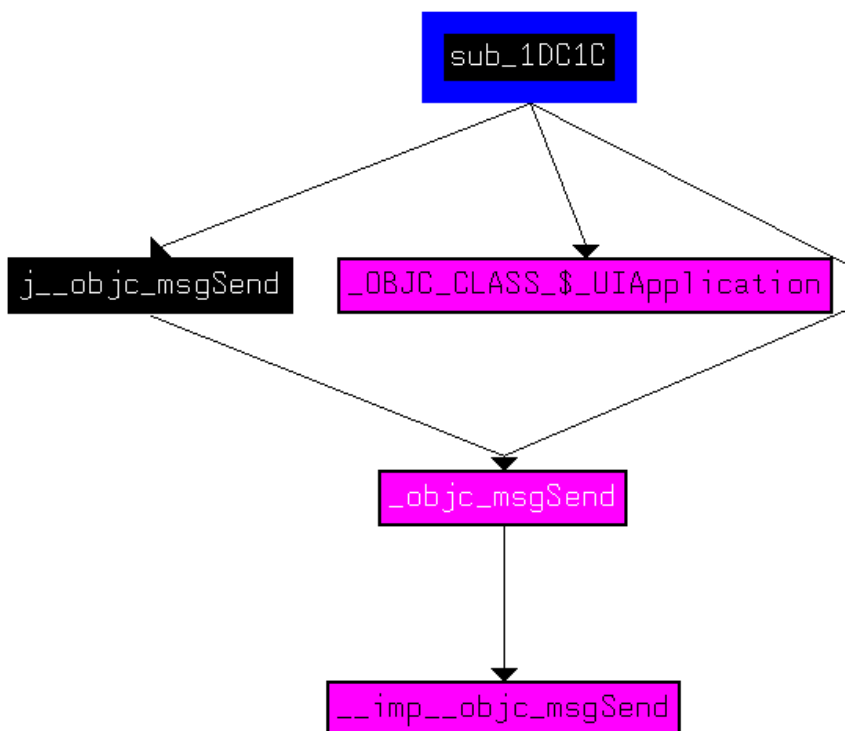


Figure 3-42 Xrefs graph from...

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:0000D3DC1o ...
IMPORT __objc_empty_cache
; DATA XREF: __objc_data: OBJC_CLASS_$_SMSNin
; __objc_data: OBJC_METACLASS_$_SMSNinjaAppli
IMPORT __imp_objc_autoreleasePoolPop
; CODE XREF: __objc_autoreleasePoolPop1j
; DATA XREF: __objc_autoreleasePoolPop1o ...
IMPORT __imp_objc_autoreleasePoolPush
; CODE XREF: __objc_autoreleasePoolPush1j
; DATA XREF: __objc_autoreleasePoolPush1o ...
IMPORT __imp_objc_enumerationMutation
; CODE XREF: __objc_enumerationMutation1j
; DATA XREF: __objc_enumerationMutation1o ...
IMPORT __imp_objc_getClass
; CODE XREF: __objc_getClass1j
; DATA XREF: __objc_getClass1o ...
IMPORT __imp_objc_msgSend
; CODE XREF: __objc_msgSend1j
; DATA XREF: __objc_msgSend1o ...
IMPORT __imp_objc_msgSendSuper2
; CODE XREF: __objc_msgSendSuper21j
; DATA XREF: __objc_msgSendSuper21o ...
IMPORT __imp_objc_msgSend_stret
; CODE XREF: __objc_msgSend_stret1j
; DATA XREF: __objc_msgSend_stret1o ...
IMPORT __imp_objc_setProperty
; CODE XREF: __objc_setProperty1j
; DATA XREF: __objc_setProperty1o ...
```

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.

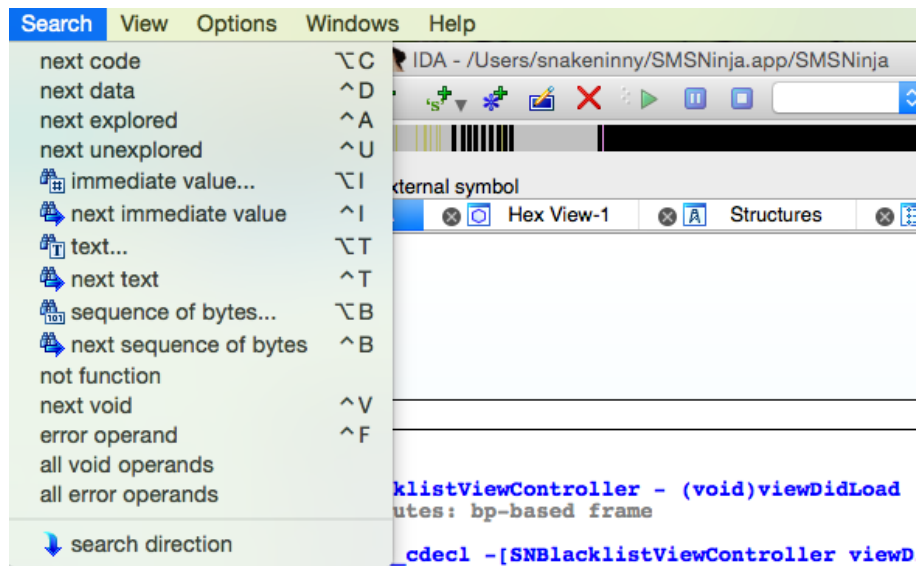


Figure 3-44 Search in Main window

Choose “text...”, a window will popup, as shown in figure 3-45.

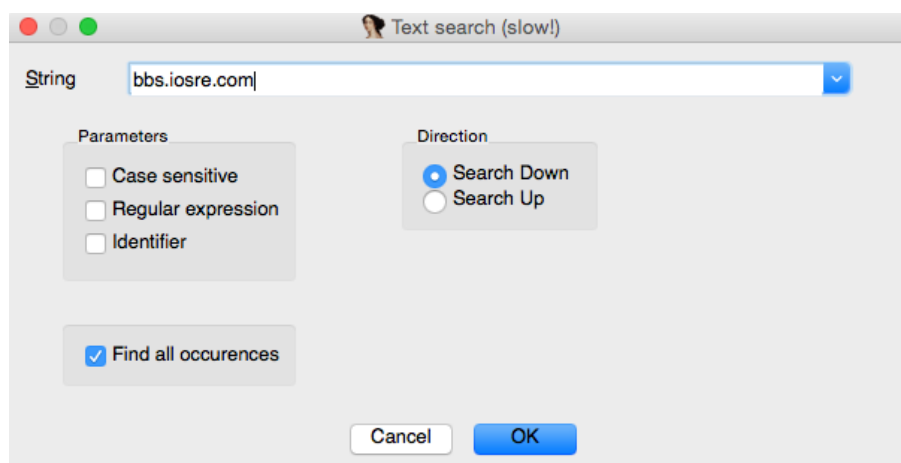


Figure 3-45 Text search

There're other searching options available, you can check them out according to your situations. Then check "Find all occurrences" 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.

```

PUSH      {R4,R5,R7,LR}
ADD       R7, SP, #8
SUB       SP, SP, #4
MOV       R4, R0
BLX      _BSLoggingInitialize
TST.W    R0, #0xFF
BEQ      loc_35DA6

MOV       R0, #(_FBWorkspaceLoggingEnabled_ptr - 0x35D4A)
ADD       R0, PC ; _FBWorkspaceLoggingEnabled_ptr
LDR      R0, [R0] ; _FBWorkspaceLoggingEnabled
LDRB     R0, [R0]
CBZ      R0, loc_35DA6

MOV       R0, #(selRef_stringWithFormat_ - 0x35D66)
MOV       R2, #(classRef_NSString - 0x35D68)
MOVW     R5, #(:lower16:(stru_42148C - 0x35D76)) ; "%@"
ADD       R0, PC ; selRef_stringWithFormat_
ADD       R2, PC ; classRef_NSString
MOVT.W   R5, #(:upper16:(stru_42148C - 0x35D76)) ; "%@"
LDR      R1, [R0] ; stringWithFormat:"
LDR      R0, [R2] ; _OBJC_CLASS_$_NSString
MOVW     R2, #(:lower16:(cfstr_S_8 - 0x35D84)) ; "%s() %@"
ADD       R5, PC ; "%@"
MOVT.W   R2, #(:upper16:(cfstr_S_8 - 0x35D84)) ; "%s() %@"
MOV       R3, #(aSbworkspacerel - 0x35D88) ; "SBWorkspaceRelaunchWhenFinishedTerminat"...
ADD       R2, PC ; "%s() %@"
STR      R5, [SP,#0xC+var_C]
ADD       R3, PC ; "SBWorkspaceRelaunchWhenFinishedTerminat"...
BLX      _objc_msgSend
MOV       R5, R0
MOV       R0, R4
BLX      _NSStringFromBOOL
MOV       R3, R0
MOV       R0, #(cfstr_Fbworkspacelog - 0x35DA2) ; "FBWorkspaceLog"
MOV       R2, R5
ADD       R0, PC ; "FBWorkspaceLog"
MOV       R1, R0
BLX      _BSFileLog

loc_35DA6
MOV       R0, #(byte_4DD880 - 0x35DB2)
ADD       R0, PC ; byte_4DD880
STRB     R4, [R0]
ADD       SP, SP, #4
POP      {R4,R5,R7,PC}
; End of function sub_35D2C

```

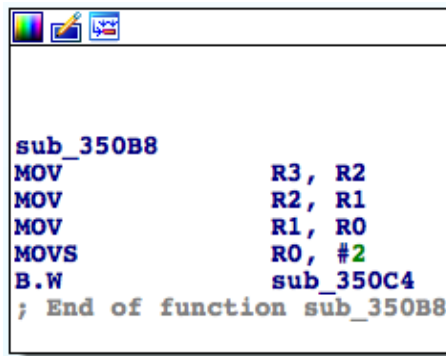
Figure 3- 48 sub_35D2C

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.

A screenshot of a window displaying assembly code. The code is as follows:

```
sub_350B8
MOV          R3, R2
MOV          R2, R1
MOV          R1, R0
MOVS        RO, #2
B.W         sub_350C4
; End of function sub_350B8
```

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_$_FBWorkspaceEvent
MOV      R3, #(sub_351F8+1 - 0x351A4)
STR      R1, [SP,#0x44+var_3C]
MOVW     R1, #0xC2000000
STR      R1, [SP,#0x44+var_38]
ADD      R9, PC ; selRef_eventWithName_handler_
MOVS     R1, #0
ADD      R4, PC ; unk_40B640
ADD      R3, PC ; sub_351F8
STMIA.W  R10, {R1,R3,R4}
ADD      R2, PC ; "TerminateApplicationGroup"
ADD      R3, SP, #0x44+var_3C
LDR.W    R1, [R9] ; "eventWithName:handler:"
STR      R6, [SP,#0x44+var_24]
STR      R5, [SP,#0x44+var_20]
STRB.W   R8, [SP,#0x44+var_1C]
STR.W    R11, [SP,#0x44+var_28]
BLX      _objc_msgSend
MOV      R4, R0
MOV      R0, #(selRef_sharedInstance - 0x351D4)
MOV      R2, #(classRef_FBWorkspaceEventQueue - 0x351D6)
ADD      R0, PC ; selRef_sharedInstance
ADD      R2, PC ; classRef_FBWorkspaceEventQueue
LDR      R1, [R0] ; "sharedInstance"
LDR      R0, [R2] ; _OBJC_CLASS_$_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.

```
sub_351F8
LDR.W      R9, [R0, #0x18]
LDR        R3, [R0, #0x14]
LDR        R1, [R0, #0x1C]
LDRSB.W   R2, [R0, #0x20]
MOV        R0, R9
B.W        j__BKSTerminateApplicationGroupForReasonAndReportWithDescription
; End of function sub_351F8
```

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

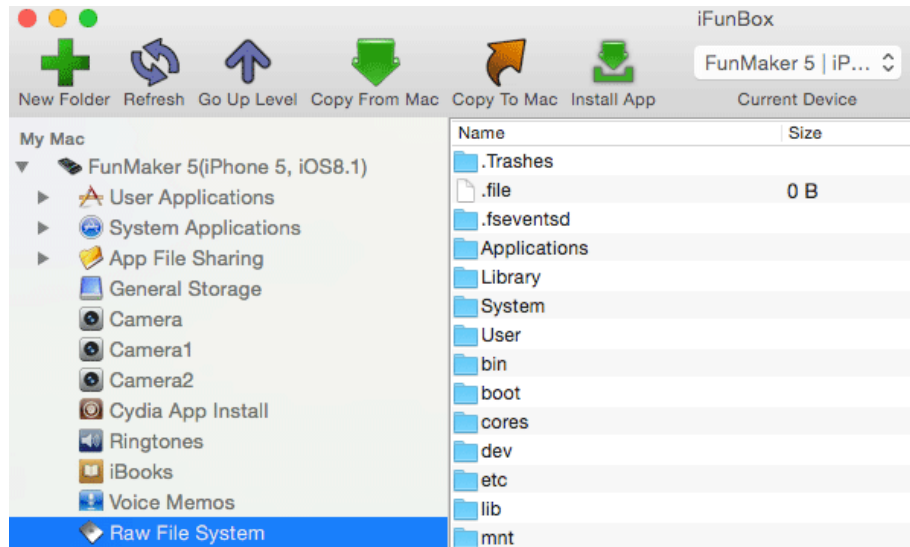


Figure 3-53 iFunBox

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.

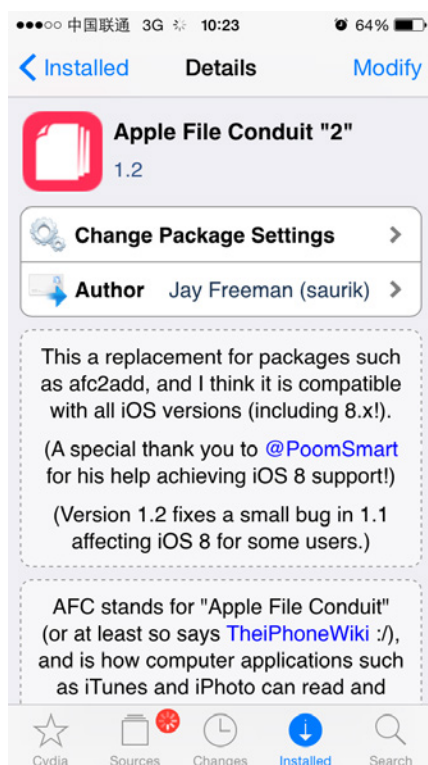


Figure 3-54 Apple File Conduit 2

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](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/AXSpeechImplementati
on'...
 1/877: Dumping
'/System/Library/AccessibilityBundles/AccessibilitySettingsLoader.bundle/AccessibilitySe
ttingsLoader'...
 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.

iOS toolkit

In chapter 3, we've introduced the OSX toolkit for iOS reverse engineering. To get our work done, we still need to install and configure several tools on iOS to combine both platforms. All operations in this chapter are finished on iPhone 5, iOS 8.1, if you encounter any problems, please talk to us on <http://bbs.iosre.com>.

4.1 CydiaSubstrate



Figure 4- 1 Logo of CydiaSubstrate

CydiaSubstrate (as shown in figure 4-1) is the infrastructure of most tweaks. It consists of MobileHooker, MobileLoader and Safe mode.

4.1.1 MobileHooker

MobileHooker is used to replace system calls, or namely, hook. There are two major functions:

```
void MSHookMessageEx(Class class, SEL selector, IMP replacement, IMP *result);  
void MSHookFunction(void* function, void* replacement, void** p_original);
```

MSHookMessageEx works on Objective-C methods. It calls `method_setImplementation` to replace the original implementation of `[class selector]` with “replacement”. What exactly does this mean? For example, if we send the message `hasSuffix:` to an `NSString` object (i.e, call `[NSString hasSuffix:]`), in normal situation, this method's implementation is to indicate whether an `NSString` object has a certain suffix. But if we change this implementation with the implementation of `hasPrefix:`, then after an `NSString` object receives `hasSuffix:` message, it

actually verifies whether an NSString object has a certain prefix. Isn't it easy to understand?

Logos syntax, which we've introduced in chapter 3, is actually an encapsulation of MSHookMessageEx. Although Logos is clean and elegant, while making it easy to write Objective-C hooks, it's still based on MSHookMessageEx. For Objective-C hooks, we recommend using Logos instead of MSHookMessageEx. If you are interested in the use of MSHookMessageEx, you can take a look at its official document, or Google "cydiasubstrate fuchsiaexample", the link starting with "http://www.cydiasubstrate.com" is what you are looking for.

MSHookFunction is used for C/C++ hooks, and works in assembly level. Conceptually, when the process is about to call "function", MSHookFunction makes it execute "replacement" instead, and allocate some memory to store the original "function" and its return address, making it possible for the process to execute "function" optionally, and guarantees the process can run as usual after executing "replacement".

Maybe it's hard to understand the above paragraph, so here comes an example. Let's take a look at figure 4-2.

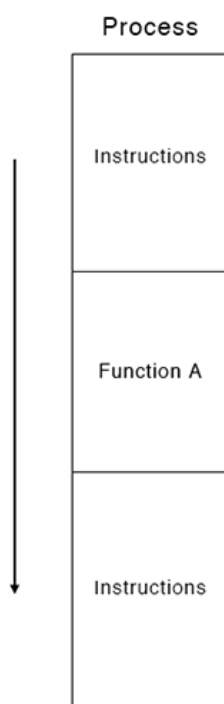


Figure 4- 2 Normal execution flow of a process

As shown in figure 4-2, a process executes some instructions, then calls function A, and afterward executes the remaining instructions. If we hook function A and replace it with function B, then this process' execution flow changes to figure 4-3.

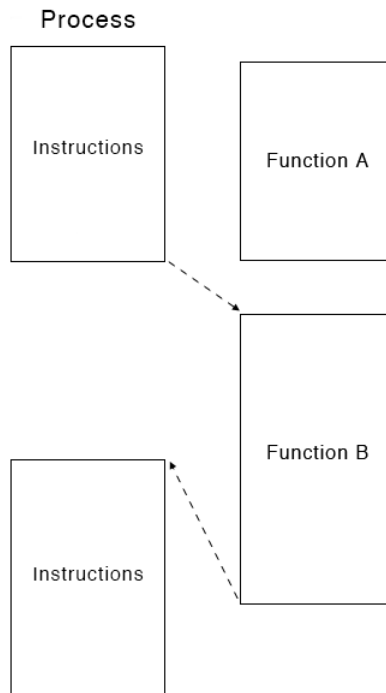


Figure 4- 3 Replace Function A with B

We can see in figure 4-3 that this process executes some instructions at first, but then calls function B at where it's supposed to call function A, with function A stored elsewhere. Inside function B, it's up to you whether and when to call function A. After function B finishes execution, the process will continue to execute the remaining instructions.

There's one more thing to notice. `MSHookFunction` has a requirement on the length of the function it hooks, the total length of all its instructions must be bigger than 8 bytes (This number is not officially acknowledged). So here comes the question, how to hook these less-than-8-byte short functions?

One workaround is hooking functions inside the short functions. The reason why a function is short is often because it calls other functions and they're doing the actual job. Some of the other functions are long enough to be hooked, so we can choose these functions to be `MSHookFunction`'s targets, then do some logical judgements in "replacement" to tell if the short function is the caller. If we can make sure the short function is calling the "replacement", then we can write our modification to the short function right inside "replacement".

If you are still confused about `MSHookFunction`, here is a simple example. To be honest, this example contains too much low-level knowledge, hence is quite hard for beginners to understand. Don't worry if you happen to be a newbie, just skip to section 4.1.2. When you encounter a similar situation later in practice, review this section and you'll know what we're

talking about. Anyway, welcome to <http://bbs.iosre.com> for further discussion.

Follow me:

1. Create iOSRETargetApp with Theos. The commands are as follows:

```
snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/library
[3.] iphone/preference_bundle
[4.] iphone/tool
[5.] iphone/tweak
Choose a Template (required): 1
Project Name (required): iOSRETargetApp
Package Name [com.yourcompany.iosretargetapp]: com.iosre.iosretargetapp
Author/Maintainer Name [snakeninny]: snakeninny
Instantiating iphone/application in iosretargetapp/...
Done.
```

2. Modify RootViewController.mm as follows:

```
#import "RootViewController.h"

class CPPClass
{
public:
    void CPPFunction(const char *);
};

void CPPClass::CPPFunction(const char *arg0)
{
    for (int i = 0; i < 66; i++) // This for loop makes this function long enough to
    validate MSHookFunction
    {
        u_int32_t randomNumber;
        if (i % 3 == 0) randomNumber = arc4random_uniform(i);
        NSProcessInfo *processInfo = [NSProcessInfo processInfo];
        NSString *hostName = processInfo.hostName;
        int pid = processInfo.processIdentifier;
        NSString *globallyUniqueString = processInfo.globallyUniqueString;
        NSString *processName = processInfo.processName;
        NSArray *junks = @[hostName, globallyUniqueString, processName];
        NSString *junk = @"";
        for (int j = 0; j < pid; j++)
        {
            if (pid % 6 == 0) junk = junks[j % 3];
        }
        if (i % 68 == 1) NSLog(@"Junk: %@", junk);
    }
    NSLog(@"iOSRE: CPPFunction: %s", arg0);
}

extern "C" void CFunction(const char *arg0)
{
    for (int i = 0; i < 66; i++) // This for loop makes this function long enough to
    validate MSHookFunction
```

```

    {
        u_int32_t randomNumber;
        if (i % 3 == 0) randomNumber = arc4random_uniform(i);
        NSProcessInfo *processInfo = [NSProcessInfo processInfo];
        NSString *hostName = processInfo.hostName;
        int pid = processInfo.processIdentifier;
        NSString *globallyUniqueString = processInfo.globallyUniqueString;
        NSString *processName = processInfo.processName;
        NSArray *junks = @[hostName, globallyUniqueString, processName];
        NSString *junk = @"";
        for (int j = 0; j < pid; j++)
        {
            if (pid % 6 == 0) junk = junks[j % 3];
        }
        if (i % 68 == 1) NSLog(@"Junk: %@", junk);
    }
    NSLog(@"iOSRE: CFunction: %s", arg0);
}

extern "C" void ShortCFunction(const char *arg0) // ShortCFunction is too short to be
hooked
{
    CPPClass cppClass;
    cppClass.CPPFunction(arg0);
}

@implementation RootViewController
- (void)loadView {
    self.view = [[[UIView alloc] initWithFrame:[UIScreen mainScreen]
applicationFrame]] autorelease];
    self.view.backgroundColor = [UIColor redColor];
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    CPPClass cppClass;
    cppClass.CPPFunction("This is a C++ function!");
    CFunction("This is a C function!");
    ShortCFunction("This is a short C function!");
}
@end

```

We've written a `CPPClass::CPPFunction`, a `CFunction`, and a `ShortCFunction` as our hooking targets. Here, we've intentionally added some useless code in `CPPClass::CPPFunction` and `CFunction` for the purpose of increasing the length of these two functions to validate `MSHookFunction`. However, `MSHookFunction` will fail on `ShortCFunction` because of its short length, and we have a plan B for this situation.

3. Modify Makefile and install the tweak:

```

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

```

```
include theos/makefiles/common.mk

APPLICATION_NAME = iOSRETargetApp
iOSRETargetApp_FILES = main.m iOSRETargetAppApplication.mm RootViewController.mm
iOSRETargetApp_FRAMEWORKS = UIKit CoreGraphics

include $(THEOS_MAKE_PATH)/application.mk

after-install::
    install.exec "su mobile -c uicache"
```

In the above code, “su mobile - C uicache” is used to refresh the UI cache of SpringBoard so that iOSRETargetApp’s icon can be shown on SpringBoard. Run “make package install” in Terminal to install this tweak on the device. Launch iOSRETargetApp, ssh into iOS after the red background shows, and see whether it outputs as expected:

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CPPFunction: This is a C++
function!
Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CFunction: This is a C function!
Nov 18 11:13:35 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CPPFunction: This is a short C
function!
```

4. Create iOSREHookerTweak with Theos, the commands are as follows:

```
snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/library
[3.] iphone/preference_bundle
[4.] iphone/tool
[5.] iphone/tweak
Choose a Template (required): 5
Project Name (required): iOSREHookerTweak
Package Name [com.yourcompany.iosrehookertweak]: com.iosre.iosrehookertweak
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.iosre.iosretargetapp
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: iOSRETargetApp
Instantiating iphone/tweak in iosrehookertweak/...
Done.
```

5. Modify Tweak.xm as follows:

```
#import <substrate.h>

void (*old__ZN8CPPClass11CPPFunctionEPKc)(void *, const char *);

void new__ZN8CPPClass11CPPFunctionEPKc(void *hiddenThis, const char *arg0)
{
    if (strcmp(arg0, "This is a short C function!") == 0)
        old__ZN8CPPClass11CPPFunctionEPKc(hiddenThis, "This is a hijacked short C function from
new__ZN8CPPClass11CPPFunctionEPKc!");
}
```



```

        else old__ZN8CPPClass11CPPFunctionEPKc(hiddenThis, "This is a hijacked C++
function!");
    }

void (*old_CFunction)(const char *);

void new_CFunction(const char *arg0)
{
    old_CFunction("This is a hijacked C function!"); // Call the original CFunction
}

void (*old_ShortCFunction)(const char *);

void new_ShortCFunction(const char *arg0)
{
    old_CFunction("This is a hijacked short C function from new_ShortCFunction!"); //
Call the original ShortCFunction
}

%ctor
{
    @autoreleasepool
    {
        MSImageRef image =
MSGetImageByName("/Applications/iOSRETargetApp.app/iOSRETargetApp");
        void *__ZN8CPPClass11CPPFunctionEPKc = MSFindSymbol(image,
"__ZN8CPPClass11CPPFunctionEPKc");
        if (__ZN8CPPClass11CPPFunctionEPKc) NSLog(@"iOSRE: Found CPPFunction!");
        MSHookFunction((void *)__ZN8CPPClass11CPPFunctionEPKc, (void
*)&new__ZN8CPPClass11CPPFunctionEPKc, (void **)&old__ZN8CPPClass11CPPFunctionEPKc);

        void *_CFunction = MSFindSymbol(image, "_CFunction");
        if (_CFunction) NSLog(@"iOSRE: Found CFunction!");
        MSHookFunction((void *)_CFunction, (void *)&new_CFunction, (void
**)&old_CFunction);

        void *_ShortCFunction = MSFindSymbol(image, "_ShortCFunction");
        if (_ShortCFunction) NSLog(@"iOSRE: Found ShortCFunction!");
        MSHookFunction((void *)_ShortCFunction, (void *)&new_ShortCFunction, (void
**)&old_ShortCFunction); // This MSHookFuntion will fail because ShortCFunction is too
short to be hooked
    }
}

```

In the above code, we should pay extra attention to some points:

- The use of MSFindSymbol

Simply put, the role of MSFindSymbol is to search the symbol to be hooked. Well, what's a symbol?

In computer, the instructions of a function are stored in memory. When the process is going to call the function, it needs to know where to locate the function in memory, and then executes its instructions at there. That is to say, the process needs to know the memory address of a function according to its name. The mapping of function names and addresses is stored in the

“symbol table”. “symbol” is the name of the function, according to which the process locates the function’s address in memory and then jumps there to execute it.

Imagine such a scenario: Your App calls a lookup function in a dylib to query information on your server. If another App gets to know the symbol of “lookup”, then it can import the dylib, and call the function as it wishes, causing great consumption of your server resources.

To avoid this, symbols are divided into 2 types, i.e. public symbols and private symbols (Besides, there are stripped symbols, but they have little to do with this chapter. If you are interested in stripped symbols, please visit the following reference links or google by yourselves). Private symbols are not property of yours, you can not make use of them as you wish. That’s to say, MSHookFunction will fail on private symbols without further manipulation. So saurik provides the MSFindSymbol function to access private symbols. If the concept of symbol is still beyond comprehension, just keep the following code pattern in mind:

```
MSImageRef image =  
MSGetImageByName("/path/to/binary/who/contains/the/implementation/of/symbol");  
void *symbol = MSFindSymbol(image, "symbol");
```

The parameter of MSGetImageByName is “The full path of the binary which contains the implementation of the function”. For example, the implementation of NSLog is in the Foundation framework, so the parameter should be

“/System/Library/Frameworks/Foundation.framework/Foundation”. Get it?

You can refer to the official document at

<http://www.cydiasubstrate.com/api/c/MSFindSymbol/> for a more detailed explanation of MSFindSymbol. As for the types and definition of symbols, please read

[http://msdn.microsoft.com/en-us/library/windows/hardware/ff553493\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff553493(v=vs.85).aspx) and

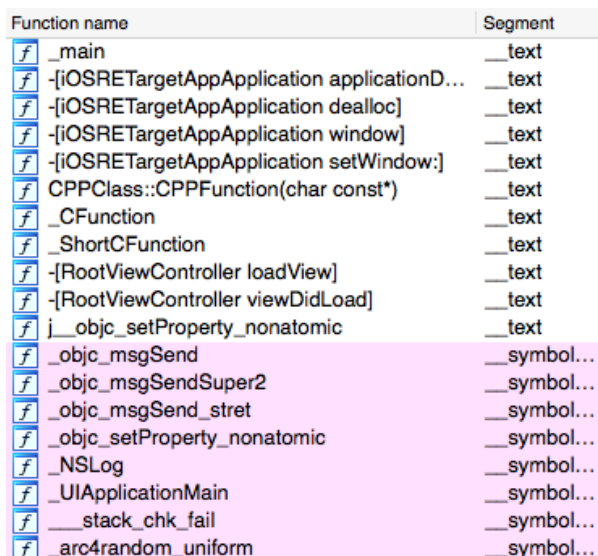
http://en.wikibooks.org/wiki/Reverse_Engineering/Mac_OS_X#Symbols_Types.

- The origin of a symbol

You may have already noticed that, the functions we defined in RootViewController.mm were CPPClass::CPPFunction, CFunction and ShortCFunction. How did they change into __ZN8CPPClass11CPPFunctionEPKc, _CFunction and _ShortCFunction respectively in Tweak.xm? In brief, that was because the compiler “mangled” (changed) the function name. It’s unnecessary here for us to know how every name is mangled, we are only concerned with the results. Where does these 3 underline prefixed symbols come from? In reverse engineering, normally we don’t have the right to access the source code of our targets, so these symbols are

all extracted via IDA, as illustrated in this example.

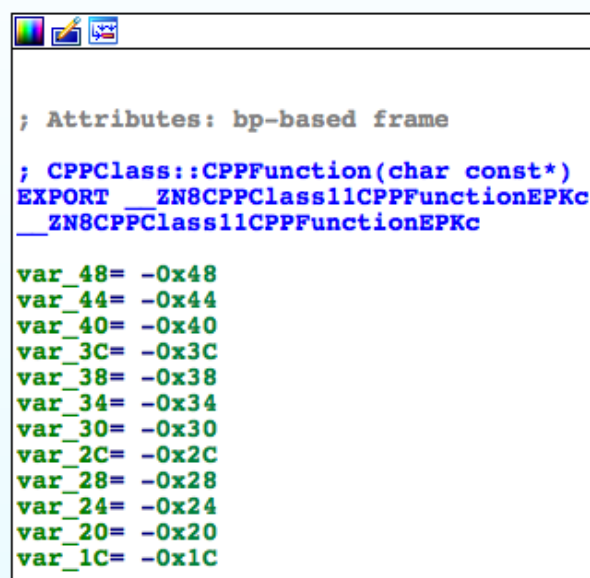
Drag and drop iOSRETargetApp’s binary into IDA. The Functions window after initial analysis is shown in figure 4-4.



Function name	Segment
_main	__text
-[iOSRETargetAppApplication applicationD...	__text
-[iOSRETargetAppApplication dealloc]	__text
-[iOSRETargetAppApplication window]	__text
-[iOSRETargetAppApplication setWindow:]	__text
CppClass::CPPFunction(char const*)	__text
_CFunction	__text
_ShortCFunction	__text
-[RootViewController loadView]	__text
-[RootViewController viewDidLoad]	__text
j_objc_setProperty_nonatomic	__text
_objc_msgSend	__symbol...
_objc_msgSendSuper2	__symbol...
_objc_msgSend_stret	__symbol...
_objc_setProperty_nonatomic	__symbol...
_NSLog	__symbol...
_UIApplicationMain	__symbol...
__stack_chk_fail	__symbol...
_arc4random_uniform	__symbol...

Figure 4- 4 Functions window

As we can see, CCppClass::CPPFunction(char const*), _CFunction and _ShortCFunction are listed here. Double click “CppClass::CPPFunction(char const*)” to go to its implementation, as shown in figure 4-5.



```
; Attributes: bp-based frame
; CCppClass::CPPFunction(char const*)
EXPORT __ZN8CppClass11CPPFunctionEPKc
__ZN8CppClass11CPPFunctionEPKc

var_48= -0x48
var_44= -0x44
var_40= -0x40
var_3C= -0x3C
var_38= -0x38
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C
```

Figure 4- 5 CCppClass::CPPFunction(char const*)

The underline prefixed string in line 4 is exactly the symbol we’re looking for. In the same way, where _CFunction and _ShortCFunction come from is obviously shown in figure 4-6 and figure 4-7.

```
; Attributes: bp-based frame

EXPORT _CFunction
_CFunction

var_48= -0x48
var_44= -0x44
var_40= -0x40
var_3C= -0x3C
var_38= -0x38
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C
```

Figure 4- 6 CFunction

```
EXPORT _ShortCFunction
_ShortCFunction
MOV          R1, R0
```

Figure 4- 7 ShortCFunction

This approach of symbol locating applies to all kinds of symbols. In the beginning stage, we suggest you keep in mind that a symbol and its corresponding function name are different, while ignore the hows and whys. During your whole process of studying reverse engineering, the concept of symbol will imperceptibly goes into your knowledge system, thus there is no need to push it for now.

- The writing pattern of MSHookFunction

The 3 parameters of MSHookFunction are: the original function to be hooked/replaced, the replacement function, and the original function saved by MobileHooker. Just like Sherlock Holmes needs Dr. Watson's assistance, MSHookFunction doesn't work alone, it only functions with a conventional writing pattern, shown as follows:

```
#import <substrate.h>

returnType (*old_symbol)(args);

returnType new_symbol(args)
{
    // Whatever
}
```

```

void InitializeMSHookFunction(void) // This function is often called in %ctor i.e.
constructor
{
    MSImageRef image =
MSGetImageByName("/path/to/binary/who/contains/the/implementation/of/symbol");
    void *symbol = MSFindSymbol(image, "symbol");
    if (symbol) MSHookFunction((void *)symbol, (void *)&new_ symbol, (void **)&old_
symbol);
    else NSLog(@"Symbol not found!");
}

```

You'll recognize this pattern if you review `Tweak.xm` in `iOSREHookerTweak`. Again, we cannot get the source code of the function to hook, so we don't know the prototype of the function: What is the returnType? How many args are there and what're their types? At this moment, we need the help of more advanced reverse engineering skills to reconstruct the prototype of the function. Chapter 6 focuses on this knowledge, so don't worry if you can't catch up for now. I strongly suggest you review this section after finishing chapter 6, I bet you will get a better understanding at that time.

6. Modify Makefile and install the tweak:

```

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

include theos/makefiles/common.mk

TWEAK_NAME = iOSREHookerTweak
iOSREHookerTweak_FILES = Tweak.xm

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 iOSRETargetApp"

```

Now please relaunch `iOSRETargetApp` and see if the output matches our expectation:

```

FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found CPPFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found CFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found ShortCFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CPPFunction: This is a hijacked
C++ function!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CFunction: This is a hijacked C
function!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CPPFunction: This is a hijacked
short C function from new__ZN8CPPClass11CPPFunctionEPKc!

```

It is worth mentioning that, we failed hooking the short function (i.e. `ShortCFunction`), otherwise it would print "This is a hijacked short C function from `new_ShortCFunction!`". But we succeeded in hooking other functions (i.e. `CPPClass::CPPFunction`) inside the short

function. We could tell if the caller was ShortCFuncation by judging the callee's argument, thus indirectly hooked short function and met our needs. The introduction of MSHookFunction above covers almost every situation a beginner may encounter. Since Theos only provides encapsulation for MSHookMessageEx, thorough understanding of the use of MSHookFunction is particularly important. If MSHookFunction still confuses you, get to us on <http://bbs.iosre.com>.

4.1.2 MobileLoader

The role of MobileLoader is to load third-party dylibs. When iOS launches, launchd will load MobileLoader into memory, then MobileLoader will call dlopen according to tweaks' plist filters to load dylibs under `/Library/MobileSubstrate/DynamicLibraries/` into different processes. The format of the plist filter here has been explained in details in the previous Theos section, which saves my words here. For most rookie iOS reverse engineers, MobileLoader works transparently, knowing the existence of it is enough.

4.1.3 Safe mode

iOS crashes when tweak sucks. A tweak is essentially a dylib residing in another process, once something goes wrong in it, the entire process crashes. If it unfortunately happens to be SpringBoard or other system processes, tweak crash leads to a system paralysis. So CydiaSubstrate introduces Safe Mode: It captures SIGTRAP, SIGABRT, SIGILL, SIGBUS, SIGSEGV and SIGSYS signals, then enter safe mode, as shown in figure 4-8.

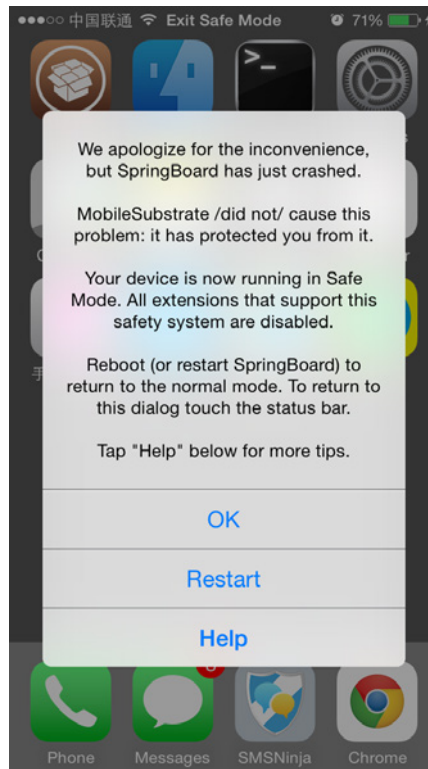


Figure 4- 8 Safe mode

In safe mode, all third-party tweaks that base on CydiaSubstrate will be disabled for troubleshooting. But safe mode can't guarantee absolute safety, in many cases, devices fail to boot because of problematic third-party dylibs. In this situation, you can perform a hard reboot by pressing and holding the home and lock buttons, then completely disable CydiaSubstrate by holding the volume "+" button. After iOS successfully reboots, you can start error checking. When the problems are fixed, reboot iOS again to enable CydiaSubstrate.

4.2 Cypcript



Figure 4- 9 Cypcript

Cypcript (As shown in figure 4-9) is a scripting language developed by saurik. You can view Cypcript as Objective-JavaScript. A lot of you may not be familiar with JavaScript, then subconsciously think Cypcript is very obscure. In fact, I, as a lazy learner, do not know JavaScript either, so in a long time, I've ignored Cypcript deliberately. It wasn't until not long ago when I was playing with MTerminal during my company's boring meeting and tested a few Objective-C methods in Cypcript, then I had a new awareness of this simple yet powerful language. In fact, for Objective-C programmers, scripting languages are not difficult to use, as long as we overcome our fear of difficulty, we will be able to handle them very quickly, and Cypcript is no exception. Cypcript has the convenience of scripting language, you can even write App in Cypcript, but saurik himself said, "This isn't quite 'ready for primetime'". In my humble opinion, the most practical usage of Cypcript is testing private methods in an easy manner, possessing both safety and efficiency. Therefore, this book will only use Cypcript to test methods. For its complete manual, please visit the official website <http://www.cypcript.org>.

We can launch Cypcript either in MTerminal or via ssh. Input "cypcript" and it outputs "cy#", which indicates Cypcript's successful launch.

```
FunMaker-5:~ root# cypcript
cy#
```


After that, you can start coding. Instead of writing Apps, we mainly use Cycrypt to test methods, so we need to inject and run code in an existing process. Let's exit Cycrypt by pressing "control + D" for now. Generally speaking, which process to inject depends on what methods we're testing: Suppose the methods to be tested are from class A, and class A exists in process B, then you should inject into process B to test the methods. Stop beating around the bush, let's see an example to make everything more straightforward.

If now we want to test the class method `+sharedNumberFormatter` in class `PhoneApplication` to reconstruct its prototype, we have to inject into the process `MobilePhone` because `PhoneApplication` only exists in `MobilePhone`; Similarly, for the instance method `[SBUIController lockFromSource:]`, we have to inject into `SpringBoard`; Naturally, for `[NSString length]`, we can inject into any process that imports `Foundation.framework`. Because most of the methods we test are private, so the general rules are that if the methods you're testing are from a process, inject right into that process; If they're from a lib, inject into the processes that import this lib.

Testing methods via process injection is rather simple. Take `SpringBoard` for an example, first we need to find out its process name or process ID (PID):

```
FunMaker-5:~ root# ps -e | grep SpringBoard
4567 ??      0:27.45 /System/Library/CoreServices/SpringBoard.app/SpringBoard
4634 ttys000   0:00.01 grep SpringBoard
```

As we can see, `SpringBoard`'s PID is 4634. Input "cycrypt -p 4634" or "cycrypt -p `SpringBoard`" to inject Cycrypt into `SpringBoard`. Now Cycrypt has been injected into `SpringBoard` and we can start method testing.

`UIAlertView` is a most frequently used UI class on iOS. Only 3 lines of code in Objective-C are needed for a popup:

```
UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"iOSRE"
message:@"snakeninny" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
[alertView show];
[alertView release];
```

It's easy to convert the above Objective-C code into Cycrypt code:

```
FunMaker-5:~ root# cycrypt -p SpringBoard
cy# alertView = [[UIAlertView alloc] initWithTitle:@"iOSRE" message:@"snakeninny"
delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil]
#<UIAlertView: 0x1700e580; frame = (0 0; 0 0); layer = <CALayer: 0x164146c0>>"
cy# [alertView show]
cy# [alertView release]
```

No need to declare the type of an object, no need to add a semicolon at the end of each line,

that's Cycrypt. If a function has a return value, Cycrypt will print its memory address and description in real time, which is very intuitive. After Cycrypt executes the above code, a popup shows on SpringBoard, as shown in figure 4-10.

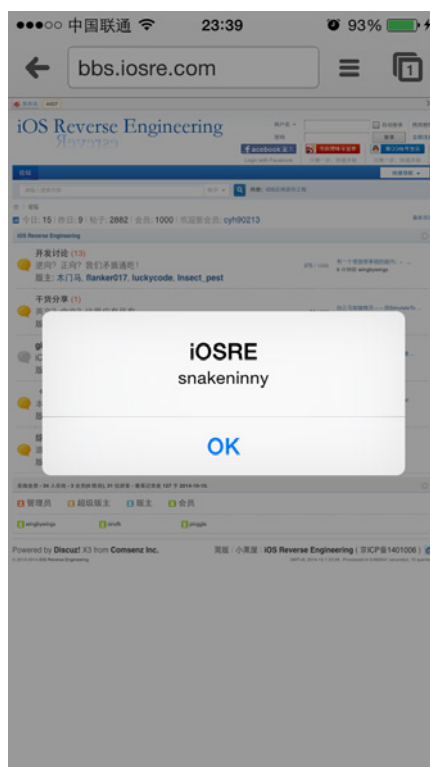


Figure 4- 10 Code execution in Cycrypt

If we already know the memory address of an object, we can use “#” operator to access the object like this:

```
cy# [[UIAlertView alloc] initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil  
cancelButtonTitle:@"OK" otherButtonTitles:nil]  
#<UIAlertView: 0x166b4fb0; frame = (0 0; 0 0); layer = <CALayer: 0x16615890>>"  
cy# [#0x166b4fb0 show]  
cy# [#0x166b4fb0 release]
```

If we know an object of a certain class exists in the current process but don't know its memory address, we cannot obtain the object with “#”. Under such circumstance, we can try “choose” out:

```
cy# choose(SBScreenShotter)  
[#<SBScreenShotter: 0x166e0e20>]"  
cy# choose(SBUIController)  
[#<SBUIController: 0x16184bf0>]"
```

“choose” a class, Cycrypt returns its objects. This command is so convenient that it doesn't succeed all the time. When it fails to get you a usable object, you're on your own. We'll talk about how to get our target objects manually in chapter 6, please stay tuned.

When it comes to testing private methods, a combination of the above Cycrypt commands

would be enough. Let's summarize the use of Cycrypt with an example of logging in to iMessage with my Apple ID. First we need to get an instance of iMessage login controller:

```
FunMaker-5:~ root# cycrypt -p SpringBoard
cy# controller = [CNFRegController controllerForServiceType:1]
#"<CNFRegController: 0x166401e0>"
```

Then login with my Apple ID:

```
cy# [controller beginAccountSetupWithLogin:@"snakeninny@gmail.com"
password:@"bbs.iosre.com" foundExisting:NO]
#"IMAccount: 0x166e7b30 [ID: 5A8E19BE-1BC9-476F-AD3B-729997FAA3BC Service:
IMService[iMessage] Login: E:snakeninny@gmail.com Active: YES LoginStatus: Connected]"
```

This is an equivalent of logging in iMessage as shown in figure 4-11.

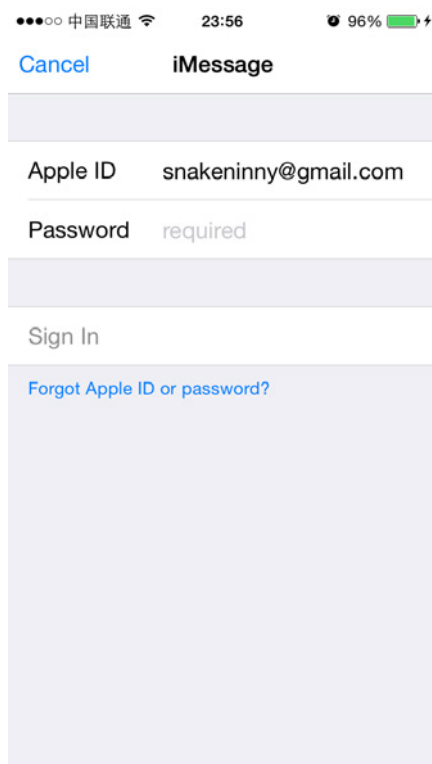


Figure 4- 11 Log in iMessage

This method returns a logged in IMAccount, i.e my iMessage account. Then select the addresses for sending and receiving iMessages:

```
cy# [controller setAliases:@[@"snakeninny@gmail.com"] onAccount:#0x166e7b30]
1
```

This is an equivalent of selecting iMessage addresses as shown in figure 4-12.

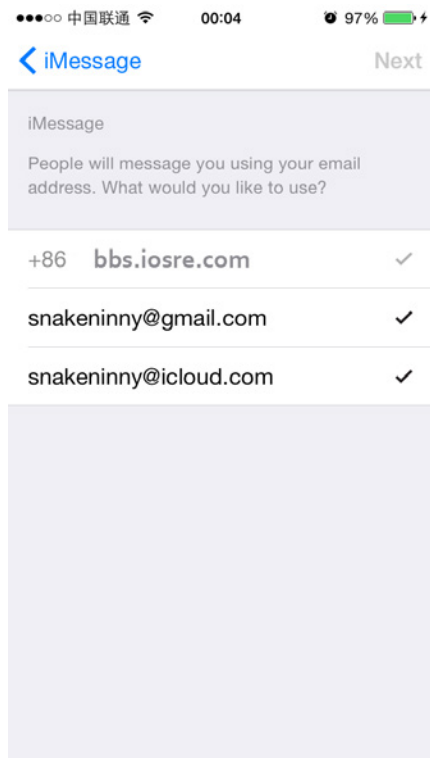


Figure 4- 12 Select iMessage addresses

The return value indicates our correctness by far. Finally, let's check if my iMessage account is ready to rock!

```
cy# [#0x166e7b30 CNFRegSignInComplete]
1
```

1 in number is YES in BOOL. We can start iMessaging others right now.

Simple and clear, right? No further explanation needed. As the exercise of this section, now it's your turn to convert the above Cycrypt code into Objective-C code, and write a tweak to verify your conversion as well get familiar with Cycrypt. One last note, remember to change my Apple ID to yours.

4.3 LLDB and debugserver

4.3.1 Introduction to LLDB

If IDA is caliburn, then LLDB is excalibur, they are at roughly the same position in iOS reverse engineering. LLDB, a production of Apple, stands for "Low Level Debugger". It's the Xcode built-in dynamic debugger supporting C, C++ and Objective-C, working on OSX, iOS and the iOS simulator.

LLDB's functionality sums up in 4 points:

- Launch the program under the conditions you specify;
- Stop the program under the conditions you specify;
- Inspect the internal status of a program when it stops;
- Modify the program when it stops, and observe the modification of its execution flow.

LLDB is a command line tool, it does not have a graphical interface. Its mass output in Terminal scares off beginners easily, but once you master the basic commands of LLDB, you'll be surprised by its formidable combination with IDA. LLDB runs in OSX, so to debug iOS, we need another tool's assistance on iOS, which is debugserver.

4.3.2 Introduction to debugserver

debugserver runs on iOS. As its name suggests, it plays the role of a server and executes the commands from LLDB (as a client), then returns the results to LLDB to show to the user. This working mode is called "remote debugging". By default, debugserver is not installed on iOS. We need to connect the device to Xcode, configure it to enable debugging in menu Window → Devices, then debugserver will be installed to "/Developer/usr/bin/" on iOS.

However, because of the lack of task_for_pid permission, the raw debugserver installed by Xcode can only debug our own Apps. Debugging our own Apps is no mystery in App development, and since we have our own Apps' source code, there is no need to reverse them. It'd only be cool if we can debug other Apps. No worry, here comes the solution. With a little hacking, debugserver and LLDB can be used to debug other Apps, maximizing their power.

4.3.3 Configure debugserver

1. Help debugserver lose some weight

Find the corresponding ARM type of your device according to table 4-1.

Name	ARM
iPhone 4s	armv7
iPhone 5	armv7s
iPhone 5c	armv7s
iPhone 5s	arm64
iPhone 6 Plus	arm64
iPhone 6	arm64

iPad 2	armv7
iPad mini	armv7
The New iPad	armv7
iPad with Retina display	armv7s
iPad Air	arm64
iPad Air 2	arm64
iPad mini with Retina display	arm64
iPad mini 3	arm64
iPod touch 5	armv7

Table 4-1 iOS 8 Compatible devices

My device is iPhone 5, its matching ARM type is armv7s. Copy the raw debugserver from iOS to “/Users/snakeninny/” on OSX.

```
snakeninnysiMac:~ snakeninny$ scp root@iOSIP:/Developer/usr/bin/debugserver
~/debugserver
```

Then help it lose some weight:

```
snakeninnysiMac:~ snakeninny$ lipo -thin armv7s ~/debugserver -output ~/debugserver
```

Note that you need to change “armv7s” here to the corresponding ARM type of your device.

- Grant task_for_pid permission to debugserver

Download <http://iosre.com/ent.xml> to “/Users/snakeninny/” on OSX, then run the following command:

```
snakeninnysiMac:~ snakeninny$ /opt/theos/bin/ldid -Sent.xml debugserver
```

Note, there is no space between “-S” and “ent.xml”.

If everything goes fine, ldid will take less than 5 seconds to finish its job. But if ldid gets stuck and times out, just try another workaround: Download <http://iosre.com/ent.plist> to “/Users/snakeninny/”, then run the following command:

```
snakeninnysiMac:~ snakeninny$ codesign -s - --entitlements ent.plist -f debugserver
```

- Copy the modified debugserver back to iOS

Copy the modified debugserver to iOS and grant it execute permission with the following commands:

```
snakeninnysiMac:~ snakeninny$ scp ~/debugserver root@iOSIP:/usr/bin/debugserver
snakeninnysiMac:~ snakeninny$ ssh root@iOSIP
```

```
FunMaker-5:~ root# chmod +x /usr/bin/debugserver
```

One thing to clarify, the reason we put the modified debugserver under “/usr/bin/” instead of overriding the original one is because, first, the original debugserver is not writable, we just cannot override it; Second, we don’t need to input full paths to execute commands under “/usr/bin/”, just run “debugserver” wherever you want, and debugserver is ready to roll out.

4.3.4 Process launching and attaching using debugserver

2 most commonly used scenarios of debugserver are process launching and attaching. Both possess very simple commands:

```
debugserver -x backboard IP:port /path/to/executable
```

debugserver will launch the specific executable and open the specific port, then wait for LLDB’s connection from IP.

```
debugserver IP:port -a "ProcessName"
```

debugserver will attach to process with the name “ProcessName” and open the specific port, then wait for LLDB’s connection from IP.

For example:

```
FunMaker-5:~ root# debugserver -x backboard *:1234 /Applications/MobileSMS.app/MobileSMS
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Listening to port 1234 for a connection from *...
```

The above command will launch MobileSMS and open port 1234, then wait for LLDB’s connection from any IP. And for the following command:

```
FunMaker-5:~ root# debugserver 192.168.1.6:1234 -a "MobileSMS"
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process MobileNotes...
Listening to port 1234 for a connection from 192.168.1.6...
```

debugserver will attach to MobileSMS and open port 1234, then wait for LLDB’s connection from 192.168.1.6.

If something goes wrong when executing the above commands, such as:

```
FunMaker-5:~ root# debugserver *:1234 -a "MobileSMS"
dyld: Library not loaded:
/Developer/Library/PrivateFrameworks/ARMDisassembler.framework/ARMDisassembler
Referenced from: /usr/bin/debugserver
Reason: image not found
Trace/BPT trap: 5
```

It means necessary debugging data under “/Developer/” is missing. This is generally because we did not enable development mode on this device in Xcode’s Window → Devices

menu. You can fix the issue by re-enabling development mode on this device.

When you exit debugserver, the process being debugged also exits. The configuration of debugserver is over for now, the following operation are performed on LLDB.

4.3.5 Use LLDB

Before introducing LLDB, we need to know a big bug in the latest LLDB: LLDB (version 320.x.xx) in Xcode 6 sometimes messes up ARM with THUMB instructions on armv7 and armv7s devices, making it impossible to debug. Before the publishing of this book, the bug has not been fixed yet. A temporary solution is to download and install Xcode 5.0.x from <https://developer.apple.com/downloads/index.action>, their built-in LLDB (version 300.x.xx) works fine on armv7 and armv7s devices. When you're installing the old version of Xcode, make sure you install it in a different path from the current Xcode, say “/Applications/OldXcode.app”, thus it won't affect the current Xcode. To launch the old LLDB, you need to specify the full path:

```
snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb
```

Then the old LLDB will launch and you can connect it to the waiting debugserver:

```
(lldb) process connect connect://iOSIP:1234
Process 790987 stopped
* thread #1: tid = 0xc11cb, 0x3995b4f0 libsystem_kernel.dylib`mach_msg_trap + 20, queue
= 'com.apple.main-thread, stop reason = signal SIGSTOP
  frame #0: 0x3995b4f0 libsystem_kernel.dylib`mach_msg_trap + 20:
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x3995b4f0: pop    {r4, r5, r6, r8}
   0x3995b4f4: bx     lr

libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x3995b4f8: mov    r12, sp
   0x3995b4fc: push  {r4, r5, r6, r8}
```

Note, the execution of “process connect connect://iOSIP:1234” will take a rather long time (approximately more than 3 minutes in a WiFi environment) to connect to debugserver, please be patient. In section 4.6, there will be an introduction to connecting to debugserver through USB, which will save a lot of time. When the process is stopped by debugserver, we can start debugging. Let's have a look at the commonly used commands in LLDB.

1. image list

“image list” is similar to “info shared” in GDB, which is used to list the main executable and all dependent libraries (hereinafter referred to as images) in the debugged process. Because of ASLR (Address Space Layout Randomization, see <http://theiphonewiki.com/wiki/ASLR>),

every time the process launches, a random offset will be added to the starting address of all images in that process, making their virtual memory addresses hard to predict.

For example, suppose there is an image B in process A, and image B is 100 bytes in size. When process A launches for the 1st time, image B may be loaded into virtual memory at 0x00 to 0x64; For the 2nd time, image B may be loaded into 0x10 to 0x74, and 0x60 to 0xC4 for the 3rd time. That is to say, although image B's size stays 100 bytes, every launch changes the starting address, which happens to be a key value in our following operations. Then comes the question, how do we get this key value?

The answer is "image list -o -f". After LLDB has connected to debugserver, run "image list -o -f" to view its output:

```
(lldb) image list -o -f
[ 0] 0x000cf000
/private/var/db/stash/_.29LMeZ/Applications/SMSNinja.app/SMSNinja(0x0000000000d3000)
[ 1] 0x0021a000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000021a000)
[ 2] 0x01645000 /usr/lib/libobjc.A.dylib(0x00000000307b5000)
[ 3] 0x01645000
/System/Library/Frameworks/Foundation.framework/Foundation(0x0000000023c4f000)
[ 4] 0x01645000
/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation(0x0000000022f0b000)
[ 5] 0x01645000 /System/Library/Frameworks/UIKit.framework/UIKit(0x00000000264c1000)
[ 6] 0x01645000
/System/Library/Frameworks/CoreGraphics.framework/CoreGraphics(0x0000000023238000)
.....
[235] 0x01645000
/System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib(0x00000000233a2000)
[236] 0x0008a000 /usr/lib/dyld(0x000000001fe8a000)
```

In the above output, the 1st column, [X], is the sequence number of the image; the 2nd column is the image's random offset generated by ASLR (hereinafter referred to as the ASLR offset); the 3rd column is the full path of this image, the content in brackets is the original starting address plus the ASLR offset. Do all these offsets and addresses confuse you? Take it easy, hopefully you'll sort it through after an example.

Suppose the virtual memory is a shooting range with 1000 target positions. You can regard the images in a process as targets and now there are 600 of them. All these targets are uniformly arranged in a row with target 1 in position 1, target 2 in position 2, target 600 in position 600, etc. And positions 601 to 1000 are all empty. You can see the layout in figure 4-13 (The number at the top is the target position number, and the target number is at the bottom).



Figure 4- 13 Shooting range (1)

The images' starting addresses in virtual memory are like the target positions of the 600 targets, which are named image base addresses in terminology. Now the owner of this shooting range thinks the previous targets are arranged rashly, shooters will hit all bulls' eyes as soon he gets familiar with the arrangement. So the owner relocates all these targets randomly. After relocation, target 1 is placed in position 5, target 2 is placed in position 6, target 3 is placed in position 8, target 4 is placed in position 13, target 5 is placed in position 15..... Target 600 is placed in position 886, as shown in figure 4-14.



Figure 4- 14 Shooting range (2)

That's to say, the offsets for target 1, 2, 3, 4, 5 and 600 are 4, 4, 5, 9, 10 and 286 respectively. This random (ASLR) offset greatly increases the shooting difficulty. For target 1, it used to be at position 1, and it is at position 5 for now, so the offset is 4, i.e.

$$\text{image base address with offset} = \text{image base address without offset} + \text{ASLR offset}$$

Back to the reverse engineering scene, let's take the 4th image (i.e. Foundation) in the output of "image list -o -f" as an example, its ASLR offset is 0x1645000, its image base address with offset is 0x23c4f000, so according to the above formula, its image base address without offset is $0x23c4f000 - 0x1645000 = 0x2260A000$.

You may wonder, where does 0x2260A000 come from? Drag and drop Foundation's binary into IDA, after the initial analysis, IDA looks like figure 4-15.

```

External symbol
IDA View-A Hex View-1 Structures Enums
HEADER:2260A000 ;
HEADER:2260A000 ; +-----+
HEADER:2260A000 ; | This file has been generated by The Interactive Disassembler (IDA) |
HEADER:2260A000 ; | Copyright (c) 2014 Hex-Rays, <support@hex-rays.com> |
HEADER:2260A000 ; | Evaluation version |
HEADER:2260A000 ; +-----+
HEADER:2260A000 ; Input MD5 : 8D58A456B36A39CBA810F25609BA4737
HEADER:2260A000 ; Input CRC32 : 49FAA649
HEADER:2260A000 ;
HEADER:2260A000 ; Processor : ARM
HEADER:2260A000 ; ARM architecture: metaarm
HEADER:2260A000 ; Target assembler: Generic assembler for ARM
HEADER:2260A000 ; Byte sex : Little endian
HEADER:2260A000 ;
HEADER:2260A000 ; =====
HEADER:2260A000 ; [00001320 BYTES: COLLAPSED SEGMENT HEADER. PRESS KEYPAD CTRL-"+" TO EXPAND]
HEADER:2260A000 ; =====
text:2260B320 ;

```

Figure 4- 15 Analyze Foundation in IDA

Scroll to the top of IDA View-A, do you see “HEADER:2260A000” in the first line? This is the origin of 0x2260A000.

Now that we’ve known “base address” means “starting address”, let’s talk about another concept which is similar to “image base address”, i.e. “symbol base address”. Return to IDA and search for “NSLog” in the Functions window, and then jump to its implementation, as shown in figure 4-16.

```

text:2261AB94
text:2261AB94 EXPORT NSLog
text:2261AB94 NSLog ; CODE XREF:
text:2261AB94 ; -[NSLock lo
text:2261AB94 var_18 = -0x18
text:2261AB94
text:2261AB94 SUB SP, SP, #0xC
text:2261AB96 PUSH {R7,LR}
text:2261AB98 MOV R7, SP
text:2261AB9A SUB SP, SP, #4
text:2261AB9C ADD.W R9, R7, #8
text:2261ABA0 STMIA.W R9, {R1-R3}
text:2261ABA4 ADD.W R1, R7, #8
text:2261ABA8 STR R1, [SP,#0x18+var_18]
text:2261ABAA BL NSLogv
text:2261ABAE ADD SP, SP, #4
text:2261ABB0 POP.W {R7,LR}
text:2261ABB4 ADD SP, SP, #0xC
text:2261ABB6 BX LR
text:2261ABB6 ; End of function NSLog

```

Figure 4- 16 NSLog

Because the base address of Foundation is a known number, and NSLog is in a fixed position inside Foundation, we can get the base address of NSLog according to the following formula:

$$\text{base address of NSLog} = \text{relative address of NSLog in Foundation} + \text{base address of Foundation}$$

What does “relative address of NSLog in Foundation” mean? Let’s go back to figure 4-16 and find the first instruction of NSLog, i.e. “SUB SP, SP, #0xC”. On the left, do you see the number 0x2261AB94? This the “address of NSLog in Foundation”. Subtract Foundation’s image base address without offset, i.e. 0x2260A000 from it, we get the “relative address of NSLog in Foundation”, i.e. 0x10B94.

Hence, the base address of NSLog is $0x10B94 + 0x23c4f000 = 0x23C5FB94$. I guess some of you have already noticed that the formula

```
image base address with offset = image base address without offset + ASLR offset
```

With tiny modifications, is a new formula for symbols:

```
symbol base address with offset = symbol base address without offset +  
ASLR offset of the image containing the symbol
```

Let's verify this formula.

NSLog's symbol base address without offset is $0x2261AB94$, ASLR offset of Foundation is $0x1645000$, add these two numbers and we get $0x23C5FB94$.

By analogy, we can also get the formula for instructions:

```
instruction base address with offset = instruction base address without offset +  
ASLR offset of the image containing the instruction
```

Naturally, symbol base address is the base address of the first instruction of the symbol's corresponding function.

In the following content, base addresses with offset will be frequently used. Make sure you understand all concepts in this section then keep in mind: Base address without offset can be viewed in IDA, ASLR offset can be viewed in LLDB, add them together we get base address with offset. As for where in IDA and LLDB to search for the values, I bet you'll get it after thoroughly reading this section.

2. breakpoint

"breakpoint" is similar to "break" in GDB, it's used to set breakpoints. In reverse engineering, we usually set breakpoints like these:

```
b function
```

Or

```
br s -a address
```

Or

```
br s -a 'ASLRoffset+address'
```

The former command is to set a breakpoint at the beginning of a function, for instance:

```
(lldb) b NSLog  
Breakpoint 2: where = Foundation`NSLog, address = 0x23c5fb94
```

The latter two commands are to set a breakpoint at a specific address, for instance:

```
(lldb) br s -a 0xCCCCC  
Breakpoint 5: where = SpringBoard`___lldb_unnamed_function3033$$SpringBoard, address =  
0x000cccc
```

```
(lldb) br s -a '0x6+0x9'
Breakpoint 6: address = 0x0000000f
```

Note that the “X” in the output “Breakpoint X:” is an integer id of that breakpoint, and we will use this number soon. When the process stops at a breakpoint, the line of code holding the breakpoint hasn’t been executed yet.

In reverse engineering, we’ll be debugging assembly code, so in most cases we’ll be setting breakpoint on a specific assembly instruction instead of a function. To set a breakpoint on an assembly instruction, we have to know its base address with offset, which we have already explained in details. Now let’s take `[SpringBoard _menuButtonDown:]` for an example and set a breakpoint on the first instruction as a demonstration.

- Find the base address without offset in IDA

Open SpringBoard’s binary in IDA, switch to Text view after the initial analysis and locate “`[SpringBoard _menuButtonDown:]`”, as shown in figure 4-17.

```
text:00017730 ; SpringBoard - (void) _menuButtonDown:(struct __IOHIDEvent *)
text:00017730 ; Attributes: bp-based frame
text:00017730
text:00017730 ; void __cdecl _[SpringBoard _menuButtonDown:](struct SpringB
text:00017730 _SpringBoard__menuButtonDown__ ; DATA XREF: __objc_c
text:00017730
text:00017730 var_68 = -0x68
text:00017730 var_64 = -0x64
text:00017730 var_60 = -0x60
text:00017730 var_5C = -0x5C
text:00017730 var_58 = -0x58
text:00017730 var_54 = -0x54
text:00017730 var_50 = -0x50
text:00017730 var_4C = -0x4C
text:00017730 var_48 = -0x48
text:00017730 var_44 = -0x44
text:00017730 var_40 = -0x40
text:00017730 var_3C = -0x3C
text:00017730 var_38 = -0x38
text:00017730 var_34 = -0x34
text:00017730 var_30 = -0x30
text:00017730 var_2C = -0x2C
text:00017730 var_28 = -0x28
text:00017730 var_24 = -0x24
text:00017730 var_20 = -0x20
text:00017730 var_1C = -0x1C
text:00017730
text:00017730 PUSH {R4-R7, LR}
text:00017732 ADD R7, SP, #0xC
```

Figure 4- 17 `[SpringBoard _menuButtonDown:]`

As we can see, the base address without offset of the first instruction “`PUSH {R4-R7, LR}`” is `0x17730`.

- Find the ASLR offset in LLDB

ssh into iOS to run debugserver with the following commands:

```
snakeninnysMac:~ snakeninny$ ssh root@iOSIP
FunMaker-5:~ root# debugserver *:1234 -a "SpringBoard"
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process SpringBoard...
Listening to port 1234 for a connection from *...
```

Then connect to debugserver with LLDB on OSX, and find the ASLR offset:

```
snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb
(lldb) process connect connect://iOSSIP:1234
Process 93770 stopped
* thread #1: tid = 0x16e4a, 0x30dee4f0 libsystem_kernel.dylib`mach_msg_trap + 20, queue
= 'com.apple.main-thread, stop reason = signal SIGSTOP
    frame #0: 0x30dee4f0 libsystem_kernel.dylib`mach_msg_trap + 20:
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x30dee4f0: pop    {r4, r5, r6, r8}
    0x30dee4f4: bx     lr

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x30dee4f8: mov    r12, sp
    0x30dee4fc: push  {r4, r5, r6, r8}
(lldb) image list -o -f
[ 0] 0x000b5000
/System/Library/CoreServices/SpringBoard.app/SpringBoard(0x00000000000b9000)
[ 1] 0x006ea000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000006ea000)
[ 2] 0x01645000
/System/Library/PrivateFrameworks/StoreServices.framework/StoreServices(0x000000002ca700
00)
[ 3] 0x01645000
/System/Library/PrivateFrameworks/AirTraffic.framework/AirTraffic(0x0000000027783000)
.....
[419] 0x00041000 /usr/lib/dyld(0x000000001fe41000)
(lldb) c
Process 93770 resuming
```

The ASLR offset of SpringBoard is 0xb5000.

- Set and trigger the breakpoint

So the base address with offset of the first instruction is $0x17730 + 0xb5000 = 0xCC730$.

Input “br s -a 0xCC730” in LLDB to set a breakpoint on the first instruction:

```
(lldb) br s -a 0xCC730
Breakpoint 1: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard, address =
0x000cc730
```

Then press the home button to trigger the breakpoint:

```
(lldb) br s -a 0xCC730
Breakpoint 1: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard, address =
0x000cc730
Process 93770 stopped
* thread #1: tid = 0x16e4a, 0x000cc730
SpringBoard`___lldb_unnamed_function299$$SpringBoard, queue = 'com.apple.main-thread,
stop reason = breakpoint 1.1
    frame #0: 0x000cc730 SpringBoard`___lldb_unnamed_function299$$SpringBoard
SpringBoard`___lldb_unnamed_function299$$SpringBoard:
-> 0xcc730: push  {r4, r5, r6, r7, lr}
    0xcc732: add   r7, sp, #12
    0xcc734: push.w {r8, r10, r11}
    0xcc738: sub   sp, #80
(lldb) p (char *)$r1
(char *) $0 = 0x0042f774 "_menuButtonDown:"
```

When the process stops, you can use “c” command to “continue” (running) the process.

Compared to GDB, a significant improvement in LLDB is that you can enter commands while the process is running. But be careful, some processes (such as SpringBoard) will automatically relaunch because of timeout after stopping for a period of time. For this kind of processes, you should try to keep it running to avoid unexpected automatic relaunching.

You can also use commands like “br dis”, “br en” and “br del” to disable, enable and delete breakpoints. The command to disable all breakpoints is as follows:

```
(lldb) br dis
All breakpoints disabled. (2 breakpoints)
```

The command to disable a specific breakpoint is as follows:

```
(lldb) br dis 6
1 breakpoints disabled.
```

The command to enable all breakpoints is as follows:

```
(lldb) br en
All breakpoints enabled. (2 breakpoints)
```

The command to enable a specific breakpoint is as follows:

```
(lldb) br en 6
1 breakpoints enabled.
```

The command to delete all breakpoints is as follows:

```
(lldb) br del
About to delete all breakpoints, do you want to do that?: [Y/n] Y
```

The command to delete a specific breakpoint is as follows:

```
(lldb) br del 8
1 breakpoints deleted; 0 breakpoint locations disabled.
```

Another useful command is that we can set a series of commands on a breakpoint to be automatically executed when we hit the breakpoint. Suppose breakpoint 1 is set on a specific objc_msgSend function, the commands to set a series of commands on breakpoint 1 are as follows:

```
(lldb) br com add 1
```

After executing the above command, LLDB will ask for a series of commands, ending with “DONE”.

```
Enter your debugger command(s). Type 'DONE' to end.
> po [$r0 class]
> p (char *)$r1
> c
> DONE
```

Here we’ve input 3 commands, once breakpoint 1 is hit, LLDB will execute them one by one:

```
(lldb) c
```



```

Process 97048 resuming
__NSArrayM
(char *) $11 = 0x26c6bbc3 "count"
Process 97048 resuming
Command #3 'c' continued the target.

```

“br com add” is often used to automatically observe the changes in the context of a breakpoint when it is hit, which often implies valuable reverse engineering clues. We’ll see how to use this command in the latter half of this book.

3. print

Thanks to “print” command, “inspecting the internal status of a program when it stops” is possible. As its name implies, this command can print the value of a register, variable, expression, etc. Again, let’s illustrate the use of “print” with “-[SpringBoard _menuButtonDown:]”, as shown in figure 4-18.

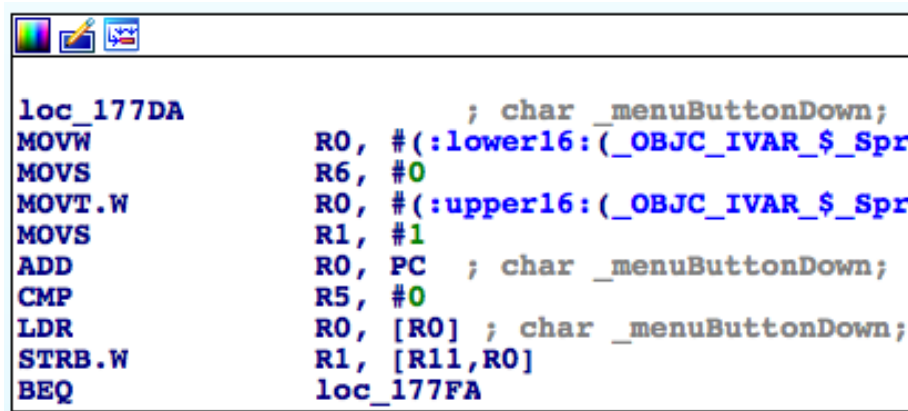


Figure 4- 18 [SpringBoard _menuButtonDown:]

The base address with offset of “MOVS R6, #0” is known to be 0xE37DE, let’s set a breakpoint on it and print R6’s value when we hit the breakpoint:

```

(lldb) br s -a 0xE37DE
Breakpoint 2: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard + 174,
address = 0x000e37de
Process 99787 stopped
* thread #1: tid = 0x185cb, 0x000e37de
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 174, queue = 'com.apple.main-
thread, stop reason = breakpoint 2.1
   frame #0: 0x000e37de SpringBoard`___lldb_unnamed_function299$$SpringBoard + 174
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 174:
-> 0xe37de: movs   r6, #0
   0xe37e0: movt   r0, #75
   0xe37e4: movs   r1, #1
   0xe37e6: add    r0, pc
(lldb) p $r6
(unsigned int) $1 = 364526080

```

After this instruction is executed, R6 should be set to 0. Input “ni” to execute this instruction

and reprint the value of R6:

```
(lldb) ni
Process 99787 stopped
* thread #1: tid = 0x185cb, 0x000e37e0
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 176, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000e37e0 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 176
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 176:
-> 0xe37e0:  movt   r0, #75
    0xe37e4:  movs   r1, #1
    0xe37e6:  add    r0, pc
    0xe37e8:  cmp    r5, #0
(lldb) p $r6
(unsigned int) $2 = 0
(lldb) c
Process 99787 resuming
```

As we can see, command “p” has printed the value of R6 correctly.

In Objective-C, the implementation of [someObject someMethod] is actually objc_msgSend(someObject, someMethod), among which the first argument is an Objective-C object, and the latter can be casted to a string (we will explain this in detail in chapter 6). As shown in figure 4-19, “BLX _objc_msgSend” executes [SBTelephonyManager sharedTelephonyManager].

```
MOV      R2, #(classRef_SBTelephonyManager - 0x178A0)
ADD      R0, PC ; selRef_sharedTelephonyManager
ADD      R2, PC ; classRef_SBTelephonyManager
LDR      R1, [R0] ; "sharedTelephonyManager"
LDR      R0, [R2] ; _OBJC_CLASS_$_SBTelephonyManager
BLX      _objc_msgSend
```

Figure 4- 19 objc_msgSend([SBTelephonyManager class], @selector(sharedTelephonyManager))

The address with offset of “BLX _objc_msgSend” is known to be 0xCC8A2. Set a breakpoint on it and print the arguments of “objc_msgSend” when we hit this breakpoint:

```
(lldb) br s -a 0xCC8A2
Breakpoint 1: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard + 370,
address = 0x000cc8a2
Process 103706 stopped
* thread #1: tid = 0x1951a, 0x000cc8a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 370, queue = 'com.apple.main-
thread, stop reason = breakpoint 1.1
    frame #0: 0x000cc8a2 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 370
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 370:
-> 0xcc8a2:  blx    0x3e3798 ; symbol stub for: objc_msgSend
    0xcc8a6:  mov    r6, r0
    0xcc8a8:  movw   r0, #31088
    0xcc8ac:  movt   r0, #74
(lldb) po [$r0 class]
SBTelephonyManager
(lldb) po $r0
SBTelephonyManager
(lldb) p (char *)$r1
(char *) $2 = 0x0042eee6 "sharedTelephonyManager"
(lldb) c
```

```
Process 103706 resuming
```

As you can see, we've used "po" command to print the Objective-C object, and "p (char *)" to print the C object by casting. Quite simple, right? It's worth mentioning that when the process stops on a "BL" instruction, LLDB will automatically parse this instruction and display the corresponding symbol:

```
-> 0xcc8a2: blx    0x3e3798          ; symbol stub for: objc_msgSend
```

However, sometimes LLDB's parsing is wrong, mistaking the symbol. In this case, please refer to IDA's static analysis of that symbol.

Finally, we can use "x" command to print the value stored in a specific address:

```
(lldb) p/x $sp
(unsigned int) $4 = 0x006e838c
(lldb) x/10 $sp
0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000
0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8
0x006e83ac: 0x000001c8 0x17a75200
(lldb) x/10 0x006e838c
0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000
0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8
0x006e83ac: 0x000001c8 0x17a75200
```

We've printed SP in hexadecimal with "p/x" command. SP is a pointer, whose value is 0x6e838c. And the "x/10" command has printed the 10 continuous words SP points to.

4. nexti and stepi

Both of "nexti" and "stepi" are used to execute the next instruction, but the biggest difference between them is that the former does not go/step inside a function but the latter does. They are two of the most used commands, and can be abbreviated as "ni" and "si" respectively. You may wonder, what does "go inside a function or not" mean? Let's still take "[SpringBoard _menuButtonDown:]" for example, as shown in figure 4-20.

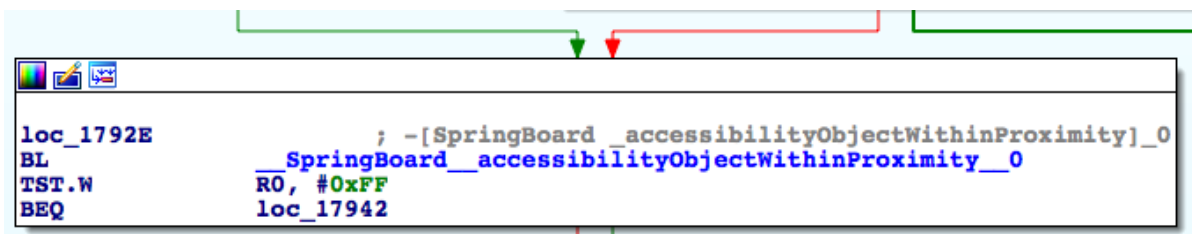


Figure 4- 20 [SpringBoard _menuButtonDown:]

The base address with offset of "BL
__SpringBoard__accessibilityObjectWithinProximity__0" is 0xEE92E, this instruction calls
__SpringBoard__accessibilityObjectWithinProximity__0. Set a breakpoint on it and execute the

“ni” command:

```
(lldb) br s -a 0xEE92E
Breakpoint 2: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510,
address = 0x000ee92e
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510, queue = 'com.apple.main-
thread, stop reason = breakpoint 2.1
    frame #0: 0x000ee92e SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510:
-> 0xee92e: bl    0x2fd654                ;
___lldb_unnamed_function16405$$SpringBoard
    0xee932: tst.w  r0, #255
    0xee936: beq   0xee942                ; ___lldb_unnamed_function299$$SpringBoard
+ 530
    0xee938: blx   0x403f08                ; symbol stub for:
BKSHIDServicesResetProximityCalibration
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee932
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 514, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000ee932 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 514
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 514:
-> 0xee932: tst.w  r0, #255
    0xee936: beq   0xee942                ; ___lldb_unnamed_function299$$SpringBoard
+ 530
    0xee938: blx   0x403f08                ; symbol stub for:
BKSHIDServicesResetProximityCalibration
    0xee93c: movs  r0, #0
(lldb) c
Process 731 resuming
```

As we can see, we haven't gone inside

`__SpringBoard__accessibilityObjectWithinProximity__0` by “ni”. Now, let's try again with “si”:

```
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510, queue = 'com.apple.main-
thread, stop reason = breakpoint 2.1
    frame #0: 0x000ee92e SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 510:
-> 0xee92e: bl    0x2fd654                ;
___lldb_unnamed_function16405$$SpringBoard
    0xee932: tst.w  r0, #255
    0xee936: beq   0xee942                ; ___lldb_unnamed_function299$$SpringBoard
+ 530
    0xee938: blx   0x403f08                ; symbol stub for:
BKSHIDServicesResetProximityCalibration
(lldb) si
Process 731 stopped
* thread #1: tid = 0x02db, 0x002fd654
SpringBoard`___lldb_unnamed_function16405$$SpringBoard, queue = 'com.apple.main-thread,
stop reason = instruction step into
    frame #0: 0x002fd654 SpringBoard`___lldb_unnamed_function16405$$SpringBoard
SpringBoard`___lldb_unnamed_function16405$$SpringBoard:
-> 0x2fd654: movw  r0, #33920
    0x2fd658: movt  r0, #43
    0x2fd65c: add  r0, pc
```

```

0x2fd65e: ldnsb.w r0, [r0]
(lldb) c
Process 731 resuming

```

The base address without offset of “movw r0, #33920” is 0x226654, as shown in figure 4-21.

```

text:00226654
text:00226654 ; -[SpringBoard_accessibilityObjectWithinProximity]_0
text:00226654 __SpringBoard__accessibilityObjectWithinProximity__0
text:00226654 ; CODE XREF: -[SpringBoard_menuButtonDown]:loc_1792E1p
text:00226654 ; -[SpringBoard_menuButtonDown:]+2AC1p ...
text:00226654 MOV R0, #(byte_4DEAE0 - 0x226660)
text:0022665C ADD R0, PC ; byte_4DEAE0
text:0022665E LDRSB.W R0, [R0]
text:00226662 BX LR
text:00226662 ; End of function -[SpringBoard__accessibilityObjectWithinProximity]_0

```

Figure 4- 21 SpringBoard__accessibilityObjectWithinProximity__0

This instruction is inside the `__SpringBoard__accessibilityObjectWithinProximity__0` function. That’s to say, the “si” command has gone inside the function, which is the meaning of “go inside a function or not”.

5. register write

“register write” is used to write a specific value to a specific register, hence “modify the program when it stops, and observe the modification of its execution flow”. According to the code in figure 4-22, the base address with offset of “TST.W R0, offset #0xFF” is known to be 0xEE7A2, if R0’s value is 0, the process will branch to the left, or to the right if R0 is not 0.

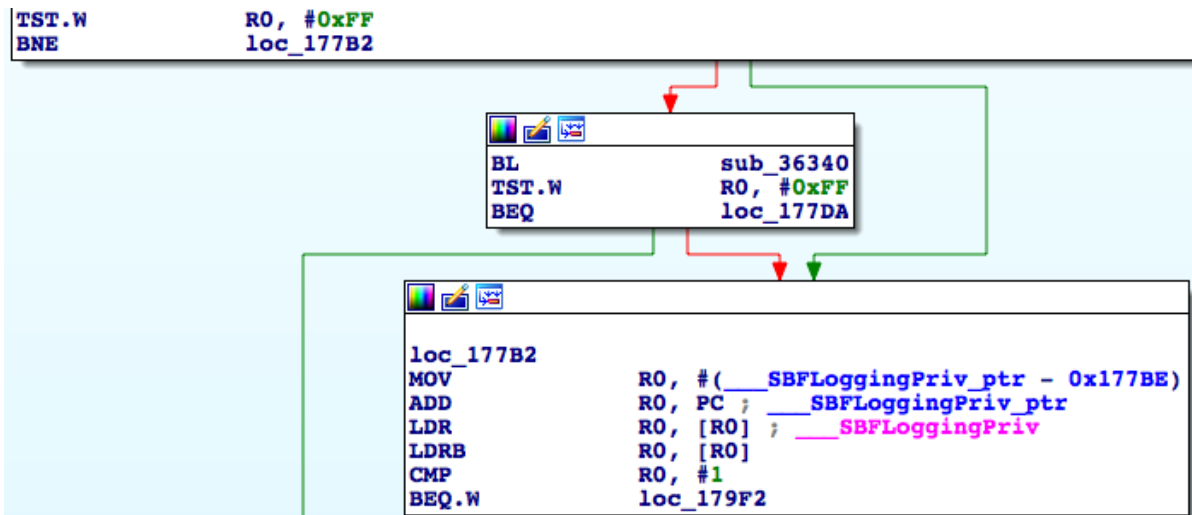


Figure 4- 22 Branches

Set a breakpoint here to see the value of R0 as follows:

```

(lldb) br s -a 0xEE7A2
Breakpoint 3: where = SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114,
address = 0x000ee7a2
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114, queue = 'com.apple.main-
thread, stop reason = breakpoint 3.1
frame #0: 0x000ee7a2 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114:

```

```

-> 0xee7a2: tst.w r0, #255
    0xee7a6: bne 0xee7b2 ; ___lldb_unnamed_function299$$SpringBoard
+ 130
    0xee7a8: bl 0x10d340 ;
___lldb_unnamed_function1110$$SpringBoard
    0xee7ac: tst.w r0, #255
(lldb) p $r0
(unsigned int) $0 = 0

```

Because the value of R0 is 0, BNE makes the process branch to the left:

```

(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000ee7a6 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118:
-> 0xee7a6: bne 0xee7b2 ; ___lldb_unnamed_function299$$SpringBoard
+ 130
    0xee7a8: bl 0x10d340 ;
___lldb_unnamed_function1110$$SpringBoard
    0xee7ac: tst.w r0, #255
    0xee7b0: beq 0xee7da ; ___lldb_unnamed_function299$$SpringBoard
+ 170
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a8
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 120, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000ee7a8 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 120
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 120:
-> 0xee7a8: bl 0x10d340 ;
___lldb_unnamed_function1110$$SpringBoard
    0xee7ac: tst.w r0, #255
    0xee7b0: beq 0xee7da ; ___lldb_unnamed_function299$$SpringBoard
+ 170
    0xee7b2: movw r0, #2174

```

Trigger that breakpoint again, change R0's value to 1 by "register write", and see if the branch changes:

```

Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114, queue = 'com.apple.main-
thread, stop reason = breakpoint 3.1
    frame #0: 0x000ee7a2 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 114:
-> 0xee7a2: tst.w r0, #255
    0xee7a6: bne 0xee7b2 ; ___lldb_unnamed_function299$$SpringBoard
+ 130
    0xee7a8: bl 0x10d340 ;
___lldb_unnamed_function1110$$SpringBoard
    0xee7ac: tst.w r0, #255
(lldb) p $r0
(unsigned int) $5 = 0
(lldb) register write r0 1
(lldb) p $r0
(unsigned int) $6 = 1
(lldb) ni

```

```

Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000ee7a6 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 118:
-> 0xee7a6: bne    0xee7b2                ; ___lldb_unnamed_function299$$SpringBoard
+ 130
    0xee7a8: bl     0x10d340                ;
___lldb_unnamed_function1110$$SpringBoard
    0xee7ac: tst.w  r0, #255
    0xee7b0: beq   0xee7da                ; ___lldb_unnamed_function299$$SpringBoard
+ 170
(lldb)
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7b2
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 130, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x000ee7b2 SpringBoard`___lldb_unnamed_function299$$SpringBoard + 130
SpringBoard`___lldb_unnamed_function299$$SpringBoard + 130:
-> 0xee7b2: movw   r0, #2174
    0xee7b6: movt   r0, #63
    0xee7ba: add   r0, pc
    0xee7bc: ldr   r0, [r0]

```

At this time, the program branches to the right as we expected.

There're much more LLDB commands that worth attention, but we're only covering 5 of the most frequently used ones in the beginning period of iOS reverse engineering, hope you can peep one spot and see the whole picture, as well feel the power of LLDB. LLDB is still under development, other than a few official websites, there is no satisfying tutorial; LLDB derives from GDB, although they have different commands, the thinking mode is almost the same. To learn LLDB in a more systematic way, I recommend you "Peter's GDB tutorial" and "RMS's gdb Debugger Tutorial". IDA is good at static analysis, while LLDB is good at dynamic analysis. Mastery of these two tools removes all obstacles on your road to a master of reverse engineering.

4.3.6 Miscellaneous LLDB

- Binaries to be debugged must be right from iOS on device

If only our static and dynamic analysis target is exactly the same that the base address without offset, ASLR offset and the base address with offset are correspondent. For binaries to be analyzed in IDA, we can use `dyld_decache` in chapter 3 to extract them from the shared cache on device. Binaries from SDK or iOS simulator usually don't meet the condition.

- Shortcuts in LLDB

If you want to repeat the last command in LLDB, you can simply press “enter”. If you want to review all history commands, just press up and down on your keyboard.

LLDB commands are simple, but it’s not easy to solve complicated problems with these simple commands. In chapter 6, we will introduce more common scenarios of using LLDB, and before that, please be sure to understand the knowledge of this section.

4.4 dumpdecrypted

When introducing class-dump, we’ve mentioned that Apple encrypts all Apps from AppStore, protecting them from being class-dumped. If we want to class-dump StoreApps, we have to decrypt their executables at first. A handy tool, dumpdecrypted, by Stefan Esser (@i0n1c) is commonly used in iOS reverse engineering.

dumpdecrypted is open sourced on GitHub, you have to compile it by yourselves. Now let’s start from scratch to class-dump a virtual target, i.e. TargetApp.app to show you the steps of decrypting an App, please follow me.

1. Download dumpdecrypted’s source code from GitHub as follows:

```
snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/Code/
snakeninnysiMac:Code snakeninny$ git clone git://github.com/stefanesser/dumpdecrypted/
Cloning into 'dumpdecrypted'...
remote: Counting objects: 31, done.
remote: Total 31 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (31/31), 6.50 KiB | 0 bytes/s, done.
Resolving deltas: 100% (15/15), done.
Checking connectivity... done
```

2. Compile the source code and get dumpdecrypted.dylib:

```
snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/Code/dumpdecrypted/
snakeninnysiMac:dumpdecrypted snakeninny$ make
`xcrun --sdk iphoneos --find gcc` -Os -Wimplicit -isysroot `xcrun --sdk iphoneos --
show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks -
F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -
arch armv7s -arch arm64 -c -o dumpdecrypted.o dumpdecrypted.c
`xcrun --sdk iphoneos --find gcc` -Os -Wimplicit -isysroot `xcrun --sdk iphoneos --
show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks -
F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -
arch armv7s -arch arm64 -dynamiclib -o dumpdecrypted.dylib dumpdecrypted.o
```

After “make”, a dumpdecrypted.dylib will be generated under the current directory. This dylib can be reused, there’s no need to recompile.

3. Locate the executable to be decrypted with “ps” command

On iOS 8, all StoreApps are under /var/mobile/Containers/, and TargetApp.app’s executable is under /var/mobile/Containers/Bundle/Application/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXXX/TargetApp.app/. Since X is unknown, it’d be a great amount of work to locate the executable manually. But a simple trick will save our days: First close all StoreApps on iOS, then launch TargetApp and ssh into iOS to print all processes:

```
snakeninnysiMac:~ snakeninny$ ssh root@iOSIP
FunMaker-5:~ root# ps -e
  PID TTY          TIME CMD
    1  ??          3:28.32 /sbin/launchd
.....
 5717  ??          0:00.21
/System/Library/PrivateFrameworks/MediaServices.framework/Support/mediaartworkd
 5905  ??          0:00.20 sshd: root@ttys000
 5909  ??          0:01.86 /var/mobile/Containers/Bundle/Application/03B61840-2349-4559-
B28E-0E2C6541F879/TargetApp.app/TargetApp
 5911  ??          0:00.07 /System/Library/Frameworks/UIKit.framework/Support/pasteboardd
 5907  ttys000    0:00.03 -sh
 5913  ttys000    0:00.01 ps -e
```

Because now there is only one running StoreApp, the only path that contains “/var/mobile/Containers/Bundle/Application/” is the full path of TargetApp’s executable.

4. Find out TargetApp’s Documents directory via Cypcript

All StoreApps’ Documents directories are under /var/mobile/Containers/Data/Application/ YYYYYYYY-YYYY-YYYY-YYYY - YYYYYYYYYYYYYY/. Note that these Ys are different from those previous Xs, and they are not obtainable via “ps”. So this time we need to mak use of Cypcript to reveal the Documents directory of TargetApp. The commands we use are as follows:

```
FunMaker-5:~ root# cypcript -p TargetApp
cy# [[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask][0]
#”file:///var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-
2146128611EE/Documents/”
```

5. Copy dumpdecrypted.dylib to TargetApp’s Documents directory:

```
snakeninnysiMac:~ snakeninny$ scp
/Users/snakeninny/Code/dumpdecrypted/dumpdecrypted.dylib
root@iOSIP:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-
2146128611EE/Documents/
dumpdecrypted.dylib
100% 193KB 192.9KB/s 00:00
```

Here we’re using scp instead of iFunBox, anyway tools don’t matter.

6. Start decrypting

The usage of `dumpdecrypted.dylib` is as follows:

```
DYLD_INSERT_LIBRARIES=/path/to/dumpdecrypted.dylib /path/to/executable
```

For instance:

```
FunMaker-5:~ root# cd /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents/
FunMaker-5:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp
mach-o decryption dumper
```

DISCLAIMER: This tool is only meant for security research purposes, not for application crackers.

```
[+] detected 32bit ARM binary in memory.
[+] offset to cryptid found: @0x81a78(from 0x81000) = a78
[+] Found encrypted data at address 00004000 of length 6569984 bytes - type 1.
[+] Opening /private/var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp for reading.
[+] Reading header
[+] Detecting header type
[+] Executable is a plain MACH-O image
[+] Opening TargetApp.decrypted for writing.
[+] Copying the not encrypted start of the file
[+] Dumping the decrypted data into the file
[+] Copying the not encrypted remainder of the file
[+] Setting the LC_ENCRYPTION_INFO->cryptid to 0 at offset a78
[+] Closing original file
[+] Closing dump file
```

A decrypted executable named `TargetApp.decrypted` will be created in the current directory:

```
FunMaker-5:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# ls
TargetApp.decrypted dumpdecrypted.dylib OtherFiles
```

Copy `TargetApp.decrypted` to OSX ASAP. `class-dump` and `IDA` have been waiting for ages!

I think these 6 steps are clear enough, but some of you may still wonder, why to copy `dumpdecrypted.dylib` to Documents directory?

Good question. We all know that StoreApps don't have write permission to most of the directories outside the sandbox. Since `dumpdecrypted.dylib` needs to write a decrypted file while residing in a StoreApp and they have the same permission, so the destination of its write operation should be somewhere writable. StoreApp can write to its Documents directory, so `dumpdecrypted.dylib` should be able to work under this directory.

Let's see what happens if `dumpdecrypted.lib` is not working under Documents directory:

```

FunMaker-5: /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-
2146128611EE/Documents root# mv dumpdecrypted.dylib /var/tmp/
FunMaker-5: /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-
2146128611EE/Documents root# cd /var/tmp
FunMaker-5:/var/tmp root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/private/var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-
0E2C6541F879/TargetApp.app/TargetApp
dyld: could not load inserted library 'dumpdecrypted.dylib' because no suitable image
found. Did find:
    dumpdecrypted.dylib: stat() failed with errno=1

Trace/BPT trap: 5

```

errno=1 means “Operation not permitted”, dumpdecrypted.dylib failed to work as expected. If you encounter any problem or have any experience using dumpdecrypted, you are welcome to share with us at <http://bbs.iosre.com>.

4.5 OpenSSH



Figure 4- 23 OpenSSH

OpenSSH will install SSH service on iOS (as shown in figure 4-23). Only 2 commands are the most commonly used: ssh is used for remote logging, scp is used for remote file transfer. The usage of ssh is as follows:

```
ssh user@iOSIP
```

For instance:

```
snakeninnysiMac:~ snakeninny$ ssh mobile@192.168.1.6
```

The usage of scp is as follows:

- Copy a local file to iOS:

```
scp /path/to/localFile user@iOSIP:/path/to/remoteFile
```

For instance:

```
snakeninnysiMac:~ snakeninny$ scp ~/1.png root@192.168.1.6:/var/tmp/
```

- Copy a file from iOS to the local system:

```
scp user@iOSIP:/path/to/remoteFile /path/to/localFile
```

For instance:

```
snakeninnysiMac:~ snakeninny$ scp root@192.168.1.6:/var/log/syslog ~/iOSlog
```

These two commands are relatively simple and intuitive. After installing OpenSSH, make sure to change the default login password “alpine”. There’re 2 users on iOS, i.e. root and mobile, we need to change both passwords like this:

```
FunMaker-5:~ root# passwd root
Changing password for root.
New password:
Retype new password:
FunMaker-5:~ root# passwd mobile
Changing password for mobile.
New password:
Retype new password:
```

If we forget to change the default password, there’re chances that viruses like Ikee login as root via ssh. This leads to very serious security disasters: all data on iOS including SMS, contacts, AppleID passwords and so on is at the risk of leaking, the intruder can take control over your device and do whatever he wants. Therefore, promise me you’ll change the default password after installing OpenSSH, OK?

4.6 usbmuxd

Most of you ssh into iOS via WiFi, which leads to slow responses in remote debugging or file copying. This is because of the instability of wireless network and the limitation of transmission speed. The well-known hacker, Nikias Bassen (@pimskeks) has written a tool named usbmuxd to forward local OSX/Windows port to remote iOS port. With this tool, we can ssh into iOS via USB, greatly increasing the speed of SSH connection. usbmuxd is easy to use:

1. Download and configure usbmuxd

Download usbmuxd from <http://cgit.sukimashita.com/usbmuxd.git/snapshot/usbmuxd->

1.0.8.tar.gz and decompress it. The files we are going to use are tcprelay.py and usbmuxd.py.

Copy them to the same directory such as:

```
/Users/snakeninny/Code/USBSSH/
```

2. Forward local port to remote port with usbmuxd

Input the following command in Terminal:

```
/Users/snakeninny/Code/USBSSH/tcprelay.py -t Remote port on iOS:Local port on OSX/Windows
```

Now usbmuxd is forwarding local port on OSX/Windows to remote port on iOS.

Here comes an example of usage scenario: ssh into iOS via USB without WiFi, then debug SpringBoard with LLDB.

- Forward local port 2222 on OSX to remote port 22 on iOS:

```
snakeninnysiMac:~ snakeninny$ /Users/snakeninny/Code/USBSSH/tcprelay.py -t 22:2222  
Forwarding local port 2222 to remote port 22
```

- ssh into iOS and attach debugserver to SpringBoard:

```
snakeninnysiMac:~ snakeninny$ ssh root@localhost -p 2222  
FunMaker-5:~ root# debugserver *:1234 -a "SpringBoard"
```

- Forward local port 1234 on OSX to remote port 1234 on iOS:

```
snakeninnysiMac:~ snakeninny$ /Users/snakeninny/Code/USBSSH/tcprelay.py -t 1234:1234  
Forwarding local port 1234 to remote port 1234
```

- Start debugging in LLDB:

```
snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb  
(lldb) process connect connect://localhost:1234
```

usbmuxd speeds up ssh connection to less than 15 seconds in general, and should be your first ssh choice.

4.7 iFile

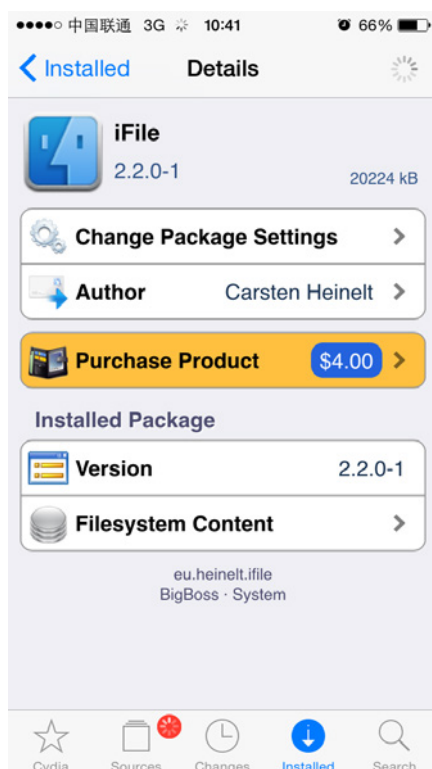


Figure 4- 24 iFile

iFile is a very powerful file management App, you can view it as Finder's parallel on iOS. iFile is capable of all kinds of file operation including browsing, editing, cutting, copying and deb installing, possessing great convenience.

iFile is rather user-friendly. Before installing a deb, remember to close Cydia at first, then tap the deb file to be installed and choose "Installer" in the action sheet, as shown in figure 4-25.

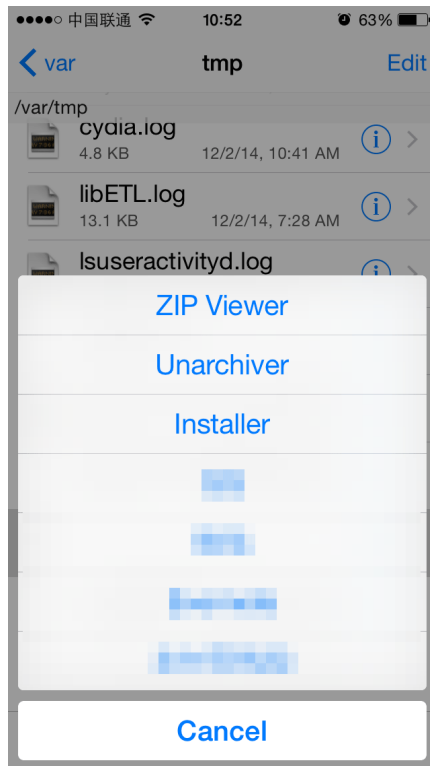


Figure 4- 25 Install deb file

4.8 MTerminal



Figure 4- 26 MTerminal

MTerminal is an open sourced Terminal on iOS with all basic functions available. The usage of MTerminal is no much difference to Terminal, if we put the screen and keyboard size aside. I

think the most practical scene of MTerminal is to test private methods in Cycript when we're blanking out on the subway or something.

4.9 syslogd to /var/log/syslog

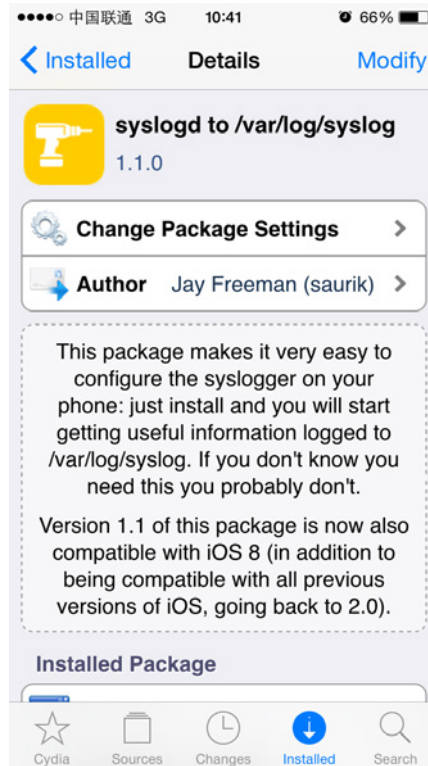


Figure 4- 27 syslogd to /var/log/syslog

syslogd is a daemon to record system logs on iOS, and “syslogd to /var/log/syslog” is used to write the logs to a file at “/var/log/syslog”. You need to reboot iOS after you install this tweak to automatically create the file “/var/log/syslog”. This file gets larger as time goes by, you can zero clear it with the following command:

```
FunMaker-5:~ root# cat /dev/null > /var/log/syslog
```

4.10 Conclusion

We've introduced 9 tools in this chapter, among which CydiaSubstrate, LLDB and Cycript are the top priorities. It is because of the existence of these iOS tools, along with the OSX toolkit in chapter 3, that we get a complete iOS reverse engineering environment. There's a famous Chinese saying that we should know how as well as know why. Now that we've already known how by finishing part 2 of this book, it's time for us to know why in the next part. Stay tuned!

Theories



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

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

5.1 How does a tweak work in Objective-C

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

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

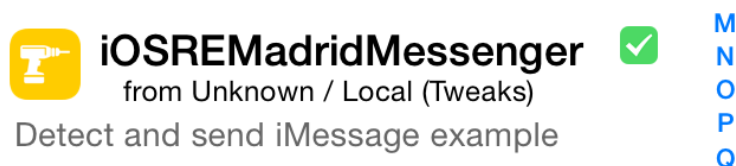


Figure 5- 1 Tweak icon

Objective-C is a typical object-oriented programming language; iOS consists of many small

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

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

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

- When here comes a phone call, we may say that “Mom, answer the phone, please”. When we want to translate this sentence into Objective-C, it will be [mom answerThePhone]. Here, the object is “mom” and the method is “answerThePhone”. The implementation could be “Mom stops cooking and goes to the sitting room to answer the phone”.
- "snakeninny, come here and help me move out this box". This could be translated into [snakeninny moveOutTheBox]. The object here is “snakeninny” and method is “moveOutTheBox” while the implementation could be “snakeninny stops working and goes to the boss’ office to move a box downstairs”.

In the above examples, if there is no specific implementation, even we call a method of an object, the object still doesn’t know what to do. So now, we can think implementation as the interpretation of method. Is it a little confusing? Don’t worry. Let’s draw an analogy between programming and dictionary. You can just imagine the method here to be a word in the

dictionary and the implementation to be the meaning of that word. When you look up the dictionary, you always want to find what does a certain abstruse word mean. When it comes to programming, the implementation of a method does exactly the same as a word's meaning in the dictionary. Easier to understand, right? Lets' move on.

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



Figure 5- 2 LowPowerBanner

Another example is SMSNinja, as shown in figure 5-3. When you receive a spam message, SMSNinja puts the spam message into trash box automatically. This feature is achieved by changing the implementation of delegate method of receiving a message; I've added extra spam

detecting function to the original method. This kind of approach is similar to changing the contents of dictionary and can be realized through the hook function provided by CydiaSubstrate. The usage of CydiaSubstrate has been explained in the last two chapters, so if you've already forgotten about it, you should go back and have a review.

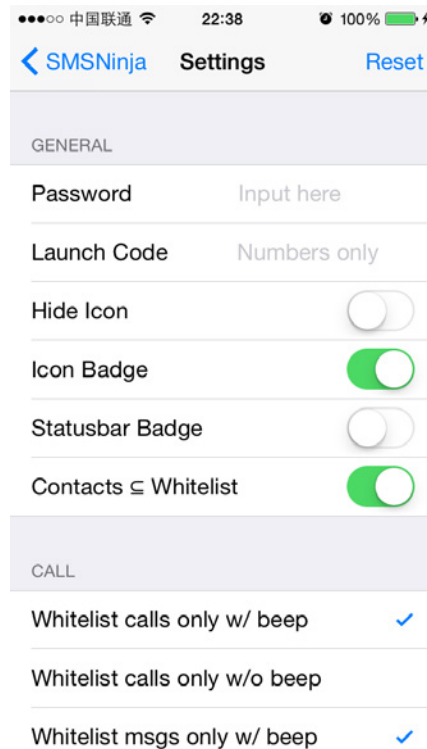


Figure 5- 3 SMSNinja

5.2 Methodology of writing a tweak

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

5.2.1 Look for inspiration

So far, some readers might have already been able to write tweaks with knowledge introduced in the previous chapters, but most may still don't know where to start. I know it's

uncomfortable when we don't know where to use our abilities, so here are some tips to help you look for inspiration for your first tweak.

- Use more, observe more

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



Figure 5- 4 Characount for Notes

- Listen to users' voice

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

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

Downloads for snakeninny			
App	Total Count	Installer	Cydia
SMSNinja (v1.2)	22934	0	22934
LowPowerBanner (v0.0.1-42)	35493	0	35493

Figure 5-5 Downloads of LowPowerBanner 1.0

- Anatomize iOS

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

```
- (void)reboot;
- (void)relaunchSpringBoard;
```

And in UIViewController.h:

```
- (void)attentionClassDumpUser:(id)arg1
yesItsUsAgain:(id)arg2
althoughSwizzlingAndOverridingPrivateMethodsIsFun:(id)arg3
itWasntMuchFunWhenYourAppStoppedWorking:(id)arg4
```

```
pleaseRefrainFromDoingSoInTheFutureOkayThanksBye:(id)arg5;
```

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

5.2.2 Locate target files

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

- Fixed location

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

- ✧ CydiaSubstrate based dylibs are all stored in `"/Library/MobileSubstrate/DynamicLibraries/"`. We can find them without effort.
- ✧ Bundles can be divided into 2 categories, which are App and framework respectively. Bundles of AppStore Apps are stored in `"/var/mobile/Containers/Bundle/Application/"`, bundles of system Apps are stored in `"/Applications/"`, and bundles of frameworks are stored in `"/System/Library/Frameworks"` and `"/System/Library/PrivateFrameworks"`. For bundles of other types, you can discuss with us on <http://bbs.iosre.com>.
- ✧ Configuration files of daemons, which are plist formatted, are all stored in `"/System/Library/LaunchDaemons/"`, `"/Library/LaunchDaemons"` and `"/Library/LaunchAgents/"`. The "ProgramArguments" fields in these files are the absolute paths of daemon executables, such as:

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
/Users/snakeninny/Desktop/com.apple.backboardd.plist
{
.....
  "ProgramArguments" => [
    0 => "/usr/libexec/backboardd"
  ]
.....
}
```

- Locate with Cydia

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

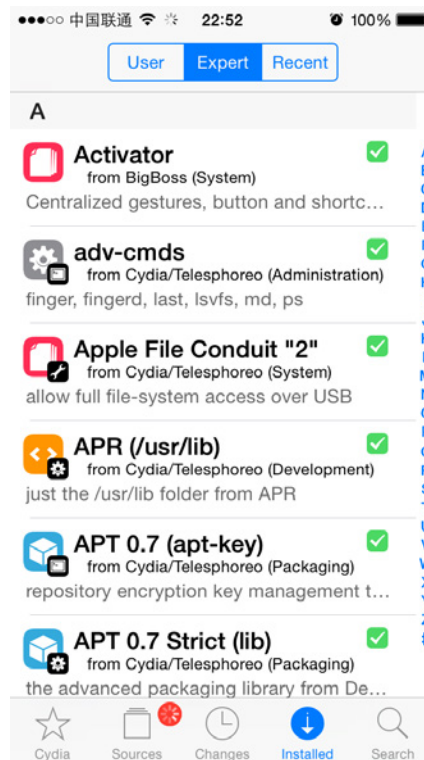


Figure 5-6 Expert view in Cydia

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

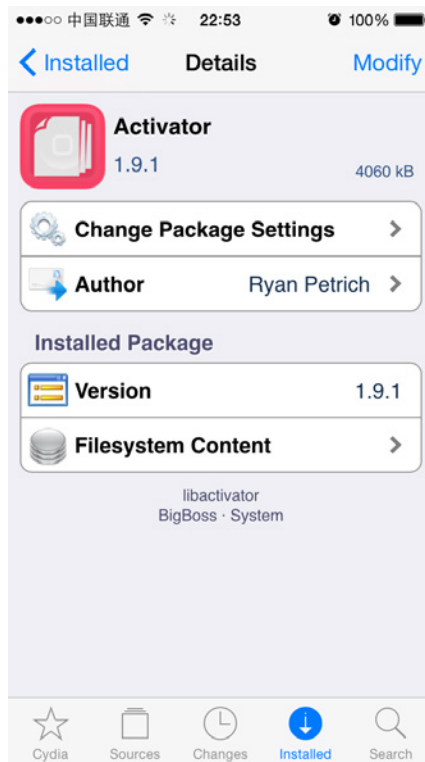


Figure 5-7 Details View

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

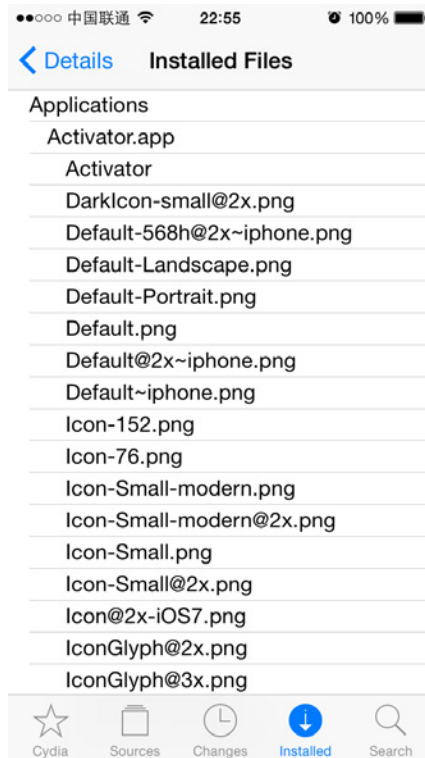


Figure 5- 8 Installed files

You can easily find each file’s location now.

- PreferenceBundle

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

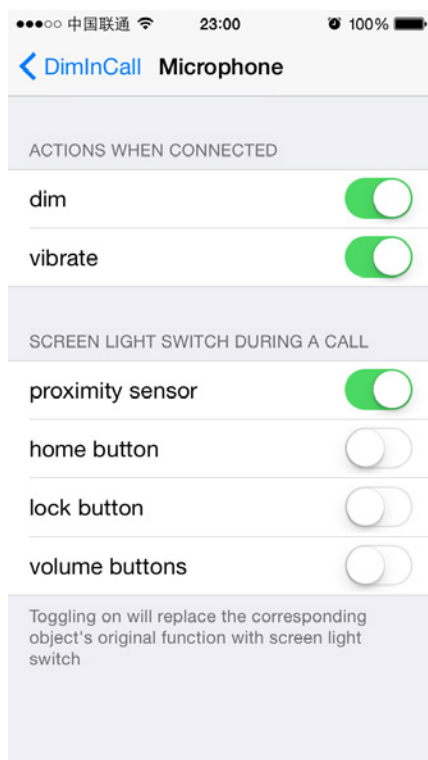


Figure 5- 9 DimInCall

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

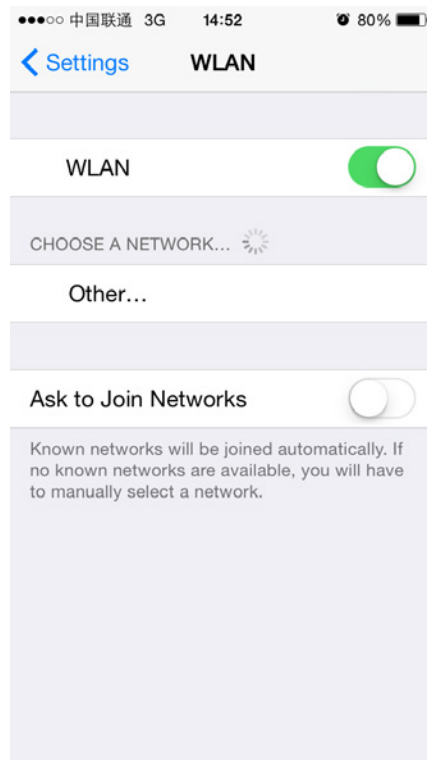


Figure 5- 10 WLAN

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

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



Figure 5- 11 About

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

- grep

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

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

```
Binary file /System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7s matches
grep: /System/Library/Caches/com.apple.dyld/enable-dylibs-to-override-cache: No such
file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGCorePDF.dylib: No
such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltin.dylib:
No such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMaps.dylib: No
such file or directory
grep: /System/Library/Frameworks/System.framework/System: No such file or directory
```

From the result, we can see that the method appears in `dyld_shared_cache_armv7s`. Now, we can use `grep` again in the detached `dyld_shared_cache_armv7s`:

```
snakeninnysiMac:~ snakeninny$ grep -r initWithAccountID:defaults:service:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/dyld_shared_cache_armv7s matches
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/System/Library/Caches/com.apple.xpc
/sdk.dylib: Too many levels of symbolic links
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/System/Library/Frameworks/OpenGL.
framework/libLLVMContainer.dylib: Too many levels of symbolic links
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/System/Library/PrivateFrameworks/IM
DaemonCore.framework/IMDaemonCore matches
```

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

5.2.3 Locate target functions

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

- Use the built-in search function in OSX

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

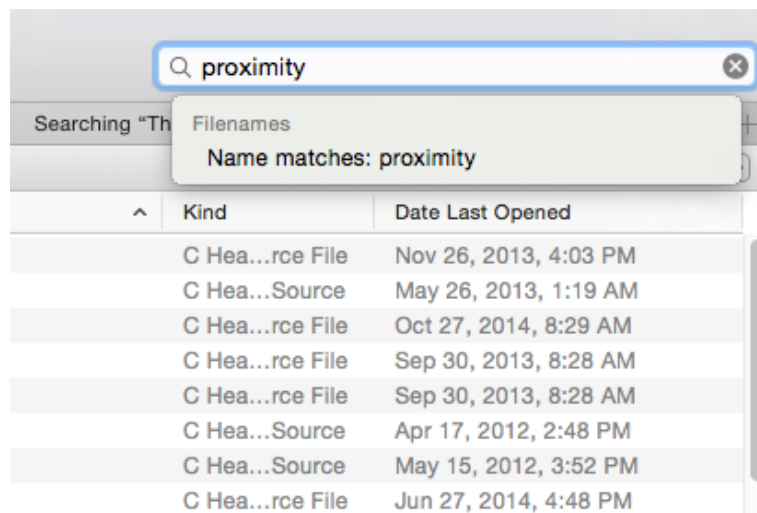


Figure 5- 12 Search in Finder

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

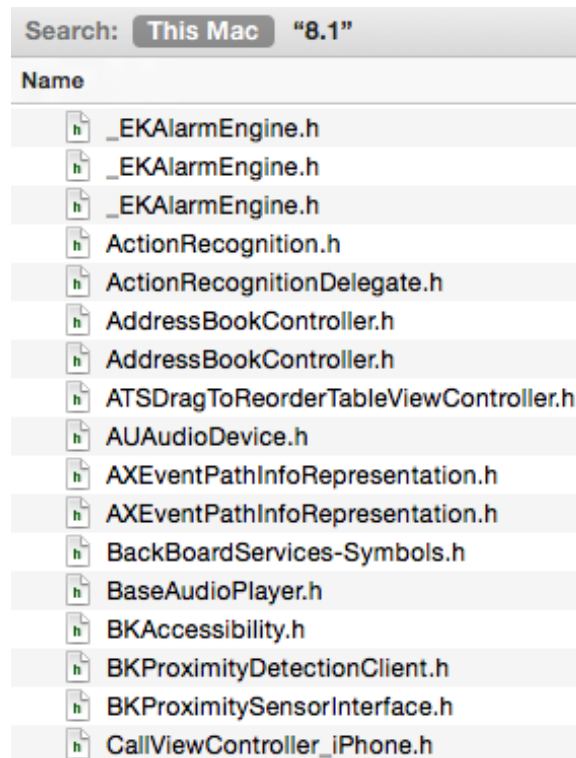


Figure 5-13 Search results in Finder

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

- grep

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

```
snakeninnysMac:~ snakeninny$ grep -r -i proximity
/Users/snakeninny/Code/iOSPrivateHeaders/8.1
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/CoreLocation/CDStructures.h:
char proximityUUID[512];
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/CoreLocation/CLBeacon.h:
NSUUID *_proximityUUID;
.....
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/SpringBoard.h:-
(_Bool)proximityEventsEnabled;
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/SpringBoard.h:-
(void)_proximityChanged:(id)arg1;
```

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

5.2.4 Test private methods

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

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

- CydiaSubstrate

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

```
%hook SBScreenShooter
- (void)saveScreenshot:(BOOL)screenshot
{
    %orig;
    NSLog(@"iOSRE: saveScreenshot: is called");
}
%end
```

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

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 24 16:22:06 FunMaker-5 SpringBoard[2765]: iOSRE: saveScreenshot: is called
```

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

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

- Cycrypt

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

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    %orig;
    SBScreenShotter *shotter = [%c(SBScreenShotter) sharedInstance];
    [shotter saveScreenshot:YES]; // For the argument here, I guess it’s YES; later
    we’ll see what happens if it’s NO
}
%end
```

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

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

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

ssh to iOS and then execute the following commands:

```
FunMaker-5:~ root# cycrypt -p SpringBoard
cy# [[SBScreenShotter sharedInstance] saveScreenshot:YES]
```


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

5.2.5 Analyze method arguments

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

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

```
%hook SBScreenFlash
- (void)flashColor:(id)arg1 withCompletion:(id)arg2
{
    %orig;
    NSLog(@"iosRE: flashColor: %s, %@", object_getClassName(arg1), arg1); // [arg1
description] can be replaced by arg1
}
%end
```

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

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

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 24 16:40:33 FunMaker-5 SpringBoard[2926]: iOSRE: flashColor:
UICachedDeviceWhiteColor, UIDeviceWhiteColorSpace 1 1
```

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

```
@interface UICachedDeviceWhiteColor : UIDeviceWhiteColor
{
}

- (void)_forceDealloc;
- (void)dealloc;
- (id)copy;
- (id)copyWithZone:(struct _NSZone *)arg1;
- (id)autorelease;
- (BOOL)retainWeakReference;
- (BOOL)allowsWeakReference;
- (unsigned int)retainCount;
- (id)retain;
- (oneway void)release;

@end
```

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

```
@interface UIDeviceWhiteColor : UIColor
{
    float whiteComponent;
    float alphaComponent;
    struct CGColor *cachedColor;
    long cachedColorOnceToken;
}

- (BOOL)getHue:(float *)arg1 saturation:(float *)arg2 brightness:(float *)arg3
alpha:(float *)arg4;
- (BOOL)getRed:(float *)arg1 green:(float *)arg2 blue:(float *)arg3 alpha:(float *)arg4;
- (BOOL)getWhite:(float *)arg1 alpha:(float *)arg2;
- (float)alphaComponent;
- (struct CGColor *)CGColor;
- (unsigned int)hash;
- (BOOL)isEqual:(id)arg1;
- (id)description;
- (id)colorSpaceName;
- (void)setStroke;
- (void)setFill;
- (void)set;
- (id)colorWithAlphaComponent:(float)arg1;
- (struct CGColor *)_createCGColorWithAlpha:(float)arg1;
- (id)copyWithZone:(struct _NSZone *)arg1;
```

```
- (void)dealloc;
- (id)initWithCGColor:(struct CGColor *)arg1;
- (id)initWithWhite:(float)arg1 alpha:(float)arg2;

@end
```

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

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

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

```
FunMaker-5:~ root# cycript -p SpringBoard
cy# [[SBScreenFlash mainScreenFlasher] flashColor:[UIColor magentaColor]
withCompletion:nil]
```

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

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

5.2.6 Limitations of class-dump

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

In section 5.2.5, we've analyzed the 1st argument of flashColor:withCompletion: in SBScreenFlash. From the effect of flashColor:withCompletion:, we guess that it should happen inside saveScreenShot:. But if we just take class-dump headers and the private methods' effects as references, we can only know the execution order of saveScreenShot: and

flashColor:withCompletion:. Neither can we know anything about implementation details and their relationship, nor can we verify our guesses.

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

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

5.3 An example tweak using the methodology

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



Figure 5- 14 Speaker SBSettings Toggle

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

5.3.1 Get inspiration

At the end of March 2012, I received an email from Shoghian, an Iranian-Canadian. In the mail, he shared an idea that iOS users could switch between microphone and speaker during a phone call while few people knew the speaker could be turned on by default. This feature was very useful for those who were cooking, driving or inconvenient to hold the phone during a call. However, such a useful feature was hidden in “Settings” → “General” → “Accessibility” → “Incoming Calls”, which was a four-level menu (as shown in figure 5-15) so the set up was very cumbersome. Various toggles in SBSettings are aimed to solve problems like this. So I planned to rewrite it as a toggle to make this good feature handier.

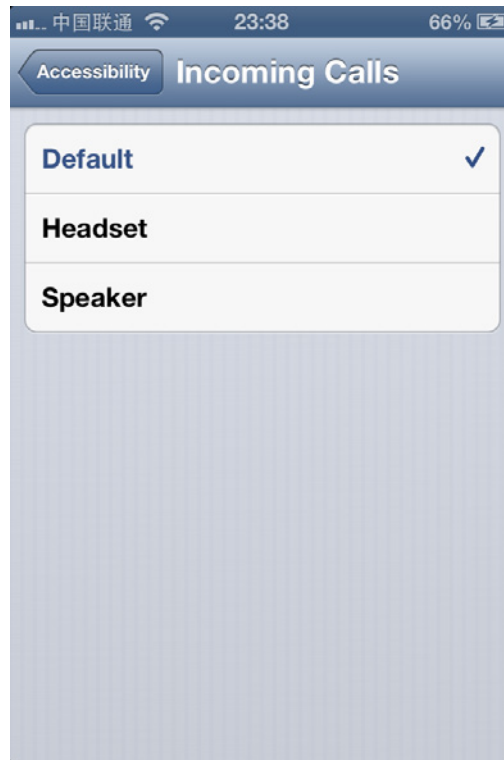


Figure 5- 15 Incoming Calls

5.3.2 Locate files

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

- Change the system language to English

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

- Discover keyword "Accessibility"

After I had changed the system language, the four-level menu has been translated from Chinese to "Settings" → "General" → "Accessibility" → "Incoming Calls". The keyword "Accessibility" caught my attention. The reason was that without combining the context, "Accessibility" was too generic to contain "Incoming Calls". So I sshed to iOS and greped the whole filesystem with keyword "Accessibility". The result was as follows:

```
FunMaker-4s:~ root# grep -r Accessibility /
grep: /Applications/Activator.app/Default-568h@2x~iphone.png: No such file or directory
grep: /Applications/Activator.app/Default.png: No such file or directory
grep: /Applications/Activator.app/Default~iphone.png: No such file or directory
grep: /Applications/Activator.app/LaunchImage-700-568h@2x.png: No such file or directory
```

```
Binary file /Applications/Activator.app/en.lproj/Localizable.strings matches
grep: /Applications/Activator.app/iOS7-Default-Landscape@2x.png: No such file or
directory
grep: /Applications/Activator.app/iOS7-Default-Portrait@2x.png: No such file or
directory
Binary file /Applications/AdSheet.app/AdSheet matches
Binary file /Applications/Compass.app/Compass matches
.....
```

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

```
Binary file /Applications/Preferences.app/English.lproj/General-Simulator.strings
matches
Binary file /Applications/Preferences.app/English.lproj/General~iphone.strings matches
Binary file /Applications/Preferences.app/General-Simulator.plist matches
Binary file /Applications/Preferences.app/General.plist matches
Binary file /Applications/Preferences.app/Preferences matches
Binary file /Applications/Preferences.app/en_GB.lproj/General-Simulator.strings matches
Binary file /Applications/Preferences.app/en_GB.lproj/General~iphone.strings matches
```

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

```
snakeninnys-MacBook:~ snakeninny$ plutil -p ~/General~iphone.strings
{
  "Videos..." => "• Videos..."
  "Wallpaper" => "Wallpaper"
  "TV_OUT" => "TV Out"
  "SOUND_EFFECTS" => "Sound Effects"
  "d_MINUTES" => "%@ Minutes"
  .....
  "ACCESSIBILITY" => "Accessibility"
  "Multitasking_Gestures" => "Multitasking Gestures"
  .....
}
```

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

- Discover General.plist

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

```
FunMaker-4s:~ root# grep -r ACCESSIBILITY /
grep: /Applications/Activator.app/Default-568h@2x~iphone.png: No such file or directory
grep: /Applications/Activator.app/Default.png: No such file or directory
grep: /Applications/Activator.app/Default~iphone.png: No such file or directory
grep: /Applications/Activator.app/LaunchImage-700-568h@2x.png: No such file or directory
grep: /Applications/Activator.app/iOS7-Default-Landscape@2x.png: No such file or
directory
grep: /Applications/Activator.app/iOS7-Default-Portrait@2x.png: No such file or
directory
Binary file /Applications/Preferences.app/Dutch.lproj/General-Simulator.strings matches
Binary file /Applications/Preferences.app/Dutch.lproj/General~iphone.strings matches
Binary file /Applications/Preferences.app/English.lproj/General-Simulator.strings
matches
Binary file /Applications/Preferences.app/English.lproj/General~iphone.strings matches
```

```
Binary file /Applications/Preferences.app/French.lproj/General-Simulator.strings matches
Binary file /Applications/Preferences.app/French.lproj/General~iphone.strings matches
Binary file /Applications/Preferences.app/General-Simulator.plist matches
Binary file /Applications/Preferences.app/General.plist matches
Binary file /Applications/Preferences.app/German.lproj/General-Simulator.strings matches
Binary file /Applications/Preferences.app/German.lproj/General~iphone.strings matches
.....
```

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

```
snakeninnys-MacBook:~ snakeninny$ plutil -p ~/General.plist
{
  "title" => "General"
  "items" => [
    0 => {
      "cell" => "PSGroupCell"
    }
    1 => {
      "detail" => "AboutController"
      "cell" => "PSLinkCell"
      "label" => "About"
    }
    2 => {
      "cell" => "PSLinkCell"
      "id" => "SOFTWARE_UPDATE_LINK"
      "detail" => "SoftwareUpdatePrefController"
      "label" => "SOFTWARE_UPDATE"
      "cellClass" => "PSBadgedTableCell"
    }
    .....
    24 => {
      "detail" => "PSInternationalController"
      "cell" => "PSLinkCell"
      "label" => "INTERNATIONAL"
    }
    25 => {
      "cell" => "PSLinkCell"
      "bundle" => "AccessibilitySettings"
      "label" => "ACCESSIBILITY"
      "requiredCapabilities" => [
        0 => "accessibility"
      ]
      "isController" => 1
    }
    26 => {
      "cell" => "PSGroupCell"
    }
    .....
  ]
}
```

- Discover AccessibilitySetting.bundle

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

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

```
FunMaker-4s:~ root# ls -la
/System/Library/PreferenceBundles/AccessibilitySettings.bundle
total 240
drwxr-xr-x 37 root wheel  2414 Mar 10  2013 .
drwxr-xr-x 40 root wheel  1360 Jan 14  2014 ..
-rw-r--r--  1 root wheel  2146 Mar 10  2013 Accessibility.plist
-rwxr-xr-x  1 root wheel 438800 Mar 10  2013 AccessibilitySettings
-rw-r--r--  1 root wheel   238 Dec 22  2012 BluetoothDeviceConfig.plist
-rw-r--r--  1 root wheel   252 Mar 10  2013 BrailleStatusCellSettings.plist
-rw-r--r--  1 root wheel  4484 Dec 22  2012 ColorWellRound@2x.png
-rw-r--r--  1 root wheel   916 Dec 22  2012 ColorWellSquare@2x.png
drwxr-xr-x  2 root wheel   646 Feb  7  2013 Dutch.lproj
drwxr-xr-x  2 root wheel   646 Dec 22  2012 English.lproj
drwxr-xr-x  2 root wheel   646 Feb  7  2013 French.lproj
drwxr-xr-x  2 root wheel   646 Dec 22  2012 German.lproj
-rw-r--r--  1 root wheel   703 Mar 10  2013 GuidedAccessSettings.plist
-rw-r--r--  1 root wheel   807 Mar 10  2013 HandSettings.plist
-rw-r--r--  1 root wheel   652 Mar 10  2013 HearingAidDetailSettings.plist
-rw-r--r--  1 root wheel   507 Mar 10  2013 HearingAidSettings.plist
-rw-r--r--  1 root wheel   383 Dec 22  2012 HomeClickSettings.plist
-rw-r--r--  1 root wheel   447 Dec 22  2012 IconPlay@2x.png
-rw-r--r--  1 root wheel  1113 Dec 22  2012 IconRecord@2x.png
-rw-r--r--  1 root wheel   170 Dec 22  2012 IconStop@2x.png
-rw-r--r--  1 root wheel   907 Mar 10  2013 Info.plist
drwxr-xr-x  2 root wheel   646 Feb  7  2013 Italian.lproj
drwxr-xr-x  2 root wheel   646 Feb  7  2013 Japanese.lproj
-rw-r--r--  1 root wheel   364 Dec 22  2012 LargeFontsSettings.plist
-rw-r--r--  1 root wheel   217 Mar 10  2013 NavigateImagesSettings.plist
-rw-r--r--  1 root wheel  1030 Dec 22  2012 QuickSpeakSettings.plist
-rw-r--r--  1 root wheel   346 Dec 22  2012 RegionNamesNonLocalized.strings
drwxr-xr-x  2 root wheel   646 Feb  7  2013 Spanish.lproj
-rw-r--r--  1 root wheel   394 Dec 22  2012 SpeakerLoad1@2x.png
-rw-r--r--  1 root wheel   622 Mar 10  2013 TripleClickSettings.plist
-rw-r--r--  1 root wheel   467 Dec 22  2012 VoiceOverBrailleOptions.plist
-rw-r--r--  1 root wheel  2477 Mar 10  2013 VoiceOverSettings.plist
-rw-r--r--  1 root wheel   540 Mar 10  2013 VoiceOverTypingFeedback.plist
-rw-r--r--  1 root wheel   480 Dec 22  2012 ZoomSettings.plist
drwxr-xr-x  2 root wheel   102 Dec 22  2012 _CodeSignature
drwxr-xr-x  2 root wheel   646 Feb  7  2013 ar.lproj
-rw-r--r--  1 root wheel  8371 Dec 22  2012 bottombar@2x~iphone.png
-rw-r--r--  1 root wheel  2701 Dec 22  2012 bottombarblue@2x~iphone.png
-rw-r--r--  1 root wheel  2487 Dec 22  2012 bottombarblue_pressed@2x~iphone.png
-rw-r--r--  1 root wheel  2618 Dec 22  2012 bottombarred@2x~iphone.png
-rw-r--r--  1 root wheel  2426 Dec 22  2012 bottombarred_pressed@2x~iphone.png
-rw-r--r--  1 root wheel  2191 Dec 22  2012 bottombarwhite@2x~iphone.png
-rw-r--r--  1 root wheel  2357 Dec 22  2012 bottombarwhite_pressed@2x~iphone.png
```

```

drwxr-xr-x  2 root wheel  646 Feb  7  2013 ca.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 cs.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 da.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 el.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 en_GB.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 fi.lproj
-rw-r--r--  1 root wheel  955 Dec 22  2012 hare@2x.png
drwxr-xr-x  2 root wheel  646 Feb  7  2013 he.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 hr.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 hu.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 id.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 ko.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 ms.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 no.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 pl.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 pt.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 pt_PT.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 ro.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 ru.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 sk.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 sv.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 th.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 tr.lproj
-rw-r--r--  1 root wheel  998 Dec 22  2012 turtle@2x.png
drwxr-xr-x  2 root wheel  646 Feb  7  2013 uk.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 vi.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 zh_CN.lproj
drwxr-xr-x  2 root wheel  646 Feb  7  2013 zh_TW.lproj

```

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



Figure 5- 16 Matching keywords

- Discover keyword “ACCESSIBILITY_DEFAULT_HEADSET”

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

```
FunMaker-4s:~ root# grep -r Incoming
/System/Library/PreferenceBundles/AccessibilitySettings.bundle
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.bundle/English.lproj/Accessibili
ty~iphone.strings matches
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.bundle/en_GB.lproj/Accessibility
~iphone.strings matches
```

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

```
snakeninnys-MacBook:~ snakeninny$ plutil -p ~/Accessibility/~iphone.strings
{
  "HAC_MODE_POWER_REDUCTION_N90" => "Hearing Aid Mode improves performance with some
hearing aids, but may reduce cellular reception."
  "LEFT_RIGHT_BALANCE_SPOKEN" => "Left-Right Stereo Balance"
  "QUICKSPEAK_TITLE" => "Speak Selection"
  "LeftStereoBalanceIdentifier" => "L"
  "ACCESSIBILITY_DEFAULT_HEADSET" => "Incoming Calls"
  "HEADSET" => "Headset"
  "CANCEL" => "Cancel"
  "ON" => "On"
  "CUSTOM_VIBRATIONS" => "Custom Vibrations"
  "CONFIRM_INVERT_COLORS_REMOVAL" => "Are you sure you want to disable inverted colors?"
  "SPEAK_AUTOCORRECTIONS" => "Speak Auto-text"
  "DEFAULT_HEADSET_FOOTER" => "Choose route for incoming calls."
  "HEARING_AID_COMPLIANCE_INSTRUCTIONS" => "Improves compatibility with hearing aids in
some circumstances. May reduce 2G cellular coverage."
  "DEFAULT_HEADSET" => "Default to headset"
  "ROOT_LEVEL_TITLE" => "Accessibility"
  "HEARING_AID_COMPLIANCE" => "Hearing Aid Mode"
  "CUSTOM_VIBES_INSTRUCTIONS" => "Assign unique vibration patterns to people in
Contacts. Change the default pattern for everyone in Sounds settings."
  "VOICEOVERTOUCH_TEXT" => "VoiceOver is for users with
blindness or vision disabilities."
  "IMPORTANT" => "Important"
  "COGNITIVE_HEADING" => "Learning"
  "HAC_MODE_EQUALIZATION_N94" => "Hearing Aid Mode improves audio quality with some
hearing aids."
  "SAVE" => "Save"
  "HOME_CLICK_TITLE" => "Home-click Speed"
  "AIR_TOUCH_TITLE" => "AssistiveTouch"
  "CONFIRM_ZOT_REMOVAL" => "Are you sure you want to disable Zoom?"
  "VOICEOVER_TITLE" => "VoiceOver"
  "OFF" => "Off"
  "GUIDED_ACCESS_TITLE" => "Guided Access"
  "ZOOMTOUCH_TEXT" => "Zoom is for users with low-vision acuity."
  "INVERT_COLORS" => "Invert Colors"
  "ACCESSIBILITY_SPEAK_AUTOCORRECTIONS" => "Speak Auto-text"
  "LEFT_RIGHT_BALANCE_DETAILS" => "Adjust the audio volume balance between left and
right channels."
  "MONO_AUDIO" => "Mono Audio"
  "CONTRAST" => "Contrast"
  "ZOOM_TITLE" => "Zoom"
  "TRIPLE_CLICK_HEADING" => "Triple-click"
```

```

"OK" => "OK"
"SPEAKER" => "Speaker"
"AUTO_CORRECT_TEXT" => "Automatically speak auto-corrections
and auto-capitalizations."
"HEARING" => "Hearing"
"LARGE_FONT" => "Large Text"
"CONFIRM_VOT_USAGE" => "VoiceOver"
"CONFIRM_VOT_REMOVAL" => "Are you sure you want to disable VoiceOver?"
"HEARING_AID_TITLE" => "Hearing Aids"
"FLASH_LED" => "LED Flash for Alerts"
"VISION" => "Vision"
"CONFIRM_ZOOM_USAGE" => "Zoom"
"DEFAULT" => "Default"
"MOBILITY_HEADING" => "Physical & Motor"
"TRIPLE_CLICK_TITLE" => "Triple-click Home"
"RightStereoBalanceIdentifier" => "R"
}

```

“ACCESSIBILITY_DEFAULT_HEADSET” => “Incoming Calls” gave me a very clear hint to continue the search.

- Locate Accessibility.plist

As you think, I’ve searched “ACCESSIBILITY_DEFAULT_HEADSET”:

```

FunMaker-4s:~ root# grep -r ACCESSIBILITY_DEFAULT_HEADSET
/System/Library/PreferenceBundles/AccessibilitySettings.bundle
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.bundle/Accessibility.plist
matches
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.bundle/Dutch.lproj/Accessibility
~iphone.strings matches
.....

```

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

```

snakeninnys-MacBook:~ snakeninny$ plutil -p ~/Accessibility.plist
{
  "title" => "ROOT_LEVEL_TITLE"
  "items" => [
    0 => {
      "label" => "VISION"
      "cell" => "PSGroupCell"
      "footerText" => "AUTO_CORRECT_TEXT"
    }
    1 => {
      "cell" => "PSLinkListCell"
      "label" => "VOICEOVER_TITLE"
      "detail" => "VoiceOverController"
      "get" => "voiceOverTouchEnabled:"
    }
    2 => {
      "cell" => "PSLinkListCell"
      "label" => "ZOOM_TITLE"
      "detail" => "ZoomController"
      "get" => "zoomTouchEnabled:"
    }
  ]
}

```

```

}
.....
18 => {
    "cell" => "PSLinkListCell"
    "label" => "HOME_CLICK_TITLE"
    "detail" => "HomeClickController"
    "get" => "homeClickSpeed:"
}
19 => {
    "detail" => "PSListItemsController"
    "set" => "accessibilitySetPreference:specifier:"
    "validValues" => [
        0 => 0
        1 => 1
        2 => 2
    ]
    "get" => "accessibilityPreferenceForSpecifier:"
    "validTitles" => [
        0 => "DEFAULT"
        1 => "HEADSET"
        2 => "SPEAKER"
    ]
    "requiredCapabilities" => [
        0 => "telephony"
    ]
    "cell" => "PSLinkListCell"
    "label" => "ACCESSIBILITY_DEFAULT_HEADSET"
    "key" => "DefaultRouteForCall"
}
]
}
}

```

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

5.3.3 Locate methods and functions

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



Figure 5- 17 A complaint email to Shoghian

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

```
FunMaker-4s:~ root# grep LibNotifyWatch: /var/log/syslog
Nov 26 00:09:20 FunMaker-4s Preferences[6488]: LibNotifyWatch: <CFNotificationCenter
0x1e875600 [0x39b4b100]> postNotificationName:UIViewAnimationDidCommitNotification
object:UIViewAnimationState userInfo:{
Nov 26 00:09:20 FunMaker-4s Preferences[6488]: LibNotifyWatch: <CFNotificationCenter
0x1e875600 [0x39b4b100]> postNotificationName:UIViewAnimationDidStopNotification
object:<UIViewAnimationState: 0x1ea74f20> userInfo:{
.....
Nov 26 00:09:21 FunMaker-4s Preferences[6488]: LibNotifyWatch:
CFNotificationCenterPostNotification center=<CFNotificationCenter 0x1dd86bd0
[0x39b4b100]> name=com.apple.accessibility.defaulttrouteforcall userInfo=(null)
deliverImmediately=1
Nov 26 00:09:21 FunMaker-4s Preferences[6488]: LibNotifyWatch: notify_post
com.apple.accessibility.defaulttrouteforcall
.....
```

I’ve found two notifications named “com.apple.accessibility.defaulttrouteforcall”. Combining them with previous mentioned deductions, there was no need to further explain.

After finding the most suspicious notification, I still had one more question: Where was the configuration file?

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

```
snakeninnys-MacBook:~ snakeninny$ plutil -p ~/com.apple.Accessibility.plist
{
    .....
    "DefaultRouteForCallPreference" => 2
    "VOTQuickNavEnabled" => 1
    "CurrentRotorTypeWeb" => 3
    "PunctuationKey" => 2
    .....
    "ScreenCurtain" => 0
    "VoiceOverTouchEnabled" => 0
    "AssistiveTouchEnabled" => 0
}
```

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

5.3.4 Test methods and functions

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

```
%hook SpringBoard
- (void)menuButtonDown:(id)down
{
    %orig;
    NSMutableDictionary *dictionary = [NSMutableDictionary
dictionaryWithContentsOfFile:@"~/var/mobile/Library/Preferences/com.apple.
Accessibility.plist"];
    [dictionary setObject:[NSNumber numberWithInt:2]
forKey:@"DefaultRouteForCallPreference"];
    [dictionary writeToFile:@"~/var/mobile/Library/Preferences/com.apple.
Accessibility.plist" atomically:YES];
    notify_post("com.apple.accessibility.defaultrouteforall");
}
%end
```


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

5.3.5 Write tweak

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

```
#import <notify.h>
#define ACCESSIBILITY @"/var/mobile/Library/Preferences/com.apple.Accessibility.plist"

// Required
extern "C" BOOL isCapable() {
    if (kCFCoreFoundationVersionNumber >= kCFCoreFoundationVersionNumber_iOS_5_0 &&
        [[[UIDevice currentDevice] model] isEqualToString:@"iPhone"])
        return YES;
    return NO;
}

// Required
extern "C" BOOL isEnabled() {
    NSMutableDictionary *dictionary = [[NSMutableDictionary alloc] initWithContentsOfFile:ACCESSIBILITY];
    BOOL result = [[dictionary objectForKey:@"DefaultRouteForCallPreference"] intValue] == 0 ? NO : YES;
    [dictionary release];
    return result;
}

// Optional
// Faster isEnabled. Remove this if it's not necessary. Keep it if isEnabled() is expensive and you can make it faster here.
extern "C" BOOL getStateFast() {
    return isEnabled();
}

// Required
extern "C" void setState(BOOL enabled) {
    NSMutableDictionary *dictionary = [[NSMutableDictionary alloc] initWithContentsOfFile:ACCESSIBILITY];
    [dictionary setObject:[NSNumber numberWithInt:(enabled ? 2 : 0)] forKey:@"DefaultRouteForCallPreference"];
    [dictionary writeToFile:ACCESSIBILITY atomically:YES]; [dictionary release];
    notify_post("com.apple.accessibility.defaultrouteformcall");
}

// Required
// How long the toggle takes to toggle, in seconds.
extern "C" float getDelayTime() {
    return 0.6f;
}
```


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



Figure 5- 18 Shoghian is the coauthor

SMSNinja (v1.3.1)	102947	0	102947
Speaker SBSettings Toggle (v0.0.1-3)	9522	0	9522
Characount for Notes (v1.0)	14447	0	14447

Figure 5- 19 Neary 10,000 downloads

5.4 Conclusion

In this chapter, we've comprehensively introduced how a tweak works as well as the thought and process of writing a tweak, accompanied with practical examples, I believe these contents can help beginners learn iOS reverse engineering better. iOS reverse engineering in

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

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

ARM related iOS reverse engineering

In previous chapters we have already introduced the fundamental knowledge and tool usage in iOS reverse engineering. Now, you should be able to satisfy your curiosity by playing with private methods and develop some mini tweaks. However, since you've come so far, I believe you have a strong delving spirit and truly want to improve your programmatic ability. If so, it'd be better for you to try something more challenging. Well, starting from this chapter, iOS reverse engineering will enter polar night, and you'll have to face the most arcane yet magical hieroglyphics in the programming world. Take a deep breath first, and then ask yourself, "Is iOS reverse engineering a right choice for me?" After finishing this chapter, hopefully you will get the answer.

Next, we'll meet the first advanced challenge in iOS reverse engineering: reading ARM assembly. According to the previous chapters, you have already got the idea that Objective-C code would become machine code after compiling, and then will be executed directly by CPU. It is overwhelming work to read machine code let alone write them. However, it's lucky that there is assembly, which bridges Objective-C code with machine code. Even though the readability of assembly is not as good as Objective-C, it's much better than machine code. If you can crash this hard nut, congratulations, you have the talents to be a reverse engineer. Conversely, if you cannot, AppStore may suit you better.

6.1 Introduction to ARM assembly

ARM assembly is a brand new language to most iOS developers. If your major in college is computer related, you may already have some impression about assembly. Actually, assembly is too esoteric for most college students; we're nervous and uncomfortable dealing with it. Is assembly really too hard to learn? Yes, it's obscure and difficult to understand. On the other

hand, however, as a human readable language, it is no much difference with other human languages, namely, if we use it more often, we will get familiar with it quicker.

As App developers, chances are rare for us to deal with assembly in our daily work. In this situation, if we don't practice deliberately, we cannot handle it for sure. In a nutshell, it's all about whether our time and energy is poured into learning it. Well, iOS reverse engineering offers us a great chance to learn ARM assembly. When we're reversing a function, we need to analyze massive lines of ARM assembly, and translate them to high-level language manually to reconstruct the functions. Even though there is no need to write assembly yet, a vast reading will definitely improve our understanding of it. ARM assembly is a necessity in iOS reverse engineering; you have to master it if you really want to be a member of this field. Like English, basic ARM assembly concepts correspond to 26 letters and phonetic symbols in English; its instructions correspond to words, and instructions' variants correspond to different word tenses; its calling conventions correspond to grammars, which define the connection between words. Sounds not that bad, right? Let's delve into it step by step.

6.1.1 Basic concepts

For a thorough introduction to ARM assembly, the ARM Architecture Reference Manual does a great job. However, as rookies, most of us don't need a thorough introduction at all, the thousands pages ARM Architecture Reference Manual is no better than my limited knowledge about ARM assembly, which is enough and fits junior iOS reverse engineers better. With the release of iPhone 5s, Apple brings in the more powerful 64-bit processor, arm64. However, the tools introduced in the previous chapters do not fully support arm64. Therefore, the following chapters will still focus on 32-bit processors, i.e. armv7 and armv7s. Nonetheless, the general methods and thoughts work on both 32-bit and 64-bit processors.

- Register, memory, and stack

In high-level languages like Objective-C, C, and C++, our operands are variables; whereas in ARM assembly, the operands are registers, memory, and stack. Registers can be regarded as CPU built-in variables; their amounts are often very limited. If we need more variables, we can put them in memory. However, this is a trade off between performance and amounts; memory operation is slower than register operation.

In fact, stack is in memory as well. But it works like a stack, i.e. follows the “first in last out” rule. The stack of ARM is full descending, meaning that the stack grows towards lower address, the latest object is placed at the bottom, which is at the lowest address, as shown in the figure 6-1.

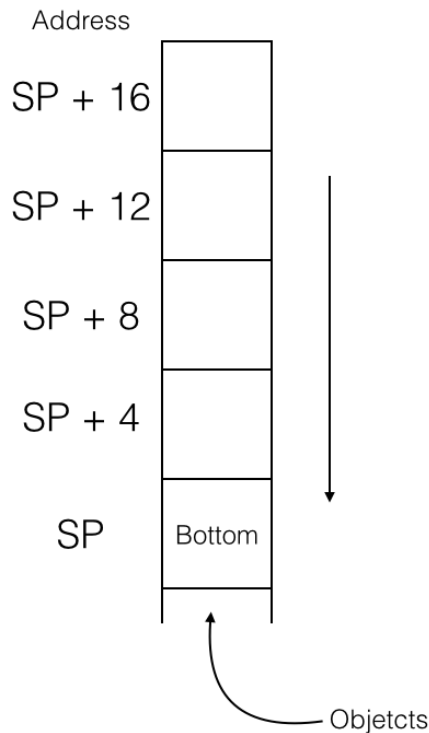


Figure 6-1 The stack of ARM

A register, named “stack pointer” (hereafter referred to as SP), holds the bottom address of stack, i.e. the stack address. We can push a register into stack to save its value, or pop a register out of stack to load its value. During process running, SP changes a lot, but before and after a block of code is executed, SP should stay the same, otherwise there will be a fatal problem.

Why? Let’s take an example:

```
static int global_var0;
static int global_var1;

...

void foo(void)
{
    bar();
    // other operations;
}
```

In the above code snippet, suppose that `foo()` uses registers A, B, C, and D; `foo()` calls `bar()`, and suppose that `bar()` uses registers A, B, and C. Because registers A, B and C are overlapped in `foo()` and `bar()`, `bar()` needs to save values of A, B, and C into stack before it starts execution.

Also, it needs to restore these 3 registers from stack before it ends execution, to make sure `foo()` can work correctly. Let's look at some pseudo code:

```
// foo()
foo:
    // Push A, B, C, D into stack, save their values
    push    {A, B, C, D}
    // Use A ~ D
    move    A, #1        // A = 1
    move    B, #2        // B = 2
    move    C, #3        // C = 3
    call    bar
    move    D, global_var0
    // global_var1 = A + B + C + D
    add     A, B         // A = A + B, notice A's value
    add     A, C         // A = A + C, notice A's value
    add     A, D         // A = A + D, notice A's value
    move    global_var1, A
    // Pop A, B, C, D out of stack, restore their values
    pop     {A-D}
    return

// bar()
bar:
    // Push A,B,C into the stack, store their values
    push    {A-C}
    // Use A ~ C
    move    A, #2        // Do you know what this instruction do?
    move    B, #5
    move    C, A
    add     C, B         // C = 7
    // global_var0 = A + B + C (== 2 * C)
    add     C, C
    move    global_var0, C        // A = 2, B = 5, C = 14

    // Do you get the meaning of push and pop now?
    pop     {A-C}
    return
```

Let's shortly explain this snippet of pseudo code: firstly, `foo()` sets registers A, B and C to 1, 2 and 3 respectively, then calls `bar()`, which changes values of A, B and C as well sets `global_var0`, a global variable, to the sum of registers A, B and C. If we directly use the current values of A, B and C to calculate the value of `global_var1` for now, then the result would be wrong. So before executing `bar()`, values of A, B and C should be pushed into stack first, and pop them out after the execution of `bar()` for restoration, then we can get a correct `global_var1`. Notice that, for the same reason, `foo()` has done the same operations on A, B, C and D, which saves its callers' days.

- Preserved registers

Some registers in ARM processors must preserve their values after a function call, as shown below:

R0-R3	Passes arguments and return values
R7	Frame pointer, which points to the previously saved stack frame and the saved link register
R9	Reserved by system before iOS 3.0
R12	IP register, used by dynamic linker
R13	Stack Pointer, i.e. SP
R14	Link Register, i.e. LR, saves function return address
R15	Program Counter, i.e. PC

We're not writing ARM assembly yet, so treat the above table as a reference would be enough.

- Branches

The process saves the address of the next instruction in PC register. Usually, CPU will execute instructions in order. When it has done with one instruction, PC will increase 1 to point to the next instruction, as shown in figure 6-2.

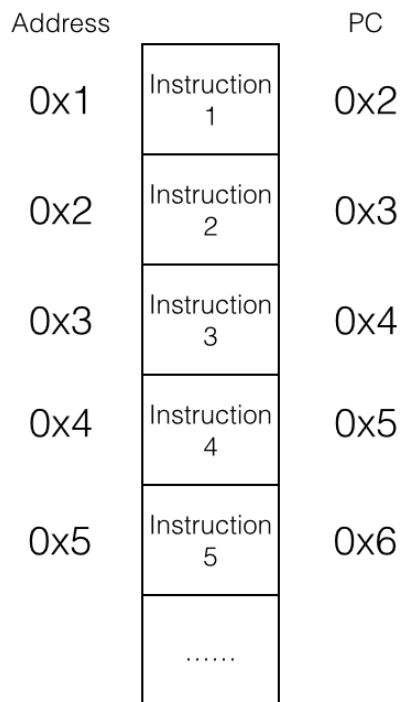


Figure 6-2 Execute instructions in order

The processor will execute instructions from 1 to 5 in a plain and trivial way. However, if we change the value of PC, the execution order will be very different, as shown in figure 6-3.

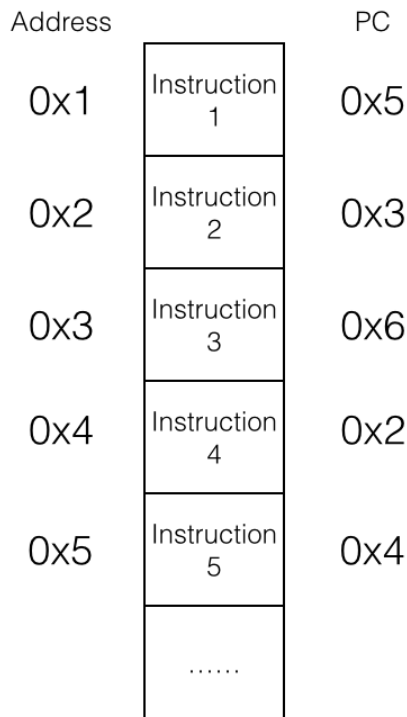


Figure 6-3 Execute instructions out of order

The instructions' execution has been disordered to 1, 5, 4, 2, 3 and 6, which is bizarre and remarkable. This kind of "disorder" is officially called "branch" or "jump", which makes loop and subroutine possible. For example:

```
// endless()
endless:
    operate    op1, op2
    branch    endless
    return    // Dead loop, we cannot reach here!
```

In actual cases, conditional branches, which are triggered under some specific conditions, are the most practical branches. "if else" and "while" are both based on conditional branches. In ARM assembly, there are 4 kinds of conditional branches:

- ✧ The result of operation is zero (or non-zero).
- ✧ The result of operation is negative.
- ✧ The result of operation has carry.
- ✧ The operation overflows (for example, the sum of two positive numbers exceeds 32 bits).

These operation results are often represented as flags and are saved in the Program Status Register (PSR). Some instructions will change these flags according to their operation results, and conditional branches decide whether to branch according to these flags. The pseudo code below shows an example of for loop:


```

for:
    add     A, #1
    compare A, #16
    bne    for // If A - 16 != 0 then jump to for

```

The above code compares A and #16, if they're not equal, increase A by 1 and compare again. Otherwise break out the loop and go on to the next instruction.

6.1.2 Interpretation of ARM/THUMB instructions

ARM processors use 2 different instruction sets: ARM and THUMB. The length of ARM instructions is universally 32 bits, whereas it's 16 bits for THUMB instructions. Broadly, both sets have 3 kinds of instructions: data processing instructions, register processing instructions, and branch instructions.

- Data processing instructions

There're 2 rules in data processing instructions:

- ✧ All operands are 32 bits.
- ✧ All results are 32 bits, and can only be stored in registers.

In a nutshell, the basic syntax of data processing instructions is:

```
op{cond}{s} Rd, Rn, Op2
```

“cond” and “s” are two optional suffixes. “cond” decides the execution condition of “op”, and there are 17 conditions:

EQ	The result equals to 0 (Equal to 0)
NE	The result doesn't equal to 0 (Not Equal)
CS	The operation has carry or borrow (Carry Set)
HS	Same to CS (unsigned Higher or Same)
CC	The operation has no carry or borrow (Carry Clear)
LO	Same to CC (unsigned LOwer)
MI	The result is negative (MInus)
PL	The result is greater than or equal to 0 (PLus)
VS	The operation overflows (oVerflow Set)
VC	The operation doesn't overflow (oVerflow Clear)
HI	If operand1 is unsigned HIgher than operand2
LS	If operand1 is unsigned LOwer or Same than operand2
GE	If operand1 is signed Greater than or Equal to operand2
LT	If operand1 is signed Less Than operand2
GT	If operand1 is signed Greater Than operand2
LE	If operand1 is signed Less than or Equal operand2
AL	Always, this is the default

“cond” is easy to use, for example:

```

compare R0, R1
moveGE  R2, R0
moveLT  R2, R1

```

Compare R0 with R1, if R0 is greater than or equal to R1, then R2 = R0, otherwise R2 = R1.

“s” decides whether “op” sets flags or not, there are 4 flags:

N (Negative)

If the result is negative then assign 1 to N, otherwise assign 0 to N.

Z (Zero)

If the result is zero then assign 1 to Z, otherwise assign 0 to Z.

C (Carry)

For add operations (including CMN), if they have carry then assign 1 to C, otherwise assign 0 to C; for sub operations (including CMP), Carry acts as Not-Borrow, if borrow happens then assign 0 to C, otherwise assign 1 to C; for shift operations (excluding add or sub), assign C the last bit to be shifted out; for the rest of operations, C stays unchanged.

V (oVerflow)

If the operation overflows then assign 1 to V, otherwise assign 0 to V.

One thing to note, C flag works on unsigned calculations, whereas V flag works on signed calculations.

Data processing instructions can be divided into 4 kinds:

- Arithmetic instructions

```
ADD R0, R1, R2      ; R0 = R1 + R2
ADC R0, R1, R2      ; R0 = R1 + R2 + C(carry)
SUB R0, R1, R2      ; R0 = R1 - R2
SBC R0, R1, R2      ; R0 = R1 - R2 - !C
RSB R0, R1, R2      ; R0 = R2 - R1
RSC R0, R1, R2      ; R0 = R2 - R1 - !C
```

All arithmetic instructions are based on ADD and SUB. RSB is the abbreviation of “Reverse SuB”, which just reverse the two operands of SUB; instructions ending with “C” stands for ADD with carry or SUB with borrow, and they will assign 1 to C flag when there is carry or there isn’t borrow.

- Logical operation instructions

```
AND R0, R1, R2      ; R0 = R1 & R2
ORR R0, R1, R2      ; R0 = R1 | R2
EOR R0, R1, R2      ; R0 = R1 ^ R2
BIC R0, R1, R2      ; R0 = R1 &~ R2
MOV R0, R2           ; R0 = R2
MVN R0, R2           ; R0 = ~R2
```

There is not much to explain about these instructions with their corresponding C operators. You may have noticed that there’s no shift instruction, because ARM uses barrel shift with 4 instructions:

```
LSL           Logical Shift Left, as shown in figure 6-4
```



Figure 6-4 LSL

LSR Logical Shift Right, as shown in figure 6-5



Figure 6-5 LSR

ASR Arithmetic Shift Right, as shown in figure 6-6

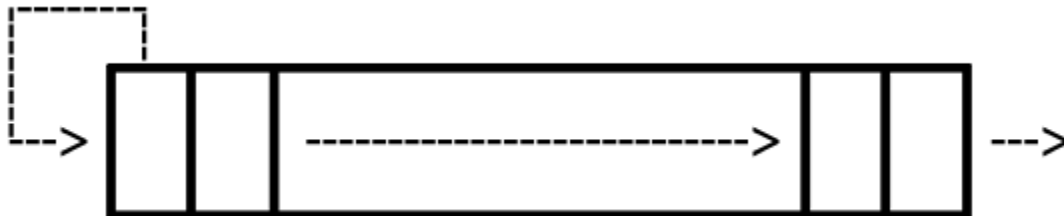


Figure 6-6 ASR

ROR Rotate Right, as shown in figure 6-7

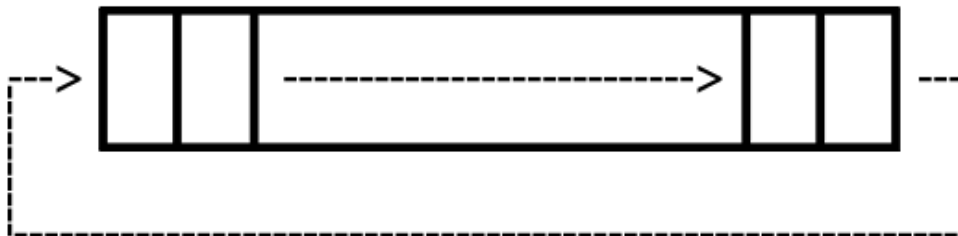


Figure 6-7 ROR

- Compare instructions

```

CMP R1, R2 ; Set flag according to the result of R1 - R2
CMN R1, R2 ; Set flag according to the result of R1 + R2
TST R1, R2 ; Set flag according to the result of R1 & R2
TEQ R1, R2 ; Set flag according to the result of R1 ^ R2

```

Compare instructions are just arithmetic or logical operation instructions that change flags, but they don't save the results in registers.

- Multiply instructions

```

MUL R4, R3, R2 ; R4 = R3 * R2
MLA R4, R3, R2, R1 ; R4 = R3 * R2 + R1

```

The operands of multiply instructions must come from registers.

- Register processing instructions

The basic syntax of register processing instructions is:

```
op{cond}{type} Rd, [Rn, Op2]
```

Rn, the base register, stores base address; the function of “cond” is the same to data processing instructions; “type” decides the data type which “op” operates, there are 4 types:

B (unsigned Byte)

Extends to 32 bits when executing, filled with 0.

SB (Signed Byte)

For LDR only; extends to 32 bits when executing, filled with the sign bit.

H (unsigned Halfword)

Extends to 32 bits when executing, filled with 0.

SH (Signed Halfword)

For LDR only; extends to 32 bits when executing, filled with the sign bit.

The default data type is word if no “type” is specified.

There are only 2 basic register processing instructions: LDR (LoaD Register), which reads data from memory then write to register; and STR (STore Register), which reads data from register then write to memory. They’re used like this:

◇ LDR

```
LDR Rt, [Rn {, #offset}] ; Rt = *(Rn {+ offset}), {} is optional
LDR Rt, [Rn, #offset]! ; Rt = *(Rn + offset); Rn = Rn + offset
LDR Rt, [Rn], #offset ; Rt = *Rn; Rn = Rn + offset
```

◇ STR

```
STR Rt, [Rn {, #offset}] ; *(Rn {+ offset}) = Rt
STR Rt, [Rn, #offset]! ; *(Rn {+ offset}) = Rt; Rn = Rn + offset
STR Rt, [Rn], #offset ; *Rn = Rt; Rn = Rn + offset
```

Besides, LDRD and STRD, the variants of LDR and STR, can operate doubleword, namely, LDR or STR two registers at once. The syntax of them is:

```
op{cond} Rt, Rt2, [Rn {, #offset}]
```

The use of LDRD and STRD is just like LDR and STR:

◇ STRD

```
STRD R4, R5, [R9, #offset] ; *(R9 + offset) = R4; *(R9 + offset + 4) = R5
```

◇ LDRD

```
LDRD R4, R5, [R9, #offset] ; R4 = *(R9 + offset); R5 = *(R9 + offset + 4)
```

Beside LDR and STR, LDM (LoaD Multiple) and STM (STore Multiple) can process several registers at the same time like this:

```
op{cond}{mode} Rd{!}, reglist
```

Rd is the base register, and the optional “!” decides whether the modified Rd is written back to the original Rd if “op” modifies Rd; reglist is a list of registers which are curly braced and separated by “,”, or we can use “-” to represent a scope, such as {R4 – R6, R8} stands for R4, R5, R6 and R8; these registers are ordered according to their numbers, regardless of their positions inside the braces.

Attention, the operation direction of LDM and STM is opposite to LDR and STR; LDM reads memory starting from Rd then write to reglist, while STM reads from reglist then write to memory starting from Rd. This is a little confusing; please don’t mess up.

The function of “cond” is the same to data processing instructions. And, “mode” specifies how Rd is modified, including 4 cases:

IA (Increment After)
Increment Rd after “op”.

IB (Increment Before)
Increment Rd before “op”.

DA (Decrement After)
Decrement Rd after “op”.

DB (Decrement Before)
Decrement Rd before “op”.

What do they mean? We will use LDM as an example. As figure 6-8 shows, R0 points to 5 currently.

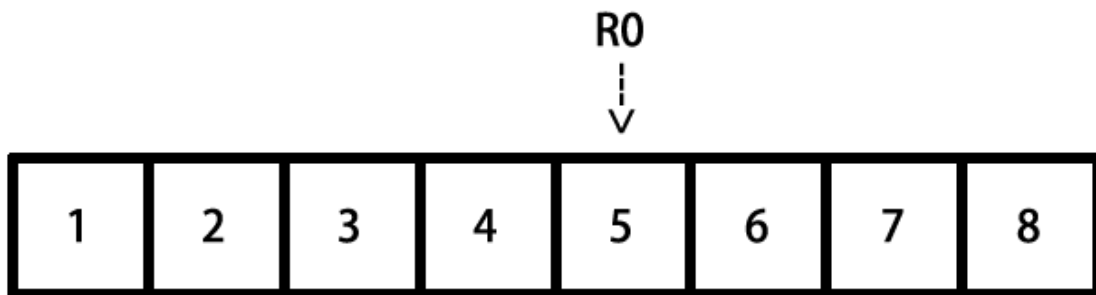


Figure 6-8 Simulation of LDM

After executing the following instructions, R4, R5 and R6 will change to:

```
LDMIA R0, {R4 – R6} ; R4 = 5, R5 = 6, R6 = 7
LDMIB R0, {R4 – R6} ; R4 = 6, R5 = 7, R6 = 8
LDMDA R0, {R4 – R6} ; R4 = 5, R5 = 4, R6 = 3
LDMDB R0, {R4 – R6} ; R4 = 4, R5 = 3, R6 = 2
```

STM works similarly. Notice again, the operation direction of LDM and STM is just opposite to LDR and STR.

- Branch instructions

Branch instructions can be divided into 2 kinds: unconditional branches and conditional branches.

✧ Unconditional branches

```
B Label          ; PC = Label
BL Label         ; LR = PC - 4; PC = Label
BX Rd           ; PC = Rd ,and switch instruction set
```

Unconditional branches are easy to understand, for example:

```
foo():
    B Label          ; Jump to Label to keep executing
    .....          ; Can't reach here
Label:
    .....
```

✧ Conditional branches

The “cond” of conditional branches are decided by the 4 flag mentioned in section 6.2.1, their correspondences are:

cond	flag
EQ	Z = 1
NE	Z = 0
CS	C = 1
HS	C = 1
CC	C = 0
LO	C = 0
MI	N = 1
PL	N = 0
VS	V = 1
VC	V = 0
HI	C = 1 & Z = 0
LS	C = 0 Z = 1
GE	N = V
LT	N != V
GT	Z = 0 & N = V
LE	Z = 1 N != V

Before every conditional branch there will be a data processing instruction to set the flag, which determines if the condition is met or not, hence influence the code execution flow.

```
Label:
    LDR R0, [R1], #4
    CMP R0, 0          ; If R0 == 0 then Z = 1; else Z = 0
    BNE Label         ; If Z == 0 then jump
```

- THUMB instructions

THUMB instruction set is a subset of ARM instruction set. Every THUMB instruction is 16 bits long, so THUMB instructions are more space saving than ARM instructions, and can be faster transferred on 16-bit data bus. However, you can't make an omelet without breaking eggs. All THUMB instructions except “b” can't be executed conditionally; barrel shift can't

cooperate with other instructions; most THUMB instructions can only make use of registers R0 to R7, etc. Compared with ARM instructions, the features of THUMB instructions are:

- ✧ There're less THUMB instructions than ARM instructions

Since THUMB is just a subset, the number of THUMB instructions is definitely less. For example, among all multiply instructions, only MUL is kept in THUMB.

- ✧ No conditional execution

Except branch instructions, other instructions cannot be executed conditionally.

- ✧ All THUMB instructions set flags by default

- ✧ Barrel shift cannot cooperate with other instructions

Shift instructions can only be executed alone, say:

```
LSL R0, #2
```

But cannot:

```
ADD R0, R1, LSL #2
```

- ✧ Limitation of registers

Unless declared explicitly, THUMB instructions can only make use of R0 to R7. However, there are exceptions: ADD, MOV, and CMP can use R8 to R15 as operands; LDR and STR can use PC or SP; PUSH can use LR, POP can use PC; BX can use all registers.

- ✧ Limitation of immediate values and the second operand

Most of THUMB instructions' formats are "op Rd, Rm", excluding shift instructions, ADD, SUB, MOV and CMP.

- ✧ Doesn't support data write back

All THUMB instructions do not support data write back i.e. "!", except LDMIA and STMIA.

We will see the instructions mentioned above a lot during the junior stage of iOS reverse engineering. If you only have a smattering of the knowledge so far, take it easy. Get your hands dirty and analyze several binaries from now on, you will gradually get familiar with ARM assembly. This section is just an introduction, if you have any questions about instructions in practice, ARM Architecture Reference Manual on <http://infocenter.arm.com> will always be the best reference for you. Of course, things discussed on <http://bbs.iosre.com> are also worth to have a look.

6.1.3 ARM calling conventions

After a brief look at the commonly used ARM instructions, I believe you can barely read the assembly of a function for now. When a function calls another function, arguments and return values need to be passed between the caller and the callee. The rule of how to pass them is called ARM calling conventions.

- Prologs and epilogs

We've mentioned in section 6.1.1 that "before and after a block of code is executed, SP should stay the same, otherwise there will be a fatal problem". This goal is achieved by the cooperation of prolog and epilog of this code block. Generally, prolog does these:

- ✧ PUSH LR;
- ✧ PUSH R7;
- ✧ R7 = SP;
- ✧ PUSH registers that must be preserved;
- ✧ Allocates space in the stack frame for local storage.

And epilog does an opposite job to prolog:

- ✧ Deallocates space that the prolog allocates;
- ✧ POP preserved registers;
- ✧ POP R7;
- ✧ POP LR, and PC = LR.

However, the work of prolog and epilog is not indispensable. If the code block doesn't make use of a register at all, then there is no need to push it onto stack. In iOS reverse engineering, prologs and epilogs may change the value of SP, which deserves our attention. We'll come across this situation in chapter 10; review this section when you get there.

- Pass arguments and return values

If you want to delve deeper into how arguments and return values are passed, you can read http://infocenter.arm.com/help/topic/com.arm.doc.ih0042e/IHI0042E_aapcs.pdf. However, in the majority of cases, you just need to remember "sentence of the book":

“The first 4 arguments are saved in R0, R1, R2 and R3; the rest are saved on the stack; the return value is saved in R0.”

A concise but informative sentence, right? To make a deeper impression, let’s see an example:

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --show-sdk-path` -o MainBinary main.m

#include <stdio.h>

int main(int argc, char **argv)
{
    printf("%d, %d, %d, %d, %d", 1, 2, 3, 4, 5);
    return 6;
}
```

Save this code snippet as main.m, and compile it with the sentence in comments. Then drag and drop MainBinary into IDA and locate to main, as shown in figure 6-9.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT __main
__main
var_20= -0x20
var_1C= -0x1C
var_18= -0x18
var_14= -0x14
var_10= -0x10
var_C= -0xC

PUSH      {R4,R5,R7,LR}
ADD       R7, SP, #8
SUB       SP, SP, #0x18
MOV       R2, #(aDDDDD - 0xBF6A) ; "%d, %d, %d, %d, %d"
ADD       R2, PC ; "%d, %d, %d, %d, %d"
MOVS      R3, #1
MOV       R9, #2
MOV       R12, #3
MOV       LR, #4
MOVS      R4, #5
MOVS      R5, #0
STR       R5, [SP,#0x20+var_C]
STR       R0, [SP,#0x20+var_10]
STR       R1, [SP,#0x20+var_14]
MOV       R0, R2 ; char *
MOV       R1, R3
MOV       R2, R9
MOV       R3, R12
STR.W     LR, [SP,#0x20+var_20]
STR       R4, [SP,#0x20+var_1C]
BLX       __printf
MOVS      R1, #6
STR       R0, [SP,#0x20+var_18]
MOV       R0, R1
ADD       SP, SP, #0x18
POP       {R4,R5,R7,PC}
; End of function __main
; __text ends
```

Figure 6-9 main in assembly

“BLX _printf” calls printf, and its 6 arguments are stored in R0, R1, R2, R3, [SP, #0x20 + var_20], and [SP, #0x20 + var_1C] respectively; the return value is stored in R0. Because var_20 = -0x20, var_1C = -0x1C, 2 arguments in the stack are at [SP] and [SP, #0x4].

I don’t think we need further explanation.

“The first 4 arguments are saved in R0, R1, R2 and R3; the rest are saved on the stack; the return value is saved in R0.”

Promise me you'll remember "sentence of the book", which is the key to most problems in iOS reverse engineering!

This section just walked you through the most basic knowledge about ARM assembly; there were omissions for sure. However, to be honest, with "sentence of the book" and the official site of ARM, you can start reversing 99% of all Apps. Next, it's time for us to figure out how to use the knowledge we have just learned in practical iOS reverse engineering.

6.2 Advanced methodology of writing a tweak

In "Methodology of writing a tweak" of chapter 5, we have concluded the methodology into 5 steps: 1. look for inspiration; 2. locate target files; 3. locate target functions; 4. test private methods; 5. analyze method arguments. These steps seem reasonable, but the most important step "locate target functions" is lame and untenable. Can we refer to "look for interesting keywords in class-dump headers" as "locate target functions"? No.

In the vast majority of cases, only 2 elements of an App attract our interests: its function and its data. What if we discover an interesting function, but fail to find the related keywords in class-dump headers? And how can we track an interesting data till we know how it's generated? In these cases, class-dump is all thumbs. Thus, "look for interesting keywords in class-dump headers" is just one scenario in "locate target functions", we've overgeneralized. Therefore, in more general cases, how should we locate target functions?

Functions and data that we're interested in, are all presented in software in some intuitive forms that we can see or feel. For example, figure 6-10 shows Mail App (hereafter referred to as Mail), and the button at the right bottom has the function of composing an email; figure 6-11 shows phone settings view in Settings App (hereafter referred to as MobilePhoneSettings), its top cell shows my number. App functions are provided by programmatic functions, and data is generated by programmatic functions as well. That's to say, from programmatic point of view, the nature of what we're interested in is programmatic functions. So, "locate target functions" is actually the process of how we locate the source functions of our interested Apps' visual expressions.

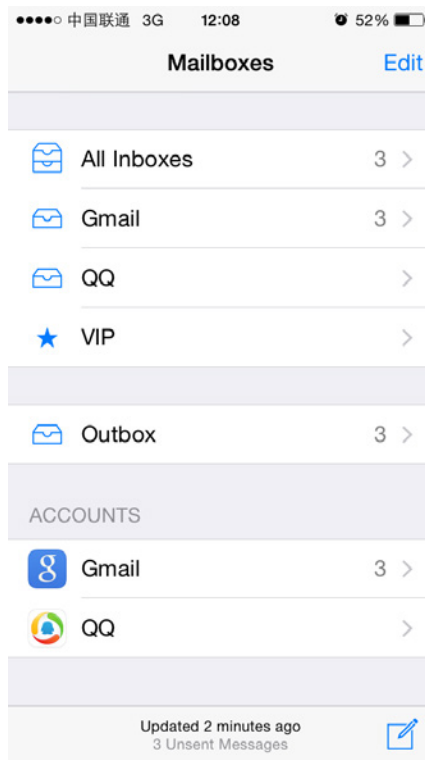


Figure 6- 10 Mail



Figure 6- 11 MobilePhoneSettings

Facing such demands, class-dump is quite helpless. Luckily, we have already learned how to use Cycript, IDA and LLDB, and gained some basic knowledge about ARM assembly; with their help, there are patterns to follow for “locate target functions”. For most of us, among all iOS software, we know Apps the best, so if we’re to choose something as our junior reverse targets,

nothing is more appropriate than Apps. As a result, in the following sections, we will take Apps as examples, and try to refine “locate target functions” with ARM level reverse engineering, as well enhance the methodology of writing a tweak.

6.2.1 Cut into the target App and find the UI function

For an App, what we’re interested in are regularly presented on UI, which exhibits execution processes and results. The relationship between function and UI is very tight, if we can get the UI object that interests us, we can find its corresponding function, which is referred to as “UI function”. The process of getting the programmatic UI object of our interested visual UI control object, then further getting the UI function of the programmatic UI object is usually implemented with Cycript, with the magic private method “recursiveDescription” in UIView and the undervalued public method “nextResponder” in UIResponder. In the rest of this chapter, I will explain this process by taking Mail as the example to summarize the methodology, and then apply the methodology to MobilePhoneSettings to give you a deeper impression. All the work is finished on iPhone 5, iOS 8.1.

1. Inject Cycript into Mail

Firstly use the skill mentioned in section “dumpdecrypted” to locate the process name of Mail, and inject with Cycript:

```
FunMaker-5:~ root# ps -e | grep /Applications
363 ??      0:06.94 /Applications/MobileMail.app/MobileMail
596 ??      0:01.50
/Applications/MessagesNotificationViewService.app/MessagesNotificationViewService
623 ??      0:08.50 /Applications/InCallService.app/InCallService
713 ttys000  0:00.01 grep /Applications
FunMaker-5:~ root# cycript -p MobileMail
```

2. Examine the view hierarchy of “Mailboxes” view, and locate “compose” button

The private method [UIView recursiveDescription] returns the view hierarchy of UIView. Normally, the current view is consists of at least one UIWindow object, and UIWindow inherits from UIView, so we can use this private method to examine the view hierarchy of current view. Its usage follows this pattern:

```
cy# ?expand
expand == true
```

First of all, execute “?expand” in Cypript to turn on “expand”, so that Cypript will translate control characters such as “\n” to corresponding formats and give the output a better readability.

```
cy# [[UIApp keyWindow] recursiveDescription]
```

UIApp is the abbreviation of [UIApplication sharedApplication], they’re equivalent. Calling the above method will print out view hierarchy of keyWindow, and output like this:

```
@"<UIWindow: 0x14587a70; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x147166b0>; layer = <UIWindowLayer: 0x14587e30>>
  | <UIView: 0x146e6180; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x146e98d0>; layer = <CALayer: 0x146e61f0>>
    | | <UIView: 0x146e5f60; frame = (0 0; 320 568); layer = <CALayer: 0x1460ec40>>
      | | | <_MFACTORItemView: 0x14506a30; frame = (0 0; 320 568); layer = <CALayer: 0x14506c10>>
        | | | | <UIView: 0x145074b0; frame = (-0.5 -0.5; 321 569); alpha = 0; layer = <CALayer: 0x14507520>>
          | | | | <_MFACTORSnapshotView: 0x14506f70; baseClass = UISnapshotView; frame = (0 0; 320 568); clipsToBounds = YES; hidden = YES; layer = <CALayer: 0x145071c0>>
            .....
              | | <MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568); userInteractionEnabled = NO; gestureRecognizers = <NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>
                | | | <UIScrollView: 0x146bfa90; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x146e1e90>; layer = <CALayer: 0x146c8740>; contentOffset: {0, 0};
                  contentSize: {320, 77.5}>
                    | | | <_TabGradientView: 0x146e7010; frame = (-320 -508; 960 568); alpha = 0;
                      userInteractionEnabled = NO; layer = <CAGradientLayer: 0x146e7d80>>
                        | | | <UIView: 0x146e29c0; frame = (-10000 568; 10320 10000); layer = <CALayer: 0x146e2a30>>"
```

Description of every subview and sub-subview of keyWindow will be completely presented in <.....>, including their memory addresses, frames and so on. The indentation spaces reflect the relationship between views. Views on the same level will have same indentation spaces, such as UIScrollView, _TabGradientView and UIView at the bottom; and less indented views are the superviews of more indented views, for example, UIScrollView, _TabGradientView, and UIView are subviews of MFTiltedTabView. By using “#” in Cypript, we can get any view object in keyWindow like this:

```
cy# tableView = #0x146e1af0
#"<MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568); userInteractionEnabled = NO; gestureRecognizers = <NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>"
```

Of course, through other methods of UIApplication and UIView, it is also feasible to get views we are interested in, for example:

```
cy# [UIApp windows]
@[#"<UIWindow: 0x14587a70; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x147166b0>; layer = <UIWindowLayer: 0x14587e30>>",#"<UITextEffectsWindow: 0x15850570; frame = (0 0; 320 568); opaque = NO; gestureRecognizers = <NSArray: 0x147503e0>; layer = <UIWindowLayer: 0x1474ff10>>"]
```

The above code can get all windows of this App:

```

cy# [#0x146e1af0 subviews]
@[#"<UIScrollView: 0x146bfa90; frame = (0 0; 320 568); gestureRecognizers = <NSArray:
0x146e1e90>; layer = <CALayer: 0x146c8740>; contentOffset: {0, 0}; contentSize: {320,
77.5}>"#,#"<_TabGradientView: 0x146e7010; frame = (-320 -508; 960 568); alpha = 0;
userInteractionEnabled = NO; layer = <CAGradientLayer: 0x146e7d80>>"#,#"<UIView:
0x146e29c0; frame = (-10000 568; 10320 10000); layer = <CALayer: 0x146e2a30>>"]
cy# [#0x146e29c0 superview]
#"<MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568); userInteractionEnabled = NO;
gestureRecognizers = <NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>"

```

The above code can get subviews and superviews. In a word, we can get any view objects that are visible on UI by combining the above methods, which lays the foundation for our next steps.

In order to locate “compose” button, we need to find out the corresponding control object. To accomplish this, the regular approach is to examine control objects one by one. For views like <UIView: viewAddress; ...>, we call [#viewAddress setHidden:YES] for everyone of them, and the disappeared control object is the matching one. Of course, some tricks could accelerate the examination; because on the left side of this button there’re two lines of sentences, we can infer that the button shares the same superview with this two sentences; if we can find out the superview, the rest of work is only examining subviews of this superview, hence reduce our work burden. Commonly, texts will be printed in description, so we can directly search “3 Unsent Messages” in recursiveDescription:

```

| | | | | | | | | | <MailStatusUpdateView: 0x146e6060; frame = (0 0;
182 44); opaque = NO; autoresize = W+H; layer = <CALayer: 0x146c8840>>
| | | | | | | | | | <UILabel: 0x14609610; frame = (40 21.5; 102
13.5); text = ‘3 Unsent Messages’; opaque = NO; userInteractionEnabled = NO; layer =
<UILabelLayer: 0x146097f0>>

```

Thereby, we get its superview, i.e. MailStatusUpdateView. If the button is a subview of MailStatusUpdateView, then when we call [MailStatusUpdateView setHidden:YES], the button would disappear. Let’s try it out:

```

cy# [#0x146e6060 setHidden:YES]

```

However, only the sentences are hidden, the button remains visible, as shown in figure 6-12:


```

| | | | | | | | <UIImageView: 0x14725730; frame = (0 -0.5; 320 0.5);
autorelease = W+BM; userInteractionEnabled = NO; layer = <CALayer: 0x1472be40>>
| | | | | | | | <MailStatusBarView: 0x146c4110; frame = (69 0; 182
44); opaque = NO; autoresize = BM; layer = <CALayer: 0x146f9f90>>

```

Let's repeat the operation to hide UIToolBar:

```

cy# [#0x146c4110 setHidden:NO]
cy# [#0x146f62a0 setHidden:YES]

```

The effect is shown in figure 6-13:

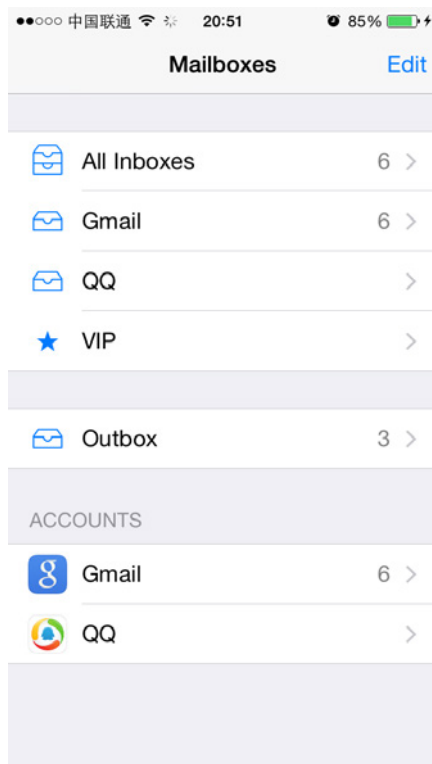


Figure 6-13 UIToolBar is hidden

This time, the button is hidden, which means the button is a subview of UIToolBar. Look for keyword “button” in subviews of UIToolBar, and we can easily locate UIToolbarButton:

```

| | | | | | | | <MailStatusBarView: 0x146c4110; frame = (69 0; 182
44); opaque = NO; autoresize = BM; layer = <CALayer: 0x146f9f90>>
| | | | | | | | <MailStatusUpdateView: 0x146e6060; frame = (0 0;
182 44); opaque = NO; autoresize = W+H; layer = <CALayer: 0x146c8840>>
| | | | | | | | <UILabel: 0x14609610; frame = (40 21.5; 102
13.5); text = '3 Unsent Messages'; opaque = NO; userInteractionEnabled = NO; layer =
<UILabelLayer: 0x146097f0>>
| | | | | | | | <UILabel: 0x145f3020; frame = (43 8; 96.5
13.5); text = 'Updated Just Now'; opaque = NO; userInteractionEnabled = NO; layer =
<UILabelLayer: 0x145f2e50>>
| | | | | | | | <UIToolbarButton: 0x14798410; frame = (285 0; 23 44);
opaque = NO; gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>

```

Let's see whether it is “compose” button with the following commands:

```

cy# [#0x146f62a0 setHidden:NO]
cy# [#0x14798410 setHidden:YES]

```

The button is hidden as expected, as shown in figure 6-14:

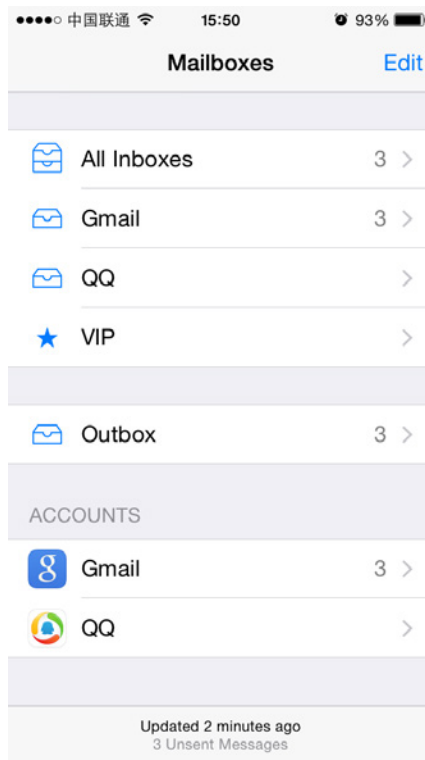


Figure 6-14 Button is hidden

Now, we have successfully located “compose” button, and its description is `<UIToolbarButton: 0x14798410; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>`. Next, we need to find out its UI function.

3. Find out UI function of “compose” button

UI function of a button is its response method after tapping it. Usually we use `[UIControl addTarget:action:forControlEvents:]` to add a response method to a UIView object (I haven’t seen any exceptions so far). Meanwhile, the method `[UIControl actionsForTarget:forControlEvents:]` offers a way to get the response method of a UIControl object. Based on this, as long as the view we get in the last step is a subclass of UIControl (Again, I haven’t seen any exceptions so far), we can find out its response method. More specifically in this example, we do it like this:

```
cy# button = #0x14798410
#"<UIToolbarButton: 0x14798410; frame = (285 0; 23 44); hidden = YES; opaque = NO;
gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>"
cy# [button allTargets]
[NSSet setWithArray:@[#"<ComposeButtonItem: 0x14609d00>"]]
cy# [button allControlEvents]
64
cy# [button actionsForTarget:#0x14609d00 forControlEvents:64]
```


It's easy to locate the control object that shows "+86PhoneNumber", and we can say for sure its cell is a PSTableCell object without test. Try to hide this cell to verify our guesses:

```
cy# [#0x17f92890 setHidden:YES]
```

Now, MobilePhoneSettings looks like figure 6-15:



Figure 6-15 Hide the top cell

So the description of the top cell is `<PSTableCell: 0x17f92890; baseClass = UITableViewCell; frame = (0 35; 320 44); text = 'My Number'; autoresize = W; tag = 2; layer = <CALayer: 0x17f92a60>>`. Unlike "compose" button, our current target is not the response method of this cell (i.e. function), but the content (i.e. data) it shows, hence `actionsForTarget:forControlEvents:` is no longer our choice. Facing this kind of situation, what shall we do?

In most cases, data we are interested in would not be a constant. If this data is constantly 1, I believe you won't be interested at all. So, when our target is a variable, one question needs to be thought about: where does the variable come from?

Any variable does not come from nowhere. It originates from a data source and is generated by a specific algorithm. Usually, our focus is on that algorithm, namely, how the data source becomes the variable. This process is usually comprised of multiple functions, which form a call chain like the pseudo code below:

```
id dataSource = ?; // head
```

```

id a = function(dataSource);
id b = function(a);
id c = function(b);
...
id z = function(y);
NSString *myPhoneNumber = function(z); // tail

```

The variable is already known, and we're at the tail of the call chain. Reverse engineering, as its name suggests, enables us to track from the tail back to the head. In this process we will find out every function in this chain, so that we can regenerate the whole algorithm. In a nutshell, to regenerate the algorithm is to record every data source (data source's data source, and so on. Hereafter referred to as the Nth data source) and the trace of function calls along the trip. When the Nth data source of the variable is a determined data (say in this chapter, the Nth data source is the SIM card), the functions between Nth data source and variable is the algorithm. Confused? It'll become clearer after this example.

3. Find the UI function of the top cell

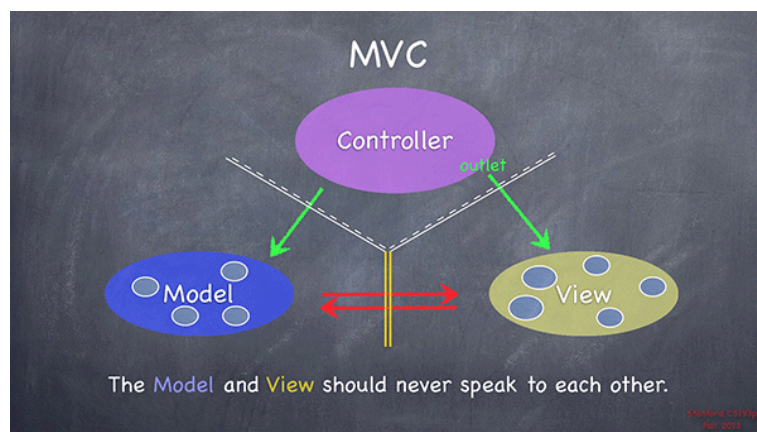


Figure 6-16 MVC design pattern, taken from Stanford CS 193P

According to MVC design pattern (as shown in figure 6-16), M stands for model, namely, the data source, which is unknown; V stands for view, namely, the top cell, which is known; C stands for controller, which is unknown. M and V has no direct relations, while C can directly access both M and V, hence is the communication center of MVC. If we can make use of the known V to acquire C, can't we access M via C to get the data source? This method is logically accessible, is it practicable?

Based on my professional experiences so far, getting C from V is 100% doable; the key is the public method `[UIResponder nextResponder]`, which has the same position to `recursiveDescription` in my heart. Its description is:

“The UIResponder class does not store or set the next responder automatically, instead returning nil by default. Subclasses must override this method to set the next responder. UIView implements this method by returning the UIViewController object that manages it (if it has one) or its superview (if it doesn't); UIViewController implements the method by returning its view's superview; UIWindow returns the application object, and UIApplication returns nil.”

It means that for a V, the return value of nextResponder is either the corresponding C or its superview. Because none of M, V or C can be absent in an App, C exists for sure, namely, there must be a [V nextResponder] that returns a C. Besides, we can get all Vs from recursiveDescription, so getting C from known V is approachable, then M is not far from us.

Therefore, our current target is to get C of the top cell, and it's relatively easy; keep calling nextResponder from cell, until a C is returned:

```
cy# [#0x17f92890 nextResponder]
#"<UITableViewWrapperView: 0x17eb4fc0; frame = (0 0; 320 504); gestureRecognizers =
<NSArray: 0x17ee5230>; layer = <CALayer: 0x17ee5170>; contentOffset: {0, 0};
contentSize: {320, 504}>"
cy# [#0x17eb4fc0 nextResponder]
#"<UITableView: 0x16c69e00; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers
= <NSArray: 0x17f4ace0>; layer = <CALayer: 0x17f4ac20>; contentOffset: {0, -64};
contentSize: {320, 717.5}>"
cy# [#0x16c69e00 nextResponder]
#"<UIView: 0x17ebf2b0; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer:
0x17ebf320>>"
cy# [#0x17ebf2b0 nextResponder]
#"<PhoneSettingsController 0x17f411e0: navItem <UINavigationController: 0x17dae890>, view
<UITableView: 0x16c69e00; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers =
<NSArray: 0x17f4ace0>; layer = <CALayer: 0x17f4ac20>; contentOffset: {0, -64};
contentSize: {320, 717.5}>>"
```

As soon as we get C, we can search in C's header for clues of M. In this example, first we need to locate the binary that contains PhoneSettingsController, we aren't sure whether it comes from Preferences.app or a certain PreferenceBundle. In this case, a simple test would be all good:

```
FunMaker-5:~ root# grep -r PhoneSettingsController /Applications/Preferences.app/
FunMaker-5:~ root# grep -r PhoneSettingsController /System/Library/
Binary file /System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7s matches
grep: /System/Library/Caches/com.apple.dyld/enable-dylibs-to-override-cache: No such
file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGCorePDF.dylib: No
such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltin.dylib:
No such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMaps.dylib: No
such file or directory
grep: /System/Library/Frameworks/System.framework/System: No such file or directory
```

```
Binary file /System/Library/PreferenceBundles/MobilePhoneSettings.bundle/Info.plist
matches
```

It seems that this class comes from MobilePhoneSettings.bundle. Next, class-dump its binary and open PhoneSettingsController.h:

```
@interface PhoneSettingsController : PhoneSettingsListController
<TPSetPINViewControllerDelegate>
.....
- (id)myNumber:(id)arg1;
- (void)setMyNumber:(id)arg1 specifier:(id)arg2;
.....
- (id)tableView:(id)arg1 cellForRowAtIndexPath:(id)arg2;
@end
```

From the above snippet, we know the first 2 methods have obvious relationships with my number. While in a more general manner, the 3rd method is used for initializing all cells, it can be regarded as the UI function of cells. Therefore, data source of the top cell certainly lies in these 3 methods, and we'll take the 3rd method as an example. Let's set a breakpoint at the end of [PhoneSettingsController tableView:cellForRowAtIndexPath:] with LLDB, and see if the return value contains my number. Attach debugserver to Preferences, then connect LLDB to debugserver, and check the ASLR offset of MobilePhoneSettings:

```
(lldb) image list -o -f
[ 0] 0x00078000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.app/Preferences(0x000000000007c000)
[ 1] 0x00231000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000231000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard
[ 3] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
.....
[322] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PreferenceBundles/MobilePhoneSettings.bundle/MobilePhone
Settings
.....
```

As we can see, the ASLR offset of MobilePhoneSettings is 0x06db3000. Then check the address of the last instruction in [PhoneSettingsController tableView:cellForRowAtIndexPath:], as shown in figure 6-17:

```

text:25BB2C2A loc_25BB2C2A ; CODE XREF: -[PhoneSettingsController
text:25BB2C2A MOV R0, R4
text:25BB2C2C ADD SP, SP, #8
text:25BB2C2E POP {R4-R7,PC}
text:25BB2C2E ; End of function -[PhoneSettingsController tableView:cellForRowAtIndexPath:]

```

Figure 6-17 [PhoneSettingsController tableView:cellForRowAtIndexPath:]

Because the return value is stored in R0, let's set the breakpoint at "ADD SP, SP, #8", then re-enter MobilePhoneSettings to trigger the breakpoint. Print R0 out when the process stops, an initialized cell should be ready by then:

```

(lldb) br s -a 0x2c965c2c
Breakpoint 2: where = MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236, address = 0x2c965c2c
Process 115525 stopped
* thread #1: tid = 0x1c345, 0x2c965c2c MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236, queue = 'com.apple.main-thread, stop reason =
breakpoint 2.1
    frame #0: 0x2c965c2c MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236
MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 236:
-> 0x2c965c2c: add    sp, #8
    0x2c965c2e: pop    {r4, r5, r6, r7, pc}

MobilePhoneSettings`-[PhoneSettingsController applicationWillSuspend]:
    0x2c965c30: push  {r7, lr}
    0x2c965c32: mov   r7, sp
(lldb) po $r0
<PSTableCell: 0x15f41440; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My
Number'; tag = 2; layer = <CALayer: 0x15f4c930>>
(lldb) po [$r0 subviews]
<__NSArrayM 0x17060e50>(
<UITableViewCellContentView: 0x15ed0660; frame = (0 0; 320 44); gestureRecognizers =
<NSArray: 0x15f491e0>; layer = <CALayer: 0x15ed06d0>>,
<UIButton: 0x15f26f50; frame = (302 16; 8 13); opaque = NO; userInteractionEnabled = NO;
layer = <CALayer: 0x15f27050>>
)

(lldb) po [$r0 detailTextLabel]
<UILabel: 0x15eb3480; frame = (0 0; 0 0); text = '+86PhoneNumber';
userInteractionEnabled = NO; layer = <UILabelLayer: 0x15eb3540>>

```

As the output suggests, UI function of the top cell is indeed [PhoneSettingsController tableView:cellForRowAtIndexPath:], we have done a great job so far. We are confident that by digging into PhoneSettingsController we'll finally get M, and there must be clues about M in tableView:cellForRowAtIndexPath:. We'll witness this in the next section.

One thing to note, iOS games' UI are generally not constructed with UIKit, so recursiveDescription and nextResponder don't work on games. As rookie reverse engineers, I don't suggest you take games as targets. After understanding this book, if you want to reverse games, welcome to <http://bbs.iosre.com> for discussion.

6.2.2 Locate the target function from the UI function

Successfully getting the UI function is a perfect beginning. UI functions have close ties with UI, namely, if we call [ComposeButtonItem _sendAction:withEvent:] to compose an email, or call [PhoneSettingsController tableView:cellForRowAtIndexPath:] to get my number, a lot of correlated events will happen on UI, such as the view will be refreshed, the layout will be updated, etc. It is over reacting. In most cases, we just want to stay low and perform the functions without interrupting the UI. So what should we do when facing this kind of challenge?

As developers, I assume you have the most basic programmatic knowledge: the lowest level functions are written directly in assembly, which are far from us for now; the remaining functions are all nested called. Since UI functions are rather high level functions, they certainly nested call our target functions, which can be shown as the following pseudo code:

```
drink GetRegular(water arg)
{
    Functions();
    return MakeRegular(arg);
}

drink GetDiet(void)
{
    Functions();
    return MakeDiet();
}

drink GetZero(void)
{
    Functions();
    return MakeZero();
}

drink GetCoke(sugar arg1, water arg2, color arg3)
{
    if (arg1 > 0 && arg1 < 3) return GetDiet();
    else if (arg1 == 0) return GetZero();
    return GetRegular(arg2);
}

drink Get7Up(void)
{
    Functions();
    return Make7Up();
}

drink GetMirinda(void)
{
    Functions();
    return MakeMirinda();
}

drink GetPepsi(sugar arg1, water arg2, color arg3)
```



```

{
    if (arg3 == clear) Get7Up();
    else if (arg3 == orange) GetMirinda();
    return GetRegular(arg2);
}

array GetDrinks(sugar arg1, color arg2) // UIFunction
{
    drink coke = GetCoke(arg1, 100, arg3);
    drink pepsi = GetPepsi(arg1, 105, arg3);
    return arrayWithComponents(coke, pepsi)
}

```

We don't want to be served with coke and pepsi at the same time (you can regard them as UI functions). If we only want to drink 7Up (data), we need to find Get7Up (target function which generates the data); if we want to know how Zero is made (function), we need to find MakeZero (target function which provides function). Actually, the "nest" of nested called functions are also consists of chains, so if we can get to know any link of the chain, we can regenerate the whole chain by reverse engineering, and the tools we mainly use are IDA and LLDB. Let's continue with the previous 2 examples to learn how to find target functions of "compose email" and "get my number" by referring to [ComposeButtonItem _sendAction:withEvent:] and [PhoneSettingsController tableView:cellForRowAtIndexPath:].

1. Look for the target function of "compose email"

Drag and drop MobileMail in IDA, and search [ComposeButtonItem _sendAction:withEvent:] in functions window, as shown in figure 6-18.

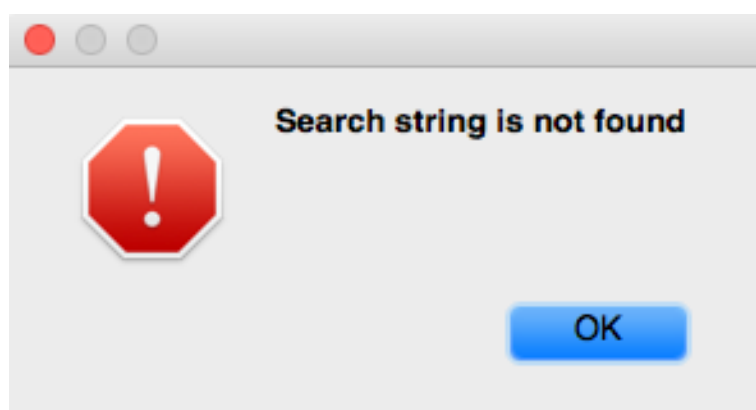


Figure 6-18 [ComposeButtonItem _sendAction:withEvent:] is not found

Where is [ComposeButtonItem _sendAction:withEvent:]? Now that ComposeButtonItem doesn't implement this method, it's supposed to be implemented in its super class. Open ComposeButtonItem.h and see which class it inherits from:

```
@interface ComposeButtonItem : LongPressableButtonItem
```

```
+(id)composeButtonItem;
@end
```

Then open LongPressableButtonItem.h, and see whether it implements
_sendAction:withEvent:.

```
@interface LongPressableButtonItem : UIBarButtonItem
{
    id _longPressTarget;
    SEL _longPressAction;
}

- (void)_attachGestureRecognizerToView:(id)arg1;
- (id)createViewForNavigationItem:(id)arg1;
- (id)createViewForToolbar:(id)arg1;
- (void)longPressGestureRecognized:(id)arg1;
- (void)setLongPressTarget:(id)arg1 action:(SEL)arg2;

@end
```

It doesn't implement this method either, so let's proceed to its super class. Open
UIBarButtonItem.h:

```
@interface UIBarButtonItem : UIBarItem <NSCoding>
.....
- (void)_sendAction:(id)arg1 withEvent:(id)arg2;
.....
@end
```

UIBarButtonItem does implement this method, so it's UIKit that we should analyze. Drag
and drop the binary into IDA, since UIKit is big in size, it takes a rather long time to be analyzed.
During waiting time, how about dropping in <http://bbs.iosre.com> for a chat?

After the initial analysis of UIKit, let's go to the implementation of [UIBarButtonItem
_sendAction:withEvent:], as shown in figure 6-19.

```
; UIBarButtonItem - (void)_sendAction:(id) withEvent:(id)
; Attributes: bp-based frame
; void __cdecl -[UIBarButtonItem _sendAction:withEvent:](struct U
UIBarButtonItem __sendAction_withEvent__
PUSH        {R4,R5,R7,LR}
ADD         R7, SP, #8
PUSH.W     {R10,R11}
SUB         SP, SP, #8
MOV        R10, R0
MOV        R0, #(selRef_action - 0x2501F69E) ; selRef_action
MOV        R11, R3
ADD        R0, PC ; selRef_action
LDR        R4, [R0] ; "action"
MOV        R0, R10
MOV        R1, R4
BLX.W     _objc_msgSend
CBZ       R0, loc_2501F6FC
```

Figure 6-19 [UIBarButtonItem _sendAction:withEvent:]

The first function to be called is objc_msgSend. Its official documentation is:

*“When it encounters a method call, the compiler generates a call to one of the functions
objc_msgSend, objc_msgSend_stret, objc_msgSendSuper, or objc_msgSendSuper_stret.
Messages sent to an object's superclass (using the super keyword) are sent using*

objc_msgSendSuper; other messages are sent using objc_msgSend. Methods that have data structures as return values are sent using objc_msgSendSuper_stret and objc_msgSend_stret.”

According to the relationship of “object”, “method” and “implementation” in chapter 5, [receiver message] becomes `objc_msgSend(receiver, @selector(message))` after compilation; when there are arguments in the method, [receiver message:arg1 foo:arg2 bar:arg3] becomes `objc_msgSend(receiver, @selector(message), arg1, arg2, arg3)`, etc. Based on this, the first `objc_msgSend` actually executes an Objective-C method. So what exactly is the method? Who’s the receiver, and what are the arguments?

Still remember “sentence of the book”?

“The first 4 arguments are saved in R0, R1, R2 and R3; the rest are saved on the stack; the return value is saved in R0.”

According to the sentence, at ARM level, `objc_msgSend` works in the format of `objc_msgSend(R0, R1, R2, R3, *SP, *(SP + sizeofLastArg), ...)`, and the corresponding Objective-C method is [R0 R1:R2 foo:R3 bar:*SP baz:*(SP + sizeofLastArg) qux:...]. :Let’s apply this format to the first `objc_msgSend`; if we’re to reproduce its corresponding Objective-C method, you have to find out what’s in R0, R1, R2, R3 and SP before “BLX.W _objc_msgSend”. This kind of backward analysis is worthy of the name reverse engineering. Let’s try it out.

Before “BLX.W _objc_msgSend”, the latest assignment of R0 comes from “MOV R0, R10”, thus R0 comes from R10; the latest assignment of R10 comes from “MOV R10, R0”, thus R10 comes from R0. Before “MOV R10, R0”, R0 is directly used without assignment; this seems illogical, but such an obvious “bug” is impossible to exist, it’s us that may have made a mistake. So R0 must be assigned somewhere. Here comes the question, where is this “somewhere”?

Given that there is no assignment of R0 inside [UIButtonItem _sendAction:withEvent:], the only possibility is that it’s assigned in the caller of [UIButtonItem _sendAction:withEvent:]. [UIButtonItem _sendAction:withEvent:] becomes `objc_msgSend(UIButtonItem, @selector(_sendAction:withEvent:), action, event)` after compilation, and 4 arguments are stored separately in R0~R3. So when [UIButtonItem _sendAction:withEvent:] gets called, R0 is UIButtonItem, so is R0 in “MOV R10, R0” and “BLX.W _objc_msgSend”. Still confused? Refer to figure 6-20, I bet you can understand.

```

; UIButtonItem - (void)_sendAction:(id) withEvent:(id)
; Attributes: bp-based frame

; void __cdecl -[UIButtonItem_sendAction:withEvent:](struct U
UIButtonItem_sendAction_withEvent__
PUSH      {R4,R5,R7,LR}
ADD       R7, SP, #8
PUSH.W   {R10,R11}
SUB       SP, SP, #8
MOV       R10, R0
MOV       RO, #(selRef_action - 0x2501F69E) ; selRef_action
MOV       R11, R3
ADD       RO, PC ; selRef_action
LDR       R4, [R0] ; "action"
MOV       RO, R10
MOV       R1, R4
BLX.W    _objc_msgSend
CBZ       RO, loc_2501F6FC

```

Figure 6-20 R0's evolution

Similarly, before “BLX.W _objc_msgSend”, the latest assignment of R1 comes from “MOV R1, R4”, thus R1 comes from R4; the latest assignment of R4 comes from “LDR R4, [R0]”, thus R4 comes from *R0, i.e. “action” which is already commented out in IDA. The evolution of R1 is shown in figure 6-21:

```

; UIButtonItem - (void)_sendAction:(id) withEvent:(id)
; Attributes: bp-based frame

; void __cdecl -[UIButtonItem_sendAction:withEvent:](struct U
UIButtonItem_sendAction_withEvent__
PUSH      {R4,R5,R7,LR}
ADD       R7, SP, #8
PUSH.W   {R10,R11}
SUB       SP, SP, #8
MOV       R10, R0
MOV       RO, #(selRef_action - 0x2501F69E) ; selRef_action
MOV       R11, R3
ADD       RO, PC ; selRef_action
LDR       R4, [R0] ; "action"
MOV       RO, R10
MOV       R1, R4
BLX.W    _objc_msgSend
CBZ       RO, loc_2501F6FC

```

Figure 6-21 R1's change process

So after reproduction, the first objc_msgSend becomes [self action], and the return value is stored in R0, right? Next, the process judges whether [self action] is 0. If it is 0, there will be no actions; otherwise, it branches to figure 6-22:

```

MOV     R0, #(selRef_sharedApplication - 0x2501F6BC) ; selRef_sharedApplication
MOV     R2, #(classRef_UIApplication - 0x2501F6BE) ; classRef_UIApplication
ADD     R0, PC ; selRef_sharedApplication
ADD     R2, PC ; classRef_UIApplication
LDR     R1, [R0] ; "sharedApplication"
LDR     R0, [R2] ; _OBJC_CLASS_$_UIApplication
BLX.W  _objc_msgSend
MOV     R5, R0
MOV     R0, R10
MOV     R1, R4
BLX.W  _objc_msgSend
MOV     R4, R0
MOV     R0, #(selRef_target - 0x2501F6DC) ; selRef_target
ADD     R0, PC ; selRef_target
LDR     R1, [R0] ; "target"
MOV     R0, R10
BLX.W  _objc_msgSend
MOV     R3, R0
MOV     R0, #(selRef_sendAction_to_from_forEvent_ - 0x2501F6F2) ; selRef_sendAction_to_from_forEvent_
MOV     R2, R4
ADD     R0, PC ; selRef_sendAction_to_from_forEvent_
LDR     R1, [R0] ; "sendAction:to:from:forEvent:"
MOV     R0, R5
STRD.W R10, R11, [SP]
BLX.W  _objc_msgSend

```

Figure 6-22 [UIBarButtonItem _sendAction:withEvent:]

There're 4 objc_msgSends, let's analyze them with the same thought one by one:

R0 of the 1st objc_msgSend comes from "LDR R0, [R2]", and IDA has already figured out that [R2] is a UIApplication class; R1 comes from "LDR R1, [R0]", i.e. "sharedApplication". So the 1st objc_msgSend is actually [UIApplication sharedApplication], and the return value is stored in R0.

R0 of the 2nd objc_msgSend comes from "MOV R0, R10", i.e. R10; in figure 6-20, we can see that R10 is UIBarButtonItem; R1 comes from "MOV R1, R4", i.e. R4; in figure 6-21, R4 is "action". So, the 2nd objc_msgSend is actually [UIBarButtonItem action], and the return value is stored in R0.

R0 of the 3rd objc_msgSend comes from "MOV R0, R10", i.e. UIBarButtonItem; R1 comes from "LDR R1, [R0]", i.e. "target". Therefore, the 3rd objc_msgSend is actually [UIBarButtonItem target], and the return value is stored in R0.

R0 of the 4th objc_msgSend comes from "MOV R0, R5", i.e. R5; R5 comes from "MOV R5, R0" under the 1st objc_msgSend, i.e. R0. What's R0? Because the 1st objc_msgSend stores its return value in R0, R0 is the return value of [UIApplication sharedApplication] as well the 1st argument of the 4th objc_msgSend. R1 comes from "LDR R1, [R0]", i.e.

"sendAction:to:from:forEvent:", which has 4 arguments. Since objc_msgSend already has 2 arguments, there're 6 arguments in total, R0~R3 are not enough to hold all arguments, the last 2 arguments have to be stored on the stack. R2 comes from "MOV R2, R4", i.e. R4; R4 comes from "MOV R4, R0" under the 2nd objc_msgSend, i.e. R0; R0 comes from the return value of the 2nd objc_msgSend, i.e. [UIBarButtonItem action], which is the 3rd argument. R3 comes

from “MOV R3, R0” under the 3rd objc_msgSend, i.e. R0; R0 comes from the return value of the 3rd objc_msgSend, i.e. [UIBarButtonItem target], which is the 4th argument. The rest 2 arguments come from the stack, and before the 4th objc_msgSend, the latest change of stack comes from “STRD.W R10, R11, [SP]”, i.e. R10 and R11 are saved onto the stack; therefore, the rest 2 arguments are R10 and R11. R10 is UIBarButtonItem, which is discussed several times; whereas R11 comes from “MOV R11, R3” in figure 6-21, i.e. R3, which is another unassigned register, so it must come from the caller of [UIBarButtonItem _sendAction:withEvent:]. Based on our previous analysis, R11 is the 2nd argument of _sendAction:withEvent:, i.e. event. The relationship of these 4 arguments is a little complicated, hope figure 6-23 and 6-24 can give you a better illustration.

```

; UIBarButtonItem - (void)_sendAction:(id) withEvent:(id)
; Attributes: bp-based frame

; void __cdecl -[UIBarButtonItem _sendAction:withEvent:](struct U
UIBarButtonItem _sendAction_withEvent__
PUSH      {R4,R5,R7,LR}
ADD       R7, SP, #8
PUSH.W   {R10,R11}
SUB       SP, SP, #8
MOV       R10, R0
MOV       R0, #(selRef_action - 0x2501F69E) ; selRef_action
MOV       R11, R3
ADD       R0, PC ; selRef_action
LDR       R4, [R0] ; "action"
MOV       R0, R10
MOV       R1, R4
BLX.W    _objc_msgSend
CBZ       R0, loc_2501F6FC

```

Figure 6-23 The relationship of objc_msgSend’s arguments

```

MOV       R0, #(selRef_sharedApplication - 0x2501F6BC) ; selRef_s
MOV       R2, #(classRef_UIApplication - 0x2501F6BE) ; classRef_U
ADD       R0, PC ; selRef_sharedApplication
ADD       R2, PC ; classRef_UIApplication
LDR       R1, [R0] ; "sharedApplication"
LDR       R0, [R2] ; _OBJC_CLASS_$_UIApplication
BLX.W    _objc_msgSend
MOV       R5, R0
MOV       R0, R10
MOV       R1, R4
BLX.W    _objc_msgSend
MOV       R4, R0
MOV       R0, #(selRef_target - 0x2501F6DC) ; selRef_target
ADD       R0, PC ; selRef_target
LDR       R1, [R0] ; "target"
MOV       R0, R10
BLX.W    _objc_msgSend
MOV       R3, R0
MOV       R0, #(selRef_sendAction_to_from_forEvent_ - 0x2501F6F2)
MOV       R2, R4
ADD       R0, PC ; selRef_sendAction_to_from_forEvent_
LDR       R1, [R0] ; "sendAction:to:from:forEvent:"
MOV       R0, R5
STRD.W   R10, R11, [SP]
BLX.W    _objc_msgSend

```

Figure 6-24 The relationship of objc_msgSend’s arguments

So, seems the core of [UIBarButtonItem _sendAction:withEvent:] is [[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:event]. Since we have already known that [UIBarButtonItem _sendAction:withEvent:] will perform “compose mail” operation, [[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:event] is sure to get called. Although with IDA, we’ve sorted out where every argument comes from, IDA can’t tell us what their values are during execution. So, it’s time to bring LLDB on stage to do some dynamic debugging.

Attach debugserver to MobileMail, and connect with LLDB, then print out the ASLR offset of UIKit:

```
(lldb) image list -o -f
```

```

[ 0] 0x0008e000
/private/var/db/stash/_.29LMeZ/Applications/MobileMail.app/MobileMail(0x0000000000092000
)
[ 1] 0x00393000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000393000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/usr/lib/libarchive.2.dylib
.....
[ 45] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/UIKit.framework/UIKit
.....

```

ASLR offset of UIKit is 0x6db3000. Let's check out the address of the 4th objc_msgSend, as shown in figure 6-25.

```

text:2501F6F0      LDR      R1, [R0] ; "sendAction:to:from:forEvent:"
text:2501F6F2      MOV      R0, R5
text:2501F6F4      STRD.W  R10, R11, [SP]
text:2501F6F8      BLX.W   _objc_msgSend
text:2501F6FC      ; CODE XREF: -[UIBarButtonItem _s
loc_2501F6FC      ADD     SP, SP, #8
text:2501F6FC      POP.W   {R10, R11}
text:2501F6FE      POP     {R4, R5, R7, PC}
text:2501F702      ; End of function -[UIBarButtonItem _sendAction:withEvent:]
text:2501F702

```

Figure 6-25 Check out address of objc_msgSend

Set a breakpoint at $0x6db3000 + 0x2501F6F8 = 0x2BDD26F8$, then tap “compose” button to trigger it and inspect the arguments of `[[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:eventFromArg2]:`

```

(lldb) br s -a 0x2BDD26F8
Breakpoint 4: where = UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] + 116,
address = 0x2bdd26f8
Process 44785 stopped
* thread #1: tid = 0xae1, 0x2bdd26f8 UIKit`-[UIBarButtonItem(UIInternal)
_sendAction:withEvent:] + 116, queue = 'com.apple.main-thread, stop reason = breakpoint
4.1
   frame #0: 0x2bdd26f8 UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] +
116
UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] + 116:
-> 0x2bdd26f8: blx    0x2c3539f8      ; symbol stub for: roundf$shim
   0x2bdd26fc: add     sp, #8
   0x2bdd26fe: pop.w  {r10, r11}
   0x2bdd2702: pop    {r4, r5, r7, pc}
(lldb) p (char *)$r1
(char *) $48 = 0x2c3de501 "sendAction:to:from:forEvent:"
(lldb) po $r0
<MailAppController: 0x176a8820>
(lldb) po $r2
[no Objective-C description available]
(lldb) p (char *)$r2
(char *) $51 = 0x2d763308 "composeButtonClicked:"
(lldb) po $r3
<nil>
(lldb) x/10 $sp
0x00391198: 0x1776d640 0x176a8ce0 0x1760f5e0 0x00000000
0x003911a8: 0x2c4140f2 0x1776ff50 0x003911cc 0x2bc6ec2b
0x003911b8: 0x176a8ce0 0x00000001
(lldb) po 0x1776d640
<ComposeButtonItem: 0x1776d640>
(lldb) po 0x176a8ce0

```



```
<UITouchesEvent: 0x176a8ce0> timestamp: 58147.4 touches: {(
  <UITouch: 0x1895e2b0> phase: Ended tap count: 1 window: <UIWindow: 0x17759c30; frame
= (0 0; 320 568); gestureRecognizers = <NSArray: 0x1775c7a0>; layer = <UIWindowLayer:
0x1752e190>> view: <UIToolbarButton: 0x1776ff50; frame = (285 0; 23 44); opaque = NO;
gestureRecognizers = <NSArray: 0x17758670>; layer = <CALayer: 0x17770160>> location in
window: {308, 534} previous location in window: {304.5, 534} location in view: {23, 10}
previous location in view: {19.5, 10}
)}
```

The first 4 arguments of `objc_msgSend`, i.e. `R0~R3` are intuitive. They're `self`, `@selector(sendAction:to:from:forEvent:)`, the argument of `sendAction:`, and the argument of `to:`. One thing to mention is that when I entered "po \$r2", LLDB said "no Objective-C description available", indicating `R2` wasn't an Objective-C object. Thus, combining with the meaning of "action", I guessed it was a SEL, so I used "p (char *)\$r2" to print it. How to analyze those arguments in the stack? Because `SP` points to the bottom of stack while the rest 2 arguments are on the stack, and they are both one word long, I've printed out the continuous 10 words from the bottom of the stack using "x/10 \$sp", and the first 2 were the arguments on stack. Most Objective-C arguments are one word long pointers, which point at Objective-C objects, so I've "po"ed the first 2 words, they were the arguments. For ease of understanding, the relationship of `SP`, values on the stack and arguments are shown in figure 6-26.

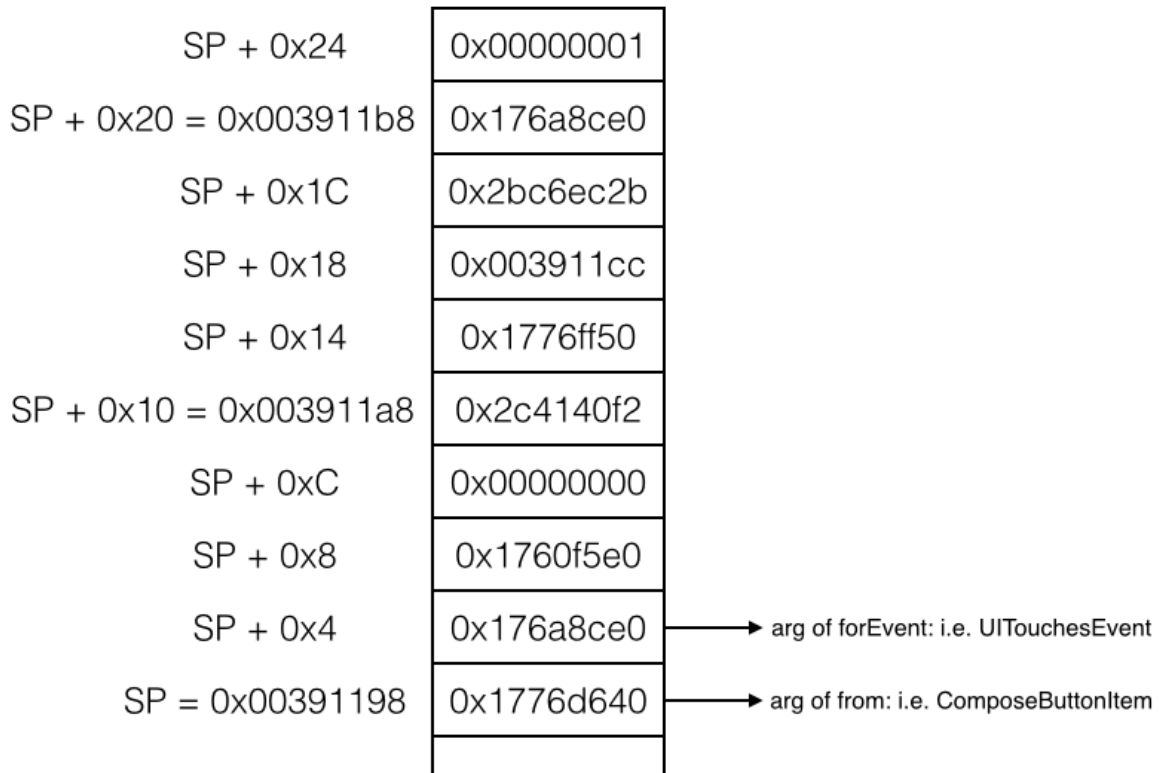


Figure 6-26 The relationship of SP, value in the stack and arguments

In most cases, the number of arguments on stack will not exceed 10, so “x/10 \$sp” is enough. Print them in order, we can get all arguments on stack.

With the combination of IDA and LLDB, we have figured out that the core in [UIBarButtonItem _sendAction:withEvent:] is [MailAppController sendAction:@selector(composeButtonClicked:) to:nil from:ComposeButtonItem forEvent:UITouchesEvent], which is one step closer to our target function of “composing email”. Next let’s figure out what does [UIApplication sendAction:to:from:forEvent:] do with IDA, as shown in figure 6-27:

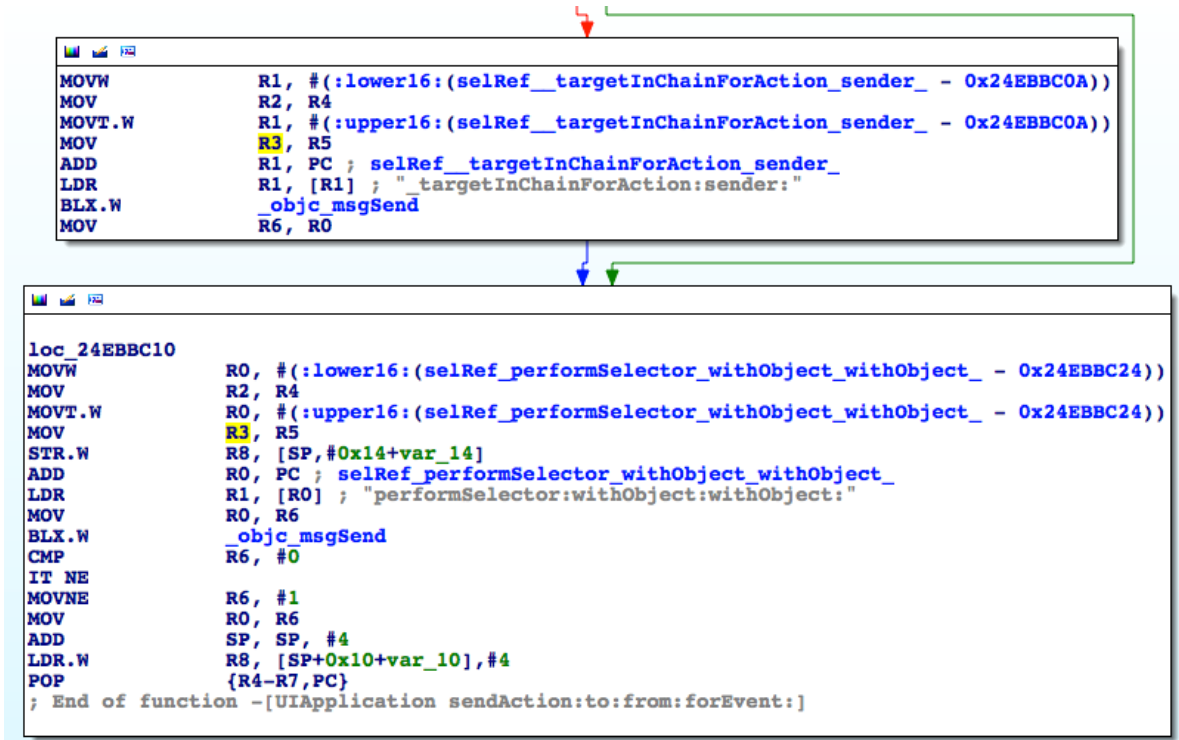


Figure 6- 27 [UIApplication sendAction:to:from:forEvent:]

Whatever, “performSelector:withObject:withObject:” in loc_24ebbc10 will get executed, so naturally we can guess it is where actual operations are performed. Just like before, let’s figure out what does this method do with LLDB. The ASLR offset of UIKit is 0x6db3000, and the address of the last objc_msgSend is 0x24EBBC26, so we set a breakpoint at $0x6db3000 + 0x24EBBC26 = 0x2BC6EC26$, then tap “compose” button to trigger the breakpoint to inspect the arguments:

```
(lldb) br s -a 0x2BC6EC26
Breakpoint 1: where = UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66, address = 0x2bc6ec26
Process 226191 stopped
* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66
UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:
-> 0x2bc6ec26: blx    0x2c3539f8          ; symbol stub for: roundf$shim
    0x2bc6ec2a: cmp    r6, #0
    0x2bc6ec2c: it     ne
    0x2bc6ec2e: movne r6, #1
(lldb) p (char *)$r1
(char *) $0 = 0x2c3dac95 "performSelector:withObject:withObject:"
(lldb) po $r0
<ComposeButtonItem: 0x14ddf5f0>
(lldb) p (char *)$r2
(char *) $2 = 0x2c4140f2 "_sendAction:withEvent:"
(lldb) po $r3
<UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>>
(lldb) x/10 $sp
```

```

0x003735a8: 0x160a6120 0x00000001 0x14d73c90 0x160a6120
0x003735b8: 0x2c3d9be5 0x003735d4 0x2bc6ebd1 0x14d73c90
0x003735c8: 0x160a6120 0x00000040
(lldb) po 0x160a6120
<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches: {(
    <UITouch: 0x14ff2f20> phase: Ended tap count: 1 window: <UIWindow: 0x14d878b0; frame
= (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x14dba890>; layer =
<UIWindowLayer: 0x14d87a30>> view: <UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44);
opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>>
location in window: {308, 545} previous location in window: {308, 545} location in view:
{23, 21} previous location in view: {23, 21}
)}

```

What the hell? `performSelector:withObject:withObject:` called `[ComposeButtonItem _sendAction:withEvent:]`, and `[ComposeButtonItem _sendAction:withEvent:]` called `performSelector:withObject:withObject:` in turn. If `performSelector:withObject:withObject:` calls `[ComposeButtonItem _sendAction:withEvent:]` again then we'll fall into an infinite call loop and the UI will be locked endlessly, which doesn't make sense and conflicts with what we've seen. Let's continue the process to trigger the breakpoint again and see what happens:

```

(lldb) c
Process 226191 resuming
Process 226191 stopped
* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-[UIApplication
sendAction:to:from:forEvent:] + 66, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66
UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:
-> 0x2bc6ec26: blx    0x2c3539f8          ; symbol stub for: roundf$shim
    0x2bc6ec2a: cmp    r6, #0
    0x2bc6ec2c: it     ne
    0x2bc6ec2e: movne r6, #1
(lldb) p (char *)$r1
(char *) $6 = 0x2c3dac95 "performSelector:withObject:withObject:"
(lldb) po $r0
<MailAppController: 0x14e7a7a0>
(lldb) p (char *)$r2
(char *) $7 = 0x2d763308 "composeButtonClicked:"
(lldb) po $r3
<ComposeButtonItem: 0x14ddf5f0>
(lldb) x/10 $sp
0x0037356c: 0x160a6120 0x160a6120 0x2d763308 0x14e7a7a0
0x0037357c: 0x14ddf5f0 0x003735a0 0x2bdd26fd 0x14ddf5f0
0x0037358c: 0x160a6120 0x160fbbdf0
(lldb) po 0x160a6120
<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches: {(
    <UITouch: 0x14ff2f20> phase: Ended tap count: 1 window: <UIWindow: 0x14d878b0; frame
= (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x14dba890>; layer =
<UIWindowLayer: 0x14d87a30>> view: <UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44);
opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>>
location in window: {308, 545} previous location in window: {308, 545} location in view:
{23, 21} previous location in view: {23, 21}
)}

```

As we can see, arguments of `performSelector:withObject:withObject:` have changed, and `[MailAppController composeButtonClicked:ComposeButtonItem]` was called. If we “c” again, the breakpoint will not be triggered, so we can confirm it’s `composeButtonClicked:` that performs the actual operation. Because inside `MobileMail`, we can get an `MailAppController` object from `[UIApplication sharedApplication]`, and at the beginning of this section, we’ve seen a class method `+composeButtonItem` in `ComposeButtonItem.h`, which returns a `ComposeButtonItem` object, now we’re able to get all necessary objects to call `[MailAppController composeButtonClicked:ComposeButtonItem]`; what’s more, we can call it anywhere inside `MobileMail`. Therefore, `composeButtonClicked:` can be regarded as the target function of “compose email”.

Finally, let’s test this method in `Cycript` to see if it works:

```
FunMaker-5:~ root# cycript -p MobileMail
cy# [UIApp composeButtonClicked:[ComposeButtonItem composeButtonItem]]
```

After the above commands, the “New Message” view shows in `Mail`. In this example, we’ve tracked the call chain with `IDA` until the target function was located, and then we’ve analyzed its arguments with `LLDB`. I call it a complex process rather than a difficult one, do you agree? In the next section, we will find out the target function of “my number” with the similar pattern, please try to sum up the experiences.

2. Look for the target function of “my number”

Let’s continue our analysis from the UI function `[PhoneSettingsController tableView:cellForRowAtIndexPath:]`. Because the return value of UI function is stored in `R0`, and according to “`MOV R0, R4`” in figure 6-17, we know `R0` comes from `R4`. As shown in figure 6-28, `R4` is only assigned once at “`MOV R4, R0`” and `R0` comes from the return value of `objc_msgSendSuper2`. `objc_msgSendSuper2` is undocumented, as we can see in figure 6-29, it comes from “`/usr/lib/libobjc.A.dylib`”.

```

; id __cdecl -[PhoneSettingsController tableView:cellForRowAtIndexPath:]
__PhoneSettingsController tableView_cellForRowAtIndexPath__

var_14= -0x14
var_10= -0x10

PUSH        {R4-R7,LR}
ADD         R7, SP, #0xC
SUB         SP, SP, #8
MOV         R6, R0
MOV         R0, #(classRef_PhoneSettingsController - 0x25BB2B58) ; c
STR         R6, [SP,#0x14+var_14]
MOV         R5, R3
ADD         R0, PC ; classRef_PhoneSettingsController
LDR         R0, [R0] ; _OBJC_CLASS_$_PhoneSettingsController
MOV         R1, #(selRef_tableView_cellForRowAtIndexPath_ - 0x25BB2B
ADD         R1, PC ; selRef_tableView_cellForRowAtIndexPath_
LDR         R1, [R1] ; "tableView:cellForRowAtIndexPath:"
STR         R0, [SP,#0x14+var_10]
MOV         R0, SP
BLX        __objc_msgSendSuper2
MOV         R4, R0

```

Figure 6-28 Source of R4

```

Imports from /usr/lib/libobjc.A.dylib

IMPORT __OBJC_CLASS_$_NSObject
; DATA XREF:
; -[PhoneSet

IMPORT __OBJC_METACLASS_$_NSObject
; DATA XREF:
; __objc_dat

IMPORT __objc_personality_v0
IMPORT __objc_empty_cache
IMPORT __imp__objc_getProperty
; CODE XREF:
; DATA XREF:

IMPORT __imp__objc_msgSend ; CODE XREF:
; DATA XREF:

IMPORT __imp__objc_msgSendSuper2

```

Figure 6-29 Source of objc_msgSendSuper2

According to the literal meaning, objc_msgSendSuper2 and objc_msgSendSuper are supposed to work similarly, namely send messages to callers' superclasses. No more guesses, let's set a breakpoint on objc_msgSendSuper2 and check out its arguments as well return value. Attach debugserver to Preference, and connect with LLDB, then print out ASLR offset of MobilePhoneSettings:

```

(lldb) image list -o -f
[ 0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.app/Preferences(0x0000000000007d00)
[ 1] 0x00232000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000232000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard
[ 3] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
.....
[330] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PreferenceBundles/MobilePhoneSettings.bundle/MobilePhone
Settings
.....

```

ASLR offset of MobilePhoneSettings is 0x6db3000. Then take a look at obj_msgSendSuper2's address, as shown in figure 6-30.

```

25BB2B40 ; id __cdecl -[PhoneSettingsController tableView:cellForRowAtIndexPath:](stru
25BB2B40 __PhoneSettingsController_tableView_cellForRowAtIndexPath_
25BB2B40 ; DATA XREF: __objc_const:3044E378␣
25BB2B40
25BB2B40 var_14 = -0x14
25BB2B40 var_10 = -0x10
25BB2B40
25BB2B40 PUSH {R4-R7,LR}
25BB2B42 ADD R7, SP, #0xC
25BB2B44 SUB SP, SP, #8
25BB2B46 MOV R6, R0
25BB2B48 MOV R0, #(classRef_PhoneSettingsController - 0x25
25BB2B50 STR R6, [SP]
25BB2B52 MOV R5, R3
25BB2B54 ADD R0, PC ; classRef_PhoneSettingsController
25BB2B56 LDR R0, [R0] ; _OBJC_CLASS_$_PhoneSettingsControl
25BB2B58 MOV R1, #(selRef_tableView_cellForRowAtIndexPath_
25BB2B60 ADD R1, PC ; selRef_tableView_cellForRowAtIndexPa
25BB2B62 LDR R1, [R1] ; "tableView:cellForRowAtIndex:"
25BB2B64 STR R0, [SP,#4]
25BB2B66 MOV R0, SP
25BB2B68 BLX _objc_msgSendSuper2

```

Figure 6-30 Check out address of objc_msgSendSuper2

The breakpoint should be set at $0x6db3000 + 0x25BB2B68 = 0x2C965B68$. Re-enter MobilePhoneSettings to trigger the breakpoint:

```

(lldb) br s -a 0x2C965B68
Breakpoint 1: where = MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40, address = 0x2c965b68
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x2c965b68 MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x2c965b68 MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40
MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 40:
-> 0x2c965b68: blx    0x2c975fb8          ; symbol stub for:
CTSettingRequest$shim
    0x2c965b6c: mov     r4, r0
    0x2c965b6e: movw   r0, #54708
    0x2c965b72: movt   r0, #2697
(lldb) p (char *)$r1
(char *) $0 = 0x2c3daf33 "tableView:cellForRowAtIndex:"
(lldb) po $r0
[no Objective-C description available]
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x2c965b6c MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 44, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x2c965b6c MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 44
MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 44:
-> 0x2c965b6c: mov     r4, r0
    0x2c965b6e: movw   r0, #54708
    0x2c965b72: movt   r0, #2697
    0x2c965b76: mov    r2, r5
(lldb) po $r0
<PSTableCell: 0x15fc6b00; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My
Number'; tag = 2; layer = <CALayer: 0x15fbbe40>>

```



```
(lldb) po [$r0 detailTextLabel]
<UITableViewLabel: 0x15fb5590; frame = (0 0; 0 0); text = '+86PhoneNumber';
userInteractionEnabled = NO; layer = <_UILabelLayer: 0x15fd87e0>>
```

It's worth mentioning that the 1st argument of `objc_msgSendSuper2` is not an Objective-C object. I'm not sure whether it is a bug of LLDB or it is the actual case. Anyway, it doesn't influence our analysis, just ignore it for now. If you're really interested in this detail, you are welcome to share your research on <http://bbs.iosre.com>.

Back on track, the output of LLDB indicates that the return value of `objc_msgSendSuper2` is an initialized cell, which contains my number already. Similar to what happened in the last section, let's check out the implementation of `tableView:cellForRowAtIndexPath:` in `PhoneSettingsController`'s superclass. First of all let's figure out who's the superclass in `PhoneSettingsController.h`:

```
@interface PhoneSettingsController : PhoneSettingsListController
<TPSetPINViewControllerDelegate>
.....
@end
```

`PhoneSettingsController` inherits from `PhoneSettingsListController`, so open `PhoneSettingsListController.h` to check out if it implements `tableView:cellForRowAtIndexPath:`.

```
@interface PhoneSettingsListController : PSListController
{
}

- (id)bundle;
- (void)dealloc;
- (id)init;
- (void)pushController:(Class)arg1 specifier:(id)arg2;
- (id)setCellEnabled:(BOOL)arg1 atIndex:(unsigned int)arg2;
- (id)setCellLoading:(BOOL)arg1 atIndex:(unsigned int)arg2;
- (id)setControlEnabled:(BOOL)arg1 atIndex:(unsigned int)arg2;
- (id)sheetSpecifierWithTitle:(id)arg1 controller:(Class)arg2 detail:(Class)arg3;
- (void)simRemoved:(id)arg1;
- (id)specifiers;
- (void)updateCellStates;
- (void)viewWillAppear:(BOOL)arg1;

@end
```

`PhoneSettingsListController` doesn't implement `tableView:cellForRowAtIndexPath:`, so just proceed to its superclass `PSListController`. The class `PSListController` is no longer inside `MobilePhoneSettings.bundle`, so let's search it in all class-dump headers, as shown in figure 6-31.

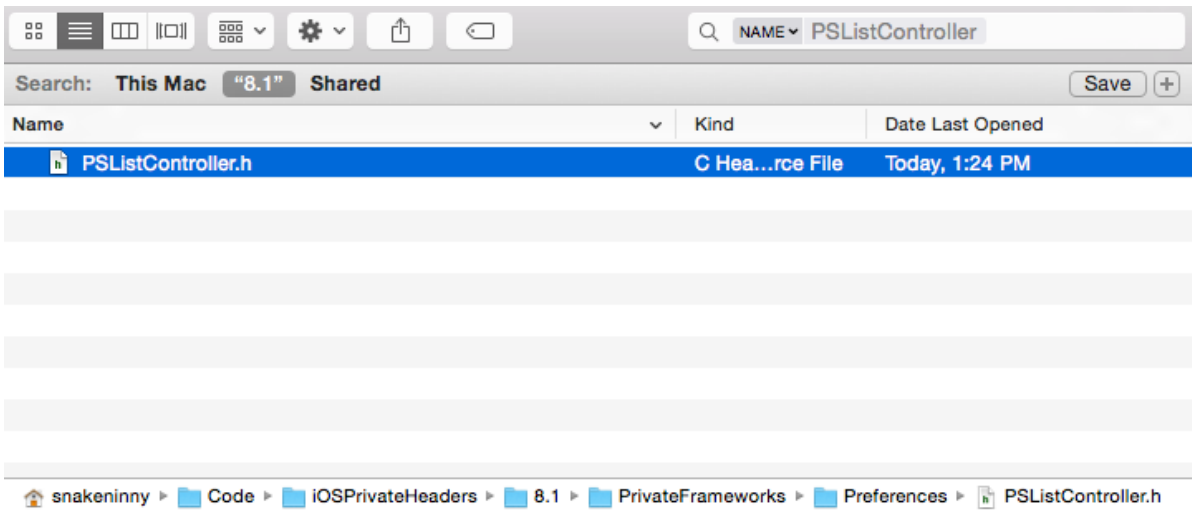


Figure 6-31 Locate PSListController.h

Note, PSListController.h comes from Preferences.framework, which shares the name with Preferences.app, make sure to distinguish them. Open it, and check if there is tableView:cellForRowAtIndexPath:.

```
@interface PSListController : PSViewController <UITableViewDelegate,
UITableViewDataSource, UIActionSheetDelegate, UIAlertViewDelegate,
UIPopoverControllerDelegate, PSSpecifierObserver, PSViewControllerOffsetProtocol>
.....
- (id)tableView:(id)arg1 cellForRowAtIndexPath:(id)arg2;
.....
@end
```

As we see, it has implemented this method, so drag and drop the binary of Preferences.framework into IDA and jump to tableView:cellForRowAtIndexPath:, as shown in figure 6-32.

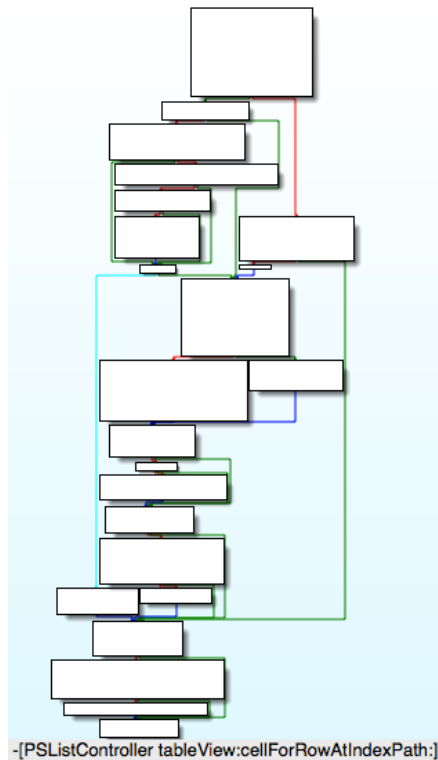


Figure 6-32 [PListController tableView:cellForRowAtIndexPath:]

Its execution logic is complicated. To play it safe, let's set a breakpoint at the end of this method to check if "my number" is contained in the return value, so that we can make sure objc_msgSendSuper2 calls [PListController tableView:cellForRowAtIndexPath:]. First, let's check out ASLR offset of Preferences.framework:

```
(lldb) image list -o -f
[ 0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.app/Preferences(0x000000000007d000)
[ 1] 0x00232000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000232000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard
[ 3] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
.....
[ 42] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Preferences.framework/Preferences
.....
```

Its ASLR offset is 0x6db3000. Then find the address of the last instruction of [PListController tableView:cellForRowAtIndexPath:], as shown in figure 6-33.

```

text:2A9F79E6 loc_2A9F79E6                                ; CODE XREF: -[PListController
text:2A9F79E6                                           ; -[PListController tableView:c
text:2A9F79E6 MOV R0, R6
text:2A9F79E8 ADD SP, SP, #0x1C
text:2A9F79EA POP.W {R8,R10,R11}
text:2A9F79EE POP {R4-R7,PC}
text:2A9F79EE ; End of function -[PListController tableView:cellForRowAtIndexPath:]

```

Figure 6-33 [PListController tableView:cellForRowAtIndexPath:]

Because the return value is stored in R0 and R0 comes from “MOV R0, R6”, we can simply set a breakpoint on this instruction and print out R6. The address of this instruction is 0x2A9F79E6, so set the breakpoint at $0x6db3000 + 0x2A9F79E6 = 0x317AA9E6$. Re-enter MobilePhoneSettings to trigger the breakpoint:

```

(lldb) br s -a 0x317AA9E6
Breakpoint 5: where = Preferences`-[PListController tableView:cellForRowAtIndexPath:] +
1026, address = 0x317aa9e6
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9e6 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 1026, queue = 'com.apple.main-thread, stop reason =
breakpoint 5.1
   frame #0: 0x317aa9e6 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 1026
Preferences`-[PListController tableView:cellForRowAtIndexPath:] + 1026:
-> 0x317aa9e6: mov    r0, r6
   0x317aa9e8: add    sp, #28
   0x317aa9ea: pop.w {r8, r10, r11}
   0x317aa9ee: pop   {r4, r5, r6, r7, pc}
(lldb) po $r6
<PSTableCell: 0x15f8c6a0; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My
Number'; tag = 2; layer = <CALayer: 0x15f7c0b0>>
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7b8d0; frame = (0 0; 0 0); text = '+86PhoneNumber';
userInteractionEnabled = NO; layer = <UILabelLayer: 0x15f7b990>>

```

Now we can confirm that objc_msgSendSuper2 calls [PListController tableView:cellForRowAtIndexPath:], and its return value does come from R6. Well, where does R6 come from? When we track back to look for the source of R6, we can see multiple occurrences of R6 as the 1st argument of multiple objc_msgSend, as shown in figure 6-34.

```

MOVW      R0, #(:lower16:(selRef_setSpecifier_ - 0x2A9F79AA))
MOV       R2, R11
MOVT.W   R0, #(:upper16:(selRef_setSpecifier_ - 0x2A9F79AA))
ADD      R0, PC ; selRef_setSpecifier_
LDR      R1, [R0] ; "setSpecifier:"
MOV      R0, R6
BLX      _objc_msgSend
MOVW      R0, #(:lower16:(selRef_refreshCellContentsWithSpecifier_ - 0x2A9F7
MOV       R2, R11
MOVT.W   R0, #(:upper16:(selRef_refreshCellContentsWithSpecifier_ - 0x2A9F7
ADD      R0, PC ; selRef_refreshCellContentsWithSpecifier_
LDR      R1, [R0] ; "refreshCellContentsWithSpecifier:"
MOV      R0, R6
BLX      _objc_msgSend
MOV      R0, #(_OBJC_IVAR_$_PListController._forceSynchronousIconLoadForCr
ADD      R0, PC ; char _forceSynchronousIconLoadForCreatedCells;
LDR      R0, [R0] ; char _forceSynchronousIconLoadForCreatedCells;
LDRB     R0, [R4, R0]
CBZ      R0, loc_2A9F79E6

MOV      R0, #(selRef_forceSynchronousIconLoadOnNextIconLoad -
ADD      R0, PC ; selRef_forceSynchronousIconLoadOnNextIconLoad
LDR      R1, [R0] ; "forceSynchronousIconLoadOnNextIconLoad"
MOV      R0, R6
BLX      _objc_msgSend

```

Figure 6-34 Multiple occurrences of R6

Keep looking upwards, you will find that R6 are assigned with various initialized objects, as shown in figure 6-35, figure 6-36, and figure 6-37.

```

STR.W      R11, [SP,#0x34+var_34]
ADD        R0, PC ; selRef_initWithStyle_reuseIdentifier_specifier_
LDR        R1, [R0] ; "initWithStyle:reuseIdentifier:specifier"...
MOV        R0, R6
BLX        _objc_msgSend
MOV        R1, #(selRef_autorelease - 0x2A9F784C) ; selRef_autorelease
ADD        R1, PC ; selRef_autorelease
LDR        R1, [R1] ; "autorelease"
BLX        _objc_msgSend
MOV        R6, R0

```

Figure 6-35 The assignment of R6

```

loc_2A9F7874
MOVW       R0, #(:lower16:(selRef_initWithStyle_reuseIdent
MOV        R2, R5
MOVT.W     R0, #(:upper16:(selRef_initWithStyle_reuseIdent
MOVS       R3, #0
ADD        R0, PC ; selRef_initWithStyle_reuseIdentifier_
LDR        R1, [R0] ; "initWithStyle:reuseIdentifier:"
MOV        R0, R6
BLX        _objc_msgSend
MOV        R1, #(selRef_autorelease - 0x2A9F7896) ; selRef_
ADD        R1, PC ; selRef_autorelease
LDR        R1, [R1] ; "autorelease"
BLX        _objc_msgSend
MOV        R6, R0

```

Figure 6-36 The assignment of R6

```

BLX        _objc_msgSend
MOV        R8, R0
MOV        R0, #(selRef_alloc - 0x2A9F77EE)
ADD        R0, PC ; selRef_alloc
LDR        R1, [R0] ; "alloc"
MOV        R0, R5
BLX        _objc_msgSend
MOV        R6, R0

```

Figure 6-37 The assignment of R6

This makes sense; the functionality of `tableView:cellForRowAtIndexPath:` is basically returning an available cell. So, its regular implementation is to create an empty cell at first, then configure it with other methods. Well, where does the configuration of “my number” happen?

Regardless of efficiency, we can investigate from the beginning of `[PSListController tableView:cellForRowAtIndexPath:]`. Since there’s a limited number of `objc_msgSends`, by printing out `[$r6 detailTextLabel]` before and after each `objc_msgSend` and comparing the differences, we can definitely locate this configuration `objc_msgSend`; if you’re good at math, dichotomy can be used in this scenario, you can inspect from the middle. Anyway, it’s just a matter of personal preferences. In this example, I use a compromised dichotomy, as shown in figure 6-38.

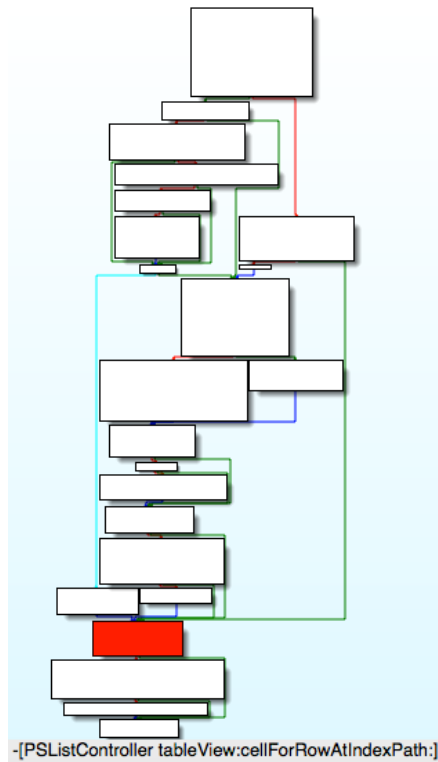


Figure 6-38 [PListController tableView:cellForRowAtIndexPath:]

Dichotomy increases the efficiency of our investigation, but it brings a new question: [PListController tableView:cellForRowAtIndexPath:] branches a lot, where should we choose as the investigation starting point to avoid missing any branches? Because [PListController tableView:cellForRowAtIndexPath:] will definitely execute code in the red block in figure 6-38, if we start from this block, we can make sure every branch is investigated. Next let's investigate the 1st objc_msgSend in this block, if [\$r6 detailTextLabel] contains my number, then we should investigate upwards, otherwise we should investigate downwards. Take a look at the assembly in the red block, as shown in figure 6-39.

```

loc_2A9F7966
MOV      R0, #(selRef_class - 0x2A9F797A) ; selRef_class
MOV      R2, #(classRef_PSTableCell - 0x2A9F797C) ; classRef_PSTableCell
ADD      R0, PC ; selRef_class
ADD      R2, PC ; classRef_PSTableCell
LDR      R1, [R0] ; "class"
LDR      R0, [R2] ; _OBJC_CLASS_$_PSTableCell
BLX     _objc_msgSend
MOV      R2, R0
MOV      R0, #(selRef_isKindOfClass_ - 0x2A9F7990) ; selRef_isKindOfClass_
ADD      R0, PC ; selRef_isKindOfClass_
LDR      R1, [R0] ; "isKindOfClass:"
MOV      R0, R6
BLX     _objc_msgSend
TST.W   R0, #0xFF
BEQ     loc_2A9F79E6

```

Figure 6-39 loc_2a9f7966

There are 2 objc_msgSends, so we start from the top one, as shown in figure 6-40.

```

2A9F7966 loc_2A9F7966                                ; CODE XREF: -[PListController
2A9F7966                                            ; -[PListController tableView:c
2A9F7966      MOV      R0, #(selRef_class - 0x2A9F797A) ; selRe
2A9F7966      MOV      R2, #(classRef_PSTableCell - 0x2A9F797C)
2A9F7976      ADD      R0, PC ; selRef_class
2A9F7978      ADD      R2, PC ; classRef_PSTableCell
2A9F797A      LDR      R1, [R0] ; "class"
2A9F797C      LDR      R0, [R2] ; _OBJC_CLASS_$_PSTableCell
2A9F797E      BLX     _objc_msgSend
2A9F7982      MOV      R2, R0
2A9F7984      MOV      R0, #(selRef_isKindOfClass_ - 0x2A9F7990
2A9F798C      ADD      R0, PC ; selRef_isKindOfClass_
2A9F798E      LDR      R1, [R0] ; "isKindOfClass:"
2A9F7990      MOV      R0, R6
2A9F7992      BLX     _objc_msgSend

```

Figure 6-40 Check out address of objc_msgSend

ASLR offset of Preferences is 0x6db3000 as we have just seen it. So the breakpoint should be set at $0x6db3000 + 0x2A9F797E = 0x317AA97E$. Trigger it and see if PSTableCell contains my number already:

```

(lldb) br s -a 0x317AA97E
Breakpoint 10: where = Preferences`-[PListController tableView:cellForRowAtIndexPath:]
+ 922, address = 0x317aa97e
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa97e Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 922, queue = 'com.apple.main-thread, stop reason =
breakpoint 10.1
   frame #0: 0x317aa97e Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 922
Preferences`-[PListController tableView:cellForRowAtIndexPath:] + 922:
-> 0x317aa97e: blx    0x31825f04          ; symbol stub for:
____NETRBCClientResponseHandler_block_invoke
   0x317aa982: mov    r2, r0
   0x317aa984: movw  r0, #59804
   0x317aa988: movt  r0, #1736
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<UILabelLayer: 0x15fd1c90>>

```

The cell doesn't hold my number yet, which indicates that my number is generated after the red block, i.e. somewhere in the last 3 blocks of code in figure 6-38. Based on this conclusion, let's keep executing "ni" command, then "po [\$r6 detailTextLabel]" before and after each objc_msgSend:

```

(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa982 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 926, queue = 'com.apple.main-thread, stop reason =
instruction step over
   frame #0: 0x317aa982 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 926
Preferences`-[PListController tableView:cellForRowAtIndexPath:] + 926:
-> 0x317aa982: mov    r2, r0
   0x317aa984: movw  r0, #59804
   0x317aa988: movt  r0, #1736
   0x317aa98c: add   r0, pc
(lldb) po [$r6 detailTextLabel]

```

```

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
.....
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa992 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 942, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa992 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 942
Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 942:
-> 0x317aa992: blx    0x31825f04                ; symbol stub for:
____NETRBClientResponseHandler_block_invoke
    0x317aa996: tst.w  r0, #255
    0x317aa99a: beq    0x317aa9e6                ; -[PSListController
tableView:cellForRowAtIndexPath:] + 1026
    0x317aa99c: movw   r0, #60302
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa996 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 946, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa996 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 946
Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 946:
-> 0x317aa996: tst.w  r0, #255
    0x317aa99a: beq    0x317aa9e6                ; -[PSListController
tableView:cellForRowAtIndexPath:] + 1026
    0x317aa99c: movw   r0, #60302
    0x317aa9a0: mov    r2, r11
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
.....
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9ac Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 968, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9ac Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 968
Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 968:
-> 0x317aa9ac: blx    0x31825f04                ; symbol stub for:
____NETRBClientResponseHandler_block_invoke
    0x317aa9b0: movw   r0, #60822
    0x317aa9b4: mov    r2, r11
    0x317aa9b6: movt   r0, #1736
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9b0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 972, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9b0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 972

```



```

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 972:
-> 0x317aa9b0: movw  r0, #60822
    0x317aa9b4: mov   r2, r11
    0x317aa9b6: movt  r0, #1736
    0x317aa9ba: add   r0, pc
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
.....
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9c0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988
Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988:
-> 0x317aa9c0: blx   0x31825f04 ; symbol stub for:
____NETRBClientResponseHandler_block_invoke
    0x317aa9c4: movw  r0, #4312
    0x317aa9c8: movt  r0, #1737
    0x317aa9cc: add   r0, pc
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9c4 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 992, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9c4 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 992
Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 992:
-> 0x317aa9c4: movw  r0, #4312
    0x317aa9c8: movt  r0, #1737
    0x317aa9cc: add   r0, pc
    0x317aa9ce: ldr   r0, [r0]
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); text = '+86PhoneNumber';
userInteractionEnabled = NO; layer = <_UILabelLayer: 0x15fd1c90>>

```

Obviously, my number appears after objc_msgSend at 0x317aa9c0. Because 0x317aa9c0 - 0x6db3000 = 0x2A9F79C0, we can locate this address in IDA, as shown in figure 6-41.

text: 2A9F79BC	LDR	R1, [R0]; "refreshCellContentsWithSpecifier:"
text: 2A9F79BE	MOV	R0, R6
text: 2A9F79C0	BLX	_objc_msgSend

Figure 6-41 The configuration objc_msgSend

As its name suggests, this method refreshes the cell contents with something specific. Let's uncover this "something specific": set a breakpoint at this objc_msgSend, then trigger it and print its argument:

```

(lldb) br s -a 0x317AA9C0
Breakpoint 11: where = Preferences`-[PSListController tableView:cellForRowAtIndexPath:]
+ 988, address = 0x317aa9c0
Process 268587 stopped

```

```

* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 988, queue = 'com.apple.main-thread, stop reason =
breakpoint 11.1
  frame #0: 0x317aa9c0 Preferences`-[PListController
tableView:cellForRowAtIndexPath:] + 988
Preferences`-[PListController tableView:cellForRowAtIndexPath:] + 988:
-> 0x317aa9c0: blx    0x31825f04          ; symbol stub for:
____NETRBClientResponseHandler_block_invoke
  0x317aa9c4: movw   r0, #4312
  0x317aa9c8: movt   r0, #1737
  0x317aa9cc: add    r0, pc
(lldb) p (char *)$r1
(char *) $97 = 0x318362d2 "refreshCellContentsWithSpecifier:"
(lldb) po $r2
My Number      ID:myNumberCell 0x170ece60      target:<PhoneSettingsController
0x170ed760: navItem <UINavigationController: 0x170d0b40>, view <UITableView: 0x16acb200; frame
= (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x15d232d0>; layer =
<CALayer: 0x15fc9110>; contentOffset: {0, -64}; contentSize: {320, 717.5}>>
(lldb) po [$r2 class]
PSSpecifier

```

As the output shows, “something specific”, i.e. specifier, is a PSSpecifier object, and it’s tightly related to my number. If you have carefully read the preferences specifier plist standard in section PreferenceBundle of the last chapter, you would know that the contents of a PSTableCell are specified by a PSSpecifier. Further more, we can acquire the setter and getter of PSSpecifier through [PSSpecifier valueForKey:@“set”] and [PSSpecifier valueForKey:@“get”] like this:

```

(lldb) po [$r2 valueForKey:@"set"]
setMyNumber:specifier:
(lldb) po [$r2 valueForKey:@"get"]
myNumber:

```

We can also get their target through [PSSpecifier target]:

```

(lldb) po [$r2 target]
<PhoneSettingsController 0x170ed760: navItem <UINavigationController: 0x170d0b40>, view
<UITableView: 0x16acb200; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers =
<NSArray: 0x15d232d0>; layer = <CALayer: 0x15fc9110>; contentOffset: {0, -64};
contentSize: {320, 717.5}>>

```

Excellent! Now we know my number on PSTableCell is set through [PhoneSettingsController setMyNumber:specifier:], and is got through [PhoneSettingsController myNumber:] (Do you still remember these 2 methods?), so there must be a method inside myNumber: that returns my number, as shown in figure 6-42.

```

; PhoneSettingsController - (id)myNumber:(id)
; Attributes: bp-based frame

; id __cdecl -[PhoneSettingsController myNumber:](struct PhoneSettingsController *self, SEL, id)
__PhoneSettingsController_myNumber__

var_10= -0x10

PUSH        {R4-R7,LR}
ADD         R7, SP, #0xC
SUB         SP, SP, #4
MOV         R4, R0
MOV         R0, #(selRef_telephony - 0x25BB3FCA) ; selRef_telephony
MOV         R6, R2
MOVW       R2, #(:lower16:(classRef_PhoneSettingsTelephony - 0x25BB3FD2))
ADD         R0, PC ; selRef_telephony
MOVT.W     R2, #(:upper16:(classRef_PhoneSettingsTelephony - 0x25BB3FD2))
LDR         R1, [R0] ; "telephony"
ADD         R2, PC ; classRef_PhoneSettingsTelephony
LDR         R0, [R2] ; _OBJC_CLASS_$_PhoneSettingsTelephony
BLX        _objc_msgSend
MOV         R1, #(selRef_myNumber - 0x25BB3FE2) ; selRef_myNumber
ADD         R1, PC ; selRef_myNumber
LDR         R1, [R1] ; "myNumber"
BLX        _objc_msgSend
MOV         R5, R0
BLX        UIUnformattedPhoneNumberFromString
MOV         R2, R0

```

Figure 6-42 [PhoneSettingsController myNumber:]

The implementation of [PhoneSettingsController myNumber:] is rather straightforward. This method simply checks whether the length of [[PhoneSettingsTelephony telephony] myNumber] is 0. If it is not 0, it is returned as my number, otherwise this method returns an “unknown number” as an error reminder. Let’s test [[PhoneSettingsTelephony telephony] myNumber] with Cypcript:

```

FunMaker-5:~ root# cypcript -p Preferences
cy# [[PhoneSettingsTelephony telephony] myNumber]
@"+86PhoneNumber"

```

Now, press home button to quit Preferences, then terminate it completely and make sure it’s not running in the background. After that, launch it again and don’t enter MobilePhoneSettings for now, let’s test this method again:

```

FunMaker-5:~ root# cypcript -p Preferences
cy# [[PhoneSettingsTelephony telephony] myNumber]
ReferenceError: Can't find variable: PhoneSettingsTelephony

```

An error happens. What’s wrong? The reason is that PhoneSettingsTelephony is a class of MobilePhoneSettings.bundle. If we don’t enter MobilePhoneSettings, this bundle will not be loaded, so this class doesn’t exist. In other words, this method will only work after MobilePhoneSettings.bundle is loaded. The way Preference.app loads MobilePhoneSettings.bundle is called lazy load, which is common in iOS reverse engineering. When you come across it, welcome to discuss with us on <http://bbs.iosre.com>.

So far, we can say we have already found the target function, because we have got both the caller and arguments of this method, plus no UI operation is involved, we can call this method neatly. However, there is still a fly in the ointment: MobilePhoneSettings.bundle must be

loaded, which weakens elegancy. Is there any way that enables us to get rid of this burden? I think so. Because ultimately, my number is stored on SIM card, the original data source of [PhoneSettingsTelephony myNumber] should come from SIM card. Whereas, SIM card accessibility is obviously not limited to MobilePhoneSettings.bundle, there must be a more common as well lower level library that can read SIM card. If we can locate this library, we can get my number without loading MobilePhoneSettings.bundle. Since it's supposed to be a lower level library, naturally, we should dig into [PhoneSettingsTelephony myNumber] to find out how it reads my number, as shown in figure 6-43.

```

; id __cdecl -[PhoneSettingsTelephony myNumber](struct PhoneSettingsTelephony *self, SEL)
__PhoneSettingsTelephony_myNumber_

var_24= -0x24
var_20= -0x20
var_1C= -0x1C
var_18= -0x18

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10}
SUB       SP, SP, #0x10
MOVW     R10, #(:lower16:(_OBJC_IVAR_$_PhoneSettingsTelephony._myNumber - 0x25BB6300)) ; NSString *_myNumber;
MOV      R4, R0
MOVT.W   R10, #(:upper16:(_OBJC_IVAR_$_PhoneSettingsTelephony._myNumber - 0x25BB6300)) ; NSString *_myNumber;
ADD      R10, PC ; NSString *_myNumber;
LDR.W    R6, [R10] ; NSString *_myNumber;
LDR      R0, [R4,R6]
CBZ      R0, loc_25BB636C

elRef_shouldLogType - 0x25BB631E) ; selRef_shouldLogType_
lassRef_TULogging - 0x25BB6320) ; classRef_TULogging
lower16:(cfstr_Phone - 0x25BB632A)) ; "Phone"

loc_25BB636C
BL       _PhoneSettingsCopyMyNumber

```

Figure 6-43 [PhoneSettingsTelephony myNumber]

This method is also very simple. It judges if the instance variable `_myNumber` is nil; if not, branches left and records “My Number requested, returning cached value: %@”, namely, returns a data in cache; or else branches right, first get my number by calling `PhoneSettingsCopyMyNumber`, then records “My Number requested, no cached value, fetched: %@”, namely, my number is not in cache, so it returns a newly fetched data. In consequence, `PhoneSettingsCopyMyNumber` is able to get my number, but as its name suggests, it is still a function inside `MobilePhoneSettings.bundle`, we can't call it from outside this bundle. We're one step further, but not far enough. Let's continue by digging into `PhoneSettingsCopyMyNumber`, as shown in figure 6-44.

```
EXPORT _PhoneSettingsCopyMyNumber
 PhoneSettingsCopyMyNumber
PUSH      {R7,LR}
MOV       R7, SP
BLX      _CTSettingCopyMyPhoneNumber
MOV       R1, #(selRef_autorelease - 0x25BB22EC) ; selRef_autorelease
ADD      R1, PC ; selRef_autorelease
LDR      R1, [R1] ; "autorelease"
BLX      _objc_msgSend
POP.W    {R7,LR}
B.W      _PhoneSettingsCopyFormattedNumberBySIMCountry
; End of function _PhoneSettingsCopyMyNumber
```

Figure 6-44 PhoneSettingsCopyMyNumber

This snippet first calls CTSettingCopyMyPhoneNumber and autoreleases the return value, then calls PhoneSettingsCopyFormattedNumberBySIMCountry, which seems to format the phone number according to the country of the SIM card. Judging from the name and context, CTSettingCopyMyPhoneNumber looks like the target function we are looking for. And the prefix CT implies that it comes from CoreTelephony rather than MobilePhoneSettings. Double click this function to see its implementation, as shown in figure 6-45.

```
; Attributes: thunk
CTSettingCopyMyPhoneNumber
LDR      R12, =(_CTSettingCopyMyPhoneNumber_ptr - 0x25BC31D4)
ADD      R12, PC, R12 ; _CTSettingCopyMyPhoneNumber_ptr
LDR      PC, [R12] ; __imp__CTSettingCopyMyPhoneNumber
; End of function _CTSettingCopyMyPhoneNumber
```

Figure 6-45 CTSettingCopyMyPhoneNumber

As expected, it's an external function. Double click “__imp__CTSettingCopyMyPhoneNumber” to check out which library it originates from; it's exactly CoreTelephony. Quit Preferences and terminate it completely in the background, then relaunch it and don't enter MobilePhoneSettings. Now let's attach debugserver to it and take a look at its image list with LLDB, we will see CoreTelephony on the list. It means that we can call CTSettingCopyMyPhoneNumber to get my unformatted number without loading MobilePhoneSettings.bundle, which perfectly meets our requirements of a target function. Finally, the last question: what're its arguments and return value?

Judging from figure 6-44, CTSettingCopyMyPhoneNumber doesn't seem to have any argument; before CTSettingCopyMyPhoneNumber, R0~R3 don't even show at all. If it has any argument, then R0~R3 come from its caller, i.e. PhoneSettingsCopyMyNumber. However, as we can see in figure 6-43, before PhoneSettingsCopyMyNumber, only R0 occurs, and if it

branches right, R0 is permanently 0, if R0 is an argument, it's meaningless. Therefore, PhoneSettingsCopyMyNumber doesn't seem to have any argument either. To play it safe, let's reconfirm our guesses by checking the implementation of CTSettingCopyMyPhoneNumber in CoreTelephony, as shown in figure 6-46.

```
EXPORT _CTSettingCopyMyPhoneNumber
_CTSettingCopyMyPhoneNumber

var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_20= -0x20
var_1C= -0x1C

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB       SP, SP, #0x1C
MOVS     R6, #0
STR       R6, [SP,#0x34+var_1C]
BL        _CTTelephonyCenterGetDefault
MOV       R4, R0
ADD.W    R11, SP, #0x34+var_1C
ADD.W    R8, SP, #0x34+var_28
ADD.W    R10, SP, #0x34+var_20
MOVS     R5, #0

loc_2226760E
MOV       R0, R8
MOV       R1, R4
```

Figure 6-46 CTSettingCopyMyPhoneNumber

According to the naming conventions of Objective-C functions, CTTelephonyCenterGetDefault is a getter and should return something; as a result, R0 under “BL _CTTelephonyCenterGetDefault” is set to the return value of CTTelephonyCenterGetDefault. Meanwhile, at the bottom of figure 6-46, R1 is set to R4 in “MOV R1, R4”. If R0 and R1 are arguments, then they are useless, which doesn't make sense. Now we can say for sure that CTSettingCopyMyPhoneNumber has no argument. What about its return value? We naturally guess it's an NSString object. Let's verify it by setting a breakpoint at the end of CTSettingCopyMyPhoneNumber, and print out R0. First locate to the end of CTSettingCopyMyPhoneNumber in IDA, as shown in figure 6-47.


```

text:2226763A loc_2226763A ; CODE XREF
text:2226763A ADD SP, SP, #0x1C
text:2226763C POP.W {R8,R10,R11}
text:22267640 POP {R4-R7,PC}
text:22267640 ; End of function _CTSettingCopyMyPhoneNumber

```

Figure 6-47 CTSettingCopyMyPhoneNumber

Then quit Preferences and terminate it completely in the background, then relaunch it and don't enter MobilePhoneSettings. Next attach debugserver to it and take a look at CoreTelephony's ASLR offset with LLDB:

```

(lldb) image list -o -f
[ 0] 0x000b3000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.app/Preferences(0x00000000000b7000)
[ 1] 0x0026c000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x000000000026c000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard
[ 51] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreTelephony.framework/CoreTelephony
.....

```

The breakpoint should be set at $0x6db3000 + 0x2226763A = 0x2901A63A$, right? Then enter MobilePhoneSettings to trigger the breakpoint:

```

(lldb) br s -a 0x2901A63A
Breakpoint 1: where = CoreTelephony`CTSettingCopyMyPhoneNumber + 78, address =
0x2901a63a
Process 330210 stopped
* thread #1: tid = 0x509e2, 0x2901a63a CoreTelephony`CTSettingCopyMyPhoneNumber + 78,
queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
  frame #0: 0x2901a63a CoreTelephony`CTSettingCopyMyPhoneNumber + 78
CoreTelephony`CTSettingCopyMyPhoneNumber + 78:
-> 0x2901a63a: add    sp, #28
0x2901a63c: pop.w  {r8, r10, r11}
0x2901a640: pop    {r4, r5, r6, r7, pc}
0x2901a642: nop
(lldb) po $r0
+86PhoneNumber
(lldb) po [$r0 class]
__NSCFString

```

It is indeed an NSString, so the prototype of this function can be reconstructed:

```
NSString *CTSettingCopyMyPhoneNumber(void);
```

This is our target function, as well the data source of PSTableCell. We've finally found it through analyzing the call chain of [PhoneSettingsController tableView:cellForRowAtIndexpath:], hurray! Just remember to release the return value when you make use of this function. At last, let's write a tweak to test this function.

1. Create tweak project "iOSREGetMyNumber" using Theos:

```
snakeninny-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl
```

NIC 2.0 - New Instance Creator

```
-----
[1.] iphone/application
[2.] iphone/cydyget
[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): iOSREGetMyNumber
Package Name [com.yourcompany.iosregetmynumber]: com.iosre.iosregetmynumber
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.Preferences
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: Preferences
Instantiating iphone/tweak in iosregetmynumber/...
Done.
```

2. Edit Tweak.xml as follows:

```
extern "C" NSString *CTSettingCopyMyPhoneNumber(void); // From CoreTelephony

%hook PreferencesAppController
- (BOOL)application:(id)arg1 didFinishLaunchingWithOptions:(id)arg2
{
    BOOL result = %orig;
    NSLog(@"iOSRE: my number = %@", [CTSettingCopyMyPhoneNumber() autorelease]);
    return result;
}
%end
```

3. Edit Makefile and control

The finalized 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 = iOSREGetMyNumber
iOSREGetMyNumber_FILES = Tweak.xml
iOSREGetMyNumber_FRAMEWORKS = CoreTelephony # CTSettingCopyMyPhoneNumber is from here

include $(THEOS_MAKE_PATH)/tweak.mk

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

The finalized control looks like this:

```
Package: com.iosre.iosregetmynumber
```



```
Name: iOSREGetMyNumber
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Get my number just like MobilePhoneSettings!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

4. Test

Compile and install the tweak on iOS, then launch Preferences without entering MobilePhoneSettings. After that, ssh into iOS and take a look at the syslog:

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 29 23:23:01 FunMaker-5 Preferences[2078]: iOSRE: my number = +86PhoneNumber
```

5. P.S.

I have set the region of my iPhone 5 to US, so PhoneSettingsCopyFormattedNumberBySIMCountry has formatted my number from “+86PhoneNumber” to “+86 Pho-neNu-mber”, which is the American phone number format.

You’ll run into CTSettingCopyMyPhoneNumber more frequently as you reverse more. Actually, the prototype of CTSettingCopyMyPhoneNumber should be:

```
CFStringRef CTSettingCopyMyPhoneNumber(void);
```

Since NSString * and CFStringRef are toll-free bridged, our prototype is OK.

Because there is a keyword “copy” in the name of CTSettingCopyMyPhoneNumber and it returns a CoreData object, we are responsible to release the return value according to Apple’s “Ownership Policy”.

In this section, we have shed considerable light to refine “locate target functions” with ARM level reverse engineering and enhanced the methodology of writing a tweak. Specifically, we’ve divided “locate target functions” into 2 steps, i.e. “cut into the target App and find the UI function” and “locate the target function from the UI function”. By combining Cycrypt, IDA and LLDB, we have not only located the target functions, but also analyzed their arguments and return values to reconstruct their prototypes. The methodology we used in the examples can work on at least 95% of all Apps; however, if you unfortunately encounter those 5%, please share and discuss with us on <http://bbs.iosre.com>.

6.3 Advanced LLDB usage

I bet the last section has opened a new chapter of iOS reverse engineering for you. The combination of IDA and LLDB can easily beat them all, and with the help of ARM architecture reference manual, you can conquer almost all Apps. I know you're already desperate to practice what you have just learned.

Hold your horses for now. Although the examples in section 6.2 have synthetically made use of IDA and LLDB, they haven't covered LLDB's common usage yet. In the next section, we'll go over some short LLDB examples for a better comprehension, which can greatly reduce our workload in practice.

6.3.1 Look for a function's caller

In the examples of the previous section, when we were restoring call chains, we primarily focused on the callees of a function, i.e. we've restored the bottom half of a call chain. When we're to restore the top half, we need to find out the caller of a function. Look at this snippet:

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --show-sdk-path` -framework
Foundation -o MainBinary main.m

#include <stdio.h>
#include <dlfcn.h>
#import <Foundation/Foundation.h>

extern void TestFunction0(void)
{
    NSLog(@"iOSRE: %u", arc4random_uniform(0));
}

extern void TestFunction1(void)
{
    NSLog(@"iOSRE: %u", arc4random_uniform(1));
}

extern void TestFunction2(void)
{
    NSLog(@"iOSRE: %u", arc4random_uniform(2));
}

extern void TestFunction3(void)
{
    NSLog(@"iOSRE: %u", arc4random_uniform(3));
}

int main(int argc, char **argv)
{
    TestFunction3();
    return 0;
}
```

Save this snippet as a file named main.m, and compile it with the sentence in the comments. Drag and drop MainBinary into IDA, and then check the cross references of NSLog, as shown in figure 6-48.

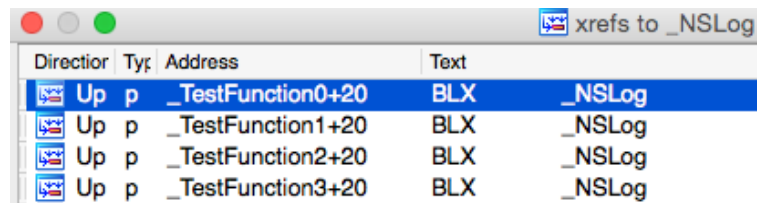


Figure 6-48 Check the cross references of NSLog

As we can see, NSLog appears in 4 functions. If we see “iOSRE: 0” in syslog when we are reversing, how can we know which NSLog it’s from? When there’re only tens lines of code, we can figure out by hand that only TestFunction3 is called, and it further calls NSLog. What if there are 20 TestFunctions that are called by 8 separate functions? When the amount of code increases, it’ll be too complicate to analyze manually. If we want to find the caller of NSLog under such circumstances, LLBD will be very helpful. Generally, there are 2 main methods.

- Inspect LR

Still remember LR register introduced in section 6.1? Its function is to save the return address of a function. So what’s a return address? Take an example:

```
void FunctionA()
{
.....
    FunctionB();
.....
}
```

In the above pseudo code, FunctionA calls FunctionB, while A and B are located in 2 different memory areas, and their addresses have no direct connection. After the execution of B, the process needs to go back to A to continue execution, as shown in figure 6-49.

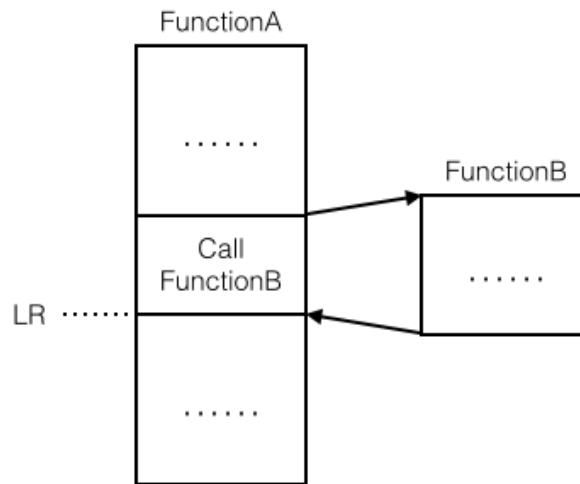


Figure 6-49 An illustration of return address

The address that the process returns to after the execution of FunctionB, is the return address, i.e. LR. Because it's inside FunctionB's caller, if we know the value of LR we can track to the caller. Let's explain this theory with an example. Drag and drop Foundation.framework's binary into IDA; locate to NSLog after the initial analysis, and check out its base address, as shown in figure 6-50.

```

text:2261AB94      EXPORT _NSLog
text:2261AB94      _NSLog                ; CODE XREF: -
text:2261AB94      ; -[NSLock loc
text:2261AB94      var_18                = -0x18
text:2261AB94      SUB                   SP, SP, #0xC
text:2261AB96      PUSH                 {R7,LR}
text:2261AB98      MOV                  R7, SP
text:2261AB9A      SUB                   SP, SP, #4
text:2261AB9C      ADD.W                R9, R7, #8
text:2261ABA0      STMIA.W              R9, {R1-R3}
text:2261ABA4      ADD.W                R1, R7, #8
text:2261ABA8      STR                   R1, [SP,#0x18+var_18]
text:2261ABAA      BL                   _NSLogv
text:2261ABAE      ADD                   SP, SP, #4
text:2261ABB0      POP.W                {R7,LR}
text:2261ABB4      ADD                   SP, SP, #0xC
text:2261ABB6      BX                   LR
text:2261ABB6      ; End of function _NSLog

```

Figure 6-50 Check out NSLog's base address

Its base address is 0x2261ab94, we will set a breakpoint on it shortly and print out the value of LR. Next, launch MainBinary with debugserver:

```

FunMaker-5:~ root# debugserver -x backboard *:1234 /var/tmp/MainBinary
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Listening to port 1234 for a connection from *...

```

Then connect with LLDB:

```

(lldb) process connect connect://localhost:1234
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7000 dyld`_dyld_start, stop reason = signal SIGSTOP
frame #0: 0x1fec7000 dyld`_dyld_start

```

```

dyld`_dyld_start:
-> 0x1fec7000: mov    r8, sp
    0x1fec7004: sub    sp, sp, #16
    0x1fec7008: bic    sp, sp, #7
    0x1fec700c: ldr    r3, [pc, #112]          ; _dyld_start + 132
(lldb) image list -f
[ 0] /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/usr/lib/dyld

```

Right at this moment, MainBinary is not run yet, and we are inside dyld. Next, keep entering “ni” until LLDB outputs “error: invalid thread”:

```

(lldb) ni
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7004 dyld`_dyld_start + 4, stop reason = instruction
step over
    frame #0: 0x1fec7004 dyld`_dyld_start + 4
dyld`_dyld_start + 4:
-> 0x1fec7004: sub    sp, sp, #16
    0x1fec7008: bic    sp, sp, #7
    0x1fec700c: ldr    r3, [pc, #112]          ; _dyld_start + 132
    0x1fec7010: sub    r0, pc, #8
(lldb)
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7008 dyld`_dyld_start + 8, stop reason = instruction
step over
    frame #0: 0x1fec7008 dyld`_dyld_start + 8
dyld`_dyld_start + 8:
-> 0x1fec7008: bic    sp, sp, #7
    0x1fec700c: ldr    r3, [pc, #112]          ; _dyld_start + 132
    0x1fec7010: sub    r0, pc, #8
    0x1fec7014: ldr    r3, [r0, r3]
.....
(lldb)
error: invalid thread

```

No more “ni” when the error occurs; now dyld begins to load MainBinary. Wait a moment, the process will stop again, and we are inside MainBinary, it’s okay to debug then:

```

Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7040 dyld`_dyld_start + 64, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x1fec7040 dyld`_dyld_start + 64
dyld`_dyld_start + 64:
-> 0x1fec7040: ldr    r5, [sp, #12]
    0x1fec7044: cmp    r5, #0
    0x1fec7048: bne    0x1fec7054              ; _dyld_start + 84
    0x1fec704c: add    sp, r8, #4

```

Check out ASLR offset of Foundation.framework:

```

(lldb) image list -o -f
[ 0] 0x000fc000 /private/var/tmp/MainBinary(0x0000000000100000)
[ 1] 0x000c6000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/usr/lib/dyld
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/Foundation.framework/Foundation
.....

```

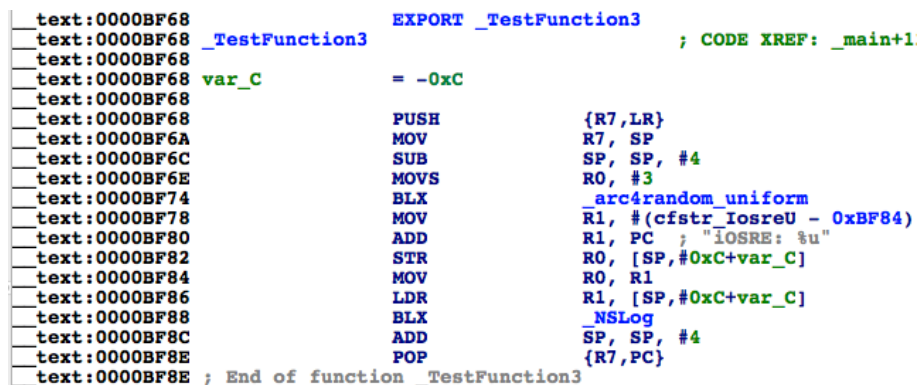
As usual, we should set the breakpoint at $0x6db3000 + 0x2261ab94 = 0x293CDB94$. Execute “c” to trigger the breakpoint:

```
(lldb) br s -a 0x293CDB94
Breakpoint 1: where = Foundation`NSLog, address = 0x293cdb94
(lldb) c
Process 450336 resuming
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x293cdb94 Foundation`NSLog, queue = 'com.apple.main-thread,
stop reason = breakpoint 1.1
    frame #0: 0x293cdb94 Foundation`NSLog
Foundation`NSLog:
-> 0x293cdb94:  sub    sp, #12
    0x293cdb96:  push   {r7, lr}
    0x293cdb98:  mov    r7, sp
    0x293cdb9a:  sub    sp, #4
```

Print out LR:

```
(lldb) p/x $lr
(unsigned int) $0 = 0x00107f8d
```

Because the base address of MainBinary is $0x000fc000$, open MainBinary in IDA and jump to $0x107f8d - 0xfc000 = 0xBF8D$, as shown in figure 6-51.



```
text:0000BF68 EXPORT _TestFunction3
text:0000BF68 _TestFunction3 ; CODE XREF: _main+1
text:0000BF68
text:0000BF68 var_C = -0xC
text:0000BF68
text:0000BF68 PUSH {R7,LR}
text:0000BF6A MOV R7, SP
text:0000BF6C SUB SP, SP, #4
text:0000BF6E MOVS R0, #3
text:0000BF74 BLX _arc4random_uniform
text:0000BF78 MOV R1, #(cfstr_iosreU - 0xBF84)
text:0000BF80 ADD R1, PC ; "iosre: %u"
text:0000BF82 STR R0, [SP,#0xC+var_C]
text:0000BF84 MOV R0, R1
text:0000BF86 LDR R1, [SP,#0xC+var_C]
text:0000BF88 BLX NSLog
text:0000BF8C ADD SP, SP, #4
text:0000BF8E POP {R7,PC}
text:0000BF8E ; End of function _TestFunction3
```

Figure 6-51 TestFunction3

$0xBF8D$ is right below “BLX NSLog”, so we have found the caller of NSLog. One thing should be noted is that because LR may change in the caller, the breakpoint should be set at the base address. Pretty easy, huh?

- Execute “ni” to get inside caller

Although “Inspect LR” is straightforward enough, we’ve played a trick: because we’ve already known NSLog is called inside MainBinary, we’ve subtracted MainBinary’s ASLR offset from LR to get the final result. But in more general cases, we don’t know which function calls NSLog, not to mention which image calls NSLog, so we don’t know whose ASLR offset should be subtracted from LR. To solve this problem, our theoretical base is still “After the execution of B, the process needs to go back to A to continue execution”; if we set a breakpoint at the end of

the callee and keep executing “ni”, we will come back to the caller. Let’s take another example: repeat the steps in last section to check out ASLR offset of Foundation.framework in MainBinary:

```
(lldb) image list -o -f
[ 0] 0x0000c000 /private/var/tmp/MainBinary(0x0000000000010000)
[ 1] 0x000c5000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/usr/lib/dyld
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/Foundation.framework/Foundation
.....
```

Its ASLR offset is 0x6db3000. According to figure 6-50, the address of the last instruction of NSLog is 0x2261ABB6, so set a breakpoint at $0x6db3000 + 0x2261ABB6 = 0x293CDDB6$, then enter “c” to trigger the breakpoint:

```
(lldb) br s -a 0x293CDDB6
Breakpoint 1: where = Foundation`NSLog + 34, address = 0x293cddb6
(lldb) c
Process 452269 resuming
(lldb) 2014-11-30 23:45:37.070 MainBinary[3454:452269] iOSRE: 1
Process 452269 stopped
* thread #1: tid = 0x6e6ad, 0x293cddb6 Foundation`NSLog + 34, queue = 'com.apple.main-
thread, stop reason = breakpoint 1.1
    frame #0: 0x293cddb6 Foundation`NSLog + 34
Foundation`NSLog + 34:
-> 0x293cddb6: bx    lr

Foundation`NSLogv:
    0x293cddb8: push  {r4, r5, r6, r7, lr}
    0x293cdbba: add   r7, sp, #12
    0x293cdbbc: sub   sp, #12
```

Notice the texts above “->”, it implies the present image. Keep executing “ni”:

```
(lldb) ni
Process 452269 stopped
* thread #1: tid = 0x6e6ad, 0x00017fa6 MainBinary`main + 22, queue = 'com.apple.main-
thread, stop reason = instruction step over
    frame #0: 0x00017fa6 MainBinary`main + 22
MainBinary`main + 22:
-> 0x17fa6: movs  r0, #0
    0x17fa8: movt  r0, #0
    0x17fac: add   sp, #12
    0x17fae: pop   {r7, pc}
```

Here comes MainBinary and the process stops at 0x17fa6. $0x17fa6 - 0xc000 = 0xbfa6$, so again, we have found NSLog’s caller TestFunction3 according to figure 6-51.

Both methods are simple and direct; choose whatever you like.

6.3.2 Change process execution flow

Why do we need to change process execution flow? Commonly it's because the code we want to debug could only be executed in specific conditions, which are hard to meet in the original execution flow. Under such circumstances, we have to change the flow to redirect the process to execute the target code for debugging. Reads awkward? Let's see an example.

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --show-sdk-path` -framework
Foundation -framework UIKit -o MainBinary main.m

#include <stdio.h>
#include <dlfcn.h>
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

extern void ImportantAndComplicatedFunction(void)
{
    NSLog(@"iOSRE: Suppose I'm a very important and complicated function");
}

int main(int argc, char **argv)
{
    if ([[UIDevice currentDevice] systemVersion] isEqualToString:@"8.1.1"])
    ImportantAndComplicatedFunction();
    return 0;
}
```

Save this snippet as main.m, and compile it with the sentence in the comments, then copy MainBinary to “/var/tmp/” on iOS:

```
snakeninnys-MacBook:6 snakeninny$ scp MainBinary root@iOSIP:/var/tmp/
MainBinary                               100%  49KB
48.6KB/s  00:00
```

Run it:

```
FunMaker-5:~ root# /var/tmp/MainBinary
FunMaker-5:~ root#
```

Because I'm using iOS 8.1, there is no output for sure. What if I am interested in ImportantAndComplicatedFunction but don't have iOS 8.1.1 in hand? Then I have to dynamically change the execution flow to make this function get called. I'll show you how, please keep focused. Drag and drop MainBinary into IDA, then locate to the branch before ImportantAndComplicatedFunction, as shown in figure 6-52.

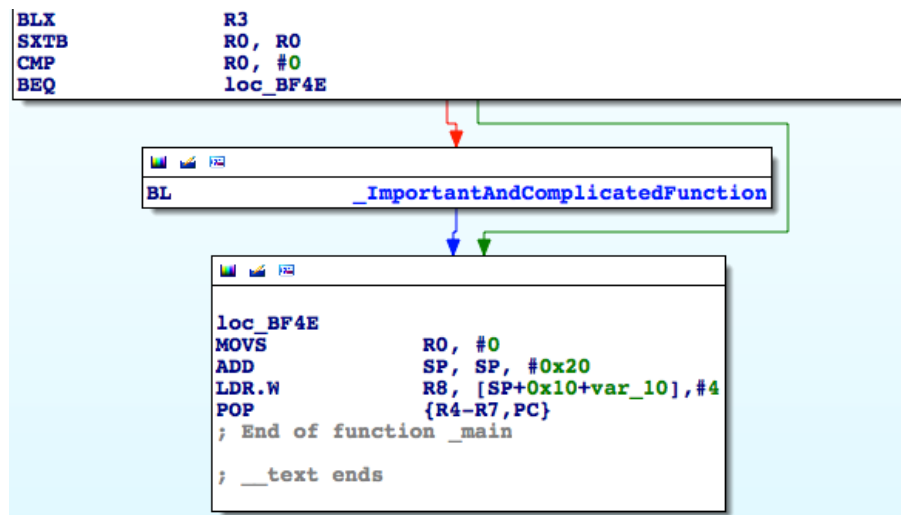


Figure 6-52 Before ImportantAndComplicatedFunction

Repeat the previous steps to check out MainBinary’s ASLR offset:

```
(lldb) image list -o -f
[ 0] 0x0000e000 /private/var/tmp/MainBinary(0x0000000000012000)
.....
```

Because the address of “CMP R0, #0” in figure 6-52 is 0xBF46, the breakpoint should be set at 0xbf46 + 0xe000 = 0x19F46. Trigger it with “c”, and print R0:

```
(lldb) br s -a 0x19F46
Breakpoint 1: where = MainBinary`main + 134, address = 0x00019f46
(lldb) c
Process 456316 resuming
Process 456316 stopped
* thread #1: tid = 0x6f67c, 0x00019f46 MainBinary`main + 134, queue = 'com.apple.main-
thread, stop reason = breakpoint 1.1
    frame #0: 0x00019f46 MainBinary`main + 134
MainBinary`main + 134:
-> 0x19f46: cmp    r0, #0
    0x19f48: beq    0x19f4e          ; main + 142
    0x19f4a: bl     0x19ea4          ; ImportantAndComplicatedFunction
    0x19f4e: movs  r0, #0
(lldb) p $r0
(unsigned int) $0 = 0
```

R0 is 0, so ImportantAndComplicatedFunction will not be executed. If we change R0 to 1, the situation changes all together:

```
(lldb) register write r0 1
(lldb) p $r0
(unsigned int) $1 = 1
(lldb) c
Process 456316 resuming
(lldb) 2014-12-01 00:41:47.779 MainBinary[3482:457105] iOSRE: Suppose I’m a very
important and complicated function
Process 456316 exited with status = 0 (0x00000000)
```

As we can see, we’ve changed the process execution flow by modifying the value of a register, thus achieved our goal.

6.4 Conclusion

The combination of IDA and LLDB is far more powerful than what we have introduced in this chapter, their usage ranges from App analysis to jailbreak, showing their omnipotence. Nonetheless, in the beginning stage of iOS reverse engineering, their usage is not likely to exceed the scope of this book. As soon as you can use them proficiently, your understanding of iOS would rise to a new level and you'll be able to summarize your own methodologies. There're still lots and lots of topics in ARM related iOS reverse engineering to further explore, and we're unable to cover them all in one book. Therefore, we will leave them to <http://bbs.iosre.com>, please stay focused.

To be honest, this chapter is rather difficult to understand, but this is the only path to be a real iOS reverse engineer. In part 4 of this book, we will turn methodologies in part 3 into practices and write 4 tweaks. I hope you know from the bottom of your heart whether you are talented enough to be an iOS reverse engineer after finishing all 4 practices. As Steve Jobs said, "It's more fun to be a pirate than to join the Navy". IMHO, being an iOS reverse engineer is way more fun than being just an App developer, but after all, it's up to you. Good luck!

Practices

The first 3 parts of this book have introduced the concepts, tools and theories of iOS reverse engineering, along with examples to give you a better understanding of them. I believe you have the same feeling that only if concepts, tools and theories are combined together can we get the best out reverse engineering.

So far, you may still feel unsatisfied with the fragmented and conservative examples. So in this part, we've prepared 4 original and serialized examples to show you the combination of concepts, tools and theories. They are:

- Characount for Notes 8
- Mark user specific emails as read automatically
- Save and share Sight in WeChat
- Detect and send iMessage

Now, welcome to the most splendid part of this book. Let's enjoy the art of iOS reverse engineering!

Practice 1: Characount for Notes 8

7.1 Notes

I bet Notes App (hereafter referred to as Notes) is one of your most familiar iOS Apps. Its UI and functionality have experienced very few changes since iOS came out. The simplicity and convenience of Notes win my heart, all my secrets are sealed in it, as shown in figure 7-1.

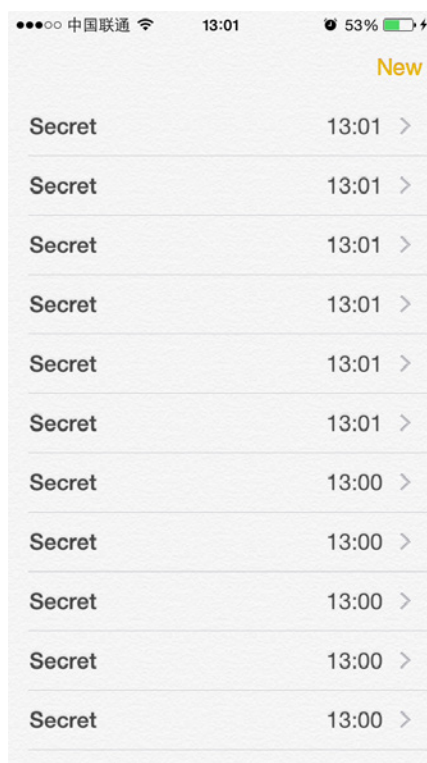


Figure 7- 1 Notes

Being a power user of Notes, not only do I save secrets in it, but also compose SMS or tweets in it. Since there is word limit on SMS and tweets, I really wish Notes can display each note's character count as a reminder. DIY is a born spirit of reverse engineers, so I've developed Characount for Notes, which is one of my daily necessities on iOS 6. It's not a difficult tweak, hence can be a stepping-stone for beginners like you. Our goal in this chapter is to rewrite

Characount for Notes on iOS 8, and all the following operations are performed on iPhone 5, iOS 8.1.

7.2 Tweak prototyping

On iOS 8, the original note browsing view looks like figure 7-2.

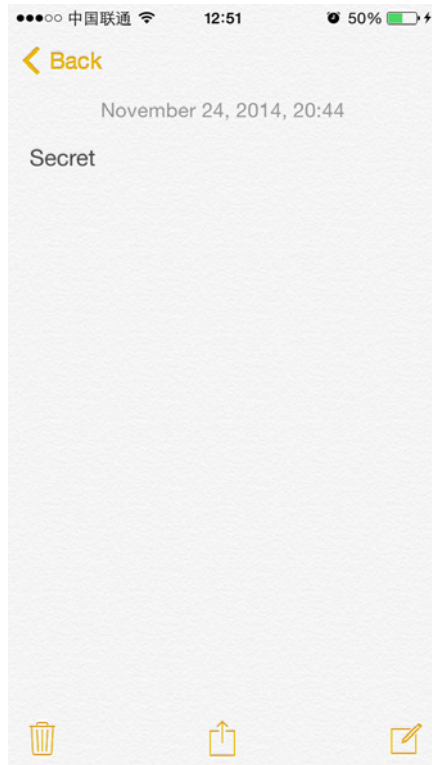


Figure 7- 2 Note browsing view on iOS 8

If we're to choose a place to display the character count of this note, where do you think looks better? If you used to be an iOS 6 user, do you remember that each note has a centered title as shown in figure 7-3?



Figure 7- 3 Note browsing view on iOS 6

However, Notes on iOS 8 has removed the title, leaving a blank navigation bar. Why don't we just display the character count here, as shown in figure 7-4?

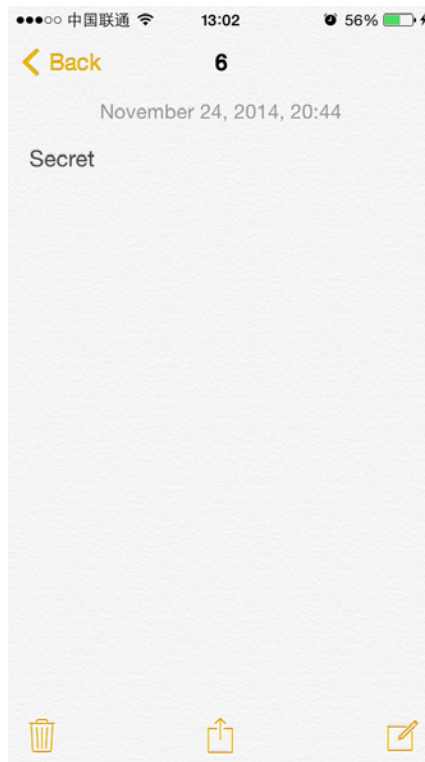


Figure 7- 4 Note browsing view with a title

It looks good! So, what exactly should we do to make Notes look like this? Hope you remember the saying in chapter 5 that everything you see on iOS is an object. Keep that in mind and think with me:

Every note is an object, and note browsing view contains the content and modification time of a note object. Since note browsing view is a subclass of `UIView`, we can trace back to its view controller via `nextResponder`, and further access all note concerned data via its view controller according to MVC design pattern. With the note data, we can initialize the character count when this view appears.

While we are editing a note, a “Done” button will appear on the right side of the navigation bar, as shown in figure 7-5.

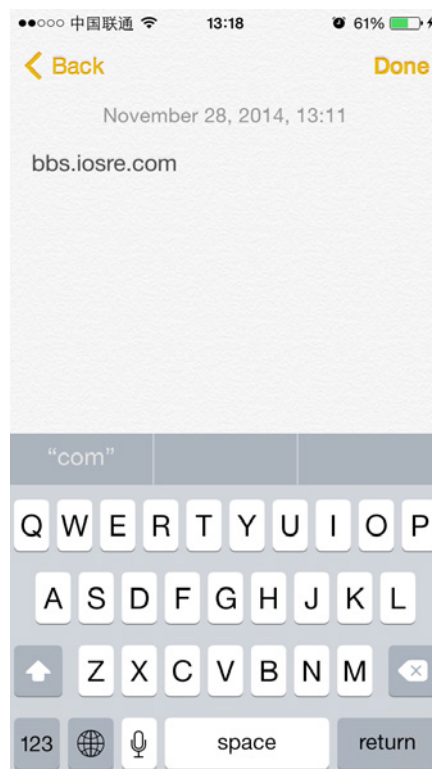


Figure 7- 5 “Done” button

After tapping “Done”, the current note is saved. This phenomenon indicates that a note is not saved in real time during editing, or we just don’t need this button at all. Of course, character count changes instantly with the editing content would be the ideal visual effect, so to accomplish this goal, we need to find a specific method which monitors the changes of the current note. In addition, we should be able to get the character count of this note and update the title just in time within this method. Because this kind of methods are usually defined in protocols, we should keep an eye on protocols in Notes.

Suppose we can get the current note’s character count, how do we put it on the navigation bar? Usually, the note browsing view controller inherits from UIViewController, which possesses a “title” property. So, “setTitle:” is the answer.

If we managed to solve these 3 problems, there’ll be no more technical difficulties for Characount for Notes. Code speaks louder than words, let’s move it!

7.2.1 Locate Notes’ executable

There’s no Notes.app under /Applications/ at all. Besides searching blindly, what else can we do to locate its executable? Do you still remember the trick of getting an App’s path in

dumpdecrypted section? Yeah, it's ps command again: first close all Apps, then open Notes and ssh to iOS to list all system processes with ps:

```
FunMaker-5:~ root# ps -e | grep /Applications/
 592 ??      0:37.70 /Applications/MobileMail.app/MobileMail
 761 ??      0:02.78
/Applications/MessagesNotificationViewService.app/MessagesNotificationViewService
1807 ??      0:00.55
/private/var/db/stash/_.29LMeZ/Applications/MobileSafari.app/webbookmarksd
2016 ??      0:05.23 /Applications/InCallService.app/InCallService
2619 ??      0:02.66 /Applications/MobileSMS.app/MobileSMS
2672 ??      0:01.20 /Applications/MobileNotes.app/MobileNotes
2678 ttys000    0:00.01 grep /Applications/
```

Among those processes, MobileNotes attracts us most. How to verify our guess? We can simply kill it and see whether Notes quit.

```
FunMaker-5:~ root# killall MobileNotes
```

Notes has quit as we expected, which clearly means that “/Applications/MobileNotes.app/MobileNotes” is Notes’ executable. Meanwhile, we’ve discovered some Apps that’re running in the background. Copy MobileNotes to OSX and get ready to class-dump it.

7.2.2 class-dump MobileNotes’ headers

Because Notes is a stock App, its executable is not encrypted, enabling us to class-dump it directly:

```
snakeninnys-MacBook:~ snakeninny$ class-dump -S -s -H
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/MobileNotes.app/MobileNotes -o
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/MobileNotes
```

We’ve got 88 headers in total. Let’s take a brief look to see what we can discover, as shown in figure 7-6.

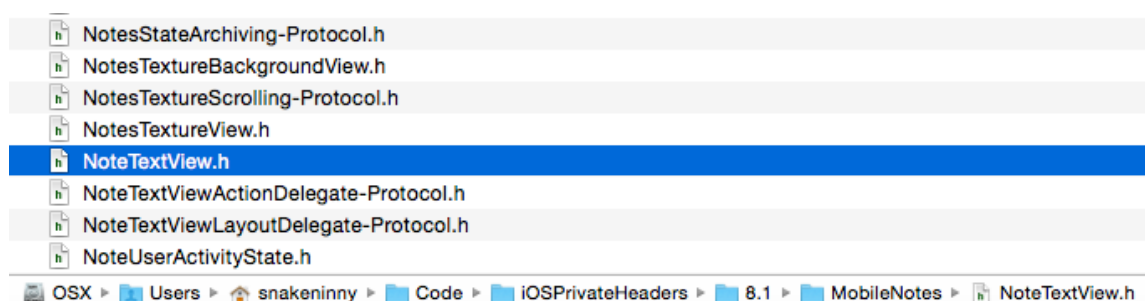


Figure 7- 6 Headers of Notes

Do you see the selected file in figure 7-6? I am not sure if it is a key clue of this chapter for now, but we’ll see.

7.2.3 Find the controller of note browsing view using Cycrypt

Again, recursiveDescription makes our days:

```
FunMaker-5:~ root# cycrypt -p MobileNotes
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<UIWindow: 0x17688db0; frame = (0 0; 320 568); gestureRecognizers = <NSArray:
0x17689620>; layer = <UIWindowLayer: 0x17688fc0>>
  | <UILayoutContainerView: 0x175bb880; frame = (0 0; 320 568); autoresize = W+H; layer
= <CALayer: 0x175bb900>>
    | | <UILayoutContainerView: 0x17699350; frame = (0 0; 320 568); clipsToBounds =
YES; gestureRecognizers = <NSArray: 0x1769cf60>; layer = <CALayer: 0x17699530>>
      | | | <UINavigationController: 0x176564c0; frame = (0 0; 320 568);
clipsToBounds = YES; autoresize = W+H; layer = <CALayer: 0x17658ec0>>
        | | | | <UIViewControllerWrapperView: 0x176d13b0; frame = (0 0; 320 568);
layer = <CALayer: 0x176d1530>>
          | | | | | <UILayoutContainerView: 0x1769dd80; frame = (0 0; 320 568);
clipsToBounds = YES; gestureRecognizers = <NSArray: 0x176a16f0>; layer = <CALayer:
0x1769de00>>
            | | | | | | <UINavigationController: 0x1769ebb0; frame = (0 0; 320
568); clipsToBounds = YES; autoresize = W+H; layer = <CALayer: 0x1769ec40>>
              | | | | | | | <UIViewControllerWrapperView: 0x175109e0; frame = (0
0; 320 568); layer = <CALayer: 0x175109b0>>
                | | | | | | | | <NotesBackgroundView: 0x175ee3e0; frame = (0 0;
320 568); gestureRecognizers = <NSArray: 0x17510a70>; layer = <CALayer: 0x175ee580>>
                  | | | | | | | | | <NotesTextureBackgroundView: 0x175ee5b0;
frame = (0 0; 320 568); clipsToBounds = YES; layer = <CALayer: 0x175ee630>>
                    | | | | | | | | | | <NotesTextureView: 0x175ee940; frame =
(0 -64; 320 640); layer = <CALayer: 0x175ee9c0>>
                      | | | | | | | | | | | <NoteContentLayer: 0x176c5110; frame = (0
0; 320 568); layer = <CALayer: 0x176ca850>>
                        | | | | | | | | | | | | <UIView: 0x175f2130; frame = (16 0;
288 0); hidden = YES; layer = <CALayer: 0x175dd2b0>>
                          | | | | | | | | | | | | | <NotesScrollView: 0x175f2a10;
baseClass = UIScrollView; frame = (0 0; 320 568); clipsToBounds = YES;
gestureRecognizers = <NSArray: 0x175f1b70>; layer = <CALayer: 0x175f28d0>;
contentOffset: {0, -64}; contentSize: {320, 460}>
                            | | | | | | | | | | | | | | <UIView: 0x175f09a0; frame = (0
0; 320 0); layer = <CALayer: 0x175f2790>>
                              | | | | | | | | | | | | | | | <UIView: 0x175f27e0; frame = (0
0; 0 460); layer = <CALayer: 0x175f2850>>
                                | | | | | | | | | | | | | | | | <NoteDateLabel: 0x175f3400;
baseClass = UILabel; frame = (69 5.5; 182 18); text = 'November 24, 2014, 20:44';
userInteractionEnabled = NO; layer = <UILabelLayer: 0x175f3560>>
                                  | | | | | | | | | | | | | | | | | <NoteTextView: 0x175ee3e0;
baseClass = _UICompatibilityTextView; frame = (6 28; 308 418); text = 'Secret';
clipsToBounds = YES; gestureRecognizers = <NSArray: 0x176c7ed0>; layer = <CALayer:
0x176d88e0>; contentOffset: {0, 0}; contentSize: {308, 52}>
.....
```

Look! There is a NoteTextView with the keyword “Secret”. Call nextResponder continuously until we get its controller:

```
cy# [#0x175ee3e0 nextResponder]
#"<NotesScrollView: 0x17d307c0; baseClass = UIScrollView; frame = (0 0; 320 568);
clipsToBounds = YES; gestureRecognizers = <NSArray: 0x17e502a0>; layer = <CALayer:
0x17d30b60>; contentOffset: {0, -64}; contentSize: {320, 251}>"
cy# [#0x17d307c0 nextResponder]
```

```

#"<NoteContentLayer: 0x17e505b0; frame = (0 0; 320 568); layer = <CALayer: 0x17e50470>>"
cy# [#0x17e505b0 nextResponder]
#"<NotesBackgroundView: 0x17e52320; frame = (0 0; 320 568); gestureRecognizers =
<NSArray: 0x17d0c940>; layer = <CALayer: 0x17e522f0>>"
cy# [#0x17e52320 nextResponder]
#"<NotesDisplayController: 0x17edc340>"

```

Okay, NoteDisplayController is the one. Let's see if setTitle: really changes the title of note browsing view:

```
cy# [#0x17edc340 setTitle:@"Characount = Character count"]
```

The UI after setTitle: is shown in figure 7-7.

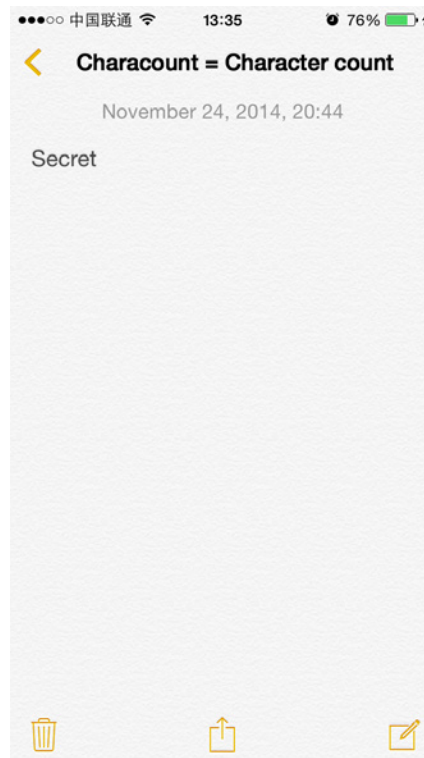


Figure 7- 7 UI After setTitle:

Neet! Mission 1, completed!

7.2.4 Get the current note object from NoteDisplayController

Strike while the iron is hot, let's overview NoteDisplayController.h.

```

@interface NotesDisplayController : UIViewController <NoteContentLayerDelegate,
UISheetDelegate, AFContextProvider, UIPopoverPresentationControllerDelegate,
UINavigationControllerDelegate, UIImagePickerControllerDelegate,
NotesQuickLookActivityItemDelegate, ScrollViewKeyboardResizerDelegate,
NSUserActivityDelegate, NotesStateArchiving>
{
.....
@property(nonatomic, getter=isVisible) BOOL visible; // @synthesize visible=_visible;
- (void)loadView;
@property(retain, nonatomic) NoteObject *note; // @synthesize note=_note;
.....
}

```

After going over this large header, we've found a property of NoteObject type. Since a note is exactly an object, NoteObject seems to be too obvious to believe... Hehe, let's print it in Cycript:

```
cy# [#0x17edc340 note]
#'<NoteObject: 0x176aa170> (entity: Note; id: 0x176a9040 <x-coredata://4B88CC7C-7A5F-4F15-9275-53C6D0ABE0C3/Note/p15> ; data: {\n  attachments = (\n  );\n  author = nil;\n  body = "0x176a8b20 <x-coredata://4B88CC7C-7A5F-4F15-9275-53C6D0ABE0C3/NoteBody/p15>";\n  containsCJK = 0;\n  contentType = 0;\n  creationDate = "2014-11-24 05:00:59 +0000";\n  deletedFlag = 0;\n  externalFlags = 0;\n  externalSequenceNumber = 0;\n  externalServerIntId = "-4294967296";\n  guid = "781B6C87-2855-4512-8864-50618754333A";\n  integerId = 3865;\n  isBookkeepingEntry = 0;\n  modificationDate = "2014-11-24 12:44:08 +0000";\n  serverId = nil;\n  store = "0x175a2b60 <x-coredata://4B88CC7C-7A5F-4F15-9275-53C6D0ABE0C3/Store/p1>";\n  summary = nil;\n  title = Secret;\n})'
```

Needless to say, NoteObject is exactly the current note. Each field in the description is explicit, let's take a look at its header:

```
@interface NoteObject : NSManagedObject
{
}

- (BOOL)belongsToCollection:(id)arg1;
@property(nonatomic) unsigned long long sequenceNumber;
- (BOOL)containsAttachments;
@property(retain, nonatomic) NSString *externalContentRef;
@property(retain, nonatomic) NSData *externalRepresentation;
@property(readonly, nonatomic) BOOL isValidServerIntId;
@property(nonatomic) long long serverIntId;
@property(nonatomic) unsigned long long flags;
@property(readonly, nonatomic) NSURL *noteId;
@property(readonly, nonatomic) BOOL isBeingMarkedForDeletion;
@property(readonly, nonatomic) BOOL isMarkedForDeletion;
- (void)markForDeletion;
@property(nonatomic) BOOL isPlainText;
- (id)contentAsPlainTextPreservingNewlines;
@property(readonly, nonatomic) NSString *contentAsPlainText;
@property(retain, nonatomic) NSString *content;

// Remaining properties
@property(retain, nonatomic) NSSet *attachments; // @dynamic attachments;
@property(retain, nonatomic) NSString *author; // @dynamic author;
@property(retain, nonatomic) NoteBodyObject *body; // @dynamic body;
@property(retain, nonatomic) NSNumber *containsCJK; // @dynamic containsCJK;
@property(retain, nonatomic) NSNumber *contentType; // @dynamic contentType;
@property(retain, nonatomic) NSDate *creationDate; // @dynamic creationDate;
@property(retain, nonatomic) NSNumber *deletedFlag; // @dynamic deletedFlag;
@property(retain, nonatomic) NSNumber *externalFlags; // @dynamic externalFlags;
@property(retain, nonatomic) NSNumber *externalSequenceNumber; // @dynamic
externalSequenceNumber;
@property(retain, nonatomic) NSNumber *externalServerIntId; // @dynamic
externalServerIntId;
@property(readonly, retain, nonatomic) NSString *guid; // @dynamic guid;
@property(retain, nonatomic) NSNumber *integerId; // @dynamic integerId;
@property(retain, nonatomic) NSNumber *isBookkeepingEntry; // @dynamic
isBookkeepingEntry;
@property(retain, nonatomic) NSDate *modificationDate; // @dynamic modificationDate;
```

```

@property(retain, nonatomic) NSString *serverId; // @dynamic serverId;
@property(retain, nonatomic) NoteStoreObject *store; // @dynamic store;
@property(retain, nonatomic) NSString *summary; // @dynamic summary;
@property(retain, nonatomic) NSString *title; // @dynamic title;

@end

```

Great! Lots of properties indicate that NoteObject is a very standard model. How do we get its text? Among its properties, we can see a possible property named `contentAsPlainText`. Let's check what it is:

```

cy# [#0x176aa170 contentAsPlainText]
@"Secret"

```

For further confirmation, let's change the text of this note and add a picture, as shown in figure 7-8.



Figure 7- 8 Change this note

Then call `contentAsPlainText` again:

```

cy# [#0x176aa170 contentAsPlainText]
@"bbs.iosre.com"

```

Now we're certain that this method can correctly return the text of the current note. With a further length method, we're able to get the character count of this note:

```

cy# [[#0x176aa170 contentAsPlainText] length]
13

```

We're almost done.

7.2.5 Find a method to monitor note text changes in real time

At the beginning of this chapter we've mentioned that "this kind of methods are usually defined in protocols". Because both setTitle: and NoteObject are found in NotesDisplayController.h, if we can find the "monitor" method inside this header too, our code will be greatly simplified. Open NotesDisplayController.h and check what protocols it has implemented.

```
@interface NotesDisplayController : UIViewController <NoteContentLayerDelegate,
UISheetDelegate, AFContextProvider, UIPopoverPresentationControllerDelegate,
UINavigationControllerDelegate, UIImagePickerControllerDelegate,
NotesQuickLookActivityItemDelegate, UIScrollViewKeyboardResizerDelegate,
NSUserActivityDelegate, NotesStateArchiving>
.....
@end
```

Among those protocols, UISheetDelegate, UIPopoverPresentationControllerDelegate, UINavigationControllerDelegate and UIImagePickerControllerDelegate are all documented, they have nothing to do with the changes of the current note, hence can be ignored. The remaining ones, i.e. NoteContentLayerDelegate, AFContextProvider, NotesQuickLookActivityItemDelegate, UIScrollViewKeyboardResizerDelegate, NSUserActivityDelegate and NotesStateArchiving are worth attention, we should inspect them one by one. Let's start with NoteContentLayerDelegate-Protocol.h:

```
@protocol NoteContentLayerDelegate <NSObject>
- (BOOL)allowsAttachmentsInNoteContentLayer:(id)arg1;
- (BOOL)canInsertImagesInNoteContentLayer:(id)arg1;
- (void)insertImageInNoteContentLayer:(id)arg1;
- (BOOL)isNoteContentLayerVisible:(id)arg1;
- (BOOL)noteContentLayer:(id)arg1 acceptContentsFromPasteboard:(id)arg2;
- (BOOL)noteContentLayer:(id)arg1 acceptStringIncreasingContentLength:(id)arg2;
- (BOOL)noteContentLayer:(id)arg1 canHandleLongPressOnElement:(id)arg2;
- (void)noteContentLayer:(id)arg1 containsCJK:(BOOL)arg2;
- (void)noteContentLayer:(id)arg1 contentScrollViewWillBeginDragging:(id)arg2;
- (void)noteContentLayer:(id)arg1 didChangeContentSize:(struct CGSize)arg2;
- (void)noteContentLayer:(id)arg1 handleLongPressOnElement:(id)arg2 atPoint:(struct
CGPoint)arg3;
- (void)noteContentLayer:(id)arg1 setEditing:(BOOL)arg2 animated:(BOOL)arg3;
- (void)noteContentLayerContentDidChange:(id)arg1 updatedTitle:(BOOL)arg2;
- (BOOL)noteContentLayerShouldBeginEditing:(id)arg1;

@optional
- (void)noteContentLayerKeyboardDidHide:(id)arg1;
@end
```

2 methods are quite suspicious, they're noteContentLayer:didChangeContentSize: and noteContentLayerContentDidChange:updatedTitle:. While we are editing a note, the content

and size of it are indeed changing, thus those 2 methods may be called when changes occur, and actually they're implemented in NotesDisplayController.h. Let's use LLDB to make sure they're called when a note changes.

Attach to MobileNotes with LLDB, and check its ASLR offset:

```
(lldb) image list -o -f
[ 0] 0x00035000
/private/var/db/stash/_.29LMeZ/Applications/MobileNotes.app/MobileNotes(0x0000000000039000)
[ 1] 0x00197000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000197000)
[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/Frameworks/QuickLook.framework/QuickLook
.....
```

The ASLR offset is 0x35000. Drag and drop MobileNotes into IDA, then check the base addresses of [NotesDisplayController noteContentLayer:didChangeContentSize:] and [NotesDisplayController noteContentLayerContentDidChange:updatedTitle:] after the initial analysis, as shown in figure 7-9 and figure 7-10.

```
text:00016E70 ; NotesDisplayController - (void)noteContentLayer:(id) didChangeContentSize:(struct
text:00016E70
text:00016E70 ; void __cdecl -[NotesDisplayController noteContentLayer:didChangeContentSize:](str
text:00016E70 __NotesDisplayController_noteContentLayer_didChangeContentSize
text:00016E70 ; DATA XREF: __objc_const:0004C680j0
text:00016E70 MOV R1, #(selRef_reloadSearchedTermHighlight - 0x16E7C
text:00016E78 ADD R1, PC ; selRef_reloadSearchedTermHighlight
text:00016E7A LDR R1, [R1] ; "reloadSearchedTermHighlight"
text:00016E7C B.W j__objc_msgSend
text:00016E7C ; End of function -[NotesDisplayController noteContentLayer:didChangeContentSize:]
```

Figure 7- 9 [NotesDisplayController noteContentLayer:didChangeContentSize:]

```
text:0001AEB8 ; NotesDisplayController - (void)noteContentLayerContentDidChange:(id) updatedTitle:(char)
text:0001AEB8 ; Attributes: bp-based frame
text:0001AEB8
text:0001AEB8 ; void __cdecl -[NotesDisplayController noteContentLayerContentDidChange:updatedTitle:](str
text:0001AEB8 __NotesDisplayController_noteContentLayerContentDidChange_updatedTitle
text:0001AEB8 ; DATA XREF: __objc_const:0004C614j0
text:0001AEB8 var_1C = -0x1C
text:0001AEB8
text:0001AEB8 PUSH {R4-R7,LR}
text:0001AEB8 ADD R7, SP, #0xC
text:0001AEB8 PUSH.W {R8,R10,R11}
text:0001AEC0 SUB SP, SP, #4
text:0001AEC2 MOV R4, R0
```

Figure 7- 10 [NotesDisplayController noteContentLayerContentDidChange:updatedTitle:]

The base addresses are 0x16E70 and 0x1AEB8 respectively, so breakpoints should be set at 0x4BE70 and 0x4FEB8. Then try to edit a note and see whether these breakpoints are triggered:

```
(lldb) br s -a 0x4BE70
Breakpoint 1: where = MobileNotes`___lldb_unnamed_function382$$MobileNotes, address = 0x0004be70
(lldb) br s -a 0x4FEB8
Breakpoint 2: where = MobileNotes`___lldb_unnamed_function458$$MobileNotes, address = 0x0004feb8
```

Great eyes see alike: These two breakpoints are hit a lot! The reason a protocol method gets called is generally that the corresponding event mentioned in the method name happened. And

what triggers the event is usually the method's arguments. In this case, [NotesDisplayController noteContentLayer:didChangeContentSize:] and [NotesDisplayController noteContentLayerContentDidChange:updatedTitle:] are called because didChangeContentSize and ContentDidChange events happened, and content itself is probably the arguments of both methods. Let's verify our guess in LLDB.

```
(lldb) br com add 1
Enter your debugger command(s). Type 'DONE' to end.
> po $r2
> c
> DONE
(lldb) br com add 2
Enter your debugger command(s). Type 'DONE' to end.
> po $r2
> c
> DONE
(lldb) c
```

We can see quite a few occurrences of NoteContentLayer:

```
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568); animations =
{ bounds.origin=<CABasicAnimation: 0x16fee090>; bounds.size=<CABasicAnimation:
0x16fee4a0>; position=<CABasicAnimation: 0x16fee500>; }; layer = <CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568); animations =
{ bounds.origin=<CABasicAnimation: 0x16fee090>; bounds.size=<CABasicAnimation:
0x16fee4a0>; position=<CABasicAnimation: 0x16fee500>; }; layer = <CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568); layer = <CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
```

If NoteContentLayer comes, can NoteContent be far behind? Let's search in NoteContentLayer.h for NoteContent:

```
@interface NoteContentLayer : UIView <NoteTextViewActionDelegate,
NoteTextViewLayoutDelegate, UITextViewDelegate>
.....
@property(retain, nonatomic) NoteTextView *textView; // @synthesize textView=_textView;
.....
@end
```

There's a property of NoteTextView type in NoteContentLayer. In the beginning of this chapter, we have printed the view hierarchy of note browsing view in Cypriat, and found the note text was displayed right on a NoteTextView. So, let's change the commands on the breakpoints and print NoteTextView:

```
(lldb) br com add 1
Enter your debugger command(s). Type 'DONE' to end.
> po [$r2 textView]
```



```

> c
> DONE
(lldb) br com add 2
Enter your debugger command(s). Type 'DONE' to end.
> po [$r2 textView]
> c
> DONE

```

Continue editing this note and keep watching the output. Our editing shows in the output in real time:

```

Process 24577 resuming
Command #2 'c' continued the target.
<NoteTextView: 0x15aace00; baseClass = _UICompatibilityTextView; frame = (6 28; 308 209); text = 'Secre'; clipsToBounds = YES; gestureRecognizers = <NSArray: 0x14eddfc0>; layer = <CALayer: 0x14ee7da0>; contentOffset: {0, 0}; contentSize: {308, 52}>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteTextView: 0x15aace00; baseClass = _UICompatibilityTextView; frame = (6 28; 308 209); text = 'Secret'; clipsToBounds = YES; gestureRecognizers = <NSArray: 0x14eddfc0>; layer = <CALayer: 0x14ee7da0>; contentOffset: {0, 0}; contentSize: {308, 52}>

```

One last step is to get “text” from NoteTextView. Open NoteTextView.h:

```

@interface NoteTextView : _UICompatibilityTextView <UIGestureRecognizerDelegate>
{
    id <NoteTextViewActionDelegate> _actionDelegate;
    id <NoteTextViewLayoutDelegate> _layoutDelegate;
    .....
}
.....
@property(n nonatomic) __weak id <NoteTextViewActionDelegate> actionDelegate; //
@synthesize actionDelegate=_actionDelegate;
.....
@property(n nonatomic) __weak id <NoteTextViewLayoutDelegate> layoutDelegate; //
@synthesize layoutDelegate=_layoutDelegate;
.....
@end

```

There’s not much content in this header, and there’re only 2 delegates with the keyword “text”. Obviously, delegates don’t return NSString objects. If we cannot get text in NoteTextView, it gets to be in NoteTextView’s super class. Open _UICompatibilityTextView then:

```

@interface _UICompatibilityTextView : UIScrollView <UITextLinkInteraction, UITextInput>
.....
@property(n nonatomic) int textAlignment;
@property(copy, nonatomic) NSString *text;
- (BOOL)hasText;
@property(retain, nonatomic) UIColor *textColor;
@property(retain, nonatomic) UIFont *font;
@property(copy, nonatomic) NSAttributedString *attributedText;
.....

```

OK, here comes NSString *text. Let’s use LLDB for a final confirmation:

```

(lldb) br com add 1
Enter your debugger command(s). Type 'DONE' to end.

```

```

> po [[$r2 textView] text]
> c
> DONE
(lldb) br com add 2
Enter your debugger command(s). Type 'DONE' to end.
> po [[$r2 textView] text]
> c
> DONE
Secret
Process 24577 resuming
Command #2 'c' continued the target.
Secret i
Process 24577 resuming
Command #2 'c' continued the target.

```

By now, we've successfully found 2 methods to monitor note text changes in real time, you can choose either of them, and `[NotesDisplayController noteContentLayerContentDidChange:updatedTitle:]` is my choice. All 3 previous problems are solved, iOS reverse engineering is way easier than you originally thought, isn't it?

7.3 Result interpretation

The mission of this chapter is to reverse a stock App, Notes. We've successfully prototyped the tweak with only Cycrypt and LLDB, and actually we can replace LLDB with Theos too. You may call it luck and it's true that reverse engineering depends on fortune. To rewrite Characount for Notes 8, the general thoughts are as follows.

1. Find a proper location on UI and a method to display the character count
Upgrading from iOS 6 to iOS 8 eliminates Notes' title, where is a good place to display the character count. In this chapter, we've cut into the code from the note browsing view and got `NoteDisplayController` with Cycrypt, therefore managed to solve the 1st problem.

2. Browse the class-dump headers and find methods in controller to access model

Accessing model via controller conforms to MVC design pattern, which Apple made Apps should apply. Therefore, `NoteDisplayController` should be able to access note objects. By just looking through headers and examining some suspicious properties with Cycrypt, we've got `NoteObject`, thus got the character count of a note.

3. Find protocol methods to monitor note text changes in real time

Event related methods with keywords like “did” or “will” are often defined in protocols.

Due to the high readability of Objective-C methods’ names, we didn’t use IDA or LLDB to find methods that meet our needs, but instead went over all headers with the keyword “protocol”.

With a 1st round filtering by header names and a 2nd round filtering by LLDB, we’ve found our target methods. This is the charm of reverse engineering, regardless of fortune or guess.

7.4 Tweak writing

This example is relatively easy, all operations can be done inside the class NotesDisplayController.

7.4.1 Create tweak project “CharacountforNotes8” using Theos

The Theos commands are as follows:

```
snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/cydyget
[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): CharacountForNotes8
Package Name [com.yourcompany.characountfornotes8]: com.naken.characountfornotes8
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.mobilenotes
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: MobileNotes
Instantiating iphone/tweak in characountfornotes8/...
Done.
```

7.4.2 Compose CharacountForNotes8.h

The finalized CharacountForNotes8.h looks like this:

```
@interface NoteObject : NSObject
@property (readonly, nonatomic) NSString *contentAsPlainText;
@end

@interface NoteTextView : UIView
@property (copy, nonatomic) NSString *text;
@end
```

```

@interface NoteContentLayer : UIView
@property (retain, nonatomic) NoteTextView *textView;
@end

@interface NotesDisplayController : UIViewController
@property (retain, nonatomic) NoteContentLayer *contentLayer;
@property (retain, nonatomic) NoteObject *note;
@end

```

This header is composed by picking snippets from other class-dump headers. The existence of this header is simply for avoiding any warnings or errors when compiling the tweak.

7.4.3 Edit Tweak.xml

The finalized Tweak.xml looks like this:

```

#import "CharacountForNotes8.h"

%hook NotesDisplayController
- (void)viewWillAppear:(BOOL)arg1 // Initialize title
{
    %orig;
    NSString *content = self.note.contentAsPlainText;
    NSString *contentLength = [NSString stringWithFormat:@"%lu", (unsigned
long)[content length]];
    self.title = contentLength;
}

- (void)viewDidDisappear:(BOOL)arg1 // Reset title
{
    %orig;
    self.title = nil;
}

- (void)noteContentLayerContentDidChange:(NoteContentLayer *)arg1
updatedTitle:(BOOL)arg2 // Update title
{
    %orig;
    NSString *content = self.contentLayer.textView.text;
    NSString *contentLength = [NSString stringWithFormat:@"%lu", (unsigned
long)[content length]];
    self.title = contentLength;
}
%end

```

7.4.4 Edit Makefile and control files

The finalized 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 = CharacountForNotes8
CharacountForNotes8_FILES = Tweak.xml

```

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

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

The finalized control looks like this:

```
Package: com.naken.characountfornotes8
Name: CharacountForNotes8
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Add a character count to Notes
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

7.4.5 Test

After packaging and installing Characount for Notes 8, let's test it by editing a random note and see if the character count changes in real time, as shown in figure 7-11 to figure 7-17.

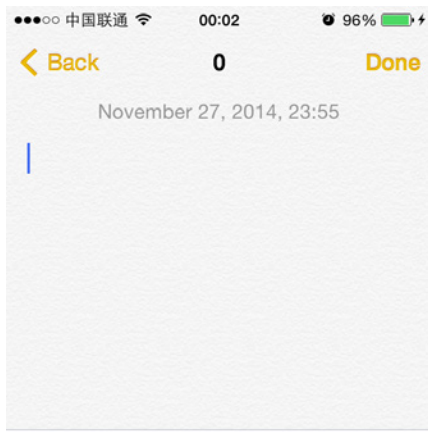


Figure 7- 11 Characount for Notes 8

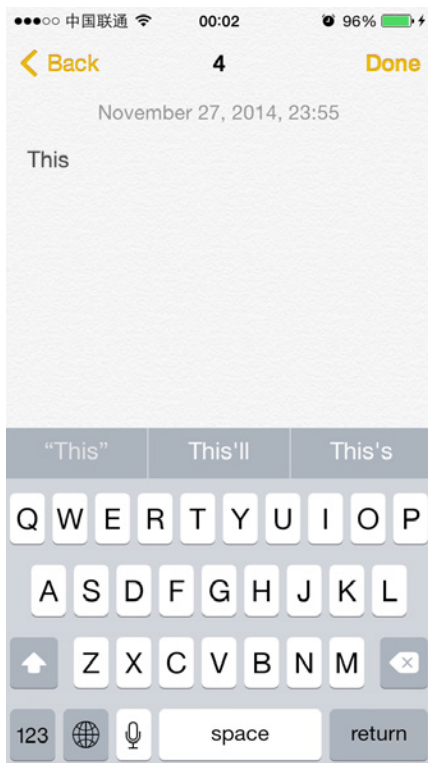


Figure 7- 12 Characount for Notes 8

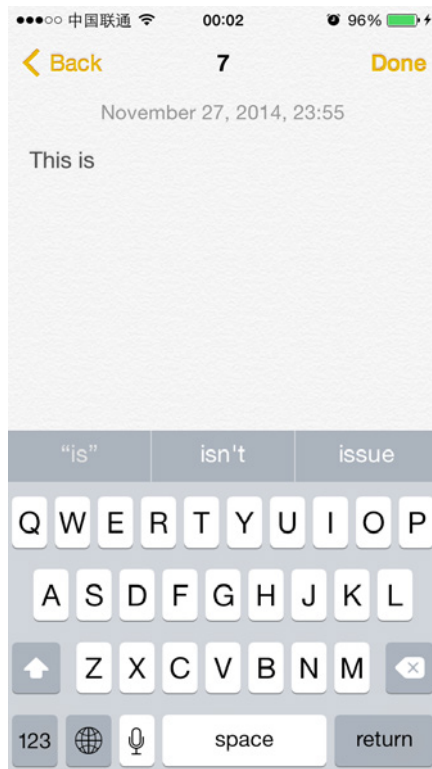


Figure 7- 13 Characount for Notes 8

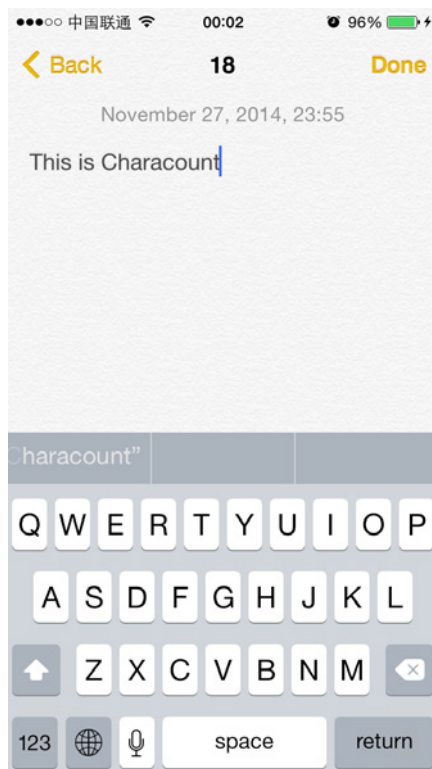


Figure 7- 14 Characount for Notes 8

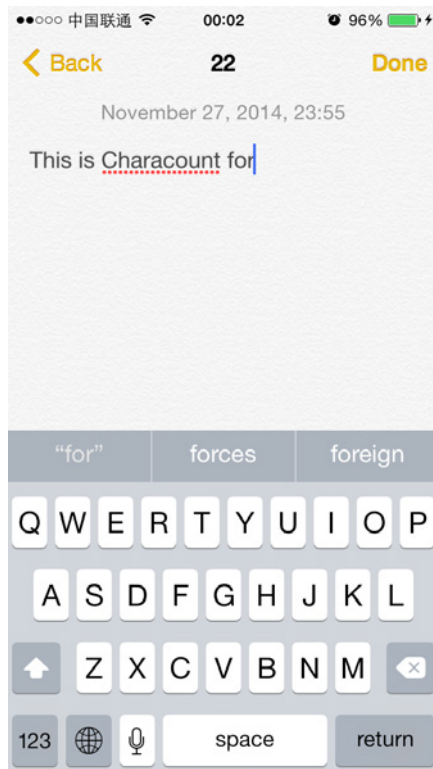


Figure 7- 15 Characount for Notes 8

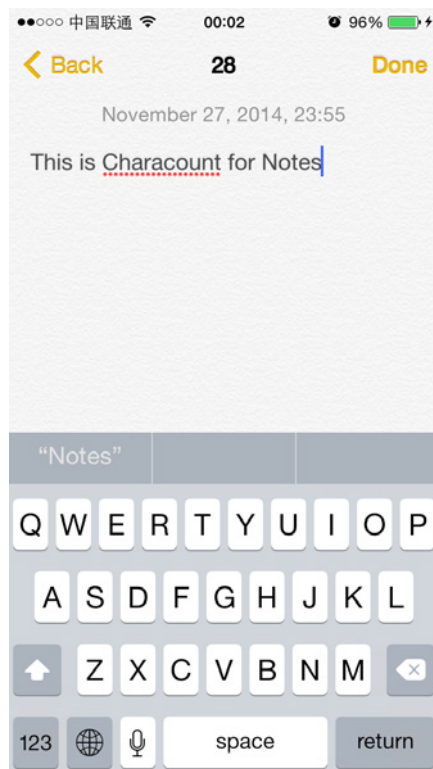


Figure 7- 16 Characount for Notes 8

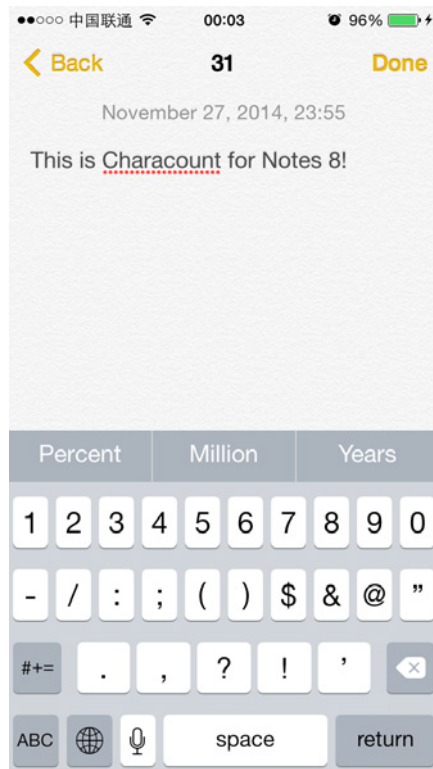


Figure 7- 17 Characount for Notes 8

It works as we expected.

7.5 Conclusion

As a veteran on iOS, Notes is simple yet popular, a great number of people use this App frequently in their daily lives. Characount for Notes 8 is so simple that we don't even need advanced reverse engineering tools to finish the whole project, I hope you don't have difficulty reading this chapter. It's energy-and-time-consuming to learn assembly-level reverse engineering when you are not familiar with IDA and LLDB, I suggest beginners carry out some simple reverse engineering projects just like the example in this chapter first. In this way, not only can you form a thinking pattern of reverse engineering, but also gain a sense of achievement, so why not get your hands dirty right now?

8.1 Mail

Email is one of the most popular communication channels in the era of Internet. Many people send and receive emails every day. Although there are lots of good email Apps on AppStore, such as Sparrow, Inbox, etc, they are not as highly integrated as the stock Mail App (hereafter referred to as Mail). Therefore, Mail is still the top choice during my daily life.

Among all emails we receive every day, most of them are valueless subscription emails like notifications and advertisements, which comes from our inadvertently clicks of subscriptions on various websites, as shown in figure 8-1.



Figure 8- 1 Mail

These emails always make me entangled. If we are kind enough to not think of them as spam messages, they are actually distracting our attention. However, if we mark them as spam

messages, we may miss some useful information. So how to deal with these messages can be a real headache. I have an idea that we can add a whitelist feature to Mail, which saves our frequent contacts. Other emails outside whitelist will be marked as read automatically. With this solution, we can highlight the most valuable messages while not missing any useful information, as shown in figure 8-2.



Figure 8- 2 Mark messages outside whitelist as read

Our task for this chapter is to finish this tweak. We can divide the task into following 2 steps.

- Add a button on the Mail UI and present an editable whitelist after pressing the button in order to add or delete entries in whitelist.
- Every time the inbox get refreshed, mark all emails outside whitelist as read.

Simple and clear, let's get started. All operations in this chapter are carried out on iPhone 5, iOS 8.1.1.

8.2 Tweak prototyping

The initial view of Mail is shown in figure 8-3.

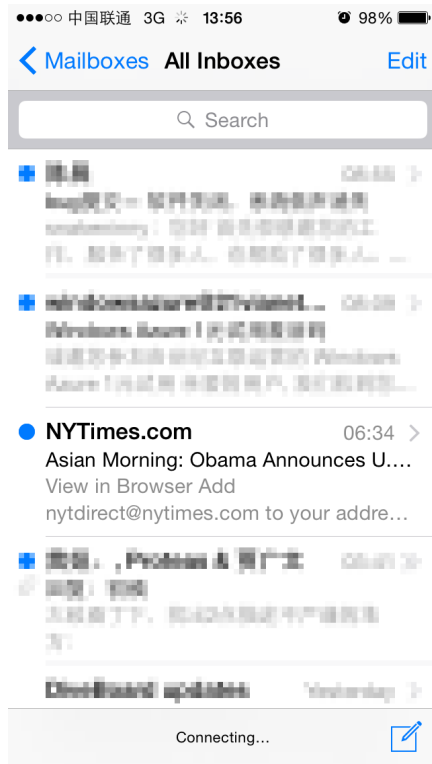


Figure 8- 3 Initial view of Mail

Where should we place the whitelist button for a better user-experience? In the “All Inboxes” view in figure 8-3, we can see that the left bottom corner is blank; maybe we can put the button here. Let’s try it out and the effect is shown in figure 8-4.

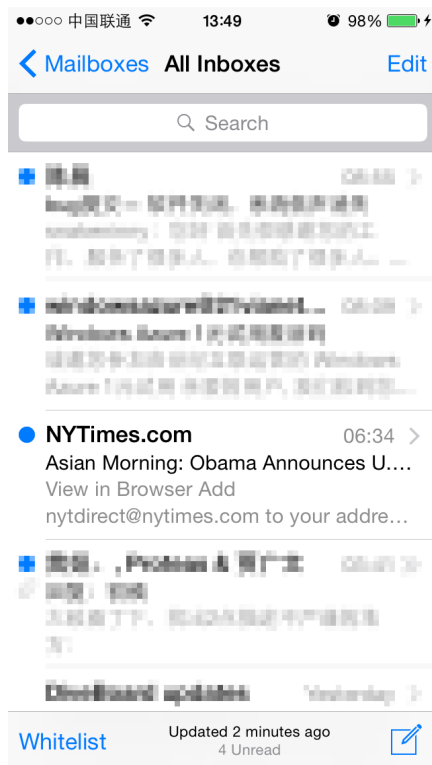


Figure 8- 4 Add whitelist button at the left bottom corner

Although the whitelist button is aligned with the compose button in right bottom corner,

the former is text and the latter is an icon. They are in different forms and looks inharmonious. Therefore, we can see the left bottom corner is not suitable for text button. How about changing it to an icon? The problem is that there isn't an accustomed icon to represent whitelist, while a random one may cause confusion. So in this view, no matter icon or text we use, we cannot get both understandability and harmony. Let's click "Mailboxes" and go to the upper view, as shown in figure 8-5.

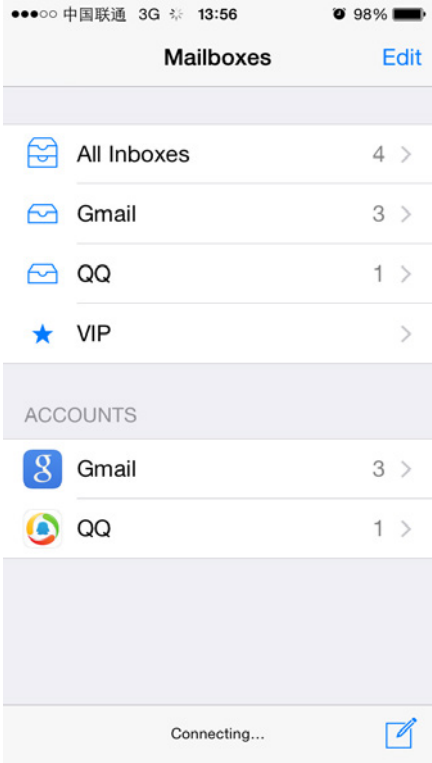


Figure 8- 5 Mailboxes

The top left and bottom left areas are both empty, as shown in figure 8-5. The bottom left is not suitable for the whitelist button as we've discussed just now. So let's put the button on top left corner to see how it looks, as shown in figure 8-6.

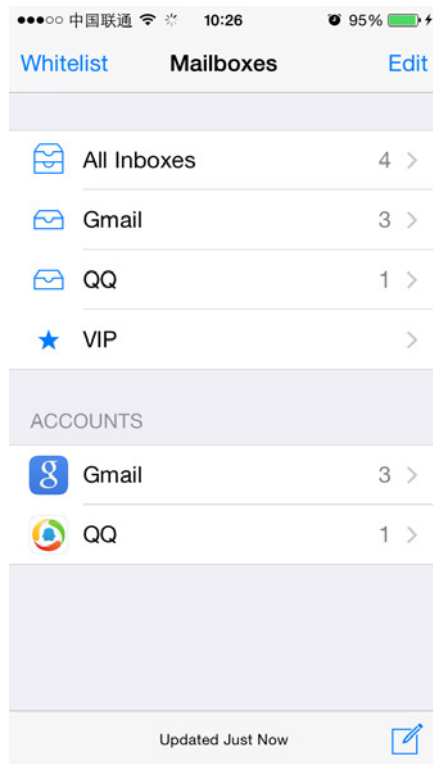


Figure 8- 6 Add whitelist button at top left corner

Not bad, this is it. To customize the view like figure 8-6, we just need to find the controller of “Mailboxes” view and then add the button by calling `[controller.navigationItem setLeftBarButtonItem:]`. We have repeated the process of finding C from V for many times previously and it has been proved as a feasible solution. After we know how to add the button, we can try to implement the function of whitelist. It can be divided into three steps.

- 1) Get all emails.
- 2) Extract their senders' addresses.
- 3) Mark them as read according to whitelist.

Let's analyze them step by step, hope you can still catch up.

How can we get all emails? As we know, we can pull to refresh the inbox, as shown in figure 8-7.

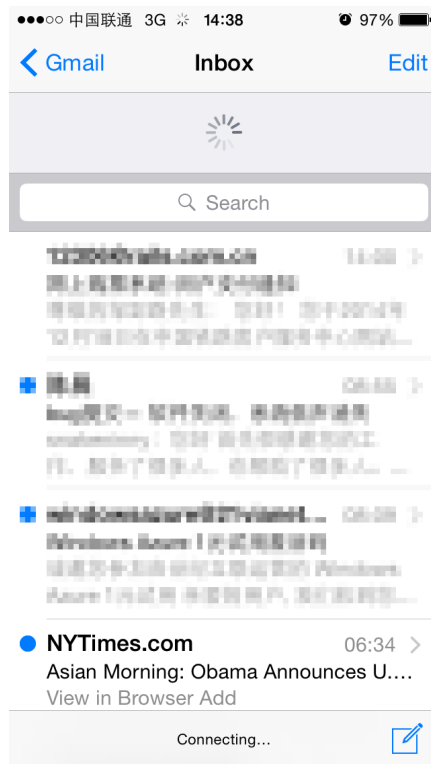


Figure 8-7 Pull to refresh

During refreshing, Mail will fetch all latest emails from mail servers. After refreshing, the UI will restore to the normal state as shown in figure 8-3, and at this moment, we've got all emails. As long as we can catch the refresh completion event and read the inbox after that, we can get all emails. Therefore, we can divide "getting all emails" into 2 steps: first, try to capture the refresh completion event; second, read the inbox. Normally, the refresh completion event handler would be a callback method in some protocols. So when analyzing the class-dump headers, we should pay attention to whether there are protocol methods with keywords like "didRefresh", "didUpdate" or "didReload" in their names. By hooking such methods and read the inbox after their execution, we'll be able to get all emails.

An email is an object and it is generally abstracted as a class. From this class, we can extract information like the receiver, sender, title, content and whether it is read. If we can get this object, we can finish the second and third step together.

The overall ideas are not complicated, let's realize them one by one.

8.2.1 Locate and class-dump Mail's executable

We can easily locate the executable of Mail, `"/Applications/MobileMail.app/MobileMail"`, using "ps". Since Mail is a stock App on iOS, it is not encrypted and we can class-dump it directly without decryption:

```
snakeninnys-MacBook:~ snakeninny$ class-dump -S -s -H
/Users/snakeninny/Code/iOSSystemBinaries/8.1.1_iPhone5/MobileMail.app/MobileMail -o
/Users/snakeninny/Code/iOSPrivateHeaders/8.1.1/MobileMail
```

There're 393 headers in total, as shown in figure 8-8.

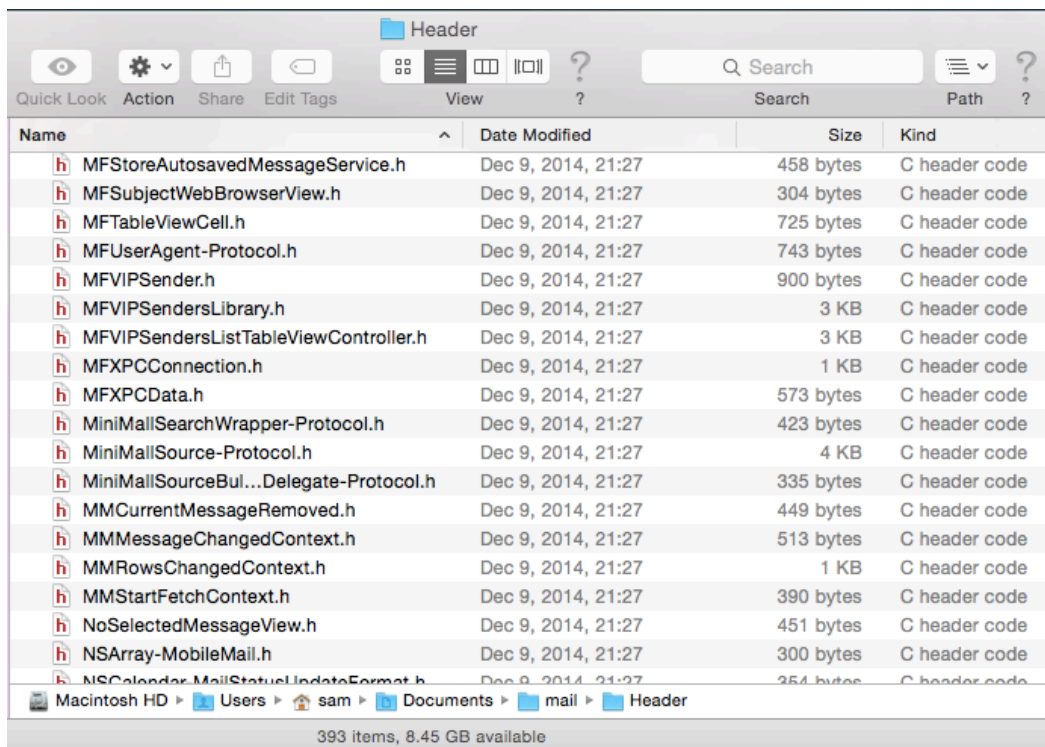


Figure 8- 7 class-dump headers

8.2.2 Import headers into Xcode

The search and code highlighting features in Xcode are competent to present lots of headers, as shown in figure 8-9.

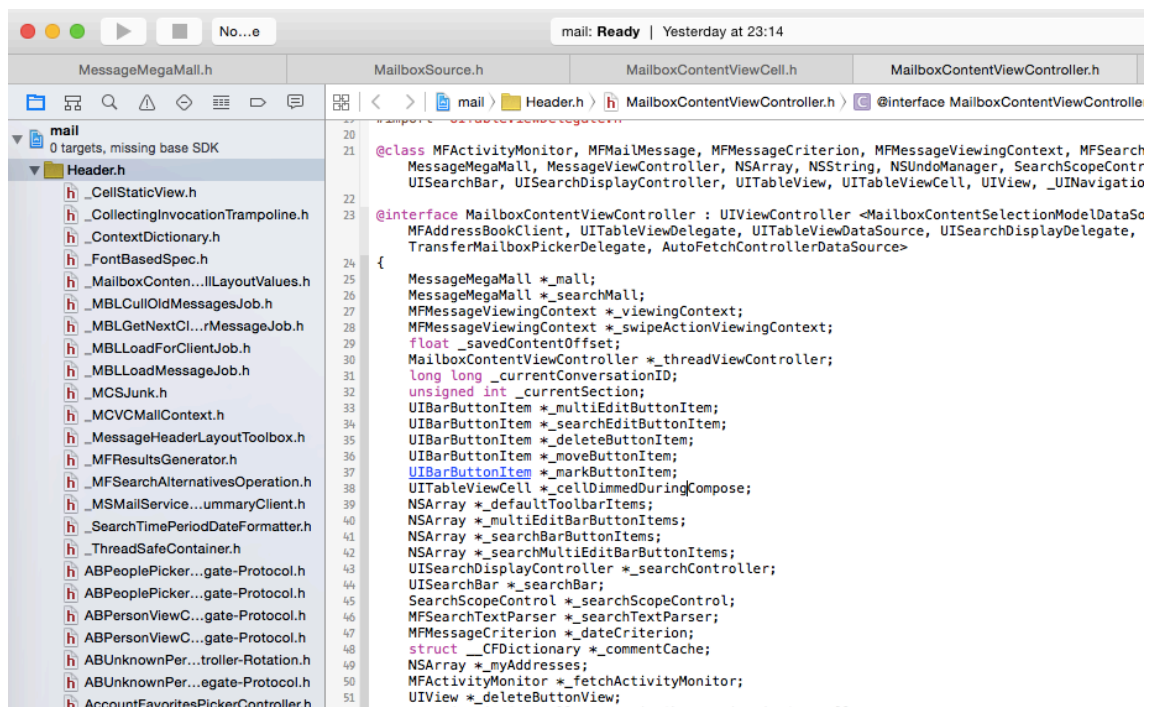


Figure 8- 8 Import headers into Xcode

Next, let's start to find the point to cut into code from UI.

8.2.3 Find the controller of “Mailboxes” view using Cypcript

Firstly, use recursiveDescription to print out the view hierarchy of “Mailboxes” view, as shown below:

```

FunMaker-5:~ root# cypcript -p MobileMail
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<UIWindow: 0x156bffe0; frame = (0 0; 320 568); gestureRecognizers = <NSArray:
0x156bd390>; layer = <UIWindowLayer: 0x156c1be0>>
  | <UIView: 0x15611490; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers =
<NSArray: 0x15618e70>; layer = <CALayer: 0x15611420>>
    | | <UIView: 0x15611210; frame = (0 0; 320 568); layer = <CALayer: 0x15611280>>
      | | | <_MFActorItemView: 0x15614660; frame = (0 0; 320 568); layer = <CALayer:
0x15614840>>
        | | | | <UIView: 0x156150f0; frame = (-0.5 -0.5; 321 569); alpha = 0; layer
= <CALayer: 0x15615160>>
          | | | | | <_MFActorSnapshotView: 0x15614bb0; baseClass = UISnapshotView; frame
= (0 0; 320 568); clipsToBounds = YES; hidden = YES; layer = <CALayer: 0x15614e00>>
            | | | | | <UIView: 0x15614f40; frame = (-1 -1; 322 570); layer =
<CALayer: 0x15614fb0>>
              | | | | | <UILayoutContainerView: 0x1572ec40; frame = (0 0; 320 568);
clipsToBounds = YES; autoresize = LM+W+RM+TM+H+BM; layer = <CALayer: 0x1572ecc0>>
                | | | | | <UIView: 0x1683d890; frame = (0 0; 320 0); layer = <CALayer:
0x16848140>>
                  | | | | | <UILayoutContainerView: 0x157246b0; frame = (0 0; 320 568);
clipsToBounds = YES; gestureRecognizers = <NSArray: 0x156088e0>; layer = <CALayer:
0x15724890>>
.....

```

```

| | | | | | | | | | | <MailboxTableCell: 0x1572ad50;
baseClass = UITableViewCell; frame = (0 28; 320 44.5); autoresize = W; layer = <CALayer:
0x168299f0>>
| | | | | | | | | | | | <UITableViewCellContentView:
0x16829b70; frame = (0 0; 286 44); gestureRecognizers = <NSArray: 0x1682b060>; layer =
<CALayer: 0x16829be0>>
| | | | | | | | | | | | | <UILabel: 0x1682b0a0; frame
= (55 12; 84.5 20.5); text = 'All Inboxes'; userInteractionEnabled = NO; layer =
<UILabelLayer: 0x1682b160>>
.....

```

The text of the UILabel at the bottom is “All Inboxes”, indicating its corresponding MailBoxTableCell is the top one in figure 8-5. Keep calling nextResponder until we get the controller:

```

cy# [#0x1572ad50 nextResponder]
#"<UITableViewWrapperView: 0x1572fe60; frame = (0 0; 320 568); gestureRecognizers =
<NSArray: 0x15730370>; layer = <CALayer: 0x157301a0>; contentOffset: {0, 0};
contentSize: {320, 568}>"
cy# [#0x1572fe60 nextResponder]
#"<UITableView: 0x1585a000; frame = (0 0; 320 568); clipsToBounds = YES; autoresize =
W+H; gestureRecognizers = <NSArray: 0x1572fa20>; layer = <CALayer: 0x1572f540>;
contentOffset: {0, -64}; contentSize: {320, 371}>"
cy# [#0x1585a000 nextResponder]
#"<MailboxPickerController: 0x156e9260>"

```

Aha. It's very easy to get MailboxPickerController. Let's try whether we can add a leftBarButtonItem:

```

cy# #0x156e9260.navigationItem.leftBarButtonItem =
#0x156e9260.navigationItem.rightBarButtonItem
#"<UIBarButtonItem: 0x15729f00>"

```

The effect is shown in figure 8-10.

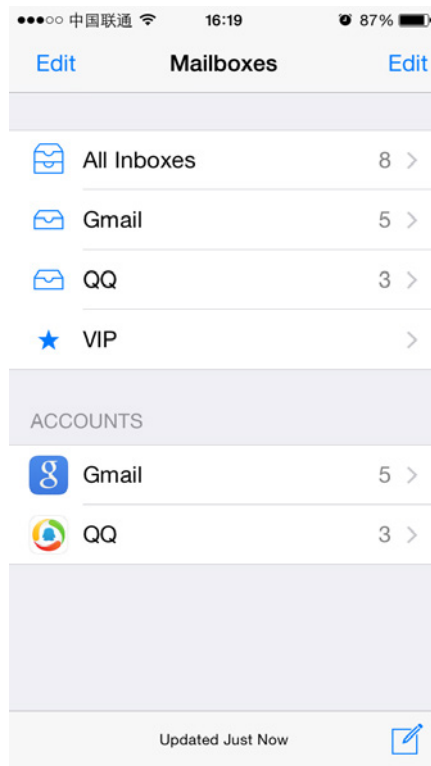


Figure 8- 9 After setLeftBarButtonItem:

No problem! We’ve successfully added the button. Therefore, we can confirm that MailboxPickerController is the controller of “Mailboxes” view.

8.2.4 Find the delegate of “All Inboxes” view using Reveal and Cycrypt

After adding the whitelist button, we need to implement the function of it. First let’s take a look at how to capture the refresh completion event. Since the event is straightly showed on “All Inboxes” view, it is very likely that the callback method is defined in the delegate of this view. Now let’s turn to “All Inboxes” view in figure 8-3 and use Reveal rather than repeating what we’ve done with Cycrypt in section 8.2.3, to locate a cell of this view, and then turn back to Cycrypt to find its associated UITableView as well delegate.

With Reveal, we can easily locate the top cell, as shown in figure 8-11.

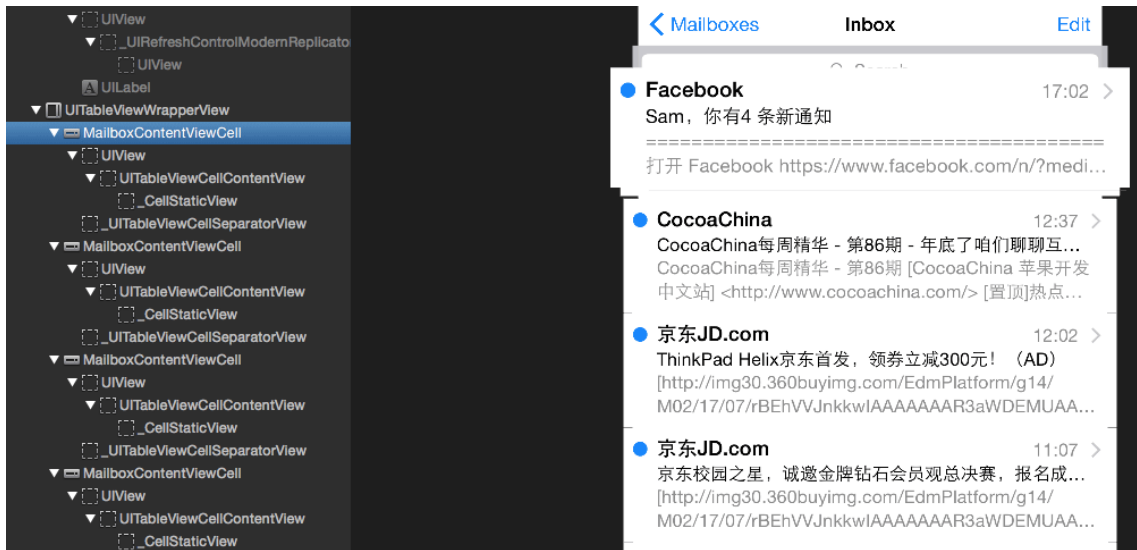


Figure 8- 10 See the view hierarchy using Reveal

MailboxContentTableViewCell is the cell class to show the sender, title and summary of an email. Next, we use Cyscript to find its associated UITableView. Since we know there must be at least one MailboxContentTableViewCell object in current view, we can try to find these cells through command “choose” without using recursiveDescription.

```
FunMaker-5:~ root# cyscript -p MobileMail
cy# choose(MailboxContentTableViewCell)
[#"<MailboxContentTableViewCell: 0x161f4000> cellContent",#"<MailboxContentTableViewCell:
0x1621c400> cellContent",#"<MailboxContentTableViewCell: 0x1621d000>
cellContent",#"<MailboxContentTableViewCell: 0x16234800>
cellContent",#"<MailboxContentTableViewCell: 0x1623ee00>
cellContent",#"<MailboxContentTableViewCell: 0x1623f200>
cellContent",#"<MailboxContentTableViewCell: 0x159c2c00> cellContent"]
```

“choose” has returned an NSArray of MailboxContentTableViewCell objects. Pick anyone and keep calling nextResponder.

```
cy# [choose(MailboxContentTableViewCell)[0] nextResponder]
#"<UITableViewWrapperView: 0x15660b80; frame = (0 0; 320 612); gestureRecognizers =
<NSArray: 0x16855170>; layer = <CALayer: 0x16888f20>; contentOffset: {0, 0};
contentSize: {320, 612}>"
cy# [#0x15660b80 nextResponder]
#"<MFMailboxTableView: 0x16095000; baseClass = UITableView; frame = (0 0; 320 568);
clipsToBounds = YES; autoresize = W+H; gestureRecognizers = <NSArray: 0x15607850>; layer
= <CALayer: 0x16838210>; contentOffset: {0, -64}; contentSize: {320, 52364}>"
```

Its associated UITableView is an MFMailboxTableView object. Let’s take a look at its delegate.

```
cy# [#0x16095000 delegate]
#"<MailboxContentViewController: 0x16106000>"
```

Its delegate is MailboxContentViewController. Keep calling nextResponder and find what its controller is.

```
cy# [#0x16095000 nextResponder]
```

```
#" <MailboxContentViewController: 0x16106000>"
```

From the output, we can see that both the controller and delegate of MFMailboxTableView are MailboxContentViewController. Let's validate the controller as below.

```
cy# [#0x16106000 setTitle:@"iOSRE"]
```

The effect is shown in 8-12.



Figure 8- 11 After setTitle:

So far, we can confirm that our deduction is correct. Playing 2 important roles at the same time, it is very likely that we can find both the refresh completion event handler and inbox reading method in MailboxContentViewController. Let's focus on this class from now on.

8.2.5 Locate the refresh completion callback method in MailboxContentViewController

Like what we did in Chapter 7, we should take a look at what protocol does MailboxContentViewController confirm to at first and then try to find our target method.

```
@interface MailboxContentViewController : UIViewController
<MailboxContentSelectionModelDataSource, MFSearchTextParserDelegate,
MessageMegaMailObserver, MFAddressBookClient, MFMailboxTableViewDelegate,
UIPopoverPresentationControllerDelegate, UITableViewDelegate, UITableViewDataSource,
UISearchDisplayDelegate, UISearchBarDelegate, TransferMailboxPickerDelegate,
AutoFetchControllerDataSource>
```

We can exclude MFSearchTextParserDelegate, MFAddressBookClient, UIPopoverPresentationControllerDelegate, UITableViewDelegate, UITableViewDataSource,

UISearchDisplayDelegate and UISearchBarDelegate just by name, because they seemingly have no relation with refresh completion. The rest protocols, MailboxContentSelectionModelDataSource, MessageMegaMallObserver, MFMailboxTableViewDelegate, TransferMailboxPickerDelegate and AutoFetchControllerDataSource are hard to determine by names. Let's check them one by one from MailboxContentSelectionModelDataSource.

```
@protocol MailboxContentSelectionModelDataSource <NSObject>
- (BOOL)selectionModel:(id)arg1 deleteMovesToTrashForTableIndexPath:(id)arg2;
- (void)selectionModel:(id)arg1 getConversationStateAtTableIndexPath:(id)arg2
hasUnread:(char *)arg3 hasUnflagged:(char *)arg4;
- (void)selectionModel:(id)arg1 getSourceStateHasUnread:(char *)arg2 hasUnflagged:(char
*)arg3;
- (id)selectionModel:(id)arg1 indexPathForMessageInfo:(id)arg2;
- (id)selectionModel:(id)arg1 messageInfosAtTableIndexPath:(id)arg2;
- (id)selectionModel:(id)arg1 messagesForMessageInfos:(id)arg2;
- (BOOL)selectionModel:(id)arg1 shouldArchiveByDefaultForTableIndexPath:(id)arg2;
- (id)selectionModel:(id)arg1 sourceForMessageInfo:(id)arg2;
- (BOOL)selectionModel:(id)arg1 supportsArchivingForTableIndexPath:(id)arg2;
- (id)sourcesForSelectionModel:(id)arg1;
@end
```

It looks like the function of this protocol is to read the data source rather than refresh it.

Let's move on to MessageMegaMallObserver, its contents are as below:

```
@protocol MessageMegaMallObserver <NSObject>
- (void)megaMallCurrentMessageRemoved:(id)arg1;
- (void)megaMallDidFinishSearch:(id)arg1;
- (void)megaMallDidLoadMessages:(id)arg1;
- (void)megaMallFinishedFetch:(id)arg1;
- (void)megaMallGrowingMailboxesChanged:(id)arg1;
- (void)megaMallMessageCountChanged:(id)arg1;
- (void)megaMallMessagesAtIndexesChanged:(id)arg1;
- (void)megaMallStartFetch:(id)arg1;
@end
```

There are many perfect tense verbs in the method names. Meanwhile, judging from the name like "LoadMessages", "FinishedFetch" and "MessageCountChanged", we guess that they may get called before or after refresh completion. So let's set breakpoints at the beginning of these three methods using LLDB and pull to refresh the inbox to check if these methods are called. In the first place, attach LLDB to MobileMail and inspect its ASLR offset.

```
(lldb) image list -o -f
[ 0] 0x000b2000
/private/var/db/stash/_.lnBgU8/Applications/MobileMail.app/MobileMail(0x00000000000b6000)
)
[ 1] 0x003b7000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000003b7000)
[ 2] 0x090d1000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/usr/lib/libarchive.2.dylib
[ 3] 0x090c3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1.1
(12B435)/Symbols/System/Library/Frameworks/CloudKit.framework/CloudKit
.....
```


We can see the ASLR offset is 0x000b2000. Then drag and drop MobileMail into IDA and after the initial analysis has been finished, check the base addresses of [MailboxContentViewController megaMallDidLoadMessages:], [MailboxContentViewController megaMallFinishedFetch:] and [MailboxContentViewController megaMallMessageCountChanged:], as shown in figure 8-13, 8-14 and 8-15.

```

__text:0003DCE0 ; MailboxContentViewController - (void)megaMallDidLoadMessages:(id)
__text:0003DCE0 ; Attributes: bp-based frame
__text:0003DCE0
__text:0003DCE0 ; void __cdecl -[MailboxContentViewController megaMallDidLoadMessag
__text:0003DCE0 __MailboxContentViewController_megaMallDidLoadMessages__
__text:0003DCE0 ; DATA XREF: __objc_const:0
__text:0003DCE0
__text:0003DCE0 var_20 = -0x20
__text:0003DCE0 var_1C = -0x1C
__text:0003DCE0
__text:0003DCE0 PUSH {R4-R7,LR}
__text:0003DCE2 ADD R7, SP, #0xC
__text:0003DCE4 PUSH.W {R8,R10,R11}

```

Figure 8- 12 [MailboxContentViewController megaMallDidLoadMessages:]

```

__text:0003D860 ; MailboxContentViewController - (void)megaMallFinishedFetch:(id)
__text:0003D860
__text:0003D860 ; void __cdecl -[MailboxContentViewController megaMallFinishedFet
__text:0003D860 __MailboxContentViewController_megaMallFinishedFetch__
__text:0003D860 ; DATA XREF: __objc_const
__text:0003D860
__text:0003D860 var_20 = -0x20
__text:0003D860 var_1C = -0x1C
__text:0003D860 var_18 = -0x18
__text:0003D860 var_14 = -0x14
__text:0003D860 var_10 = -0x10
__text:0003D860 var_C = -0xC
__text:0003D860
__text:0003D860 PUSH {R7,LR}
__text:0003D862 MOV R7, SP
__text:0003D864 SUB SP, SP, #0x18

```

Figure 8- 13 [MailboxContentViewController megaMallFinishedFetch:]

```

__text:0003DE48 PUSH {R4-R7,LR}
__text:0003DE4A ADD R7, SP, #0xC
__text:0003DE4C PUSH.W {R8,R10,R11}
__text:0003DE50 SUB.W R4, SP, #0x18
__text:0003DE54 BIC.W R4, R4, #0xF
__text:0003DE58 MOV SP, R4
00039E48 0003DE48: -[MailboxContentViewController megaMallMessageCountChanged:]

```

Figure 8- 14 [MailboxContentViewController megaMallMessageCountChanged:]

Their base addresses are 0x3dce0, 0x3d860 and 0x3de48 respectively. Set breakpoints on these addresses with LLDB and refresh the inbox to trigger the breakpoints:

```

(lldb) br s -a '0x000b2000+0x3dce0'
Breakpoint 1: where = MobileMail`___lldb_unnamed_function992$$MobileMail, address = 0x000efce0
(lldb) br s -a '0x000b2000+0x3d860'
Breakpoint 2: where = MobileMail`___lldb_unnamed_function987$$MobileMail, address = 0x000ef860
(lldb) br s -a '0x000b2000+0x3de48'
Breakpoint 3: where = MobileMail`___lldb_unnamed_function993$$MobileMail, address = 0x000efe48

```

Some of you may meet the same problem as me, which is none of three breakpoints get

triggered. If you have experience in network development, you may have the idea that in order to reduce the burden of servers and save the network traffic of iOS, Mail may not fetch emails remotely on every refresh. If the time interval between two refreshes is very short, it will use cached content as data source of inbox; and as a result, methods in MessageMegaMailObserver will not get called. In order to validate our assumption, send an email to yourself and refresh to check whether breakpoints get triggered:

```
Process 73130 stopped
* thread #44: tid = 0x14c10, 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail, stop reason = breakpoint 2.1
  frame #0: 0x000ef860 MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860: push   {r7, lr}
    0xef862: mov    r7, sp
    0xef864: sub    sp, #24
    0xef866: movw   r1, #44962
(lldb) c
Process 73130 resuming
Process 73130 stopped
* thread #44: tid = 0x14c10, 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail, stop reason = breakpoint 2.1
  frame #0: 0x000ef860 MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860: push   {r7, lr}
    0xef862: mov    r7, sp
    0xef864: sub    sp, #24
    0xef866: movw   r1, #44962
(lldb) c
Process 73130 resuming
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMail.0x157c2d90',
stop reason = breakpoint 3.1
  frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48: push   {r4, r5, r6, r7, lr}
    0xefe4a: add    r7, sp, #12
    0xefe4c: push.w {r8, r10, r11}
    0xefe50: sub.w  r4, sp, #24
(lldb)
Process 73130 resuming
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMail.0x157c2d90',
stop reason = breakpoint 3.1
  frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48: push   {r4, r5, r6, r7, lr}
    0xefe4a: add    r7, sp, #12
    0xefe4c: push.w {r8, r10, r11}
    0xefe50: sub.w  r4, sp, #24
(lldb)
Process 73130 resuming
Process 73130 stopped
```



```

* thread #44: tid = 0x14c10, 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail, stop reason = breakpoint 2.1
  frame #0: 0x000ef860 MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860: push   {r7, lr}
   0xef862: mov    r7, sp
   0xef864: sub    sp, #24
   0xef866: movw   r1, #44962
(lldb) c
Process 73130 resuming

```

As expected, `megaMallFinishedFetch:` and `megaMallMessageCountChanged:` are called alternately. From their names we can see that an email is a message, `megaMallFinishedFetch:` will be called when iOS has fetched emails from servers successfully, and `megaMallMessageCountChanged:` will get called when email count changes, i.e. when we receive or delete emails. These two methods will definitely get called after refreshing; we can choose either one as the refresh completion callback method. We'll take `megaMallMessageCountChanged:` in this chapter and our next task is to find the method for getting all emails.

8.2.6 Get all emails from MessageMegaMall

Do you still remember the saying in chapter 7 that “The reason a protocol method gets called is generally that the corresponding event mentioned in the method name happened. And the thing that triggers the event is usually the method’s arguments”? So let’s delete the first two breakpoints and keep the last one on `megaMallMessageCountChanged:`, and take a look at its argument:

```

Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90',
stop reason = breakpoint 3.1
  frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48: push   {r4, r5, r6, r7, lr}
   0xefe4a: add    r7, sp, #12
   0xefe4c: push.w {r8, r10, r11}
   0xefe50: sub.w  r4, sp, #24
(lldb) po $r2
NSConcreteNotification 0x157e8af0 {name = MegaMallMessageCountChanged; object =
<MessageMegaMall: 0x1576c320>; userInfo = {
    "added-message-infos" = (
        "<MFMessageInfo: 0x157c86d0> uid=1185, conversation=2777228998582613276"
    );
    destination = "{(\n)}";
    inserted = "{(\n    <NSIndexPath: 0x157e8ac0> {length = 2, path = 0 - 0}\n)}";
    relocated = "{(\n)}";
    updated = "{(\n)}";
}}

```

We can see that the argument is an `NSConcreteNotification` object. Checking its header file, we can learn that it inherits from `NSNotification`. Its name is `MegaMallMessageCountChanged`, object is `MessageMegaMall` and `userInfo` is its changelog. The thing that interests us is the name “MegaMall”, which seemingly has nothing to do with emails but is always next to “Message”, so I guess it’s a mega mall for emails instead of merchandises. Let’s see what’s in `MessageMegaMall.h`:

```
@interface MessageMegaMall : NSObject <MessageMiniMallObserver,
MessageSelectionDataSource>
.....
- (id)copyAllMessages;
@property (retain, nonatomic) MFMailMessage *currentMessage;
- (void)loadOlderMessages;
- (unsigned int)localMessageCount;
- (unsigned int)messageCount;
- (void)markAllMessagesAsNotViewed;
- (void)markAllMessagesAsViewed;
- (void)markMessagesAsNotViewed:(id)arg1;
- (void)markMessagesAsViewed:(id)arg1;
.....
@end
```

We’ve got some new clues: `copyAllMessages`, `currentMessage`, `loadOlderMessages`, `localMessageCount`, `messageCount`, `markAllMessagesAsViewed`, etc. From these methods and properties, we can confirm that `MessageMegaMall` is a model class in charge of all emails; a mega mall is a vivid analogy from Apple for its responsibility. So, can we get all emails with `copyAllMessages`? Let’s try it out in LLDB:

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90,
stop reason = breakpoint 3.1
   frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xfe48: push   {r4, r5, r6, r7, lr}
   0xfe4a: add    r7, sp, #12
   0xfe4c: push.w {r8, r10, r11}
   0xfe50: sub.w  r4, sp, #24
(lldb) po [[[$r2 object] copyAllMessages]
{(
  <MFLibraryMessage 0x15612030: library id 89, remote id 13020, 2014-11-25 20:32:16
+0000, 'Cydia/APT(A): LowPowerBanner (1.4.5)'\>,
  <MFLibraryMessage 0x1572ef10: library id 604, remote id 12718, 2014-10-01 21:34:28
+0000, 'Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand
Them'\>,
  <MFLibraryMessage 0x168bd170: library id 906, remote id 13142, 2014-12-17 22:34:30
+0000, 'Asian Morning: Obama Announces U.S. and Cuba Will Resume Relations'\>,
  .....
)}
(lldb) p (int)[[[$r2 object] copyAllMessages] count]
(int) $7 = 580
(lldb) p (int)[[[$r2 object] localMessageCount]
```

```
(int) $8 = 580
(lldb) p (int)[[$r2 object] messageCount]
(int) $0 = 553
(lldb) po [[[$r2 object] copyAllMessages] class]
__NSSetM
```

copyAllMessages has returned an NSSet with 580 MFLibraryMessage objects. There is an email summary in each MFLibraryMessage object and the count of this NSSet is the same to localMessageCount. Actually, 580 is far less than all email count, but this number is reasonable that to save network traffic and local storage, iOS doesn't have to really fetch all emails and store them locally, several hundreds of emails would be enough. If users want to read more, iOS will fetch more with loadOlderMessages. Therefore, copyAllMessages can be considered the right method for getting all emails. Aha, we have achieved our 2nd goal. For the 3rd goal, we should pay attention to [MessageMegaMall markMessagesAsViewed:]. If nothing goes wrong, this is the method for marking emails as read and its argument seems to be an NSArray or NSSet with MFLibraryMessage objects. Is that so? We'll see shortly.

8.2.7 Get sender address from MFLibraryMessage and mark email as read using MessageMegaMall

From the analysis in section 8.2.4, we can see that an email is an MFLibraryMessage object, whose description contains the summary of that email. However, you can't find MFLibraryMessage.h in MobileMail headers. Why? Because MFLibraryMessage originates from an external dylib. Search "MFLibraryMessage" in iOS 8 class-dump headers, you will find it in Messages.framework, as shown in figure 8-16.

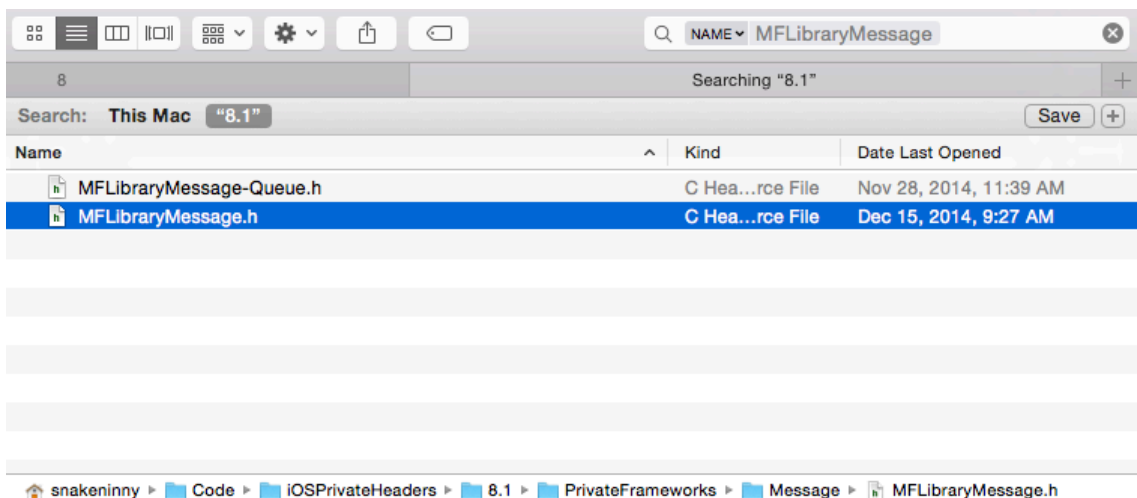


Figure 8- 15 Find MFLibraryMessage

Take a look at MFLibraryMessage.h:

```

@interface MFLibraryMessage : MFMailMessage
.....
- (id)copyMessageInfo;
.....
- (void)markAsNotViewed;
- (void)markAsViewed;
- (id)account;
.....
- (unsigned long long)uniqueRemoteId;
- (unsigned long)uid;
- (unsigned int)hash;
- (id)remoteID;
- (void)_updateUID;
- (unsigned int)messageSize;
- (id)originalMailboxURL;
- (unsigned int)originalMailboxID;
- (unsigned int)mailboxID;
- (unsigned int)libraryID;
- (id)persistentID;
- (id)messageID;
@end

```

In MFLibraryMessage.h, there are various IDs, but our target information seems to be missing. This doesn't make sense: we have already found the email summary in the description of MFLibraryMessage, but haven't found the corresponding methods to read the summary in MFLibraryMessage.h. Therefore, something must be ignored in our analysis. Recheck MFLibraryMessage.h, we notice that there is a method called copyMessageInfo. Let's take a look at it.

```

Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMail.0x157c2d90',
stop reason = breakpoint 3.1
   frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xfe48: push   {r4, r5, r6, r7, lr}
   0xfe4a: add    r7, sp, #12
   0xfe4c: push.w {r8, r10, r11}
   0xfe50: sub.w  r4, sp, #24
(lldb) po [[[$r2 object] copyAllMessages] anyObject] copyMessageInfo]
<MFMessageInfo: 0x157c8040> uid=89, conversation=594030790676622907

```

We've got an object of MFMessageInfo, which has been mentioned in section 8.2.5. Is email summary in MFMessageInfo.h? Let's try it.

```

@interface MFMessageInfo : NSObject
{
    unsigned int _flagged:1;
    unsigned int _read:1;
    unsigned int _deleted:1;
    unsigned int _uidIsLibraryID:1;
    unsigned int _hasAttachments:1;
    unsigned int _isVIP:1;
    unsigned int _uid;
    unsigned int _dateReceivedInterval;
}

```

```

    unsigned int _dateSentInterval;
    unsigned int _mailboxID;
    long long _conversationHash;
    long long _generationNumber;
}

+ (long long)newGenerationNumber;
@property(readonly, nonatomic) long long generationNumber; // @synthesize
generationNumber=_generationNumber;
@property(nonaatomic) unsigned int mailboxID; // @synthesize mailboxID=_mailboxID;
@property(nonaatomic) long long conversationHash; // @synthesize
conversationHash=_conversationHash;
@property(nonaatomic) unsigned int dateSentInterval; // @synthesize
dateSentInterval=_dateSentInterval;
@property(nonaatomic) unsigned int dateReceivedInterval; // @synthesize
dateReceivedInterval=_dateReceivedInterval;
@property(nonaatomic) unsigned int uid; // @synthesize uid=_uid;
- (id)description;
- (unsigned int)hash;
- (BOOL)isEqual:(id)arg1;
- (int)generationCompare:(id)arg1;
- (id)initWithUid:(unsigned int)arg1 mailboxID:(unsigned int)arg2
dateReceivedInterval:(unsigned int)arg3 dateSentInterval:(unsigned int)arg4
conversationHash:(long long)arg5 read:(BOOL)arg6 knownToHaveAttachments:(BOOL)arg7
flagged:(BOOL)arg8 isVIP:(BOOL)arg9;
- (id)init;
@property(nonaatomic) BOOL isVIP;
@property(nonaatomic, getter=isKnownToHaveAttachments) BOOL knownToHaveAttachments;
@property(nonaatomic) BOOL uidIsLibraryID;
@property(nonaatomic) BOOL deleted;
@property(nonaatomic) BOOL flagged;
@property(nonaatomic) BOOL read;

@end

```

MFMessageInfo can tell if an email is read, but it still doesn't contain the summary. Go back to MFLibraryMessage.h again, we see it inherits from MFMailMessage. Judging from its name, MailMessage is certainly more appropriate to represent an email than LibraryMessage. Take a look at MFMailMessage.h:

```

@interface MFMailMessage : MFMessage
.....
- (BOOL)shouldSetSummary;
- (void)setSummary:(id)arg1;
- (void)setSubject:(id)arg1 to:(id)arg2 cc:(id)arg3 bcc:(id)arg4 sender:(id)arg5
dateReceived:(double)arg6 dateSent:(double)arg7 messageIDHash:(long long)arg8
conversationIDHash:(long long)arg9 summary:(id)arg10 withOptions:(unsigned int)arg11;
- (id)subject;
@end

```

summary, subject, sender, cc, bcc and some other frequently used phrases in emails come into our eyes. However, except subject, most of them are only setters, where are the getters? If you still remember how we've shifted our attention from MFLibraryMessage.h to MFMailMessage.h, you will notice that MFMailMessage inherits from MFMessage. Before

inspecting MFMessage.h, let's take a look at the return value of [MFMailMessage subject] through LLDB to verify the analysis by now.

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90',
stop reason = breakpoint 3.1
    frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xfe48: push   {r4, r5, r6, r7, lr}
    0xfe4a: add    r7, sp, #12
    0xfe4c: push.w {r8, r10, r11}
    0xfe50: sub.w  r4, sp, #24
(lldb) po [[[$r2 object] copyAllMessages] anyObject] subject]
Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand Them
```

We can see that the return value of [MFMailMessage subject] is exactly the email title. Take a look at MFMessage.h (Attention, MFMessage is a class in MIME.framework).

```
@interface MFMessage : NSObject <NSCopying>
.....
- (id)headerData;
- (id)bodyData;
- (id)summary;
- (id)bccIfCached;
- (id)bcc;
- (id)ccIfCached;
- (id)cc;
- (id)toIfCached;
- (id)to;
- (id)firstSender;
- (id)sendersIfCached;
- (id)senders;
- (id)dateSent;
- (id)subject;
- (id)messageData;
- (id)messageBody;
- (id)headers;
.....
@end
```

to, sender, subject, messageBody, getters for all email information are available now. It's time to check their values with LLDB.

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90',
stop reason = breakpoint 3.1
    frame #0: 0x000efe48 MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xfe48: push   {r4, r5, r6, r7, lr}
    0xfe4a: add    r7, sp, #12
    0xfe4c: push.w {r8, r10, r11}
    0xfe50: sub.w  r4, sp, #24
(lldb) po [[[$r2 object] copyAllMessages] anyObject] firstSender]
NYTimes.com <nytdirect@nytimes.com>
(lldb) po [[[$r2 object] copyAllMessages] anyObject] sendersIfCached]
<__NSArrayI 0x16850850>(<
```

```

NYTimes.com <nytdirect@nytimes.com>
)

(lldb) po [[[$r2 object] copyAllMessages] anyObject] senders]
<__NSArrayI 0x16850850>(
NYTimes.com <nytdirect@nytimes.com>
)

(lldb) po [[[$r2 object] copyAllMessages] anyObject] to]
<__NSArrayI 0x16850840>(
snakeninny@gmail.com
)

(lldb) po [[[$r2 object] copyAllMessages] anyObject] dateSent]
2014-10-01 21:30:32 +0000
(lldb) po [[[$r2 object] copyAllMessages] anyObject] subject]
Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand Them
(lldb) po [[[$r2 object] copyAllMessages] anyObject] messageBody]
<MFMimeBody: 0x16852fc0>

```

Everything is too distinct to explain. `firstSender` returns a single sender, while `sendersIfCached` and `senders` both return an `NSArray`, which means on iOS, there could be multiple senders in an email. Although this situation is quite rare (at least for me, I have never seen multiple senders), to avoid missing any sender, I'll still use "senders" to get all possible senders. The final task is to mark messages as read; do you still remember `[MessageMegaMall markMessagesAsViewed:]` in section 8.2.5? Is it the right method for marking messages as read? Let's set a breakpoint on this method and check whether it will be called when we mark an email as read.

At first, we need to locate `[MessageMegaMall markMessagesAsViewed:]` in IDA and check its base address, as shown in figure 8-17.

```

__text:0013B648 ; MessageMegaMall - (void)markMessagesAsViewed:(id)
__text:0013B648 ; Attributes: bp-based frame
__text:0013B648
__text:0013B648 ; void __cdecl -[MessageMegaMall markMessagesAsViewed:
__text:0013B648 __MessageMegaMall_markMessagesAsViewed__ ; DATA XREF:
__text:0013B648
__text:0013B648 var_10          = -0x10
__text:0013B648
__text:0013B648          PUSH          {R4-R7,LR}
__text:0013B648          ADD           R7, SP, #0xC
__text:0013B648          STR.W        R8, [SP,#0xC+var_10]!
__text:0013B648

```

Figure 8- 16 `[MessageMegaMall markMessagesAsViewed:]`

Its base address is `0x13b648`. Since the ASLR offset of `MobileMail` is `0xb2000`, we can set a breakpoint like this:

```

(lldb) br s -a '0x000b2000+0x0013B648'
Breakpoint 4: where = MobileMail`___lldb_unnamed_function7357$MobileMail, address =
0x001ed648
Process 103910 stopped
* thread #1: tid = 0x195e6, 0x001ed648
MobileMail`___lldb_unnamed_function7357$MobileMail, queue = 'com.apple.main-thread,
stop reason = breakpoint 4.1

```



```

frame #0: 0x001df648 MobileMail`___lldb_unnamed_function7357$$MobileMail
MobileMail`___lldb_unnamed_function7357$$MobileMail:
-> 0x1ed648: push   {r4, r5, r6, r7, lr}
   0x1ed64a: add    r7, sp, #12
   0x1ed64c: str    r8, [sp, #-4]!
   0x1ed650: mov    r8, r0
(lldb) po $r2
{(
  <MFLibraryMessage 0x157b70b0: library id 906, remote id 13142, 2014-12-17 22:34:30
+0000, 'Asian Morning: Obama Announces U.S. and Cuba Will Resume Relations'>
)}
(lldb) po [$r2 class]
__NSSetI

```

The output of LLDB validates our assumption. `[MessageMegaMall markMessagesAsViewed:]` is the right method for marking messages as read and its argument is an `NSSet` of `MFLibraryMessage` objects. Till now, we have successfully added the whitelist button, captured the refresh completion event, got all emails and their senders, as well marked them as read. Tweak prototyping comes to an end; let's comb our thoughts before writing code.

8.3 Result interpretation

The practice in this chapter is highly modularized; every part in Mail has a clear division of work, which speeds up our tweak prototyping.

1. Find the place and method for adding whitelist button

Sticking to the pursuit of both understandability and harmony, we have tried several solutions and finally decided to put the whitelist button at the top left corner of “Mailboxes” view. We were all familiar with the pattern to get `MailboxPickerController` with `Cycript`, so there was no difficulty for us to add a button on its navigation bar.

2. Find the refresh completion callback methods in protocol

Again in this chapter, we've used the protocols in `MailboxContentViewController.h` as clues, walked through all corresponding headers and guessed the keywords, then finally found the refresh completion callback methods, just like what we've done in “find a method to monitor note text changes in real time”, chapter 7. After testing, `megaMallMessageCountChanged:` was called when email count changes, thus met our requirements.

3. Get all emails from MessageMegaMall.

According to the experience that “The reason a protocol method gets called is generally that the corresponding event mentioned in the method name happened. And the thing that triggers the event is usually the method’s arguments”, we’ve found class MessageMegaMall from the argument of megaMallMessageCountChanged:. The name, MegaMall, was very obscure. With wild guesses and programmatic checks, we’ve discovered that it was the model for email managements. By calling [MessageMegaMall copyAllMessages], we could get all emails.

4. Get the sender’s address from MFLibraryMessage

[MessageMegaMall copyAllMessages] returned an array of MFLibraryMessage objects. By inspecting MFLibraryMessage.h and related headers, as well testing some suspicious properties and methods, we could easily get the sender’s addresses from this class.

5. Mark emails as read with MessageMegaMall

When we were studying MessageMegaMall.h, we have noticed the uncertain target method, markMessagesAsViewed:. We could even say for sure it was what we were looking for without any test. Of course, the result from LLDB proved our conclusion directly.

Notice: In order to simplify the tweak, the whitelist in section 8.4 consists of only one single email address, and it’s presented as a UIAlertView. As an exercise, it’s your turn to extend it with more addresses and use a UITableView to present it, make this tweak more useful.

8.4 Tweak writing

All difficulties have been overcome during the stage of prototyping. Now we just need to follow the conclusion we get in section 8.3 and write the tweak with elegant code.

8.4.1 Create tweak project “iOSREMailMarker” using Theos

The Theos commands are as follows:

```
hangcom-mba:Documents sam$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/cydyget
[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): iOSREMailMarker
Package Name [com.yourcompany.iosreemailmarker]: com.iosre.mailmarker
Author/Maintainer Name [sam]: sam
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.mobilemail
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: MobileMail
Instantiating iphone/tweak in iosreemailmarker/...
Done.

```

8.4.2 Compose iOSREMailMarker.h

The finalized iOSREMailMarker.h looks like this:

```

@interface MailboxPickerController : UITableViewController
@end

@interface NSConcreteNotification : NSNotification
@end

@interface MessageMegaMail : NSObject
- (void)markMessagesAsViewed:(NSSet *)arg1;
- (NSSet *)copyAllMessages;
@end

@interface MFMessageInfo : NSObject
@property (nonatomic) BOOL read;
@end

@interface MFLibraryMessage : NSObject
- (NSArray *)senders;
- (MFMessageInfo *)copyMessageInfo;
@end

```

This header is composed by picking snippets from other class-dump headers. The existence of this header is simply for avoiding any warnings or errors when compiling the tweak.

8.4.3 Edit Tweak.xm

The finalized Tweak.xm looks like this:

```

#import "iOSREMailMarker.h"

%hook MailboxPickerController
%new
- (void)iOSREShowWhitelist
{
    UIAlertController *alertController = [UIAlertController
alertControllerWithTitle:@"Whitelist" message:@"Please input an email address"
preferredStyle:UIAlertControllerStyleAlert];

```

```

        UIAlertAction *okAction = [UIAlertAction actionWithTitle:@"OK"
style:UIAlertActionStyleDefault handler:^(UIAlertAction * action) {
        UITextField *whitelistField = alertController.textFields.firstObject;
        if ([whitelistField.text length] != 0) [[NSUserDefaults standardUserDefaults]
setObject:whitelistField.text forKey:@"whitelist"];
        }];
        UIAlertAction *cancelAction = [UIAlertAction actionWithTitle:@"Cancel"
style:UIAlertActionStyleCancel handler:nil];
        [alertController addAction:okAction];
        [alertController addAction:cancelAction];
        [alertController addTextFieldWithConfigurationHandler:^(UITextField *textField) {
        textField.placeholder = @"snakeninny@gmail.com";
        textField.text = [[NSUserDefaults standardUserDefaults]
objectForKey:@"whitelist"];
        }];
        [self presentViewController:alertController animated:YES completion:nil];
    }

- (void)viewWillAppear:(BOOL)arg1
{
    self.navigationItem.leftBarButtonItem = [[[UIBarButtonItem alloc]
initWithTitle:@"Whitelist" style:UIBarButtonItemStylePlain target:self
action:@selector(iOSREShowWhitelist)] autorelease];
    %orig;
}
%end

%hook MailboxContentViewController
- (void)megaMallMessageCountChanged:(NSConcreteNotification *)arg1
{
    %orig;
    NSMutableSet *targetMessages = [NSMutableSet setWithCapacity:600];
    NSString *whitelist = [[NSUserDefaults standardUserDefaults]
objectForKey:@"whitelist"];
    MessageMegaMall *mall = [arg1 object];
    NSSet *messages = [mall copyAllMessages]; // Remember to release it later
    for (MFLibraryMessage *message in messages)
    {
        MFMessageInfo *messageInfo = [message copyMessageInfo]; // Remember to
release it later
        for (NSString *sender in [message senders]) if (!messageInfo.read && [sender
rangeOfString:[NSString stringWithFormat:@"%<%@>", whitelist]].location == NSNotFound)
[targetMessages addObject:message];
        [messageInfo release];
    }
    [messages release];
    [mall markMessagesAsViewed:targetMessages];
}
%end

```

8.4.4 Edit Makefile and control files

The finalized 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 = iOSREMailMarker
iOSREMailMarker_FILES = Tweak.xml
iOSREMailMarker_FRAMEWORKS = UIKit

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 MobileMail"

```

The finalized control looks like this:

```

Package: com.iosre.mailmarker
Name: iOSREMailMarker
Depends: mobilessubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Mark non-whitelist emails as read!
Maintainer: sam
Author: sam
Section: Tweaks
Homepage: http://bbs.iosre.com

```

8.4.5 Test

Compile the tweak and install it on iOS. Open Mail but it seems nothing changed. That is because we haven't configured iOSREMailMarker yet. As shown in figure 8-18, there are 44 unread messages currently.

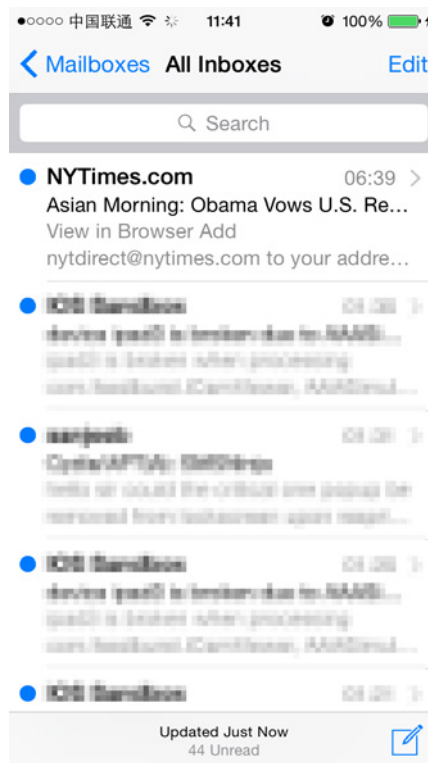


Figure 8- 17 44 unread emails

After entering the “Mailboxes” view, there is a new whitelist button on the left side of navigation bar. Press it and a new whitelist dialog will pop up, as shown in 8-19.

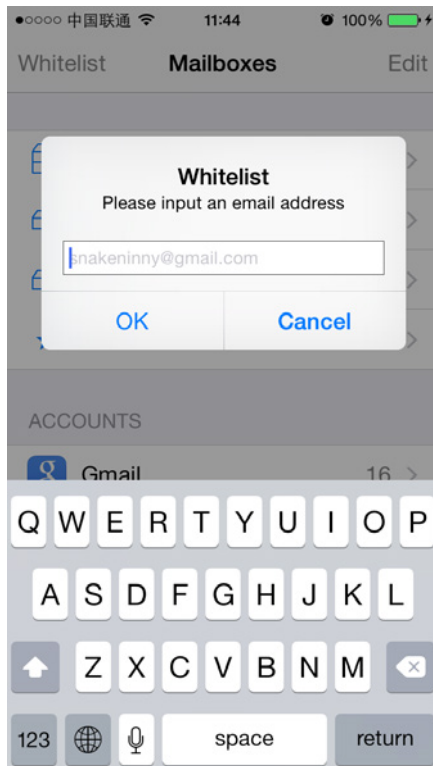


Figure 8- 18 Whitelist dialog

I've subscribed a copy of NYTimes and will spend about 15 minutes reading it every day. Let's add NYTimes into whitelist, as shown in figure 8-20.

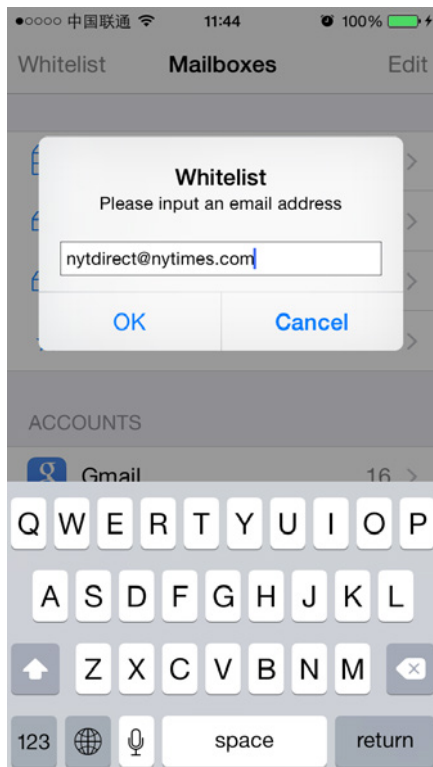


Figure 8- 19 Add NYTimes into whitelist

At last, send an email to myself to trigger megaMallMessageCountChanged:. After receiving the email, all emails except NYTimes are marked as read, as shown in 8-21.

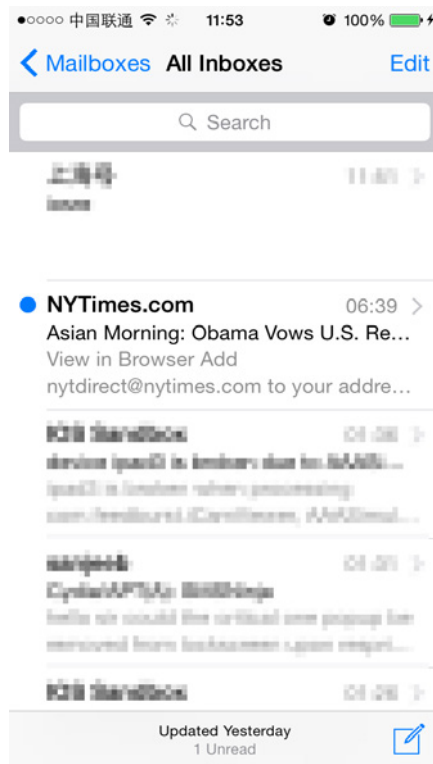


Figure 8-21 iOSREMailMarker marked all emails except NYTimes as read

So far, we have achieved all of our goals successfully.

8.5 Conclusion

In this chapter, we've taken Mail as an example and added a feature that can automatically mark emails outside whitelist as read, which helps us highlight the important emails. The filter condition of iOSREMailMarker is somewhat simple, and it may not be a good solution for everyone to simply mark emails as read. So I hope you can learn this chapter by analogy and intimidate the ideas to make your own unique tweaks. Of course, you are welcome to share your works on our website.

So far, we have gone through 2 practices. I hope everyone enjoyed them and had the feeling that our brains should keep one step ahead of our hands in iOS reverse engineering. Only when you get fully prepared during early stage analysis can you write an excellent tweak later. TiGa, a veteran reverse engineer, once said: "A reverser should know how/what is done before grabbing tools to complete the tasks automatically." I believe that everyone will gradually realize the meaning of this sentence during continuously studying reverse engineering.

Practice 3: Save and share Sight in WeChat

9.1 WeChat

WeChat is one of the most outstanding IM App in the mobile Internet industry. In China, it is the daily necessity of most netizens. WeChat's launch image is as shown in figure 9-1; it seems that there is a little sorrow in its great power.



Figure 9-1 Launch image of WeChat

In October 3rd, 2014, WeChat has updated to version 6.0 and added a new feature, Sight i.e. short videos. It was so fun that my WeChat friends started to share all kinds of Sights, as shown in figure 9-2.

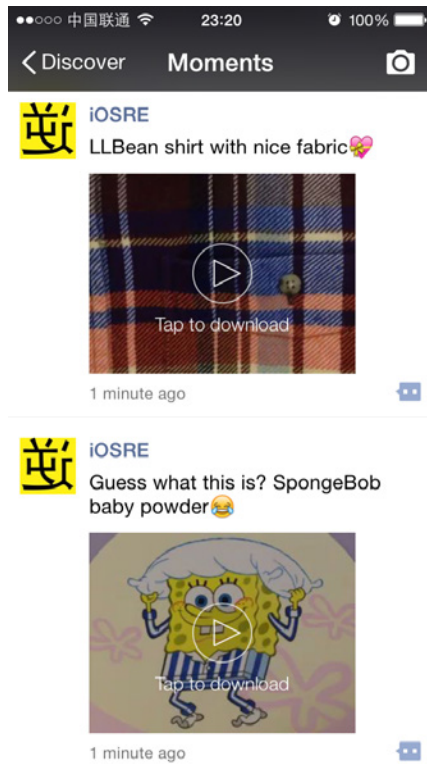


Figure 9-2 Sight

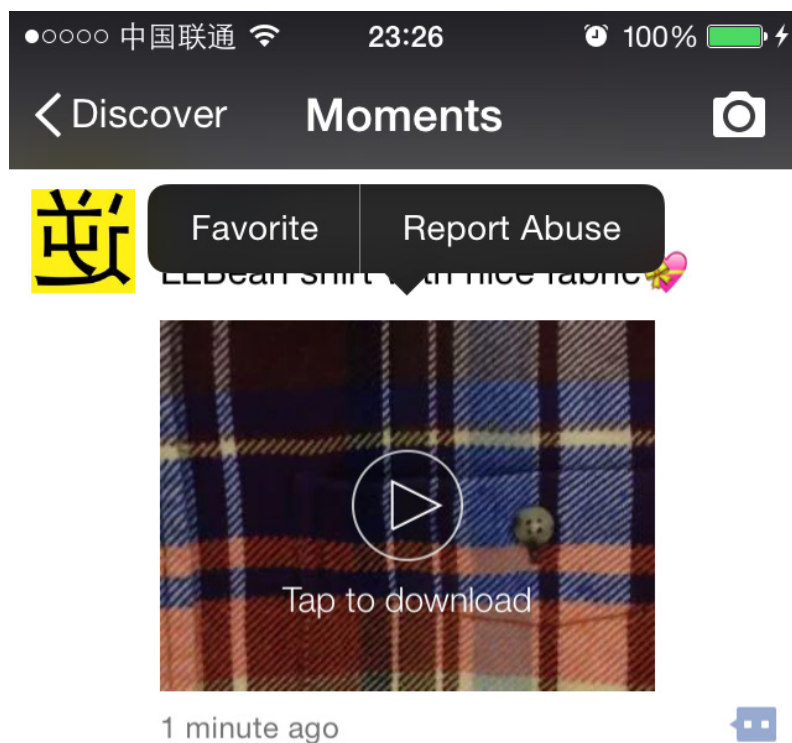


Figure 9-3 Menu of Sight

Although we can already mark our interested Sights via long press menus (as shown in figure 9-3), I'm not satisfied yet; it'd be better if those Sights can be downloaded or shared on other platforms. So, the goal of this chapter is adding two options to long press menu of Sight, which are "Save to Disk" and "Copy URL" respectively.

The size of WeChat 6.0 is bigger than 80 MB; it's rather complicated reversing it. As usual, before reversing, we need to analyze and modeling the target, then make a plan and carry it out. The following operations are done on WeChat 6.0 on iOS 8.1, iPhone 5. After the publication of this book, WeChat will probably update to a higher version, there will be some tiny changes in the following operations, but the general ideas stay the same. For the analysis of the latest WeChat, please keep following <http://bbs.iosre.com>.

9.2 Tweak prototyping

9.2.1 Observe Sight view and look for cut-in points

First, switch Sights' autoplay in "WeChat" → "Me" → "Settings" → "General" → "Sights in Moments" to "Never", as shown in figure 9-4.

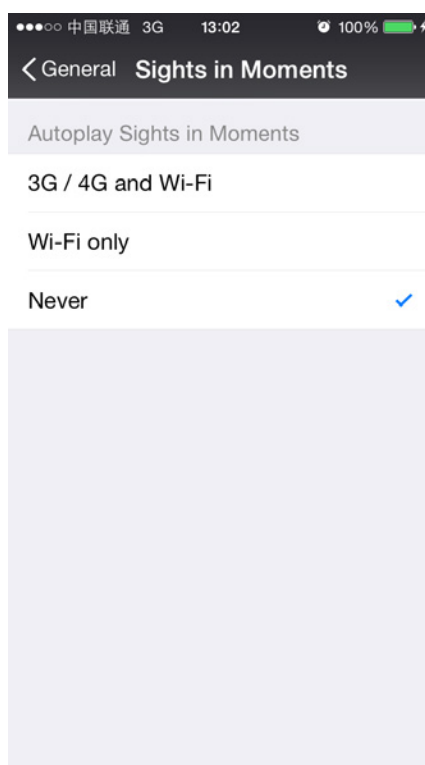


Figure 9-4 Never autoplay Sights in Moments

Let's review figure 9-3 and think together: "Favorite" and "Report Abuse" will pop up after we long press the Sight view. Doesn't this phenomenon indicate that the Sight view can already respond to long press gestures? So, we only need to find the gesture action selector and hook it, then we can pop up our custom menu with options "Save to Disk" and "Copy URL" just inside this function.

There is a line of words "Tap to download" under the play button in Sight view, which

means WeChat will download the Sight to iOS first, and then play it offline. Therefore, we can conclude that a download URL already exists in a Sight, and the downloaded Sight is saved somewhere on iOS. Luckily, the URL and the downloaded Sight happen to be our goal of this chapter, if we can find their locations in WeChat, our job is done. After the previous 2 practices, I believe your understanding of MVC has become deeper: If we manage to get the V of a Sight, we can get its M, which contains the Sight's download URL and video objects.

OK, now we know that WeChat has already invented the wheel, we just need to find and make use of it. In order to speed up our reversing process, we won't be overly sticking to the execution logic of WeChat with IDA or LLDB, but try our best to look for clues in class-dump headers, and then verify our guesses to reach the goal of locating the Sight.

9.2.2 Get WeChat headers using class-dump

First decrypt WeChat with dumpdecrypted, which is explained in details in chapter 4. It is worth mentioning that the executable name of WeChat is not "WeiXin" (which is Chinese pinyin) or "WeChat", but "MicroMessenger". After we get MicroMessenger.decrypted, drag and drop it to IDA before continuing. Then use class-dump to export its headers.

```
snakeninnysMac:~ snakeninny$ class-dump -S -s -H ~/MicroMessenger -o ~/header6.0
```

After executing the above command, 5225 headers are generated, as shown in figure 9-5.

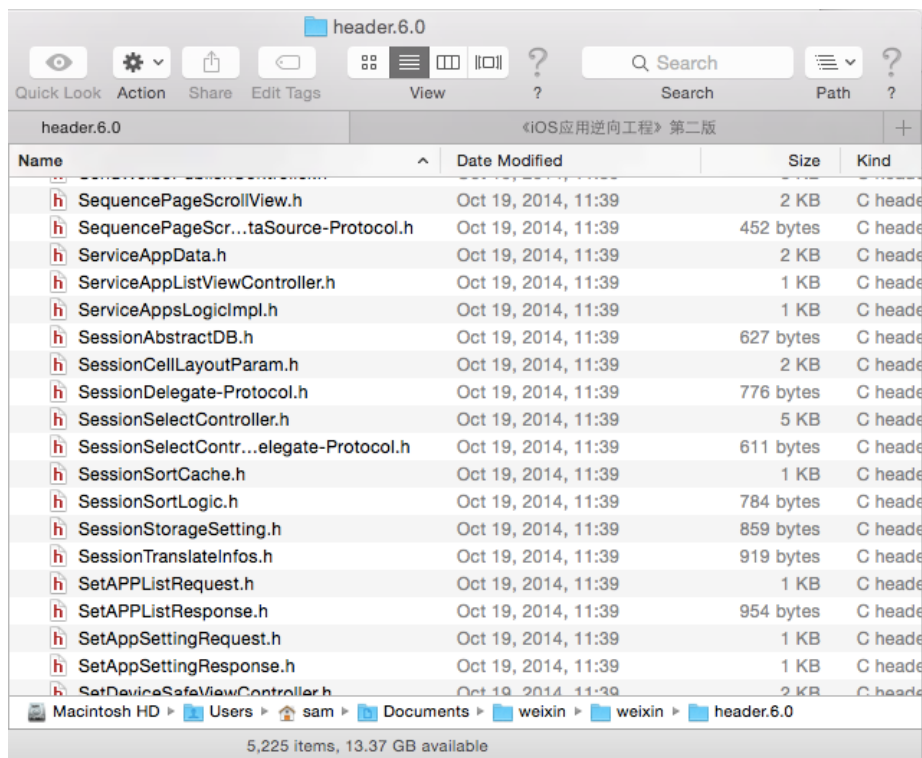


Figure 9-5 Headers of WeChat 6.0

WeChat has the most headers among all Apps I have ever seen, going through all these files one by one is mission impossible for a single person. Such a big project is unlikely to be completed by one single team, perhaps Tencent just splits WeChat into several subprojects, for example, Moments subproject, IM subproject, drift bottle subproject, Sight subproject, etc. For each subproject, there's one team in charge. At last, all subprojects are merged into one big project, namely WeChat.

9.2.3 Import WeChat headers into Xcode

Import all WeChat headers to an empty Xcode project, as shown in figure 9-6.

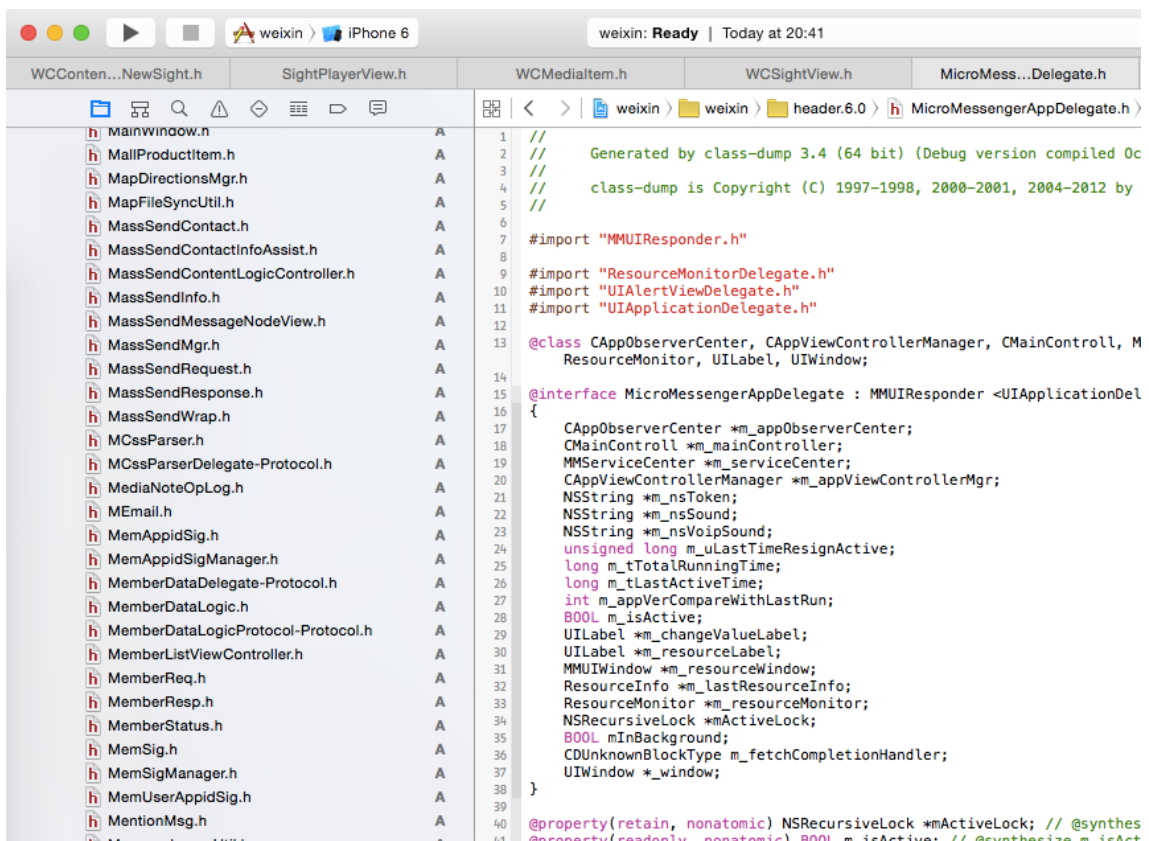


Figure 9-6 WeChat headers in Xcode

The built-in search and highlight functions in Xcode help display the code beautifully and neatly. Now, let's cut into the code via WeChat's UI.

9.2.4 Locate the Sight view using Reveal

There is no need to introduce how to configure Reveal again. Launch WeChat and enter Moments to find a Sight, then use Reveal to see the view hierarchy, as shown in figure 9-7.

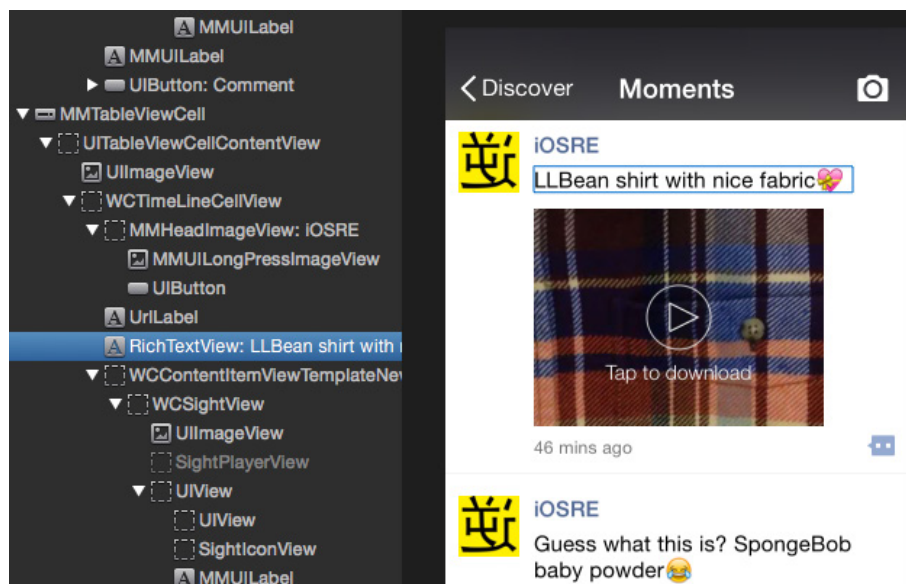


Figure 9-7 Use Reveal to see the UI layout of WeChat

In figure 9-7, text “LLBean shirt with nice fabric” can be discovered easily in both sides, indicating their correspondence. Keep checking around RichTextView, the Sight view is very conspicuous, as shown in figure 9-8.

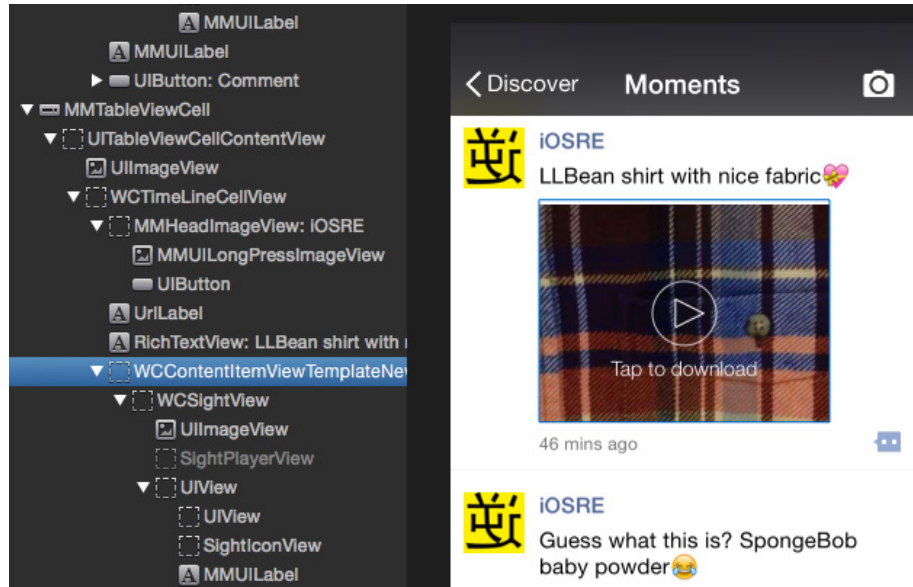


Figure 9-8 Locate the Sight view

The Sight view is an object of `WCContentItemViewTemplateNewSight`. Do you still remember the indent principle mentioned in the section of recursiveDescription? According to the rule of “the view with more indentation is the subview of the view with less indentation”, `WCContentItemViewTemplateNewSight`’s subviews include `WCSightView`, and `WCSightView`’s subviews include `UIImageView` and `SightPlayerView`. Because “Sight” is the nickname of short videos in WeChat, these classes with the name “sight” should be given more attention.

9.2.5 Find the long press action selector

Commonly we use `addGestureRecognizer:` to add a long press gesture recognizer in iOS. Since long press a Sight view shows a menu, the long press gesture is very probably to be added right on the Sight view. Since this view is an object of `WCContentItemViewTemplateNewSight`, let’s see what’s in its header file:

```
@interface WCContentItemViewTemplateNewSight : WCContentItemBaseView
<WCActionSheetDelegate, SessionSelectControllerDelegate, WCSightViewDelegate>
.....
- (void)onMore:(id)arg1;
- (void)onFavoriteAdd:(id)arg1;
- (void)onLongTouch;
```

```

- (void)onShowSightAction;
- (void)onLongPressedWCSightFullScreenWindow:(id)arg1;
- (void)onLongPressedWCSight:(id)arg1;
- (void)onClickWCSight:(id)arg1;
.....
@end

```

In the header, methods with keywords “LongTouch” and “LongPressed” are very likely to be our targets. Now, IDA should have finished the initial analysis, let’s take a look at the implementation of these methods. onLongTouch first, as shown in figure 9-9.

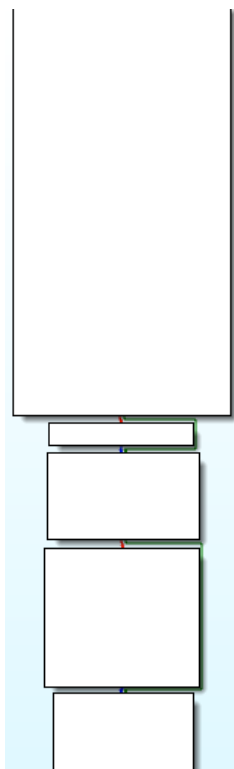


Figure 9-9 onLongTouch

The execution flow of this method is very simple. Look through the method body quickly, “UIMenuController” can be easily found, as shown in figure 9-10.

```

BLX.W      j__objc_msgSend
MOV        R0, #(selRef_becomeFirstResponder - 0x4BEBEE)
ADD        R0, PC ; selRef_becomeFirstResponder
LDR        R1, [R0] ; "becomeFirstResponder"
MOV        R0, R6
BLX.W      j__objc_msgSend
MOV        R0, #(selRef_sharedMenuController - 0x4BEC08)
MOV        R2, #(classRef_UIMenuController - 0x4BEC0A) ;
ADD        R0, PC ; selRef_sharedMenuController
ADD        R2, PC ; classRef_UIMenuController
LDR        R1, [R0] ; "sharedMenuController"
LDR        R0, [R2] ; OBJC_CLASS_$_UIMenuController
BLX.W      j__objc_msgSend
STR        R0, [SP,#0x38+var_24]
MOV        R1, #(selRef_setTargetRect_inView_ - 0x4BEC20)

```

Figure 9-10 onLongTouch

“Favorite” stands out too, as shown in figure 9-11.

```

ADD          R2, PC ; "Favorites_Add"
LDR.W       R11, [R1] ; "getStringForCurLanguage:defaultTo:"
MOV         R3, R2
MOV         R1, R11
BLX.W       j__objc_msgSend
MOV         R2, R0
MOV         R0, #(selRef_initWithTitle_action_ - 0x4BECF6) ;
MOV         R1, #(selRef_onFavoriteAdd_ - 0x4BECF8) ; selRef
ADD         R0, PC ; selRef_initWithTitle_action_
ADD         R1, PC ; selRef_onFavoriteAdd_
LDR         R5, [R0] ; "initWithTitle:action:"
MOV         R0, R4
LDR         R3, [R1] ; "onFavoriteAdd:"
MOV         R1, R5
BLX.W       j__objc_msgSend

```

Figure 9-11 onLongTouch

Unless WeChat is intentionally confusing us with these keywords, [WCContentItemViewTemplateNewSight onLongTouch] must be the response method of long press gestures. No need to hurry, let's take a look at methods with keyword "LongPressed", as shown in figure 9-12.

```

BLX.W       j__strchr
MOVW        R1, #(:lower16:(selRef_logWithLevel_module_file_line_func_format_ - 0x21E4C0
ADD.W       R3, R6, #0xA7
MOVT.W      R1, #(:upper16:(selRef_logWithLevel_module_file_line_func_format_ - 0x21E4C0
MOVW        R2, #(:lower16:(cfstr_Onlongpresse_7 - 0x21E4C6)) ; "onLongPressedWCSightFul
ADD         R1, PC ; selRef_logWithLevel_module_file_line_func_format_
MOVT.W      R2, #(:upper16:(cfstr_Onlongpresse_7 - 0x21E4C6)) ; "onLongPressedWCSightFul
ADD         R2, PC ; "onLongPressedWCSightFullScreenWindow"
MOVS        R6, #0x86
LDR         R1, [R1] ; "logWithLevel:module:file:line:func:form"...
ADDS        R0, #1
STMEA.W    SP, {R0,R6}
MOV         R0, R5
STR         R3, [SP,#0x1C+var_14]
MOVS        R3, #0
STR         R2, [SP,#0x1C+var_10]
MOVS        R2, #1
BLX.W       j__objc_msgSend
MOV         R0, #(selRef_onShowSightAction_ - 0x21E4E8) ; selRef_onShowSightAction
ADD         R0, PC ; selRef_onShowSightAction
LDR         R1, [R0] ; "onShowSightAction"
MOV         R0, R4
ADD         SP, SP, #0x10
POP.W       {R4-R7,LR}
B.W         j__j__objc_msgSend_0
; End of function -[WCContentItemViewTemplateNewSight onLongPressedWCSightFullScreenWindow:]

```

Figure 9-12 onLongPressedWCSightFullScreenWindow:

It seems that it records some information, then calls onShowSightAction. Double click onShowSightAction to see its implementation, as shown in figure 9-13.


```

; WCContentItemViewTemplateNewSight - (void)onShowSightAction
; Attributes: bp-based frame

; void __cdecl -[WCContentItemViewTemplateNewSight onShowSightAction]
__WCContentItemViewTemplateNewSight_onShowSightAction_

var_38= -0x38
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB       SP, SP, #0x20
MOV       R4, R0
STR       R4, [SP,#0x38+var_1C]
MOV       R0, #(selRef_alloc - 0x21E516) ; selRef_alloc
MOV       R2, #(classRef_WActionSheet - 0x21E518) ; classRef_W
ADD       R0, PC ; selRef_alloc
ADD       R2, PC ; classRef_WActionSheet
LDR       R1, [R0] ; "alloc"
LDR       R0, [R2] ; _OBJC_CLASS_$_WActionSheet
BLX.W    j__objc_msgSend
MOVW     R1, #(:lower16:(selRef_initWithTitle_delegate_cancelB
R2, #0
MOVT.W   R1, #(:upper16:(selRef_initWithTitle_delegate_cancelB
STR       R2, [SP,#0x38+var_38]
ADD       R1, PC ; selRef_initWithTitle_delegate_cancelButtonTi
STR       R2, [SP,#0x38+var_34]
STR       R2, [SP,#0x38+var_30]

```

Figure 9-13 onShowSightAction

In figure 9-13, we know that a WActionSheet object is created from the very beginning. From its name, we can guess that WActionSheet acts like UIActionSheet, but we didn't see any action sheet when we long press the Sight, so onLongPressedWCSightFullScreenWindow: is probably not the method we want.

The last method, onLongPressedWCSight:, is shown in figure 9-14.

```

BLX.W    j__strrchr
MOVW     R1, #(:lower16:(selRef_logWithLevel_module_file_line_func_fo
ADD.W    R3, R6, #0x6C
MOVT.W   R1, #(:upper16:(selRef_logWithLevel_module_file_line_func_fo
MOVW     R2, #(:lower16:(cfstr_Onlongpressedw - 0x21E456)) ; "onLongP
ADD       R1, PC ; selRef_logWithLevel_module_file_line_func_format_
MOVT.W   R2, #(:upper16:(cfstr_Onlongpressedw - 0x21E456)) ; "onLongP
ADD       R2, PC ; "onLongPressedWCSight"
MOVS     R6, #0x7F
LDR       R1, [R1] ; "logWithLevel:module:file:line:func:form"...
ADDS     R0, #1
STMEA.W  SP, {R0,R6}
MOV       R0, R5
STR       R3, [SP,#0x1C+var_14]
MOVS     R3, #0
STR       R2, [SP,#0x1C+var_10]
MOVS     R2, #1
BLX.W    j__objc_msgSend
MOV       R0, #(selRef_onLongTouch - 0x21E478) ; selRef_onLongTouch
ADD       R0, PC ; selRef_onLongTouch
LDR       R1, [R0] ; "onLongTouch"
MOV       R0, R4
ADD       SP, SP, #0x10
POP.W    {R4-R7,LR}
B.W      j__objc_msgSend_0
; End of function -[WCContentItemViewTemplateNewSight onLongPressedWCSight:]

```

Figure 9-14 onLongPressedWCSight:

From figure 9-14, we can see it records some information, then calls onLongTouch, which proves our conjecture indirectly. Now, let's debug onLongPressedWCSightFullScreenWindow: and onLongTouch using LLDB. Firstly, attach debugserver to MicroMessenger:


```

snakeninnysiMac:Documents snakeninny$ ssh root@localhost -p 2222
FunMaker-5:~ root# debugserver *:1234 -a MicroMessenger
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process MicroMessenger...
Listening to port 1234 for a connection from *...
Waiting for debugger instructions for process 0.
Then check the ASLR offset of WeChat:
(lldb) image list -o -f
[ 0] 0x00000000 /private/var/mobile/Containers/Bundle/Application/E4EBD049-1A75-4830-
BC65-0132C0EBC1CA/MicroMessenger.app/MicroMessenger(0x0000000000004000)
[ 1] 0x022dc000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000022dc000)
.....

```

The ASLR offset of WeChat is surprisingly 0x0. Next, let's check the base addresses of onLongPressedWCSightFullScreenWindow: and onLongTouch, as shown in figure 9-15 and 9-16.

```

text:0021E484 ; WContentItemViewTemplateNewSight - (void)onLongPressedWCSightFullScreenWindow:(id)
text:0021E484 ; Attributes: bp-based frame
text:0021E484
text:0021E484 ; void __cdecl -[WContentItemViewTemplateNewSight onLongPressedWCSightFullScreenWindow:]
text:0021E484 __WContentItemViewTemplateNewSight_onLongPressedWCSightFullScreenWindow__
text:0021E484 ; DATA XREF: __objc_const:01AF7368j
text:0021E484
text:0021E484 var_14 = -0x14
text:0021E484 var_10 = -0x10
text:0021E484
text:0021E484 PUSH {R4-R7,LR}
text:0021E486 ADD R7, SP, #0xC
text:0021E488 SUB SP, SP, #0x10
text:0021E48A MOV R4, R0
text:0021E48C MOV R0, #(classRef_iConsole - 0x21E4A0)

```

Figure 9-15 onLongPressedWCSightFullScreenWindow:

```

text:0021E7EC ; WContentItemViewTemplateNewSight - (void)onLongTouch
text:0021E7EC ; Attributes: bp-based frame
text:0021E7EC
text:0021E7EC ; void __cdecl -[WContentItemViewTemplateNewSight onLongTouch]
text:0021E7EC __WContentItemViewTemplateNewSight_onLongTouch__
text:0021E7EC ; DATA XREF: __objc_con
text:0021E7EC
text:0021E7EC var_2C = -0x2C
text:0021E7EC var_28 = -0x28
text:0021E7EC var_24 = -0x24
text:0021E7EC var_20 = -0x20
text:0021E7EC var_1C = -0x1C
text:0021E7EC
text:0021E7EC PUSH {R4-R7,LR}
text:0021E7EE ADD R7, SP, #0xC
text:0021E7F0 PUSH.W {R8,R10,R11}

```

Figure 9-16 onLongTouch

The base addresses of them are 0x21e484 and 0x21e7ec. Set 2 breakpoints on them then long press the Sight view to see whether these breakpoints are triggered:

```

(lldb) br s -a 0x21e484
Breakpoint 3: where = MicroMessenger`___lldb_unnamed_function9789$$MicroMessenger,
address = 0x0021e484
(lldb) br s -a 0x21e7ec
Breakpoint 4: where = MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger,
address = 0x0021e7ec
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger, queue = 'com.apple.main-
thread, stop reason = breakpoint 4.1

```

```

frame #0: 0x0021e7ec MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger
MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger:
-> 0x21e7ec: push  {r4, r5, r6, r7, lr}
0x21e7ee: add   r7, sp, #12
0x21e7f0: push.w {r8, r10, r11}
0x21e7f4: sub   sp, #32
(lldb) p (char *)$r1
(char *) $0 = 0x017fdc2b "onLongTouch"
(lldb) c
Process 184500 resuming
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger, queue = 'com.apple.main-
thread, stop reason = breakpoint 4.1
frame #0: 0x0021e7ec MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger
MicroMessenger`___lldb_unnamed_function9791$$MicroMessenger:
-> 0x21e7ec: push  {r4, r5, r6, r7, lr}
0x21e7ee: add   r7, sp, #12
0x21e7f0: push.w {r8, r10, r11}
0x21e7f4: sub   sp, #32
(lldb) p (char *)$r1
(char *) $1 = 0x017fdc2b "onLongTouch"

```

As we can see, onLongTouch was called twice, and onLongPressedWCSightFullScreenWindow was never called. Take another look at onLongPressedWCSight:, its base address is shown in figure 9-17.

```

text:0021E414 ; WCContentItemViewTemplateNewSight - (void)onLongPressedWCSight:(id)
text:0021E414 ; Attributes: bp-based frame
text:0021E414
text:0021E414 ; void __cdecl -[WCContentItemViewTemplateNewSight onLongPressedWCSight:]
text:0021E414 __WCContentItemViewTemplateNewSight_onLongPressedWCSight__
text:0021E414 ; DATA XREF: __objc_const:01AF735
text:0021E414
text:0021E414 var_14 = -0x14
text:0021E414 var_10 = -0x10
text:0021E414
text:0021E414 PUSH {R4-R7,LR}
text:0021E416 ADD R7, SP, #0xC
text:0021E418 SUB SP, SP, #0x10
text:0021E41A MOV R4, R0

```

Figure 9- 17 onLongPressedWCSight:

Set a breakpoint on this method to see whether it's triggered:

```

(lldb) c
Process 184500 resuming
(lldb) br del
About to delete all breakpoints, do you want to do that?: [Y/n] y
All breakpoints removed. (2 breakpoints)
(lldb) br s -a 0x21e414
Breakpoint 5: where = MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger,
address = 0x0021e414
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger, queue = 'com.apple.main-
thread, stop reason = breakpoint 5.1
frame #0: 0x0021e414 MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger
MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger:
-> 0x21e414: push  {r4, r5, r6, r7, lr}
0x21e416: add   r7, sp, #12
0x21e418: sub   sp, #16

```

```

    0x21e41a: mov    r4, r0
(lldb) p (char *)$r1
(char *) $2 = 0x0182c799 "onLongPressedWCSight:"
(lldb) c
Process 184500 resuming
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger, queue = 'com.apple.main-
thread, stop reason = breakpoint 5.1
    frame #0: 0x0021e414 MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger
MicroMessenger`___lldb_unnamed_function9788$$MicroMessenger:
-> 0x21e414: push  {r4, r5, r6, r7, lr}
    0x21e416: add   r7, sp, #12
    0x21e418: sub   sp, #16
    0x21e41a: mov   r4, r0
(lldb) p (char *)$r1
(char *) $3 = 0x0182c799 "onLongPressedWCSight:"
(lldb) po $r2
<WCSightView: 0x2454dc0; baseClass = UIControl; frame = (0 3; 200 150);
gestureRecognizers = <NSArray: 0x87e5110>; layer = <CALayer: 0xd3be460>>

```

onLongPressedWCSight: was also called twice, and its argument was an object of WCSightView. By now, we have located the response method of long press gestures, which is onLongPressedWCSight: or onLongTouch. Next, we need to go further to find the trace of the Sight.

9.2.6 Find the controller of Sight view using Cycrypt

First, click “Tap to download” in the Sight view to download the video, as shown in figure 9-18.


```

| | | | | | | | | | | | | | |
<MMUILabel: 0x7ee7530; baseClass = UILabel; frame = (0 103; 200 20); text = 'Tap to
play'; hidden = YES; userInteractionEnabled = NO; tag = 10040; layer = <_UILabelLayer:
0x7e50dd0>>
.....
cy# [#0xd3be3e0 nextResponder]
#<WCTimeLineCellView: 0x872c530; frame = (0 0; 313 243); tag = 1048577; layer =
<CALayer: 0x872ce80>>"
cy# [#0x872c530 nextResponder]
#<UITableViewControllerContentView: 0x8729d80; frame = (0 0; 320 251); gestureRecognizers =
<NSArray: 0x8729f80>; layer = <CALayer: 0x8729df0>>"
cy# [#0x8729d80 nextResponder]
#<MMTableViewCell: 0x8729be0; baseClass = UITableViewCell; frame = (0 1164.33; 320
251); autoresize = W; layer = <CALayer: 0x8729b50>>"
cy# [#0x8729be0 nextResponder]
#<UITableViewControllerWrapperView: 0xab09890; frame = (0 0; 320 568); gestureRecognizers =
<NSArray: 0xab09b00>; layer = <CALayer: 0x7e6e4b0>; contentOffset: {0, 0}; contentSize:
{320, 568}>"
cy# [#0xab09890 nextResponder]
#<MMTableView: 0x30c3200; baseClass = UITableView; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0xab09600>; layer = <CALayer: 0xab09160>; contentOffset:
{0, 1090}; contentSize: {320, 3186.3333}>"
cy# [#0x30c3200 nextResponder]
#<UIView: 0x7e3b040; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer:
0x7e3afd0>>"
cy# [#0x7e3b040 nextResponder]
#<WCTimeLineViewController: 0x28bd200>"

```

We've got WCTimeLineViewController as expected. Meanwhile, we can guess "Time Line" is the code name of "Moments".

9.2.7 Find the Sight object in WCTimeLineViewController

Look through WCTimeLineViewController.h, there are only a few properties in it; also it has no obvious methods to access M. Yet there are 2 suspicious global variables, as follows:

```

WCDatItem *_inputDataItem;
WCDatItem *_cacheDateItem;

```

But they are both NULL:

```

cy# #0x28bd200->_cacheDateItem
null
cy# #0x28bd200->_inputDataItem
null

```

Seems like we've lost. Is it time to give up? No! Let's keep calm and carry on: Because Moments are presented as table views, and there's a method named tableView:cellForRowAtIndexPath: in WCTimeLineViewController, which means this class implements UITableViewDataSource protocol, so it must have a close relationship with M. Now, jump to this method in IDA, as shown in figure 9-19.

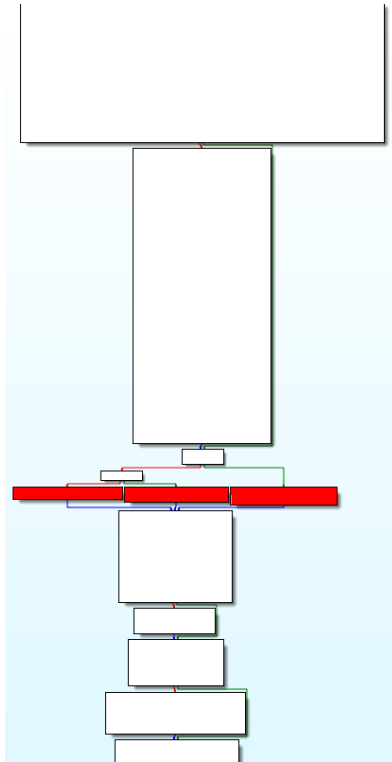


Figure 9-19 [WCTimeLineViewController tableView:cellForRowAtIndexPath:]

Look through this method, you will find 3 red squares in figure 9-17 are the core of this method, other parts are just setting the background, theme and color of the cell. Let's take a look at these 3 red squares closely, as shown in figure 9-20.

<pre> RD, #(selRef_genUploadFailCell_indexPath_ - 0x2A1CAA) RD, PC ; selRef_genUploadFailCell_indexPath_ loc_2A1CC0 </pre>	<pre> loc_2A1CB6 MOV R0, #(selRef_genNormalCell_indexPath_ - 0x2A1CC2) ADD R0, PC ; selRef_genNormalCell_indexPath_ </pre>	<pre> loc_2A1CAA MOV R0, #(selRef_genRedHeartCell_indexPath_ - 0x2A1CC4) ADD R0, PC ; selRef_genRedHeartCell_indexPath_ B </pre>
--	--	--

Figure 9-20 A close look at the 3 red squares

From left to right, the methods are `genUploadFailCell:indexPath:`, `genNormalCell:indexPath:` and `genRedHeartCell:indexPath:`. Which cell is for Sight? I guess it's "NormalCell", let's take a look at the implementation of `genNormalCell:indexPath:`, as shown in figure 9-21.

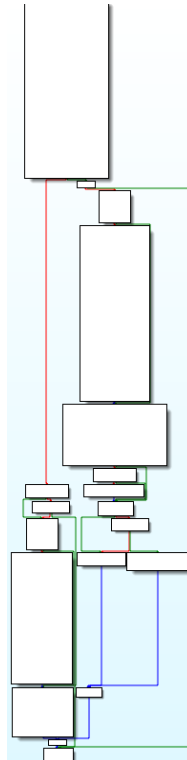


Figure 9- 21 [WCTimeLineViewController genNormalCell:indexPath:]

The logic is not complicated, if you look through the method body, a suspicious method comes up, as shown in figure 9-22.

```

MOV      R2, R0
MOV      R0, #(selRef_calcDataItemIndex_ - 0x2A08B2)
ADD      R0, PC ; selRef_calcDataItemIndex_
LDR      R1, [R0] ; "calcDataItemIndex:"
MOV      R0, R4
BLX.W   j__objc_msgSend
MOV      R5, R0
MOV      R0, #(selRef_defaultCenter - 0x2A08CE)
MOV      R2, #(classRef_MMServiceCenter - 0x2A08D0)
ADD      R0, PC ; selRef_defaultCenter
ADD      R2, PC ; classRef_MMServiceCenter
LDR      R1, [R0] ; "defaultCenter"
LDR      R0, [R2] ; _OBJC_CLASS_$_MMServiceCenter
STR      R1, [SP,#0xC8+var_A4]
BLX.W   j__objc_msgSend
MOV      R6, R0
MOV      R0, #(selRef_class - 0x2A08E6)
ADD      R0, PC ; selRef_class
LDR      R1, [R0] ; "class"
STR      R1, [SP,#0xC8+var_A8]
MOV      R0, #(classRef_WCFacade - 0x2A08F4)
ADD      R0, PC ; classRef_WCFacade
LDR      R0, [R0] ; _OBJC_CLASS_$_WCFacade
BLX.W   j__objc_msgSend
MOV      R2, R0
MOV      R0, #(selRef_getService_ - 0x2A0906)
ADD      R0, PC ; selRef_getService_
LDR      R1, [R0] ; "getService:"
MOV      R0, R6
STR      R1, [SP,#0xC8+var_AC]
BLX.W   j__objc_msgSend
MOVV    R1, #(:lower16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
MOV      R2, R5
MOVT.W  R1, #(:upper16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
ADD      R1, PC ; selRef_getTimelineDataItemOfIndex_
LDR      R1, [R1] ; "getTimelineDataItemOfIndex:"
BLX.W   j__objc_msgSend

```

Figure 9-22 [WCTimeLineViewController genNormalCell:indexPath:]

Infer from the name, getTimelineDataItemOfIndex: in figure 9-22 is probably the data

source of the current cell. Set a breakpoint at the bottom instruction, i.e. “__text:002A091C BLX.W j__objc_msgSend”, then think of a way to trigger it. We have already known that tableView:cellForRowAtIndexpath: is called when UITableView needs to display a new cell. In order to make this breakpoint break on a cell with Sight, we just need to scroll the Sight out of screen, then scroll it back. When the Sight is scrolled out, a new cell will scroll in, hence triggers the breakpoint; there’s no Sight on this cell, this kind of breakpoint doesn’t meet our requirement, so what we do is to disable the breakpoint first, then enable the breakpoint after the Sight is scrolled out of the screen completely. Now we can scroll the Sight back, the breakpoint will break on a cell with Sight:

```
(lldb) br s -a 0x2A091C
Breakpoint 6: where = MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, address = 0x002a091c
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, queue = 'com.apple.main-thread, stop reason = breakpoint 6.1
  frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$$MicroMessenger
  0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
  0x2a092a: add   r0, pc
(lldb) po $r0
Class name: WCDatItem, addr: 0x80f52b0
tid: 11896185303680028954
username: wxid_hqouu9kgsgw3e6
createtime: 1418135798
commentUsers: (
)
contentObj: <WCContentItem: 0x8724c20>
```

We’ve got a WCDatItem object, with a WCContentItem object in it. Is the “Data” in WCDatItem a Sight? Let’s test it with LLDB by setting this WCDatItem object to NULL and see what happens. Repeat the previous operations to trigger the breakpoint on a Sight cell:

```
Process 184500 stopped
```



```

* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, queue =
'com.apple.main-thread, stop reason = breakpoint 6.1
  frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$$MicroMessenger
  0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212, queue =
'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
  0x2a092a: add   r0, pc
(lldb) register write r0 0
(lldb) br del
About to delete all breakpoints, do you want to do that?: [Y/n] y
All breakpoints removed. (1 breakpoint)
(lldb) c

```



Figure 9-23 Effect of setting the return value to NULL

The first Sight totally disappeared, as shown in figure 9-23. This phenomenon indicates that the data source of the Sight is indeed WCDatItem. Before analyzing WCDatItem, there remains one problem to be solved: How can we get a WCDatItem object from the hooked

method [WCContentItemViewTemplateNewSight onLongTouch]?

9.2.8 Get a WCDataItem object from WCContentItemViewTemplateNewSight

Do you still remember how we've got an object of WCDataItem in LLDB? The answer is `getTimelineDataItemOfIndex:`. Go back to figure 9-22 to see the callers and arguments of this method.

As we can see, the caller is the return value of `getService:`, the argument is the return value of `calcDataItemIndex:`, as shown in figure 9-24.

```
MOV          R2, R0
MOV          R0, #(selRef_calcDataItemIndex_ - 0x2A08B2)
ADD          R0, PC ; selRef_calcDataItemIndex_
LDR          R1, [R0] ; "calcDataItemIndex:"
MOV          R0, R4
BLX.W       j__objc_msgSend
MOV          R5, R0
MOV          R0, #(selRef_defaultCenter - 0x2A08CE)
MOV          R2, #(classRef_MMServiceCenter - 0x2A08D0)
ADD          R0, PC ; selRef_defaultCenter
ADD          R2, PC ; classRef_MMServiceCenter
LDR          R1, [R0] ; "defaultCenter"
LDR          R0, [R2] ; _OBJC_CLASS_$_MMServiceCenter
STR          R1, [SP,#0xC8+var_A4]
BLX.W       j__objc_msgSend
MOV          R6, R0
MOV          R0, #(selRef_class - 0x2A08E6)
ADD          R0, PC ; selRef_class
LDR          R1, [R0] ; "class"
STR          R1, [SP,#0xC8+var_A8]
MOV          R0, #(classRef_WCFacade - 0x2A08F4)
ADD          R0, PC ; classRef_WCFacade
LDR          R0, [R0] ; _OBJC_CLASS_$_WCFacade
BLX.W       j__objc_msgSend
MOV          R2, R0
MOV          R0, #(selRef_getService_ - 0x2A0906)
ADD          R0, PC ; selRef_getService_
LDR          R1, [R0] ; "getService:"
MOV          R0, R6
STR          R1, [SP,#0xC8+var_AC]
BLX.W       j__objc_msgSend
MOVW        R1, #(:lower16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
MOV          R2, R5
MOVT.W      R1, #(:upper16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
ADD          R1, PC ; selRef_getTimelineDataItemOfIndex_
LDR          R1, [R1] ; "getTimelineDataItemOfIndex:"
BLX.W       j__objc_msgSend
```

Figure 9-24 Analyze `getTimelineDataItemOfIndex:`

New problems emerge: How do we call `getService:` and `calcDataItemIndex:`? Let's start from `getService:`. From the instruction "MOV R0, R6", we know the caller is R6; R6 is the return value of `[MMServiceCenter defaultCenter]`. The argument is from the return value of `[WCFacade class]`, as shown in figure 9-25.

```

MOV          R5, R0
MOV          R0, #(selRef_defaultCenter - 0x2A08CE)
MOV          R2, #(classRef_MMServiceCenter - 0x2A08D0)
ADD          R0, PC ; selRef_defaultCenter
ADD          R2, PC ; classRef_MMServiceCenter
LDR          R1, [R0] ; "defaultCenter"
LDR          R0, [R2] ; _OBJC_CLASS_$_MMServiceCenter
STR          R1, [SP, #0xC8+var_A4]
BLX.W       j__objc_msgSend
MOV          R6, R0
MOV          R0, #(selRef_class - 0x2A08E6)
ADD          R0, PC ; selRef_class
LDR          R1, [R0] ; "class"
STR          R1, [SP, #0xC8+var_A8]
MOV          R0, #(classRef_WCFacade - 0x2A08F4)
ADD          R0, PC ; classRef_WCFacade
LDR          R0, [R0] ; _OBJC_CLASS_$_WCFacade
BLX.W       j__objc_msgSend
MOV          R2, R0
MOV          R0, #(selRef_getService_ - 0x2A0906)
ADD          R0, PC ; selRef_getService_
LDR          R1, [R0] ; "getService:"
MOV          R0, R6
STR          R1, [SP, #0xC8+var_AC]
BLX.W       j__objc_msgSend

```

Figure 9-25 Analyze getService: of TimelineDataItemOfIndex:

So the caller of getService: can be obtained by `[[MMServiceCenter defaultCenter] getService:[WCFacade class]]`. Next, let's continue with calcDataItemIndex:. From the instruction "MOV R0, R4", we know the caller is R4 and R4 is "self". The argument is from the return value of [indexPath section], as shown in figure 9-26 and 9-27.

```

; void __cdecl -[WCTimelineViewController genNormalCell:indexPath:]
_WCTimelineViewController_genNormalCell_indexPath_

var_C8= -0xC8
var_C4= -0xC4
var_B4= -0xB4
var_B0= -0xB0
var_AC= -0xAC
var_A8= -0xA8
var_A4= -0xA4
var_A0= -0xA0
var_9C= -0x9C
var_98= -0x98
var_94= -0x94
var_90= -0x90
var_8C= -0x8C
var_88= -0x88
var_84= -0x84
var_68= -0x68
var_4C= -0x4C
var_48= -0x48
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_18= -0x18

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB.W    R4, SP, #0x40
BIC.W    R4, R4, #0xF
MOV      SP, R4
VST1.64  {D8-D11}, [R4@128]!
VST1.64  {D12-D15}, [R4@128]
SUB      SP, SP, #0xB0
MOV      R6, R3
MOV      R4, R0

```

Figure 9-26 Analyze getTimelineDataItemOfIndex:

```

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB.W    R4, SP, #0x40
BIC.W    R4, R4, #0xF
MOV      SP, R4
VST1.64  {D8-D11}, [R4@128]!
VST1.64  {D12-D15}, [R4@128]
SUB      SP, SP, #0xB0
MOV      R6, R3
MOV      R4, R0
STR      R6, [SP,#0xC8+var_A0]
MOV      R10, R2
STR      R4, [SP,#0xC8+var_88]
MOV      R0, #(selRef_row - 0x2A087E)
ADD      R0, PC ; selRef_row
LDR      R1, [R0] ; "row"
MOV      R0, R6
BLX.W   j__objc_msgSend
MOV      R8, R0
MOV      R0, #(selRef_section - 0x2A0892)
ADD      R0, PC ; selRef_section
LDR      R5, [R0] ; "section"
MOV      R0, R6
MOV      R1, R5
BLX.W   j__objc_msgSend
STR      R0, [SP,#0xC8+var_94]
MOV      R0, R6
MOV      R1, R5
BLX.W   j__objc_msgSend
MOV      R2, R0
MOV      R0, #(selRef_calcDataItemIndex_ - 0x2A08B2)
ADD      R0, PC ; selRef_calcDataItemIndex_
LDR      R1, [R0] ; "calcDataItemIndex:"
MOV      R0, R4
BLX.W   j__objc_msgSend

```

Figure 9-27 Analyze getTimelineDataItemOfIndex

Therefore, the argument of `getTimelineDataItemOfIndex:` can be obtained from `[WCTimeLineViewController calcDataItemIndex:[indexPath section]]`. Because we are inside `[WCContentItemViewTemplateNewSight onLongTouch]`, we can get `MMTableViewCell`, `MMTableView` and `WCTimeLineViewController` in sequence via `[self nextResponder]`, then get `indexPath` via `[MMTableView indexPathForCell:MMTableViewCell]`, and the whole process has already been proved in section 9.2.6. Although it looks inconvenient, obtaining the `WCDataItem` object from `WCContentItemViewTemplateNewSight` conforms to standard MVC design pattern at least. It is worth mentioning that the prefixes of `WCTimeLineViewController` and `WCContentItemViewTemplateNewSight` are `WC`, I guess it is short for WeChat; the prefixes of `MMTableViewCell` and `MMTableView` are `MM`, I guess it is short for MicroMessenger. The difference of prefixes may be exactly caused by different subprojects and teams. Next, we will focus on `WCDataItem` and try to get the download URL and local path of the `Sight`.

9.2.9 Get target information from `WCDataItem`

Open `WCDataItem.h` and take an overview:

```
@interface WCDataItem : NSObject <NSCoding>
{
    int cid;
    NSString *tid;
    int type;
    int flag;
    NSString *username;
    NSString *nickname;
    int createtime;
    NSString *sourceUrl;
    NSString *sourceUrl2;
    WCLocationInfo *locationInfo;
    BOOL isPrivate;
    NSMutableArray *sharedGroupIDs;
    NSMutableArray *blackUsers;
    NSMutableArray *visibleUsers;
    unsigned long extFlag;
    BOOL likeFlag;
    int likeCount;
    NSMutableArray *likeUsers;
    int commentCount;
    NSMutableArray *commentUsers;
    int withCount;
    NSMutableArray *withUsers;
    WCContentItem *contentObj;
    WCAppInfo *appInfo;
    NSString *publicUserName;
}
```

```

    NSString *sourceUserName;
    NSString *sourceNickName;
    NSString *contentDesc;
    NSString *contentDescPattern;
    int contentDescShowType;
    int contentDescScene;
    WCActionInfo *actionInfo;
    unsigned int hash;
    SnsObject *snsObject;
    BOOL isBidirectionalFan;
    BOOL noChange;
    BOOL isRichText;
    NSMutableDictionary *extData;
    int uploadErrType;
    NSString *statisticsData;
}

+ (id)fromBuffer:(id)arg1;
+ (id)fromServerObject:(id)arg1;
+ (id)fromUploadTask:(id)arg1;
@property(retain, nonatomic) WCActionInfo *actionInfo; // @synthesize actionInfo;
@property(retain, nonatomic) WCAppInfo *appInfo; // @synthesize appInfo;
@property(retain, nonatomic) NSArray *blackUsers; // @synthesize blackUsers;
@property(n nonatomic) int cid; // @synthesize cid;
@property(n nonatomic) int commentCount; // @synthesize commentCount;
@property(retain, nonatomic) NSMutableArray *commentUsers; // @synthesize commentUsers;
- (int)compareDesc:(id)arg1;
- (int)compareTime:(id)arg1;
@property(retain, nonatomic) NSString *contentDesc; // @synthesize contentDesc;
@property(retain, nonatomic) NSString *contentDescPattern; // @synthesize
contentDescPattern;
@property(n nonatomic) int contentDescScene; // @synthesize contentDescScene;
@property(n nonatomic) int contentDescShowType; // @synthesize contentDescShowType;
@property(retain, nonatomic) WCContentItem *contentObj; // @synthesize contentObj;
@property(n nonatomic) int createtime; // @synthesize createtime;
- (void)dealloc;
- (id)description;
- (id)descriptionForKeyPaths;
- (void)encodeWithCoder:(id)arg1;
@property(retain, nonatomic) NSMutableDictionary *extData; // @synthesize extData;
@property(n nonatomic) unsigned long extFlag; // @synthesize extFlag;
@property(n nonatomic) int flag; // @synthesize flag;
- (id)getDisplayCity;
- (id)getMediaWraps;
- (BOOL)hasSharedGroup;
- (unsigned int)hash;
- (id)init;
- (id)initWithCoder:(id)arg1;
@property(n nonatomic) BOOL isBidirectionalFan; // @synthesize isBidirectionalFan;
- (BOOL)isEqual:(id)arg1;
@property(n nonatomic) BOOL isPrivate; // @synthesize isPrivate;
- (BOOL)isRead;
@property(n nonatomic) BOOL isRichText; // @synthesize isRichText;
- (BOOL)isUploadFailed;
- (BOOL)isUploading;
- (BOOL)isValid;
- (id)itemID;
- (int)itemType;
- (id)keyPaths;
@property(n nonatomic) int likeCount; // @synthesize likeCount;

```

```

@property(n nonatomic) BOOL likeFlag; // @synthesize likeFlag;
@property(retain, nonatomic) NSMutableArray *likeUsers; // @synthesize likeUsers;
- (void)loadPattern;
@property(retain, nonatomic) WCLocationInfo *locationInfo; // @synthesize locationInfo;
- (void)mergeLikeUsers:(id)arg1;
- (void)mergeMessage:(id)arg1;
- (void)mergeMessage:(id)arg1 needParseContent:(BOOL)arg2;
@property(retain, nonatomic) NSString *nickname; // @synthesize nickname;
@property(n nonatomic) BOOL noChange; // @synthesize noChange;
- (void)parseContentForNetWithDataItem:(id)arg1;
- (void)parseContentForUI;
- (void)parsePattern;
@property(retain, nonatomic) NSString *publicUserName; // @synthesize publicUserName;
- (id)sequence;
- (void)setCreateTime:(unsigned long)arg1;
- (void)setHash:(unsigned int)arg1;
- (void)setIsUploadFailed:(BOOL)arg1;
- (void)setSequence:(id)arg1;
@property(retain, nonatomic) NSMutableArray *sharedGroupIDs; // @synthesize
sharedGroupIDs;
@property(retain, nonatomic) SnsObject *snsObject; // @synthesize snsObject;
@property(retain, nonatomic) NSString *sourceNickName; // @synthesize sourceNickName;
@property(retain, nonatomic) NSString *sourceUrl2; // @synthesize sourceUrl2;
@property(retain, nonatomic) NSString *sourceUrl; // @synthesize sourceUrl;
@property(retain, nonatomic) NSString *sourceUserName; // @synthesize sourceUserName;
@property(retain, nonatomic) NSString *statisticsData; // @synthesize statisticsData;
@property(retain, nonatomic) NSString *tid; // @synthesize tid;
@property(n nonatomic) int type; // @synthesize type;
@property(n nonatomic) int uploadErrType; // @synthesize uploadErrType;
@property(retain, nonatomic) NSString *username; // @synthesize username;
@property(retain, nonatomic) NSArray *visibleUsers; // @synthesize visibleUsers;
@property(n nonatomic) int withCount; // @synthesize withCount;
@property(retain, nonatomic) NSMutableArray *withUsers; // @synthesize withUsers;
- (id)toBuffer;

@end

```

There are 4 occurrences of “path” and “url” keywords:

```

- (id)descriptionForKeyPaths;
- (id)keyPaths;
@property(retain, nonatomic) NSString *sourceUrl2;
@property(retain, nonatomic) NSString *sourceUrl;

```

Now let’s inspect their return values via LLDB. Repeat the previous operations to trigger the breakpoint on a Sight cell:

```

Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, queue =
'com.apple.main-thread, stop reason = breakpoint 7.1
    frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$$MicroMessenger
    0x2a0920: mov    r11, r0
    0x2a0922: movw  r0, #32442
    0x2a0926: movt  r0, #436
(lldb) ni

```



```

Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212, queue =
'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
  212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
   0x2a0922: movw  r0, #32442
   0x2a0926: movt  r0, #436
   0x2a092a: add   r0, pc
(lldb) po [$r0 descriptionForKeyPaths]
Class name: WCDataItem, addr: 0x80f52b0
tid: 11896185303680028954
username: wxid_hqouu9kgsgw3e6
createtime: 1418135798
commentUsers: (
)
contentObj: <WCContentItem: 0x8724c20>

(lldb) po [$r0 keyPaths]
<__NSArrayI 0x87b5260>(
tid,
username,
createtime,
commentUsers,
contentObj
)

(lldb) po [$r0 sourceUrl2]
nil
(lldb) po [$r0 sourceUrl]
nil

```

Seems there is nothing special in the return values, but WCContentItem has showed up so many times and grabs my attention. Obviously, the meaning of “content” is more specific than “data”, the content of the Sight may be supplied by this object. Now, take a look at WCContentItem.h:

```

@interface WCContentItem : NSObject <NSCoding>
{
    NSString *title;
    NSString *desc;
    NSString *titlePattern;
    NSString *descPattern;
    NSString *linkUrl;
    NSString *linkUrl2;
    int type;
    int flag;
    NSString *username;
    NSString *nickname;
    int createtime;
    NSMutableArray *mediaList;
}

@property(n nonatomic) int createtime; // @synthesize createtime;
- (void)dealloc;
@property(retain, nonatomic) NSString *desc; // @synthesize desc;

```



```

@property(retain, nonatomic) NSString *descPattern; // @synthesize descPattern;
- (void)encodeWithCoder:(id)arg1;
@property(n nonatomic) int flag; // @synthesize flag;
- (id)init;
- (id)initWithCoder:(id)arg1;
- (BOOL)isValid;
@property(retain, nonatomic) NSString *linkUrl; // @synthesize linkUrl;
@property(retain, nonatomic) NSString *linkUrl2; // @synthesize linkUrl2;
@property(retain, nonatomic) NSMutableArray *mediaList; // @synthesize mediaList;
@property(retain, nonatomic) NSString *nickname; // @synthesize nickname;
@property(retain, nonatomic) NSString *title; // @synthesize title;
@property(retain, nonatomic) NSString *titlePattern; // @synthesize titlePattern;
@property(n nonatomic) int type; // @synthesize type;
@property(retain, nonatomic) NSString *username; // @synthesize username;

@end

```

There are 2 occurrences of “url”:

```

@property(retain, nonatomic) NSString *linkUrl;
@property(retain, nonatomic) NSString *linkUrl2;

```

We can get a WCContentItem object via [WCDataItem contentObj], then use LLDB to print the values of the above 2 properties. Repeat the previous operations to trigger the breakpoint on a Sight cell:

```

Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, queue =
'com.apple.main-thread, stop reason = breakpoint 8.1
    frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
    208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$$MicroMessenger
    0x2a0920: mov    r11, r0
    0x2a0922: movw  r0, #32442
    0x2a0926: movt  r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212, queue =
'com.apple.main-thread, stop reason = instruction step over
    frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
    212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
    0x2a0922: movw  r0, #32442
    0x2a0926: movt  r0, #436
    0x2a092a: add   r0, pc
(lldb) po [[[$r0 contentObj] linkUrl]
https://support.weixin.qq.com/cgi-bin/mmsupport-
bin/readtemplate?t=page/common_page__upgrade&v=1
(lldb) po [[[$r0 contentObj] linkUrl2]
nil

```

Type this URL in browser to see what we’ve got, as shown in figure 9-28.



Figure 9-28 `[[r0 contentObj] linkUrl]`

The result is not what we want. Since there is not too much content in `WCContentItem.h`, where could the Sight be? Back to this file, a property named `mediaList` attracts my attention. It is even more accurate than “content”, is the Sight hidden in it? LLDB will answer us. Repeat the previous operations to trigger the breakpoint on a Sight cell:

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$MicroMessenger + 208, queue =
'com.apple.main-thread, stop reason = breakpoint 8.1
  frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$MicroMessenger +
  208
MicroMessenger`___lldb_unnamed_function11980$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$MicroMessenger
  0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$MicroMessenger + 212, queue =
'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$MicroMessenger +
  212
MicroMessenger`___lldb_unnamed_function11980$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
  0x2a0922: movw  r0, #32442
  0x2a0926: movt  r0, #436
  0x2a092a: add   r0, pc
(lldb) po [[r0 contentObj] mediaList]
```

```
<__NSArrayM 0x8725580>C
<WCMediaItem: 0x7e78490>
)
```

Now, a new class `WCMediaItem` appears. Let's check `WCMediaItem.h`:

```
@interface WCMediaItem : NSObject <NSCoding>
{
    NSString *mid;
    int type;
    int subType;
    NSString *title;
    NSString *desc;
    NSString *titlePattern;
    NSString *descPattern;
    NSString *userData;
    NSString *source;
    NSMutableArray *previewUrls;
    WUrl *dataUrl;
    WUrl *lowBandUrl;
    struct CGSize imgSize;
    BOOL likeFlag;
    int likeCount;
    NSMutableArray *likeUsers;
    int commentCount;
    NSMutableArray *commentUsers;
    int withCount;
    NSMutableArray *withUsers;
    int createTime;
}

- (id).cxx_construct;
- (id)cityForData;
@property(n nonatomic) int commentCount; // @synthesize commentCount;
@property(retain, nonatomic) NSMutableArray *commentUsers; // @synthesize commentUsers;
- (id)comparativePathForPreview;
@property(n nonatomic) int createTime; // @synthesize createTime;
@property(retain, nonatomic) WUrl *dataUrl; // @synthesize dataUrl;
- (void)dealloc;
@property(retain, nonatomic) NSString *desc; // @synthesize desc;
@property(retain, nonatomic) NSString *descPattern; // @synthesize descPattern;
- (void)encodeWithCoder:(id)arg1;
- (BOOL)hasData;
- (BOOL)hasPreview;
- (BOOL)hasSight;
- (id)hashPathForString:(id)arg1;
- (id)imageOfSize:(int)arg1;
@property(n nonatomic) struct CGSize imgSize; // @synthesize imgSize;
- (id)init;
- (id)initWithCoder:(id)arg1;
- (BOOL)isValid;
@property(n nonatomic) int likeCount; // @synthesize likeCount;
@property(n nonatomic) BOOL likeFlag; // @synthesize likeFlag;
@property(retain, nonatomic) NSMutableArray *likeUsers; // @synthesize likeUsers;
- (CDStruct_c3b9c2ee)locationForData;
@property(retain, nonatomic) WUrl *lowBandUrl; // @synthesize lowBandUrl;
- (id)mediaID;
- (int)mediaType;
@property(retain, nonatomic) NSString *mid; // @synthesize mid;
- (id)pathForData;
- (id)pathForPreview;
```

```

- (id)pathForSightData;
@property(retain, nonatomic) NSMutableArray *previewUrls; // @synthesize previewUrls;
- (BOOL)saveDataFromData:(id)arg1;
- (BOOL)saveDataFromMedia:(id)arg1;
- (BOOL)saveDataFromPath:(id)arg1;
- (BOOL)savePreviewFromData:(id)arg1;
- (BOOL)savePreviewFromMedia:(id)arg1;
- (BOOL)savePreviewFromPath:(id)arg1;
- (BOOL)saveSightDataFromData:(id)arg1;
- (BOOL)saveSightDataFromMedia:(id)arg1;
- (BOOL)saveSightDataFromPath:(id)arg1;
- (BOOL)saveSightPreviewFromMedia:(id)arg1;
@property(retain, nonatomic) NSString *source; // @synthesize source;
@property(nonaatomic) int subType; // @synthesize subType;
@property(retain, nonatomic) NSString *title; // @synthesize title;
@property(retain, nonatomic) NSString *titlePattern; // @synthesize titlePattern;
@property(nonaatomic) int type; // @synthesize type;
@property(retain, nonatomic) NSString *userData; // @synthesize userData;
@property(nonaatomic) int withCount; // @synthesize withCount;
@property(retain, nonatomic) NSMutableArray *withUsers; // @synthesize withUsers;
- (id)videoStreamForData;
- (id)voiceStreamForData;

@end

```

There are 8 occurrences of “path”:

```

- (id)comparativePathForPreview;
- (id)hashPathForString:(id)arg1;
- (id)pathForData;
- (id)pathForPreview;
- (id)pathForSightData;
- (BOOL)saveDataFromPath:(id)arg1;
- (BOOL)savePreviewFromPath:(id)arg1;
- (BOOL)saveSightDataFromPath:(id)arg1;

```

And 3 occurrences of “url”:

```

@property(retain, nonatomic) WUrl *dataUrl;
@property(retain, nonatomic) WUrl *lowBandUrl;
@property(retain, nonatomic) NSMutableArray *previewUrls;

```

Among those methods and properties, pathForData, pathForPreview and pathForSightData are very likely to return paths; dataUrl, lowBandUrl and previewUrls are very likely to return URLs. Verify our guess ASAP with LLDB, repeat the previous operations to trigger the breakpoint on a Sight cell:

```

Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208, queue =
'com.apple.main-thread, stop reason = breakpoint 8.1
    frame #0: 0x002a091c MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 208:
-> 0x2a091c: blx    0xe08e0c    ;
___lldb_unnamed_function70162$$MicroMessenger
    0x2a0920: mov    r11, r0
    0x2a0922: movw  r0, #32442
    0x2a0926: movt  r0, #436

```

```

(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212, queue =
'com.apple.main-thread, stop reason = instruction step over
    frame #0: 0x002a0920 MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger +
212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessenger + 212:
-> 0x2a0920: mov    r11, r0
    0x2a0922: movw  r0, #32442
    0x2a0926: movt  r0, #436
    0x2a092a: add   r0, pc
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] pathForData]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e89b5c4bc9a/wc/media/5/60/2a16b0
b62baf39924448a74fa03ff2
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] pathForPreview]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e89b5c4bc9a/wc/media/5/7f/cdc793
9813d1a95feda4bed05f9b82
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] pathForSightData]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e89b5c4bc9a/wc/media/5/60/2a16b0
b62baf39924448a74fa03ff2.mp4
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] dataUrl]
type[1], url[http://vcloud1023.tc.qq.com/1023_0114929ce86949a8bfb6f7b46b6b39b8.f0.mp4]
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] lowBandUrl]
nil
(lldb) po [[[$r0 contentObj] mediaList] objectAtIndex:0] previewUrls]
<__NSArrayM 0x8725950>(
type[1],
url[http://mmsns.qpic.cn/mmsns/WiaWbRORjphSUXcNL3dNsVLDibRZ9oufPnXeJqZdLG4xhND43M87sh7DR
cxttVPxA0/0]
)

```

From the file names, I am pretty sure that they are the Sight information we've been looking for. Whatever it is ssh or iFunBox that opens the local files; whether it be MobileSafari or Chrome that opens the URL, you can come to these conclusions:

- The value of pathForData is the local path of the Sight without suffix.
- The value of pathForPreview is the path of the Sight's preview image without suffix.
- The value of pathForSightData is the local path of the Sight with suffix.
- The value of dataUrl is the Internet URL of the Sight.
- The value of lowBandUrl is nil, but I guess this value is not nil when the network condition is not good. In order to save bandwidth, file from this URL may be smaller than file from dataURL on size.
- The value of previewUrls is the Internet URL of the Sight's preview images.

The prototyping of tweak is finished for now. Let's comb our thoughts before coding.

9.3 Result interpretation

This practice covers Cycript, IDA and LLDB, we've prototyped the tweak without strictly deducing the execution logic of WeChat. Now I will do a brief summary of our thoughts.

1. Add a long press gesture to Sight view

Because there's already a long press gesture on Sight view, there's no need to reinvent the wheel, we just need to find the existing one and hook it. With Reveal, we can locate the Sight view easily, thus find the action selector of the long press gesture. What is worth mentioning is that the action selector will be called twice, leading to inefficiency. We need to take this situation into consideration when writing tweak, and compose a proper condition to make the method execute only once.

2. Find the Sight object in C

Although the MVC design pattern says that M can be accessed through C, in this example, we cannot find any obvious methods in C to access M. Therefore, we've started with the basic data source method `tableView:cellForRowAtIndexPath:` to find the suspicious data source of a cell, then looked through suspicious properties and methods in headers to locate the Sight object, and finally got the wanted information. Perhaps the procedure was not so rigorous, but we reached our goal and saved our time, it was not bad, right?

9.4 Tweak writing

The target of this section is to replace the options of the original long press menu with "Save to Disk" and "Copy URL". With a well-constructed prototype, we don't have to explain it any further, let's get coding now.

9.4.1 Create tweak project "iOSREWVideoDownloader" using Theos

The Theos commands are as follows:

```
hangcom-mba:Documents sam$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/cyidget
[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): iOSREWCVideoDownloader
Package Name [com.yourcompany.iosrewcvideodownloader]: com.iosre.iosrewcvideodownloader
Author/Maintainer Name [sam]: sam
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]: com.tencent.xin
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: MicroMessenger
Instantiating iphone/tweak in iosrewcvideodownloader/...
Done.

```

9.4.2 Compose iOSREWCVideoDownloader.h

The finalized iOSREWCVideoDownloader.h looks like this:

```

@interface WCContentItem : NSObject
@property (retain, nonatomic) NSMutableArray *mediaList;
@end

@interface WCDataItem : NSObject
@property (retain, nonatomic) WCContentItem *contentObj;
@end

@interface WCUrl : NSObject
@property (retain, nonatomic) NSString *url;
@end

@interface WCMediaItem : NSObject
@property (retain, nonatomic) WCUrl *dataUrl;
- (id)pathForSightData;
@end

@interface WCContentItemViewTemplateNewSight : UIView
- (WCMediaItem *)iOSREMediaItemFromSight;
- (void)iOSREOnSaveToDisk;
- (void)iOSREOnCopyURL;
@end

@interface MMServiceCenter : NSObject
+ (id)defaultCenter;
- (id)getService:(Class)arg1;
@end

@interface WCFacade : NSObject
- (WCDataItem *)getTimelineDataItemOfIndex:(int)arg1;
@end

@interface WCTimeLineViewController : NSObject
- (int)calcDataItemIndex:(int)arg1;
@end

@interface MMTableViewCell : UITableViewCell
@end

```



```
@interface MMTableView : UITableView
@end
```

This header is composed by picking snippets from other class-dump headers. The existence of this header is simply for avoiding any warnings or errors when compiling the tweak.

9.4.3 Edit Tweak.xml

The finalized Tweak.xml looks like this:

```
#import "iOSREWCVideoDownloader.h"

static MMTableViewCell *iOSRECell;
static MMTableView *iOSREView;
static WCTimeLineViewController *iOSREController;

%hook WCContentItemViewTemplateNewSight
%new
- (WCMediaItem *)iOSREMediaItemFromSight
{
    id responder = self;
    while (![responder isKindOfClass:NSClassFromString(@"WCTimeLineViewController")])
    {
        if ([responder isKindOfClass:NSClassFromString(@"MMTableViewCell")])
        iOSRECell = responder;
        else if ([responder isKindOfClass:NSClassFromString(@"MMTableView")])
        iOSREView = responder;
        responder = [responder nextResponder];
    }
    iOSREController = responder;
    if (!iOSRECell || !iOSREView || !iOSREController)
    {
        NSLog(@"iOSRE: Failed to get video object.");
        return nil;
    }
    NSIndexPath *indexPath = [iOSREView indexPathForCell:iOSRECell];
    int itemIndex = [iOSREController calcDataItemIndex:[indexPath section]];
    WCFacade *facade = [(MMServiceCenter *)[%c(MMServiceCenter) defaultCenter]
getService:[%c(WCFacade) class]];
    WCDataItem *dataItem = [facade getTimelineDataItemOfIndex:itemIndex];
    WCContentItem *contentItem = dataItem.contentObj;
    WCMediaItem *mediaItem = [contentItem.mediaList count] != 0 ?
(contentItem.mediaList)[0] : nil;
    return mediaItem;
}

%new
- (void)iOSREOnSaveToDisk
{
    NSString *localPath = [[self iOSREMediaItemFromSight] pathForSightData];
    UISaveVideoAtPathToSavedPhotosAlbum(localPath, nil, nil, nil);
}

%new
- (void)iOSREOnCopyURL
{
    UIPasteboard *pasteboard = [UIPasteboard generalPasteboard];
    pasteboard.string = [self iOSREMediaItemFromSight].dataUrl.url;
}
```



```

}

static int iOSRECounter;

- (void)onLongTouch
{
    iOSRECounter++;
    if (iOSRECounter % 2 == 0) return;

    [self becomeFirstResponder];

    UIBarButtonItem *saveToDiskMenuItem = [[UIBarButtonItem alloc] initWithTitle:@"Save to Disk"
action:@selector(iOSREOnSaveToDisk)];
    UIBarButtonItem *copyURLMenuItem = [[UIBarButtonItem alloc] initWithTitle:@"Copy URL"
action:@selector(iOSREOnCopyURL)];

    UINavigationController *menuController = [UINavigationController sharedMenuController];
    [menuController setMenuItems:@[saveToDiskMenuItem, copyURLMenuItem]];
    [menuController setTargetRect:CGRectZero inView:self];
    [menuController setMenuVisible:YES animated:YES];

    [saveToDiskMenuItem release];
    [copyURLMenuItem release];
}
%end

```

9.4.4 Edit Makefile and control files

The finalized Makefile looks like this:

```

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

include theos/makefiles/common.mk

TWEAK_NAME = iOSREWCVideoDownloader
iOSREWCVideoDownloader_FILES = Tweak.xm
iOSREWCVideoDownloader_FRAMEWORKS = UIKit

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 MicroMessenger"

```

The finalized control looks like this:

```

Package: com.iosre.iosrewcvideodownloader
Name: iOSREWCVideoDownloader
Depends: mobilessubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Play with Sight!
Maintainer: sam
Author: sam
Section: Tweaks
Homepage: http://bbs.iosre.com

```

9.4.5 Test

Compile and install the tweak, then launch WeChat and long press a Sight, it will show our custom menu, as shown in figure 9-29.

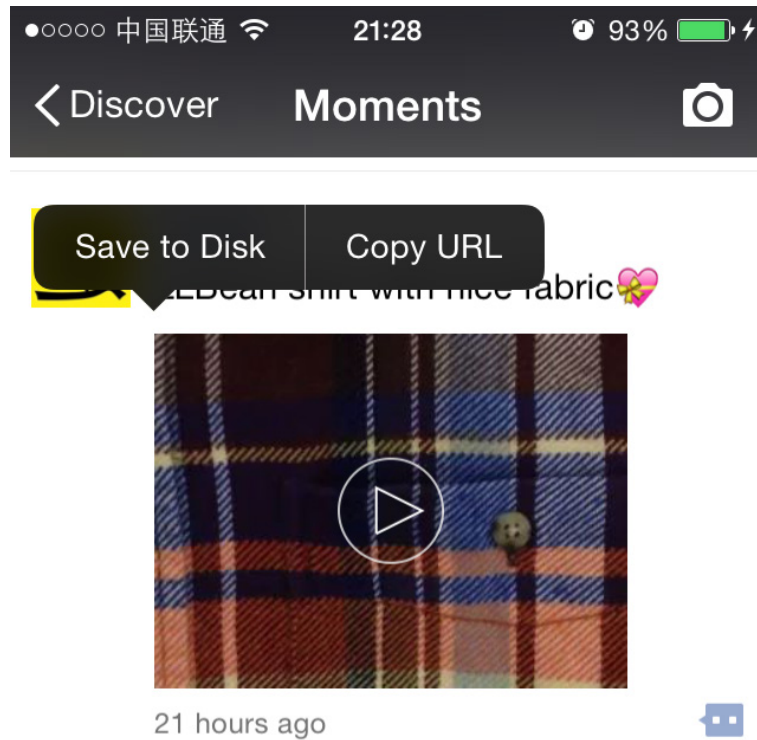


Figure 9-29 Custom menu

Click “Save to Disk”, the Sight will be saved to local album, as shown in figure 9-30.

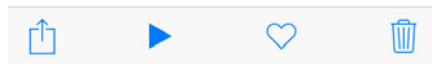
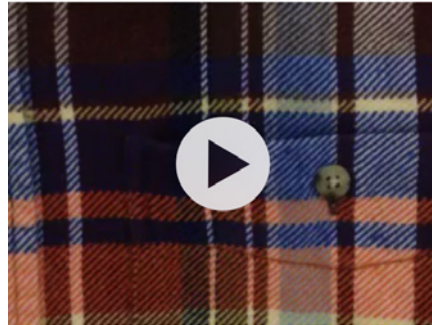
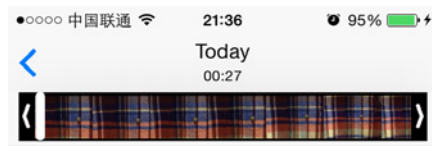


Figure 9-30 Save the Sight to local album

Click “Copy URL”, and paste it in OPlayer (or any other Apps that support online video playing), as shown in figure 9-31.

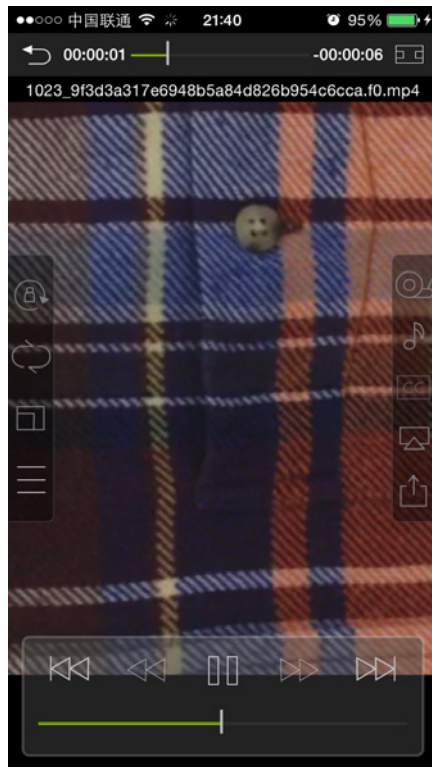


Figure 9-31 Play online Sight in OPlayer

All functions work as expected, mission accomplished!

9.5 Easter eggs

9.5.1 Find the Sight in UIMenuItem

In section 9.2.7, we've successfully found the Sight from WCTimeLineViewController. However, the whole process was not smooth: we haven't managed to find a direct way of accessing M via C, so we "had to" find some clues from tableView:cellForRowAtIndexPath: to meet our needs. If we jump out of MVC and try to think from the view of WeChat itself, things may get much easier.

Think with me: Long press the Sight view, a menu shows. Choose the menu option, a corresponding operation will be carried out on the Sight. In other words, there may be Sight related clues in the action selector of UIMenuItem. In figure 9-11, we have already seen the action selector of "Favorite", which is onFavoriteAdd:, let's check its implementation in IDA, as shown in figure 9-32.

```
; WCContentItemViewTemplateNewSight - (void)onFavoriteAdd:(id)
; Attributes: bp-based frame

; void __cdecl -[WCContentItemViewTemplateNewSight onFavoriteAdd:]
__WCContentItemViewTemplateNewSight_onFavoriteAdd__

var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB       SP, SP, #0x10
MOV       R10, R0
MOV       R0, #(off_1A002FC - 0x21EAF4) ; off_1A002FC
MOVW     R1, #(:lower16:(selRef_contentObj - 0x21EAF4))
ADD       R0, PC ; off_1A002FC
MOVT.W   R1, #(:upper16:(selRef_contentObj - 0x21EAF4))
ADD       R1, PC ; selRef_contentObj
LDR       R0, [R0] ; WCDataItem *_oDataItem;
LDR       R1, [R1] ; "contentObj"
LDR       R4, [R0]
LDR.W    R0, [R10,R4]
BLX.W    j__objc_msgSend
MOV       R5, R0
MOV       R0, #(selRef_mediaList - 0x21EB14) ; selRef_mediaL
ADD       R0, PC ; selRef_mediaList
LDR       R6, [R0] ; "mediaList"
MOV       R0, R5
MOV       R1, R6
BLX.W    j__objc_msgSend
```

Figure 9-32 [WCContentItemViewTemplateNewSight onFavoriteAdd:]

From figure 9-32, we see our familiar WCDataItem, contentObj and mediaList at the beginning of this method. If we've started with this method, the whole analysis workload would be reduced by half at least. It more or less enlightens us that although MVC design pattern is a common trail of thinking in iOS App reverse engineering, if we can occasionally think off the track, we may get something unexpected and have more fun.

9.5.2 Historical transition of WeChat's headers count

From the historical transition of WeChat's headers count as figure 9-33 to figure 9-38 show, we can see how WeChat becomes excellent step by step. A journey of a thousand miles begins with a single step, kudos to WeChat!

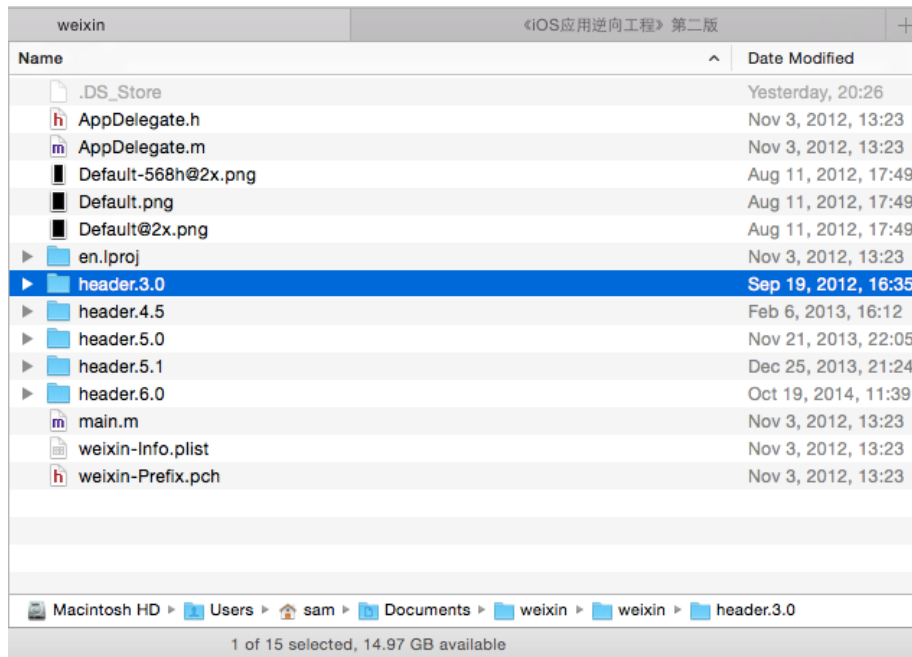


Figure 9-23 Headers directories of different WeChat versions

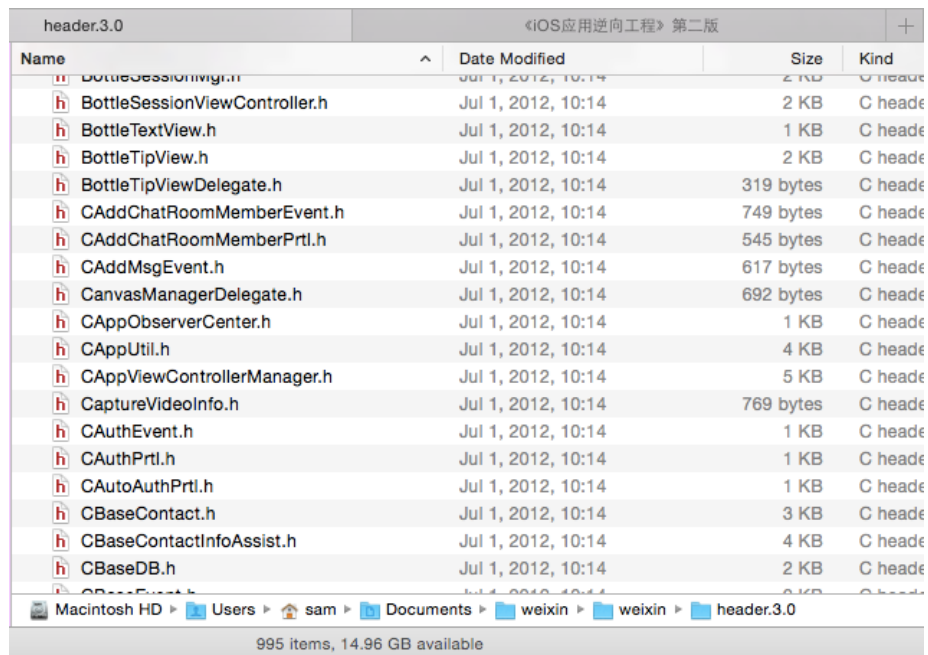


Figure 9-34 WeChat 3.0, 995 headers

header.4.5		«iOS应用逆向工程» 第二版	+
Name	^	Date Modified	
CMessageMgr.h		Feb 6, 2013, 16:12	
CMessageNodeData.h		Feb 6, 2013, 16:12	
CMessageWrap.h		Feb 6, 2013, 16:12	
CMMDB.h		Feb 6, 2013, 16:12	
CMMDBResultNew.h		Feb 6, 2013, 16:12	
CMMNotificationCenter.h		Feb 6, 2013, 16:12	
CMMVector.h		Feb 6, 2013, 16:12	
CModDisturbSettingEvent.h		Feb 6, 2013, 16:12	
CModifyHeadImgEvent.h		Feb 6, 2013, 16:12	
CModifyHeadImgPrtl.h		Feb 6, 2013, 16:12	
CModUserImgWrap.h		Feb 6, 2013, 16:12	
CModUsrInfoEvent.h		Feb 6, 2013, 16:12	
CMultiEvent.h		Feb 6, 2013, 16:12	
CNetworkMgr.h		Feb 6, 2013, 16:12	
CNetworkStatus.h		Feb 6, 2013, 16:12	
CNetworkStatusExt-Protocol.h		Feb 6, 2013, 16:12	
CNetworkStatusMgr.h		Feb 6, 2013, 16:12	
CNetworkStatusReportArchive.h		Feb 6, 2013, 16:12	
CNetworkStatusReportOplogEvent.h		Feb 6, 2013, 16:12	
Macintosh HD > Users > sam > Documents > weixin > weixin > header.4.5			
2,267 items, 14.96 GB available			

Figure 9-35 WeChat 4.5, 2267 headers

header.5.0		«iOS应用逆向工程» 第二版	+
Name	^	Date Modified	
CCTransitionZoomFlipX.h		Nov 21, 2013, 22:05	
CCTransitionZoomFlipY.h		Nov 21, 2013, 22:05	
CCTurnOffTiles.h		Nov 21, 2013, 22:05	
CCTwirl.h		Nov 21, 2013, 22:05	
CCUIViewWrapper.h		Nov 21, 2013, 22:05	
CCWaves.h		Nov 21, 2013, 22:05	
CCWaves3D.h		Nov 21, 2013, 22:05	
CCWavesTiles3D.h		Nov 21, 2013, 22:05	
CDAsynchBufferLoader.h		Nov 21, 2013, 22:05	
CDAsynchInitialiser.h		Nov 21, 2013, 22:05	
CDAudioInterruptProtocol-Protocol.h		Nov 21, 2013, 22:05	
CDAudioInterruptTargetGroup.h		Nov 21, 2013, 22:05	
CDAudioManager.h		Nov 21, 2013, 22:05	
CDAudioTransportProtocol-Protocol.h		Nov 21, 2013, 22:05	
CDBufferLoadRequest.h		Nov 21, 2013, 22:05	
CDBufferManager.h		Nov 21, 2013, 22:05	
CDFloatInterpolator.h		Nov 21, 2013, 22:05	
CDirectSend.h		Nov 21, 2013, 22:05	
CDLongAudioSource.h		Nov 21, 2013, 22:05	
Macintosh HD > Users > sam > Documents > weixin > weixin > header.5.0			
3,734 items, 14.96 GB available			

Figure 9-36 WeChat 5.0, 3734 headers

header.5.1		《iOS应用逆向工程》第二版		+
Name	^	Date Modified		
h	IVoiceReminderExt-Protocol.h	Dec 25, 2013, 21:24		
h	IVoiceSearchExt-Protocol.h	Dec 25, 2013, 21:24		
h	IVOICEExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWOIPModeSwitchExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWOIPSyncExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWOIPUILogicMgrExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWCMailControlLogicExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWCOfflinePayLogicMgrExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWCPayControlLogicExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWebViewAskAuthorizationLogicExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWXPresentExt-Protocol.h	Dec 25, 2013, 21:24		
h	IWXTalkExt-Protocol.h	Dec 25, 2013, 21:24		
h	JailBreakHelper.h	Dec 25, 2013, 21:24		
h	JKArray.h	Dec 25, 2013, 21:24		
h	JKDictionary.h	Dec 25, 2013, 21:24		
h	JKDictionaryEnumerator.h	Dec 25, 2013, 21:24		
h	JKSerializer.h	Dec 25, 2013, 21:24		
h	JoinTrackRoomRequest.h	Dec 25, 2013, 21:24		
h	JoinTrackRoomResponse.h	Dec 25, 2013, 21:24		

Macintosh HD > Users > sam > Documents > weixin > weixin > header.5.1

3,537 items, 14.96 GB available

Figure 9-37 WeChat 5.1, 3537 headers

header.6.0		《iOS应用逆向工程》第二版		+
Name	^	Date Modified	Size	Kind
h	SequencePageScrollView.h	Oct 19, 2014, 11:39	2 KB	C header
h	SequencePageScr...taSource-Protocol.h	Oct 19, 2014, 11:39	452 bytes	C header
h	ServiceAppData.h	Oct 19, 2014, 11:39	2 KB	C header
h	ServiceAppListViewController.h	Oct 19, 2014, 11:39	1 KB	C header
h	ServiceAppsLogicImpl.h	Oct 19, 2014, 11:39	1 KB	C header
h	SessionAbstractDB.h	Oct 19, 2014, 11:39	627 bytes	C header
h	SessionCellLayoutParam.h	Oct 19, 2014, 11:39	2 KB	C header
h	SessionDelegate-Protocol.h	Oct 19, 2014, 11:39	776 bytes	C header
h	SessionSelectController.h	Oct 19, 2014, 11:39	5 KB	C header
h	SessionSelectContr...delegate-Protocol.h	Oct 19, 2014, 11:39	611 bytes	C header
h	SessionSortCache.h	Oct 19, 2014, 11:39	1 KB	C header
h	SessionSortLogic.h	Oct 19, 2014, 11:39	784 bytes	C header
h	SessionStorageSetting.h	Oct 19, 2014, 11:39	859 bytes	C header
h	SessionTranslateInfos.h	Oct 19, 2014, 11:39	919 bytes	C header
h	SetAPPListRequest.h	Oct 19, 2014, 11:39	1 KB	C header
h	SetAPPListResponse.h	Oct 19, 2014, 11:39	954 bytes	C header
h	SetAppSettingRequest.h	Oct 19, 2014, 11:39	1 KB	C header
h	SetAppSettingResponse.h	Oct 19, 2014, 11:39	1 KB	C header
h	SetDeviceSafeViewController.h	Oct 19, 2014, 11:39	2 KB	C header

Macintosh HD > Users > sam > Documents > weixin > weixin > header.6.0

5,225 items, 14.96 GB available

Figure 9-38 WeChat 6.0, 5225 headers

From WeChat 3.0 to WeChat 6.0, the number of headers has increased from less than 1,000 to more than 5,000, which is a 5+ times amplification. With the global popularity of WeChat, its headers count is expected to surpass 10,000 sooner or later.

9.6 Conclusion

WeChat is the target of this chapter, we've enriched Sight by adding 2 new features, i.e. "Save to Disk" and "Copy URL". As a powerful platform, WeChat possesses complicated

structure and large amount of code; it was beautifully designed with clearly separated modules and well organized code. We have already learnt so much from it by just going through its headers, we can even see the different coding styles of different developers. I believe all of us can benefit a lot from studying WeChat's design pattern by reversing it. We will discuss what we find reversing WeChat on <http://bbs.iosre.com>, you are welcome to join us.

Practice 4: Detect And Send iMessages

10.1 iMessage

iMessage is an IM service that Apple implements seamlessly into the stock Messages App (hereafter referred to as MobileSMS). It was born in iOS 5 and became better ever since. Whether it's plain text, image, audio, or even video, iMessage can handle them with high speed, security and efficiency. We all love iMessage!

Among all functions of iMessage, detecting if an address supports iMessage, and sending an iMessage are 2 most interesting functions without doubt. Surprisingly, there are even companies that make profit from sending spam iMessages, and that's one of the main reasons that I developed the Cydia tweak "SMSNinja". You can't understand how to defense without knowing how to attack. In this chapter, we will combine all knowledge points we've studied by far and start from scratch to reverse the functions of detecting and sending iMessages, as sublimation of the book. All the following operations are finished on iPhone 5, iOS 8.1.

10.2 Detect if a number or email address supports iMessage

As usual, before using tools to start reverse engineering, let's analyze the abstract target and concretize it, then form the idea and carry it out.

10.2.1 Observe MobileSMS and look for cut-in points

As MobileSMS users, we will notice that during the process of sending a message, Apple will show us if we're currently sending an SMS or iMessage through the changes of texts and colors, say:

- When you start to compose a message by just finishing recipient's address without entering the message body, if iOS detects that the address is iMessage supportive, the placeholder will change from "Text Message" to "iMessage", as shown in figure 10-1.



Figure 10- 1 Change of placeholder

- When you start to input message body, if the address only supports SMS, the “Send” button beside the input box will be green; if it supports iMessage, the button will be blue.
- When you hit the “Send” button to send this message, if this is an SMS, the message bubble will be green, otherwise it will be blue.

These 3 phenomena will appear one after another. Since the process of detecting iMessage has already happened in the 1st phenomenon, it is enough to act as the cut-in point. We’ll focus on the 1st phenomenon from now on.

After locating the cut-in point, let’s think together to concretize the phenomenon into a reverse engineering idea.

What we can observe is visible on UI, i.e. the change from “Text Message” to “iMessage”. As we’ve already known, visualizations on UI don’t come from nowhere but the data source, hence by referring to visualizations, we can find the data source, i.e. placeholder, using Cycrypt.

Placeholder doesn’t come from nowhere but its data source either. The reason why placeholder changes is that its data source (data source’s data source, and so on. Hereafter referred to as the Nth data source) changes, like the following pseudo code presents:

```
id dataSource = ?;
id a = function(dataSource);
id b = function(a);
id c = function(b);
...
id z = function(y);
NSString *placeholder = function(z);
```

From the above snippet we can know that the original data source is dataSource, its change in turn results in the change of placeholder. Well, what’s the original data source? In the 1st phenomenon, our only input is the address, so the original data source is sure to be the address.

For detecting iMessages, there should be a conversion from dataSource to placeholder, and this conversion process is the actual meaning of “detecting iMessages” as well our target in this section, as shown in figure 10-2.

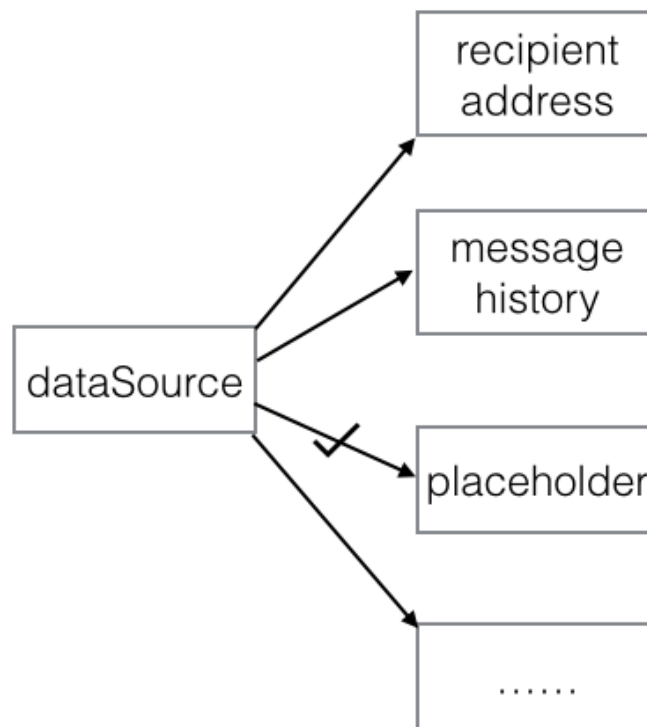


Figure 10- 2 Conversion from dataSource to placeholder

You may wonder, since figure 10-2 is so straightforward and dataSource is already known, why don't we start from it directly and trace placeholder? Then we can reproduce the process and achieve our goal. Actually, we're not living in a fairy tale, the real world is usually not idealized. For one thing, we don't have the source code of MobileSMS; for the other thing, in general cases, the conversion is much more complex, as can be illustrated in figure 10-3.

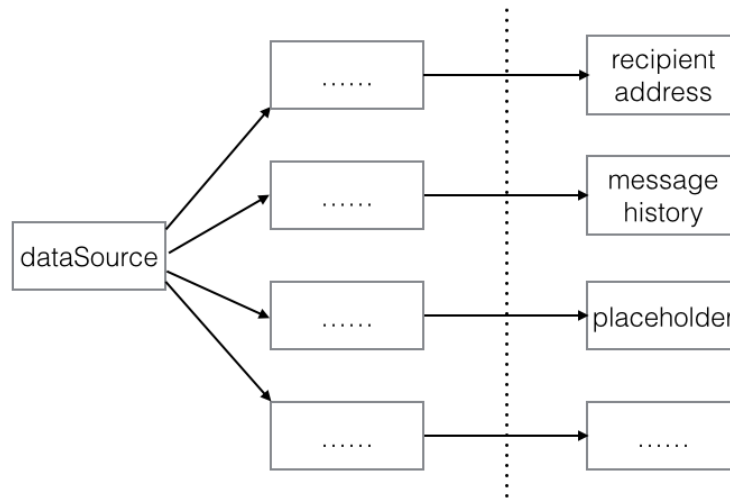


Figure 10- 3 Real conversion from dataSource to placeholder

dataSource must be converted multiple times to become placeholder, their relationship is very intricate. If we start from dataSource, how can we know which of the 4 routines leads to placeholder? Under such circumstance, because there is only one placeholder, it's more efficient and doable to start from placeholder and trace back to dataSource to reproduce the whole process.

In conclusion, the ideas of this practice are: first use Cycrypt to locate placeholder, then trace the Nth data source of placeholder using IDA and LLDB, until we get dataSource. Finally reproduce the process of how dataSource becomes placeholder. Looks as easy as a regular 3-step job? Actions not only speak louder than words, but also implement harder than words, you'll feel it soon.

10.2.2 Find placeholder using Cycrypt

Open MobileSMS and create a new message; enter "bbs.iosre.com" as the address and then tap "return" on keyboard to end editing, as shown in figure 10-4.

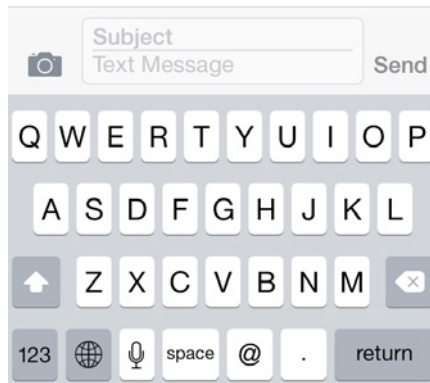
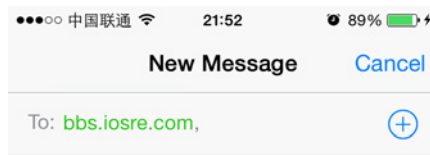


Figure 10- 4 Create a new message

Since we are using Cycrypt to find placeholder, first we should find the view that displays the current placeholder “Text Message”; they must have a close connection, so get one, get the other. Right? Let’s do it.

```
FunMaker-5:~ root# cycrypt -p MobileSMS
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
```

The view hierarchy of keyWindow is quite rich in content, so we’re not pasting it here. If you search “Text Message” in the output, you will find that there isn’t any match. Why? Maybe you’ve already guessed the answer: “Text Message” isn’t in keyWindow. For verification, let’s see how many windows are there in the current view:

```
cy# [UIApp windows]
@[#"<UIWindow: 0x1575ca10; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x15629c60>; layer = <UIWindowLayer: 0x156e36f0>>","#<UITextEffectsWindow: 0x1579ab70; frame = (0 0; 320 568); opaque = NO; autoresize = W+H; gestureRecognizers = <NSArray: 0x1579b300>; layer = <UIWindowLayer: 0x1579adf0>>","#<CKJoystickWindow: 0x1552bf90; baseClass = UIAutoRotatingWindow; frame = (0 0; 320 568); hidden = YES; gestureRecognizers = <NSArray: 0x1552b730>; layer = <UIWindowLayer: 0x1552bdc0>>","#<UITextEffectsWindow: 0x1683a2e0; frame = (0 0; 320 568); hidden = YES; gestureRecognizers = <NSArray: 0x1688b9e0>; layer = <UIWindowLayer: 0x168b9ad0>>"]
```

As we can see, each item starting with “#” is a window, there are 4 of them, and the 1st is keyWindow. Well, which one contains “Text Message”? As the names suggest, the 2nd and 4th windows with the keyword “Text” in their names may be our targets. However, the 4th

window is even invisible because of its hidden property. This leaves us with the 2nd window only, let's test it out in Cycrypt.

```
cy# [#0x1579ab70 setHidden:YES]
```

After this command, not only the input box but also the whole keyboard are hidden, as shown in figure 10-5:

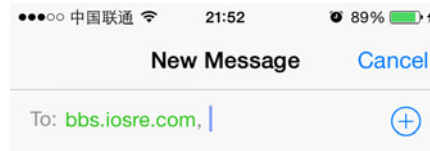


Figure 10- 5 The bottom half is hidden

Now we can confirm that “Text Message” is located right in this window. Keep looking for it using Cycrypt.

```
cy# [#0x1579ab70 setHidden:NO]
cy# [#0x1579ab70 subviews]
@[#"<UIInputSetContainerView: 0x1551fb10; frame = (0 0; 320 568); autoresize = W+H;
layer = <CALayer: 0x1551f950>>"]
cy# [#0x1551fb10 subviews]
@[#"<UIInputSetHostView: 0x1551f5e0; frame = (0 250; 320 318); layer = <CALayer:
0x1551f480>>"]
cy# [#0x1551f5e0 subviews]
@[#"<UIKBInputBackdropView: 0x16827620; frame = (0 65; 320 253); userInteractionEnabled
= NO; layer = <CALayer: 0x1681c3f0>>","#<_UIKBCompatInputView: 0x157b88d0; frame = (0
65; 320 253); layer = <CALayer: 0x157b8a10>>","#<CKMessageEntryView: 0x1682ca50; frame =
(0 0; 320 65); opaque = NO; autoresize = W; layer = <CALayer: 0x168ec520>>"]
```

There are 3 subviews in the above code, which one does “Text Message” reside? Let's test them one by one.

```
cy# [#0x16827620 setHidden:YES]
```

After the above command, the view looks like figure 10-6, indicating that this view is just

keyboard background.

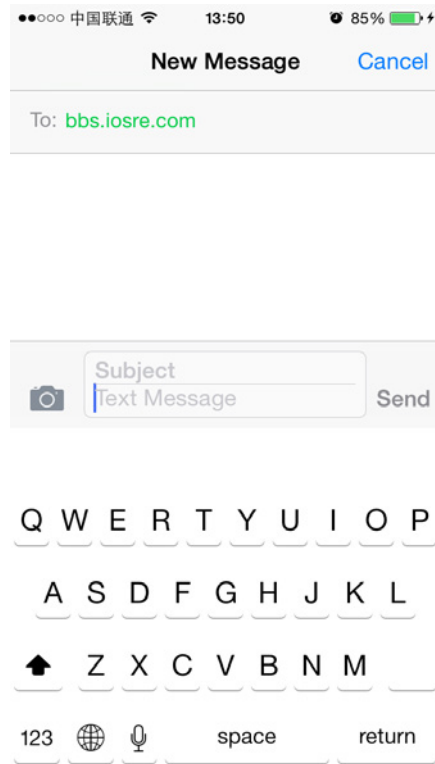


Figure 10- 6 Keyboard background is hidden

```
cy# [#0x16827620 setHidden:NO]  
cy# [#0x157b88d0 setHidden:YES]
```

After these 2 commands, the view changes to figure 10-7.

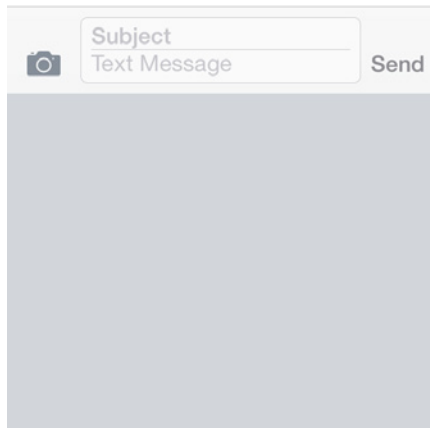
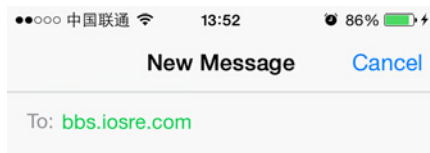


Figure 10- 7 Keyboard is hidden

OK, this view is keyboard itself. Thus, we can infer that `UIKBInputBackdropView` and `UIKBCompatInputView` work together to form a keyboard's view. This official design mode can be a good reference for 3rd-party keyboard developers and WinterBoard theme makers.

Now that there is the last subview with an explicit name “`CKMessageEntryView`”, waiting for our test:

```
cy# [#0x157b88d0 setHidden:NO]
cy# [#0x1682ca50 setHidden:YES]
```

The view looks like figure 10-8 after the above commands.

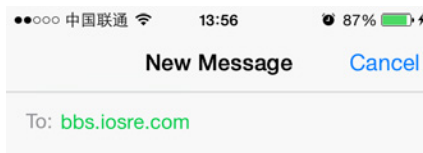


Figure 10- 8 Message entry view is hidden

According to the result, we know that “Text Message” is inside CKMessageEntryView. Go on.

```
cy# [#0x1682ca50 setHidden:NO]
cy# [#0x1682ca50 subviews]
@[#"<_UIBackdropView: 0x168ce210; frame = (0 0; 320 65); opaque = NO; autoresize = W+H;
userInteractionEnabled = NO; layer = <_UIBackdropViewLayer: 0x168f5300>>","#"<UIView:
0x168d2b70; frame = (0 0; 320 0.5); layer = <CALayer: 0x168d2be0>>","#"<UIButton:
0x1684b240; frame = (266 27; 53 33); opaque = NO; layer = <CALayer:
0x168d64b0>>","#"<UIButton: 0x168b88b0; frame = (266 30; 53 26); hidden = YES; opaque =
NO; gestureRecognizers = <NSArray: 0x16840030>; layer = <CALayer:
0x16858420>>","#"<UIButton: 0x16833ac0; frame = (15 33.5; 25 18.5); opaque = NO;
gestureRecognizers = <NSArray: 0x1682d9b0>; layer = <CALayer:
0x16838780>>","#"<UITextFieldRoundedRectBackgroundViewNeue: 0x168fba00; frame = (55 8;
209.5 49.5); opaque = NO; userInteractionEnabled = NO; layer = <CALayer:
0x1682da50>>","#"<UIView: 0x168dcf10; frame = (55 8; 209.5 49.5); clipsToBounds = YES;
opaque = NO; layer = <CALayer: 0x168e4170>>","#"<CKMessageEntryWaveformView: 0x1571b710;
frame = (15 25.5; 251 35); alpha = 0; opaque = NO; userInteractionEnabled = NO; layer =
<CALayer: 0x1578fc90>>"]
```

Again, let’s hide these views one by one to locate “Text Message”, and I’ll leave the work to you as an exercise. After locating “UIView: 0x168dcf10” (Notice, it’s the 2nd UIView object) as the target, let’s continue with its subviews.

```
cy# [#0x168dcf10 subviews]
@[#"<CKMessageEntryContentView: 0x16389000; baseClass = UIScrollView; frame = (3 -4;
203.5 57.5); clipsToBounds = YES; opaque = NO; gestureRecognizers = <NSArray:
0x168f0730>; layer = <CALayer: 0x168e41a0>; contentOffset: {0, 0}; contentSize: {203.5,
57}>"]
```

There is only one subview, keep digging.

```

cy# [#0x16389000 subviews]
@[#"<CKMessageEntryRichTextView: 0x16295200; baseClass = UITextView; frame = (0 20.5; 203.5 36.5); text = ''; clipsToBounds = YES; opaque = NO; gestureRecognizers = <NSArray: 0x168f5a60>; layer = <CALayer: 0x168f59c0>; contentOffset: {0, 0}; contentSize: {203.5, 36.5}>"#,#"<CKMessageEntryTextView: 0x15ad2a00; baseClass = UITextView; frame = (0 0; 203.5 36.5)>"#,#"<UIView: 0x157e9160; frame = (5 28; 193.5 0.5); layer = <CALayer: 0x15733bd0>>"#,#"<UIImageView: 0x157308d0; frame = (-0.5 55; 204 2.5); alpha = 0; opaque = NO; autoresize = TM; userInteractionEnabled = NO; layer = <CALayer: 0x15730950>>"#,#"<UIImageView: 0x157ef530; frame = (201 0; 2.5 57.5); alpha = 0; opaque = NO; autoresize = LM; userInteractionEnabled = NO; layer = <CALayer: 0x157ef5b0>>"]

```

By hiding these views one by one, we can find that when executing “[#0x16295200 setHidden:YES]”, only “Text Message” is hidden, other control objects are not affected, as shown in figure 10-9.

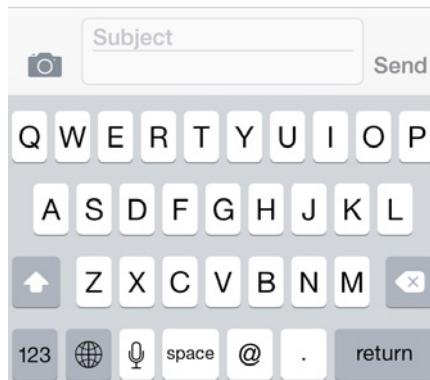
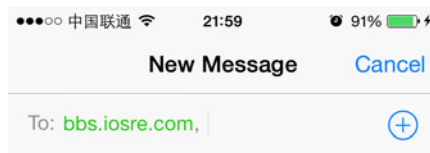


Figure 10- 9 placeholder is hidden

It means that CKMessageEntryRichTextView is our target view. Open CKMessageEntryRichTextView.h and see if there’s any “placeholder”, as shown in figure 10-10.

```

3 //
4 //   class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2013 by Steve Nygard.
5 //
6
7 #import <ChatKit/CKMessageEntryTextView.h>
8
9 #import "NSTextStorageDelegate.h"
10
11 @class CKComposition, NSMutableDictionary, NSString;
12
13 @interface CKMessageEntryRichTextView : CKMessageEntryTextView <NSTextStorageDelegate>
14 {
15     BOOL _balloonColor;
16     NSMutableDictionary *_mediaObjects;
17     NSMutableDictionary *_composeImages;
18     CKComposition *_pasteboardComposition;
19     int _pasteboardChangeCount;
20 }
21
E486: Pattern not found: \\cplaceholder

```

Figure 10- 10 CKMessageEntryRichTextView.h

Unluckily, we cannot find placeholder in CKMessageEntryRichTextView.h. Was there something wrong in our deduction? Not really. Let's have a look at its superclass, i.e. CKMessageEntryTextView, as shown in figure 10-11.

```

11 @interface CKMessageEntryTextView : UITextView
12 {
13     BOOL _showingDictationPlaceholder;
14     NSString *_autocorrectionContext;
15     NSString *_responseContext;
16     UILabel *_placeholderLabel;
17 }
18
19 @property(retain, nonatomic) UILabel *_placeholderLabel; //
20 @property(copy, nonatomic) NSString *_responseContext; // @s
21 @property(copy, nonatomic) NSString *_autocorrectionContext;
22 @property(n nonatomic, getter=isShowingDictationPlaceholder)
Placeholder;
23 - (void)textViewDidChange:(id)arg1;
24 - (void)updateTextView;
25 @property(readonly, nonatomic, getter=isSingleLine) BOOL si
26 @property(copy, nonatomic) NSString *_placeholderText;
27 @property(copy, nonatomic) NSAttributedString *_compositionT
28 - (void)removeDictationResultPlaceholder:(id)arg1 willInsert

```

Figure 10- 11 CKMessageEntryTextView.h

Aha, there are lots of placeholders in this file. Among them placeholderLabel and placeholderText are quite noticeable, is anyone of them our target placeholder? Let's verify with Cycript:

```
cy# [#0x16295200 setPlaceholderText:@"i0SRE"]
```

Now, the view looks like figure 10-12.

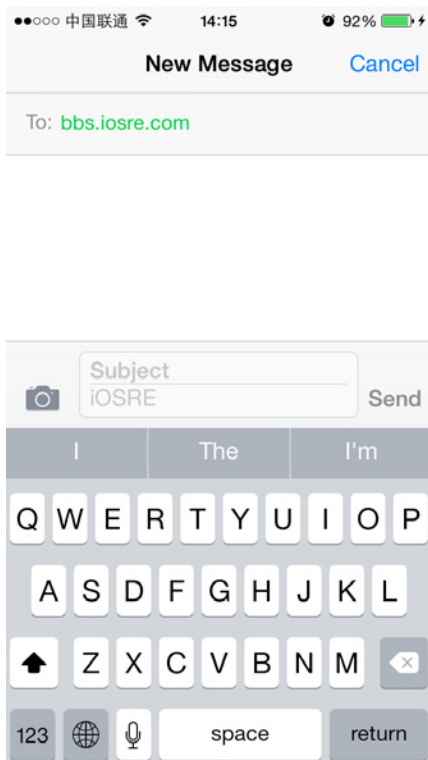


Figure 10- 12 Change placeholder to “iOSRE”

Great! placeholderText is exactly the placeholder we’re looking for. To avoid confusion, hereafter we will refer to placeholder as placeholderText. So far, we have taken the first step in a long march. Well done!

10.2.3 Find the 1st data source of placeholderText using IDA and LLDB
placeholderText is a property. To modify a property, our first reaction is to use its setter. We have already changed placeholderText from “Text Message” to “iOSRE” by calling setPlaceholderText:, does MobileSMS also call this setter to change placeholderText? To verify our guesses, we need the help of IDA and LLDB.

Since CKMessageEntryTextView comes from ChatKit, our next focus should turn to framework ChatKit in process MobileSMS, can you get it? OK, drag and drop ChatKit into IDA. After the initial analysis, locate to [CKMessageEntryTextView setPlaceholderText:], as shown in figure 10-13.

```

; CKMessageEntryTextView - (void)setPlaceholderText:(id)
; Attributes: bp-based frame

; void __cdecl -[CKMessageEntryTextView setPlaceholderText:]
CKMessageEntryTextView setPlaceholderText__
PUSH      {R4-R7,LR}
MOV       R4, R0
MOV       R0, #(selRef_placeholderLabel - 0x2693BCF2)
ADD       R7, SP, #0xC
ADD       R0, PC ; selRef_placeholderLabel
MOV       R5, R2
LDR       R1, [R0] ; "placeholderLabel"
MOV       R0, R4
BLX      _objc_msgSend

```

Figure 10- 13 [CKMessageEntryTextView setPlaceholderText:]

Attach LLDB to MobileSMS and continue the process as follows:

```

(lldb) process connect connect://iOSIP:1234
Process 200596 stopped
* thread #1: tid = 0x30f94, 0x316554f0 libsystem_kernel.dylib`mach_msg_trap + 20, queue
= 'com.apple.main-thread, stop reason = signal SIGSTOP
  frame #0: 0x316554f0 libsystem_kernel.dylib`mach_msg_trap + 20:
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x316554f0: pop    {r4, r5, r6, r8}
   0x316554f4: bx     lr

libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x316554f8: mov    r12, sp
   0x316554fc: push  {r4, r5, r6, r8}
(lldb) c
Process 200596 resuming

```

Then check the ASLR offset of ChatKit as follows:

```

(lldb) image list -o -f
[ 0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/MobileSMS.app/MobileSMS(0x000000000007d000)
[ 1] 0x0019c000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x000000000019c000)
[ 2] 0x01eac000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/Foundation.framework/Foundation
.....
[ 9] 0x01eac000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/ChatKit.framework/ChatKit

```

The ASLR offset is 0x1eac000. With this offset, we can set a breakpoint on [CKMessageEntryTextView setPlaceholderText:] to check whether it is called or not, and if it's called, who's the caller. The base address of this method is shown in figure 10-14, as we can see, it's 0x2693BCE0.

```

__text:2693BCE0 ; CKMessageEntryTextView - (void)setPlaceholderText:(id)
__text:2693BCE0 ; Attributes: bp-based frame
__text:2693BCE0
__text:2693BCE0 ; void __cdecl -[CKMessageEntryTextView setPlaceholderText:]
__text:2693BCE0 __CKMessageEntryTextView_setPlaceholderText
__text:2693BCE0 ; DATA XREF: __objc_
__text:2693BCE0          PUSH          {R4-R7,LR}
__text:2693BCE2          MOV           R4, R0

```

Figure 10- 14 [CKMessageEntryTextView setPlaceholderText:]

So the breakpoint should be set at $0x1eac000 + 0x2693BCE0 = 0x287E7CE0$.

```

(lldb) br s -a 0x287E7CE0
Breakpoint 1: where = ChatKit`-[CKMessageEntryTextView setPlaceholderText:], address =
0x287e7ce0

```

Next, let's change "bbs.iosre.com" to "snakeninny@gmail.com", an email address that supports iMessage, to see if the process stops. As a result, we can find that while we're editing the address, the breakpoint is triggered multiple times, meaning [CKMessageEntryTextView setPlaceholderText:] has been called a lot. Well, here comes a new question: among these calls, how can we know which one is the call that changes placeholderText from "Text Message" to "iMessage"? We can do a trick with LLDB's "com" command:

```

(lldb) br com add 1
Enter your debugger command(s). Type 'DONE' to end.
> po $r2
> p/x $lr
> c
> DONE

```

This command is very straightforward; when the breakpoint gets triggered, LLDB prints the Objective-C description of R2, i.e. the only argument of setPlaceholderText:, then prints LR in hexadecimal, i.e. the return address of [CKMessageEntryTextView setPlaceholderText:]. If R2 is "iMessage", it indicates that the argument is the 1st data source. Meanwhile, since LR is inside the caller, we can trace the 2nd data source from inside the caller. Clear the address entry and enter "snakeninny@gmail.com", then observe when LLDB prints "iMessage":

```

<object returned empty description>
(unsigned int) $11 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
<object returned empty description>
(unsigned int) $13 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
<object returned empty description>
(unsigned int) $15 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
Text Message
(unsigned int) $17 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.

```



```
iMessage
(unsigned int) $19 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
```

As we can see, when placeholderText turns to “iMessage”, LR’s value is 0x28768b33. 0x28768b33 - 0x1eac000 = 0x268BCB33, let’s jump to this address, as shown in figure 10-15.

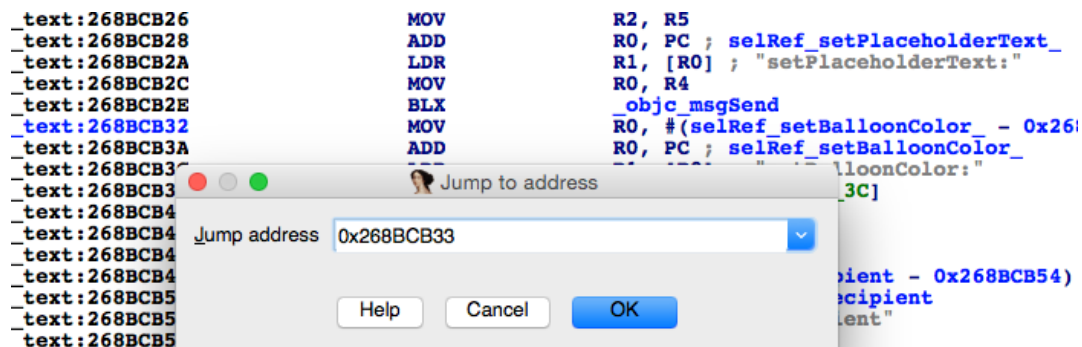


Figure 10- 15 Jump to 0x268BCB33

This address is located in ChatKit. OK, we’ve found the 1st data source of placeholder, which is the argument of setPlaceholder:, as well got on the way to find the 2nd data source. What an uneventful achievement, meh.

10.2.4 Find the Nth data source of placeholderText using IDA and LLDB

I don’t know if you’ve noticed that placeholderText was blank during address editing. Not until we’ve pressed “return” on the keyword that the placeholderText became “Text Message” or “iMessage”. In other words, iOS will not detect whether current address supports iMessage until editing is over; from the perspective of energy saving, this makes sense. Based on this design, we can firstly edit the recipient’s address, then set a breakpoint and at last press “return” to finish editing. If the breakpoint gets triggered under such circumstance, we can say that MobileSMS is stopped during the process of detecting iMessage. Now, let’s search upward from figure 10-15 to see who is the caller of [CKMessageEntryTextView setPlaceholderText:], as shown in figure 10-16.

```
; CKMessageEntryView - (void)updateEntryView
; Attributes: bp-based frame

; void __cdecl -[CKMessageEntryView updateEntryView]
__CKMessageEntryView_updateEntryView_
```

Figure 10- 16 Caller of [CKMessageEntryTextView setPlaceholderText:]

Set placeholder text when updating entry view, this is rather reasonable. However, without

any argument, how does [CKMessageEntryView updateEntryView] know whether it should set placeholderText to “Text Message” or “iMessage”? Judging from this, we can say that [CKMessageEntryView updateEntryView] must have conducted some internal judges to get the conclusion that the address supports iMessage, hence changed the 2nd data source. Let’s get back to IDA to see where the 2nd data source comes from, as shown in figure 10-17.

```

MOV      R0, R6
STR      R1, [SP, #0x6C+var_60]
BLX      _objc_msgSend
CMP      R0, #0
BEQ      loc_268BCAFA

loc_268BCAFA
STR
MOV
ADD

R0, #(selRef_contentView - 0x268BCB12) ; selRef_contentView
R0, PC ; selRef_contentView
R8, [R0] ; "contentView"
R0, R10
R1, R8
_objc_msgSend
R4, R0
R0, #(selRef_setPlaceholderText_ - 0x268BCB2C) ; selRef_setPla
R2, R5
R0, PC ; selRef_setPlaceholderText_
R1, [R0] ; "setPlaceholderText:"
R0, R4
_objc_msgSend

```

Figure 10- 17 Look for the 2nd data source

R2 is the argument of setPlaceholderText:, which is also the 1st data source. And R2 comes from R5, therefore R5 is the 2nd data source. Where does R5 come from? There is a branch here, so let’s take a look at its condition, as shown in figure 10-18.

```

loc_268BCACA
MOV      R0, #(selRef_recipientCount - 0
ADD      R0, PC ; selRef_recipientCount
LDR      R1, [R0] ; "recipientCount"
MOV      R0, R6
STR      R1, [SP, #0x6C+var_60]
BLX      _objc_msgSend
CMP      R0, #0
BEQ      loc_268BCAFA

```

Figure 10- 18 Branch condition

We can see that the branch condition is “[R0 recipientCount] == 0”. The meaning of “recipient” is very obvious that it represents the receiver of message. When the recipient count is 0, namely there’s no recipient, MobileSMS will branch right, otherwise left. In the current

case, because there is already one recipient, MobileSMS will probably branch left. It's very simple to verify our assumption: input "snakeninny@gmail.com" in the address entry, then set a breakpoint on any instruction in the right branch and at last press "return" to finish editing. We can see that the breakpoint is not triggered; as a result, we can confirm that R5 comes from [\$r8 __ck_displayName] in the left branch. In other words, [\$r8 __ck_displayName] is the 3rd data source. Where does R8 come from? Scroll up in IDA, we can find that R8 is from [[self conversation] sendingService] at the beginning of [CKMessageEntryView updateEntryView], as shown in figure 10-19.

```

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB       SP, SP, #0x54
MOV       R10, R0
MOV       R0, #(selRef_conversation - 0x26)
ADD       R0, PC ; selRef_conversation
LDR       R1, [R0] ; "conversation"
MOV       R0, R10
STR       R1, [SP,#0x6C+var_58]
BLX      _objc_msgSend
MOV       R6, R0
MOV       R0, #(selRef_sendingService - 0x)
ADD       R0, PC ; selRef_sendingService
LDR       R1, [R0] ; "sendingService"
MOV       R0, R6
STR       R1, [SP,#0x6C+var_44]
BLX      _objc_msgSend
MOV       R8, R0

```

Figure 10- 19 Look for 4th data source

Therefore, [[self conversation] sendingService] is the 4th data source. Let's verify our analysis so far with LLDB: input "snakeninny@gmail.com" in the address entry, then set a breakpoint on "MOV R8, R0" in figure 10-19 and at last press "return" to finish editing. Execute "po [\$r0 __ck_displayName]" when the breakpoint gets triggered and then see whether LLDB outputs "iMessage":

```

(lldb) br s -a 0x28768962
Breakpoint 14: where = ChatKit`-[CKMessageEntryView updateEntryView] + 54, address =
0x28768962
(lldb) br com add 14
Enter your debugger command(s). Type 'DONE' to end.
> po [$r0 __ck_displayName]
> c
> DONE
Text Message
Process 200596 resuming
Command #2 'c' continued the target.
iMessage
Process 200596 resuming
Command #2 'c' continued the target.

```

From the output, we know that the breakpoint has been triggered twice, and iMessage

support was detected in the 2nd time. Since `iMessage` comes from `[[[self conversation] sendingService] __ck_displayName]`, what is the return value of `[self conversation]` and `[[self conversation] sendingService]`? No hurry, we will get to them one by one.

Reinput the address and set 2 breakpoints on the first 2 `objc_msgSends` in `[CKMessageEntryView updateEntryView]` respectively. Then press “return” to trigger the breakpoints:

```
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b528948 ChatKit`-[CKMessageEntryView updateEntryView] +
28, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
  frame #0: 0x2b528948 ChatKit`-[CKMessageEntryView updateEntryView] + 28
ChatKit`-[CKMessageEntryView updateEntryView] + 28:
-> 0x2b528948: blx    0x2b5f5f44                ; symbol stub for:
MarcoShouldLogMadridLevel$shim
  0x2b52894c: mov     r6, r0
  0x2b52894e: movw   r0, #51162
  0x2b528952: movt   r0, #2547
(lldb) p (char *)$r1
(char *) $6 = 0x2b60cc16 "conversation"
(lldb) ni
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b52894c ChatKit`-[CKMessageEntryView updateEntryView] +
32, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b52894c ChatKit`-[CKMessageEntryView updateEntryView] + 32
ChatKit`-[CKMessageEntryView updateEntryView] + 32:
-> 0x2b52894c: mov     r6, r0
  0x2b52894e: movw   r0, #51162
  0x2b528952: movt   r0, #2547
  0x2b528956: add    r0, pc
(lldb) po $r0
CKPendingConversation<0x1587e870>{identifier:'(null)' guid:'(null)'}(null)
```

The return value of `[self conversation]` is a `CKPendingConversation` object. OK, now look at the next one:

```
(lldb) c
Process 14235 resuming
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b52895e ChatKit`-[CKMessageEntryView updateEntryView] +
50, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1
  frame #0: 0x2b52895e ChatKit`-[CKMessageEntryView updateEntryView] + 50
ChatKit`-[CKMessageEntryView updateEntryView] + 50:
-> 0x2b52895e: blx    0x2b5f5f44                ; symbol stub for:
MarcoShouldLogMadridLevel$shim
  0x2b528962: mov     r8, r0
  0x2b528964: movw   r0, #52792
  0x2b528968: movt   r0, #2547
(lldb) p (char *)$r1
(char *) $8 = 0x2b6105e1 "sendingService"
(lldb) ni
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b528962 ChatKit`-[CKMessageEntryView updateEntryView] +
54, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b528962 ChatKit`-[CKMessageEntryView updateEntryView] + 54
```

```

ChatKit`-[CKMessageEntryView updateEntryView] + 54:
-> 0x2b528962: mov    r8, r0
    0x2b528964: movw  r0, #52792
    0x2b528968: movt  r0, #2547
    0x2b52896c: add   r0, pc
(lldb) po $r0
IMService[SMS]
(lldb) po [$r0 class]
IMServiceImpl

```

Obviously, the return value of `[CKPendingConversation sendingService]` is `IMService[SMS]` (the value becomes `IMService[iMessage]` when this breakpoint gets triggered the 2nd time), whose type is `IMServiceImpl`. Therefore, the 4th data source is `[CKPendingConversation sendingService]`. Can you still follow?

Till now, we have already got a lot of useful information. So let's turn back to IDA, locate `[CKPendingConversation sendingService]` and find out how it works internally, as shown in figure 10-20.

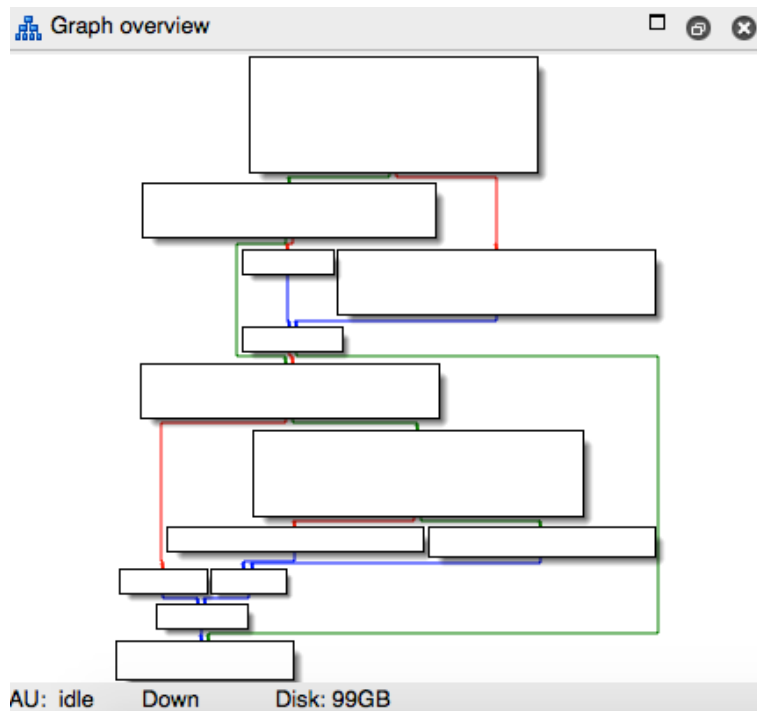


Figure 10- 20 `[CKPendingConversation sendingService]`

The implementation logic is not too complicated. But there are several branches so that we can't make sure which one `MobileSMS` actually goes. Debug again with LLDB and pay attention to every branch condition as well as the address of the next instruction.

```

Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b5f0264 ChatKit`-[CKPendingConversation sendingService],
  queue = 'com.apple.main-thread, stop reason = breakpoint 3.1
  frame #0: 0x2b5f0264 ChatKit`-[CKPendingConversation sendingService]
ChatKit`-[CKPendingConversation sendingService]:
-> 0x2b5f0264: push  {r4, r5, r7, lr}

```

```

    0x2b5f0266: add    r7, sp, #8
    0x2b5f0268: sub    sp, #8
    0x2b5f026a: mov    r4, r0
(lldb) ni
Process 14235 stopped

*****
* thread #1: tid = 0x379b, 0x2b5f027e ChatKit`-[CKPendingConversation sendingService] +
26, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b5f027e ChatKit`-[CKPendingConversation sendingService] + 26:
ChatKit`-[CKPendingConversation sendingService] + 26:
-> 0x2b5f027e: cbz    r0, 0x2b5f02a4          ; -[CKPendingConversation
sendingService] + 64
    0x2b5f0280: movw   r0, #38082
    0x2b5f0284: movt   r0, #2535
    0x2b5f0288: str    r4, [sp]
(lldb) p $r0
(unsigned int) $11 = 0
(lldb) ni
Process 14235 stopped

*****
* thread #1: tid = 0x379b, 0x2b5f02b8 ChatKit`-[CKPendingConversation sendingService] +
84, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b5f02b8 ChatKit`-[CKPendingConversation sendingService] + 84:
ChatKit`-[CKPendingConversation sendingService] + 84:
-> 0x2b5f02b8: cbz    r0, 0x2b5f02c4          ; -[CKPendingConversation
sendingService] + 96
    0x2b5f02ba: mov    r0, r4
    0x2b5f02bc: mov    r1, r5
    0x2b5f02be: blx   0x2b5f5f44          ; symbol stub for:
MarcoShouldLogMadridLevel$shim
(lldb) p $r0
(unsigned int) $12 = 341691792
(lldb) ni
Process 14235 stopped

*****
* thread #1: tid = 0x379b, 0x2b5f02c2 ChatKit`-[CKPendingConversation sendingService] +
94, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b5f02c2 ChatKit`-[CKPendingConversation sendingService] + 94:
ChatKit`-[CKPendingConversation sendingService] + 94:
-> 0x2b5f02c2: cbnz   r0, 0x2b5f032c          ; -[CKPendingConversation
sendingService] + 200
    0x2b5f02c4: movw   r0, #35464
    0x2b5f02c8: movt   r0, #2535
    0x2b5f02cc: add    r0, pc
(lldb) p $r0
(unsigned int) $13 = 341691792
(lldb) ni
Process 14235 stopped

*****
* thread #1: tid = 0x379b, 0x2b5f032e ChatKit`-[CKPendingConversation sendingService] +
202, queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2b5f032e ChatKit`-[CKPendingConversation sendingService] + 202:
ChatKit`-[CKPendingConversation sendingService] + 202:
-> 0x2b5f032e: pop    {r4, r5, r7, pc}

ChatKit`-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]:
    0x2b5f0330: push   {r4, r5, r6, r7, lr}
    0x2b5f0332: add    r7, sp, #12
    0x2b5f0334: push.w {r8, r10, r11}

```

The execution flow of MobileSMS is very evident now. There are 3 conditional branches, which are CBZ, CBZ and CBNZ respectively. At each time, the value of R0 is 0, 341691792 and 341691792 respectively. As a result, we can know that the execution flow is shown in figure 10-21.

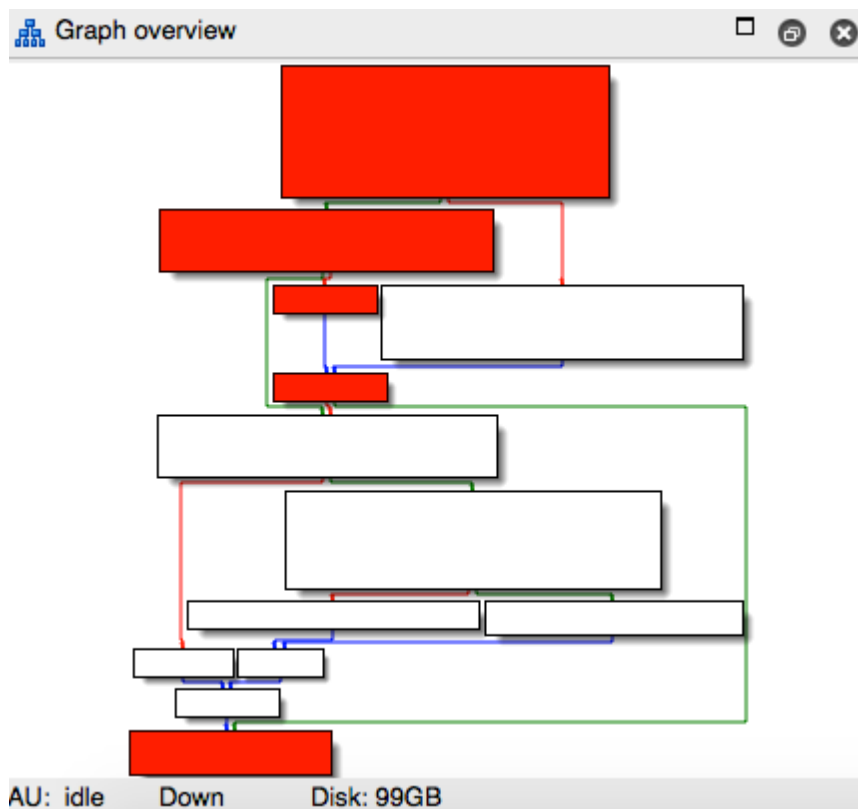


Figure 10- 21 Execution flow

So the value of [CKPendingConversation sendingService] actually comes from [CKPendingConversation composeSendingService], which is the 5th data source, right? OK, let's proceed to the new method in IDA, as shown in figure 10-22.

```

; CKPendingConversation - (id)composeSendingService
; id __cdecl -[CKPendingConversation composeSendingService](struct
  CKPendingConversation_composeSendingService_
MOV     R1, #(_OBJC_IVAR_$_CKPendingConversation._compose
ADD     R1, PC ; IService *_composeSendingService;
LDR     R1, [R1] ; IService *_composeSendingService;
LDR     R0, [R0,R1]
BX      LR
; End of function -[CKPendingConversation composeSendingService]

```

Figure 10- 22 [CKPendingConversation composeSendingService]

Obviously, [CKPendingConversation composeSendingService] merely returns the value of

instance variable `_composeSendingService`. In other words, `_composeSendingService` is the 6th data source. In that case, we just need to find where this instance variable is written and there comes the 7th data sources.

Click `_OBJC_IVAR_$_CKPendingConversation._composeSendingService` to focus the cursor on it. Then press “x” to inspect xrefs to this variable, as shown in figure 10-23.

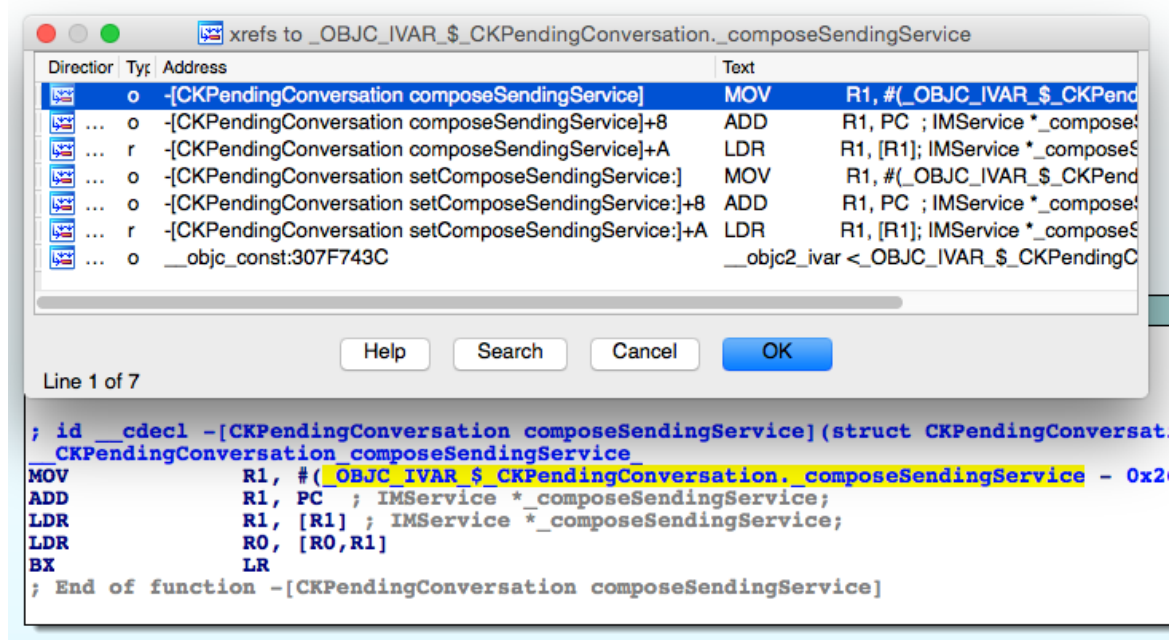


Figure 10- 23 Inspect cross references

Here, we can find 2 methods explicitly accessing `_composeSendingService`, which happens to be one setter and one getter respectively. Naturally, we guess that `_composeSendingService` is a property. Open `CKPendingConversation.h` and verify our assumption, as shown in figure 10-24.

```

11 @interface CKPendingConversation : CKConversation
12 {
13     BOOL _noAvailableServices;
14     IService *_previousSendingService;
15     IService *_composeSendingService;
16 }
17
18 @property(nonatomic) IService *composeSendingService;
19 @property(nonatomic) IService *previousSendingService;

```

Figure 10- 24 CKPendingConversation.h

In Objective-C, write operation of a property is often carried out through its setter. Thus to find the 7th data source, we should set a breakpoint on `[CKPendingConversation setComposeSendingService:]` and check out who’s the caller. Repeat our previous operations:

reinput the address, set breakpoint at the beginning of [CKPendingConversation setComposeSendingService:], and then press “return” to trigger the breakpoint:

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b3665c ChatKit`-[CKPendingConversation
setComposeSendingService:], queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
  frame #0: 0x30b3665c ChatKit`-[CKPendingConversation setComposeSendingService:]
ChatKit`-[CKPendingConversation setComposeSendingService:]
-> 0x30b3665c: movw   r1, #41004
   0x30b36660: movt   r1, #2535
   0x30b36664: add    r1, pc
   0x30b36666: ldr    r1, [r1]
(lldb) p/x $lr
(unsigned int) $0 = 0x30b3656d
```

By subtracting ASLR offset of ChatKit from LR here, we get 0x2698456D, which is LR without offset. Then jump to this address in IDA, as shown in figure 10-25.

```
MOVW   R1, #(:lower16:(selRef_setComposeSendingSe
MOV    R2, R6
MOVT.W R1, #(:upper16:(selRef_setComposeSendingSe
LDR    R0, [R4, #0x14]
ADD    R1, PC ; selRef_setComposeSendingService_
LDR    R1, [R1] ; "setComposeSendingService:"
BLX    _objc_msgSend
```

Figure 10- 25 Jump to 0x2698456D

The argument of [CKPendingConversation setComposeSendingService:], i.e. R2, is the 7th data source. R2 comes from R6, therefore R6 is the 8th data source. Search upwards to find R6’s source, as shown in figure 10-26.

```
sub_26984530
var_18= -0x18
arg_0= 8
PUSH   {R4-R7, LR}
ADD    R7, SP, #0xC
PUSH.W {R8, R10}
SUB    SP, SP, #4
MOV    R6, R1
MOV    R1, #(selRef_composeSendingService -
MOV    R4, R0
ADD    R1, PC ; selRef_composeSendingService
LDR    R0, [R4, #0x14]
MOV    R8, R3
LDR    R1, [R1] ; "composeSendingService"
MOV    R10, R2
BLX    _objc_msgSend
CMP    R0, R6
BEQ    loc_269845B0
```

Figure 10- 26 Look for the 9th data source

R6 is from R1, so R1 is the 9th data source. And where does R1 come from? Since we are inside sub_26984530 and R1 is read without being written, so R1 comes from the caller of sub_26984530, right? Let’s take a look at the cross references to sub_26984530 to look for its possible callers, as shown in figure 10-27.



Figure 10- 27 Inspect cross references

Refresh sending service? This name is very informative. Let's head directly to `-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]` as shown in figure 10-28 for more details. In this method, `sub_26984530` is obviously the 2nd argument of `refreshStatusForAddresses:withCompletionBlock:`, namely the `completionBlock`, as shown in figure 10-28.

```

ADD      LR, PC ; sub_26984530
STR      R1, [SP,#0x24+var_20]
MOVS     R1, #0
STR      R1, [SP,#0x24+var_1C]
LDR.W   R1, [R12] ; "refreshStatusForAddresses:withCompleti..."
STR.W   LR, [SP,#0x24+var_18]
STR.W   R9, [SP,#0x24+var_14]
STR      R0, [SP,#0x24+var_10]
STR      R3, [SP,#0x24+var_C]
MOV      R3, SP
BLX     _objc_msgSend

```

Figure 10- 28 `-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]`

Although `sub_26984530` appears in this method, it just acts as an argument of `objc_msgSend`, hence is not called directly. Well, who is the direct caller on earth? Actually, we've already mastered the solution of such problems: reinput the address, set a breakpoint at the beginning of `sub_26984530` and then press "return" to trigger the breakpoint.

```

Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b36530 ChatKit`__86-[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBlock:]_block_invoke, queue =
'com.apple.main-thread, stop reason = breakpoint 6.1
  frame #0: 0x30b36530 ChatKit`__86-[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBlock:]_block_invoke
ChatKit`__86-[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBlock:]_block_invoke:
-> 0x30b36530: push   {r4, r5, r6, r7, lr}
   0x30b36532: add    r7, sp, #12
   0x30b36534: push.w {r8, r10}
   0x30b36538: sub    sp, #4
(lldb) p/x $lr
(unsigned int) $38 = 0x30b364bb

```

LR without offset is $0x30b364bb - 0xa1b2000 = 0x269844BB$. Locate it in IDA, as shown in figure 10-29.

```

LDR      R6, [R0, #0xC]
MOV      R2, R5
MOV      R3, R4
STR.W    R8, [SP, #0x14+var_14]
BLX     R6

loc_269844BA
ADD      SP, SP, #4
LDR.W    R8, [SP+0x10+var_10], #4
POP      {R4-R7, PC}
; End of function sub_26984444

```

Figure 10- 29 Caller of sub_26984530

As we can see, sub_26984530 isn't called explicitly. Instead, its address is stored in R6 to where the execution flow jumps, and then sub_26984530 is called implicitly. As a result, the 9th data source comes from sub_26984444. Well done! We have achieved a lot so far. Let's keep searching for the occurrences of the 9th data source, as shown in figure 10-30.

```

;f_iMessageService - 0x26984474) ; selRef_iMessageService
;Ref_IMServiceImpl - 0x26984476) ; classRef_IMServiceImpl
;Ref_iMessageService
;lassRef_IMServiceImpl
"iMessageService"
_OBJC_CLASS_$_IMServiceImpl
14

loc_269844A4
BLX     _objc_msgSend
MOV     R1, R0
LDR     R0, [R6, #0x14]
CBZ     R0, loc_269844BA

```

Figure 10- 30 Look for the 9th data source

There are several branches inside this subroutine to determine whether it should assign [IMServiceImpl smsService] or [IMServiceImpl iMessageService] to R1. Let's figure out the branch conditions, starting from figure 10-31.

```

sub_26984444
var_14= -0x14
var_10= -0x10
arg_0= 8

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
STR.W    R8, [SP,#0xC+var_10]!
SUB       SP, SP, #4
LDR.W    R8, [R7,#arg_0]
MOV       R6, R0
MOV       R4, R3
MOV       R5, R2
UXTB.W   R0, R8
CMP       R0, #2
BNE      loc_2698447A

loc_2698447A
MOVW     R0, #(:lower16:(classRef_IMServiceImpl - 0x2698448A))
TST.W    R1, #0xFF
MOVT.W   R0, #(:upper16:(classRef_IMServiceImpl - 0x2698448A))
ADD      R0, PC ; classRef_IMServiceImpl
LDR      R0, [R0] ; OBJC_CLASS_$_IMServiceImpl
BEQ      loc_26984498

```

Figure 10- 31 Look for the 10th data source

If the value of R0 is 2, [IMServiceImpl iMessageService] is the 10th data source, otherwise we have to further check the value of R1. If R1 is 0, then [IMServiceImpl smsService] is the 10th data source, otherwise it should be [IMServiceImpl iMessageService]. The logic can be shown with the following pseudo code:

```

- (BOOL)supportIMessage
{
    if (R0 == 2 || R1 != 0) return YES;
    return NO;
}

```

That is to say, the value of the 10th data source is determined by the combination of R0 and R1, both of whom assume the responsibility of being the 11th data source, hereafter referred to as 11th data source A and 11th data source B respectively. At the same time, the above pseudo code can also be written as the following:

```

- (BOOL)supportIMessage
{
    if (11thDataSourceA == 2 || 11thDataSourceB != 0) return YES;
    return NO;
}

```

Get back to figure 10-31 to trace the 11th data source; R0 comes from "UXTB.W R0, R8".

UXTB

Zero extend Byte. Extends an 8-bit value to a 32-bit value.

▼ Syntax

`UXTB{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

Figure 10- 32 UXTB

According to the ARM official document in figure 10-32, UXTB is used to zero extend the 8-bit value in R8 to a 32-bit value and then put it into R0, who is a 32-bit register. In other words, R0 comes from R8, so R8 is the 12th data source A; and from the facts that $\text{arg_0} = 0x8$, $R8 = *(R7 + \text{arg_0}) = *(R7 + 0x8)$, $R7 = SP + 0xC$, we can know that $R8 = *(SP + 0x14)$, which means $*(SP + 0x14)$ is the 13th data source A. Well, where does $*(SP + 0x14)$ come from? It definitely doesn't come from nowhere, so before "LDR.W R8, [R7,#8]", there must be an instruction writing something into $*(SP + 0x14)$, right? That instruction is where the 14th data source A resides. As a result, we have to trace back to the instruction that writes to $*(SP + 0x14)$.

Although the idea sounds straightforward, things are much harder than you think. The reason is that SP, unlike those rarely used registers, is affected by lots of instructions. Say, push and pop both change the value of SP, so $*(SP + 0x14)$ may appear in the form of $*(SP' + \text{offset})$ in other instructions due to the change of SP. And what's even worse is that the value of offset is undetermined yet. Sounds like we're getting into troubles! From now on, we have to find every single operation that writes into $*(SP' + \text{offset})$ before "LDR.W R8, [R7,#8]", and then check whether $(SP + 0x14)$ equals to $(SP' + \text{offset})$. Thanks to the frequent and irregular changes of SP, the following section is the hardest part of this book. So please stay very close! Let's start from "LDR.W R8, [R7,#8]" and trace back every single operation that writes into $*(SP' + \text{offset})$ for now.

In sub_26984444, the first 4 instructions before "LDR.W R8, [R7,#8]" are all SP related. We use SP1~SP4 to mark the values of SP before the execution of the current instruction, as shown

in figure 10-33.

```
sub_26984444
var_14= -0x14
var_10= -0x10
arg_0= 8

PUSH     SP1     {R4-R7,LR}
ADD      SP2     R7, SP, #0xC
STR.W    SP2     R8, [SP,#0xC+var_10]!
SUB      SP2     SP, SP, #4
LDR.W    SP3     R8, [R7,#arg_0]
MOV      R6, R0
MOV      R4, R3
MOV      R5, R2
UXTB.W   R0, R8
CMP      R0, #2
BNE      loc_2698447A

loc_2698447A
MOVW     R0, #(:lower16:(classRef_IMServiceImpl - 0x2698448A))
TST.W    R1, #0xFF
MOVT.W   R0, #(:upper16:(classRef_IMServiceImpl - 0x2698448A))
ADD      R0, PC ; classRef_IMServiceImpl
LDR      R0, [R0] ; _OBJC_CLASS_$_IMServiceImpl
BEQ      loc_26984498
```

Figure 10- 33 Mark different SPs

Before and after the execution of “PUSH {R4-R7,LR}”, the values of SP are SP1 and SP2 respectively, can you understand? Next, we will try to deduce how SP changes instruction by instruction.

“PUSH {R4-R7,LR}” pushes 5 registers, i.e. R4, R5, R6, R7 and LR into stack. Every register is 32-bit i.e. 4 bytes. Since the ARM stack is full descending, therefore $SP2 = SP1 - 5 * 0x4 = SP1 - 0x14$. “ADD R7, SP, #0xC” is equivalent to $R7 = SP2 + 0xC$, which has no influence on SP. The value of var_10 in “STR.W R8, [SP,#0xC+var_10]!” is -0x10, so this instruction equals to “STR.W R8, [SP,#-4]”, i.e. $*(SP2 - 0x4) = R8$ and this instruction doesn’t have impact on SP either. “SUB SP, SP, #4” equals to $SP3 = SP2 - 0x4$. According to our marking rules, 13th data source A is $*(SP2 + 0x14)$. No instruction inside sub_26984444 has written to this address before “LDR.W R8, [R7,#8]”, so the value of $*(SP2 + 0x14)$ must come from the caller of sub_26984444. Similarly, R1 is read without being written inside sub_26984444, it must also come from the caller of sub_26984444, right? If you are still confused, please review this paragraph until you understand it clearly, and then you’re allowed to continue.

Alright, both the 13th data source A and the 11th data source B come from the caller of sub_26984444. So our next specific task is to find the 14th data source A and the 12th data source B in the caller of sub_26984444.

Reinput the recipient’s address, set a breakpoint at the beginning of sub_26984444, then

press “return” to trigger the breakpoint:

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b36444 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke, queue = 'com.apple.main-
thread, stop reason = breakpoint 7.1
  frame #0: 0x30b36444 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke:
-> 0x30b36444: push   {r4, r5, r6, r7, lr}
   0x30b36446: add    r7, sp, #12
   0x30b36448: str   r8, [sp, #-4]!
   0x30b3644c: sub   sp, #4
(lldb) p/x $lr
(unsigned int) $39 = 0x331f0d75
```

LR without offset is $0x331f0d75 - 0xa1b2000 = 0x2903ED75$, which is outside ChatKit.

Under such circumstance, how can we locate the image where $0x2903ED75$ is? We’ve talked about the solution in chapter 6, which is simply set a breakpoint at the end of `sub_26984444` and keep executing “ni” to enter the internal of caller and identify the image. The commands are as follows:

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b364c0 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 124, queue =
'com.apple.main-thread, stop reason = breakpoint 8.1
  frame #0: 0x30b364c0 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 124
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 124:
-> 0x30b364c0: pop    {r4, r5, r6, r7, pc}
   0x30b364c2: nop

ChatKit`__copy_helper_block_:
   0x30b364c4: ldr   r1, [r1, #20]
   0x30b364c6: adds r0, #20
(lldb) ni
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x331f0d74 IMCore`___lldb_unnamed_function425$$IMCore + 1360,
queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x331f0d74 IMCore`___lldb_unnamed_function425$$IMCore + 1360
IMCore`___lldb_unnamed_function425$$IMCore + 1360:
-> 0x331f0d74: movw  r0, #26972
   0x331f0d78: movt  r0, #2081
   0x331f0d7c: add   r0, pc
   0x331f0d7e: ldr   r1, [r0]
```

We’re inside IMCore now. Since we have just calculated the value of LR without offset to be $0x2903ED75$, as well IMCore shares the same ASLR offset with ChatKit, so just drag and drop IMCore into IDA and jump to $0x2903ED75$ when the initial analysis has been finished, as shown in figure 10-34.

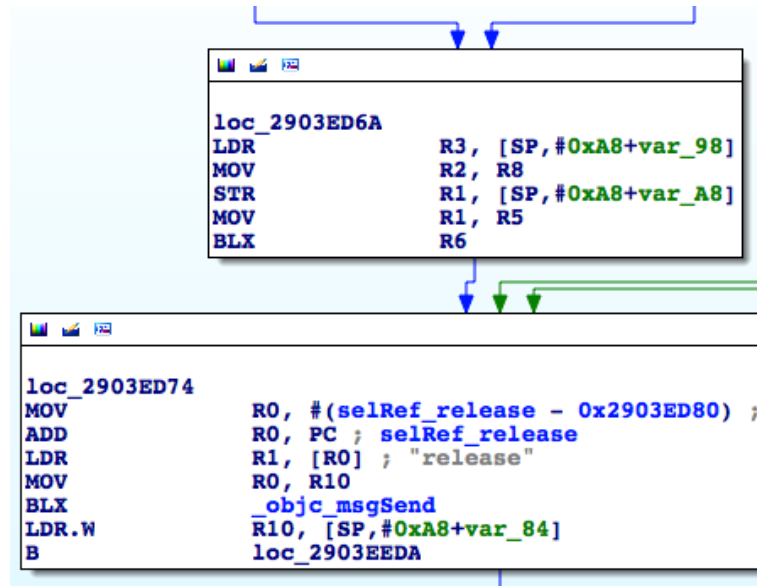


Figure 10- 34 Caller of sub_26984444

See, another implicit call from sub_2903E824, and 2 of 4 instructions before “BLX R6” has relation with SP. To make it more convenient for reading, I’ll take instructions before and after calling "BLX R6" from their respective images and put them together into one figure. The process and result is shown in figure 10-35 and figure 10-36.

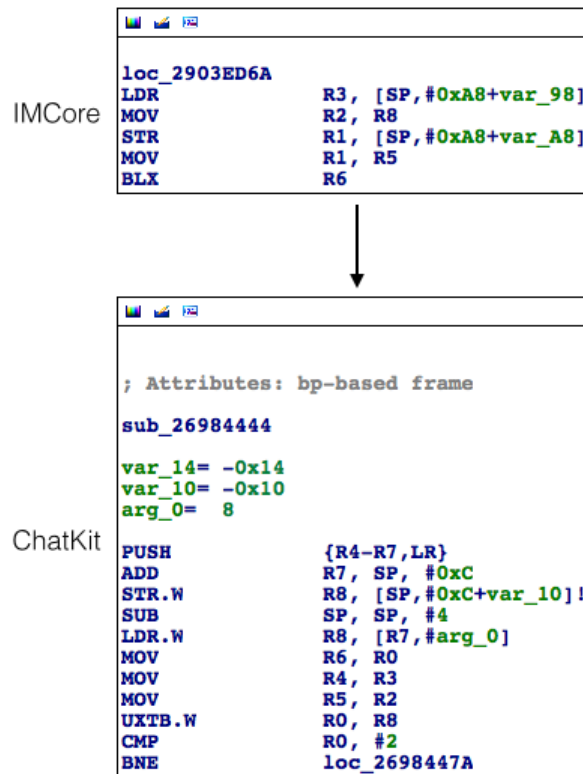


Figure 10- 35 Before instructions of 2 images are put together

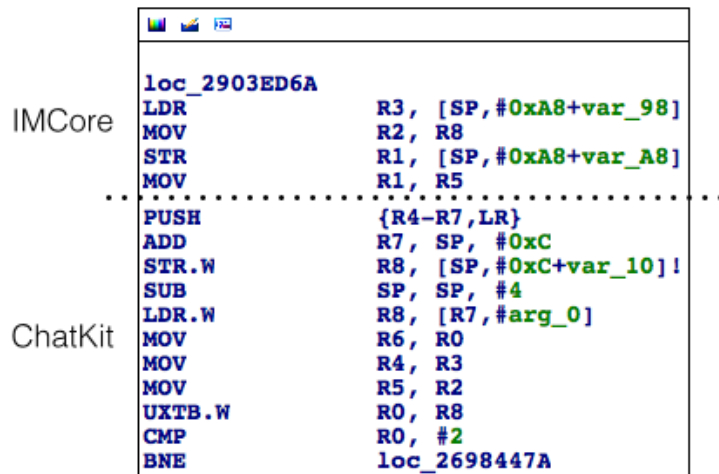


Figure 10- 36 After instructions of 2 images are put together

Let's keep looking for the 14th data source A, which has been written into $*(SP2 + 0x14)$, do you still remember? OK, mark the SPs in loc_2903ED6A just like what we've done, as shown in figure 10-37.


```

IMCore
loc_2903ED6A
LDR      sp1 R3, [SP,#0xA8+var_98]
MOV      R2, R8
STR      sp1 R1, [SP,#0xA8+var_A8]
MOV      R1, R5
-----
ChatKit
PUSH     sp1 {R4-R7,LR}
ADD      sp2 R7, SP, #0xC
STR.W    sp2 R8, [SP,#0xC+var_10]!
SUB      sp2 SP, SP, #4
LDR.W    sp3 R8, [R7,#arg_0]
MOV      R6, R0
MOV      R4, R3
MOV      R5, R2
UXTB.W   R0, R8
CMP      R0, #2
BNE     loc_2698447A

```

Figure 10- 37 Mark SPs

Then we should go through loc_2903ED6A from its 1st instruction to check how SP changes here.

“LDR R3, [SP,#0xA8+var_98]” equals to $R3 = *(SP1 + 0xA8 + var_98)$. And $var_98 = -0x98$, as shown in figure 10-38.

```

sub_2903E824
var_A8= -0xA8
var_A0= -0xA0
var_9C= -0x9C
var_98= -0x98
var_94= -0x94
var_90= -0x90
var_8C= -0x8C
var_88= -0x88
var_84= -0x84
var_80= -0x80
var_7C= -0x7C
var_78= -0x78
var_74= -0x74
var_5C= -0x5C
var_1C= -0x1C

```

Figure 10- 38 sub_2903e824

As a result, $R3 = *(SP1 + 0x10)$ and this instruction has no influence on the value of SP. “MOV R2, R8” has nothing to do with SP; the value of var_A8 in “STR R1, [SP,#0xA8+var_A8]” is -0xA8, so $*SP1 = R1$, which doesn’t influence SP too; “MOV R1, R5” has nothing to do with SP either. These SPs are really confusing for sure, so take a break and let me summarize it.

Our goal is to find where $*(SP2 + 0x14)$ is written.

Because $SP2 = SP1 - 0x14$ and $*SP1 = R1$,

Therefore, “STR R1, [SP,#0xA8+var_A8]” is the place where $*(SP2 + 0x14)$ is written, and R1 in this instruction is the 14th data source A! Also, we can easily find that R5 in “MOV R1, R5” is the 12th data source B. The logics of tracing from 13th data source A to 14th data source A and from 11th data source B to 12th data source B go across images, bringing high complexity. With the illustration of figure 10-39, I hope everything is more intuitive. We strongly suggest you comb through everything by referring to this figure before moving on to the next paragraph.

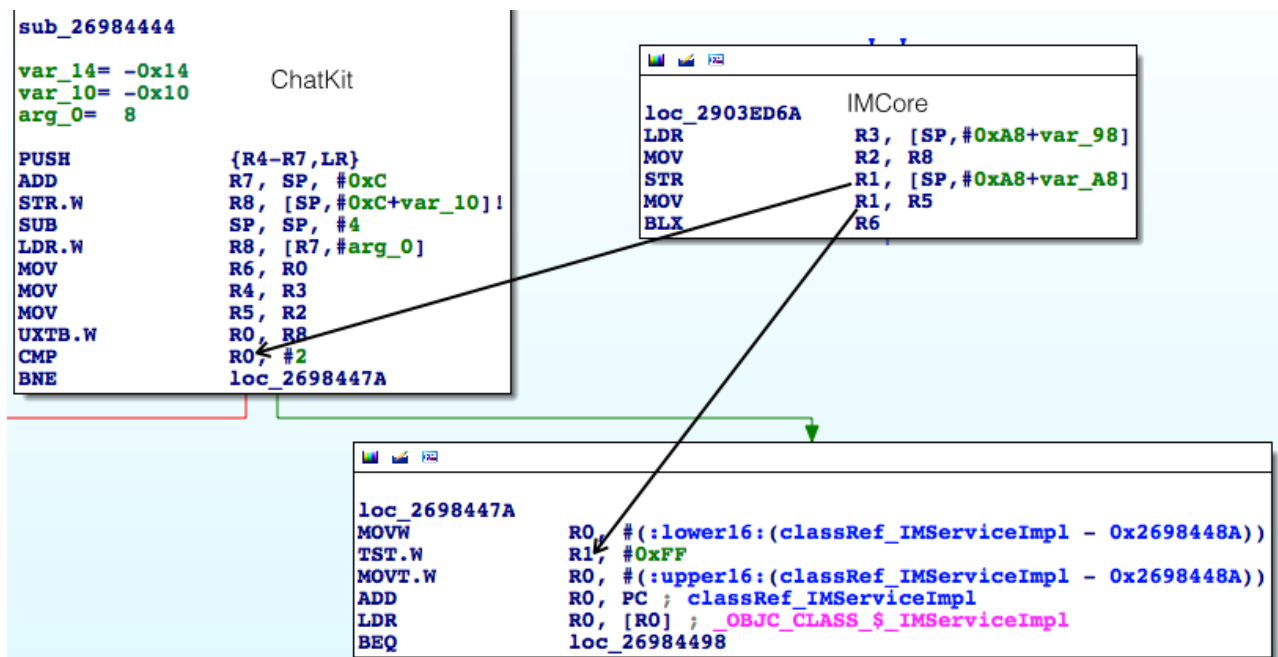


Figure 10- 39 How data sources evolve

Before we continue our analysis, let's verify our deduction so far with LLDB: reinput the address and set the breakpoint on “STR R1, [SP,#0xA8+var_A8]” to print R1, i.e. the 14th data source A. Next, execute “ni” until we reach “MOV R1, R5”, print R5 i.e. the 12th data source B. Then we'll experience an image switch from IMCore to ChatKit, so execute “si” to reach “CMP R0, #2” and print R0, i.e. the 13th data source A. Finally, we execute “ni” until “TST.W R1, #0xFF” to print R1, i.e. the 11th data source B. Press “return” to trigger the breakpoint and follow the above steps to check whether their values equal to each other like figure 10-39 shows.

```
(lldb) br s -a 0x30230D6E
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d6e IMCore`___lldb_unnamed_function425$$IMCore + 1354,
queue = 'com.apple.main-thread, stop reason = breakpoint 11.1
  frame #0: 0x30230d6e IMCore`___lldb_unnamed_function425$$IMCore + 1354
IMCore`___lldb_unnamed_function425$$IMCore + 1354:
-> 0x30230d6e: str r1, [sp]
```

```

    0x30230d70: mov    r1, r5
    0x30230d72: blx   r6
    0x30230d74: movw  r0, #26972
(lldb) p $r1
(unsigned int) $27 = 0
(lldb) ni
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d70 IMCore`___lldb_unnamed_function425$$IMCore + 1356,
queue = 'com.apple.main-thread, stop reason = instruction step over
    frame #0: 0x30230d70 IMCore`___lldb_unnamed_function425$$IMCore + 1356
IMCore`___lldb_unnamed_function425$$IMCore + 1356:
-> 0x30230d70: mov    r1, r5
    0x30230d72: blx   r6
    0x30230d74: movw  r0, #26972
    0x30230d78: movt  r0, #2081
(lldb) p $r5
(unsigned int) $28 = 1
(lldb) ni
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d72 IMCore`___lldb_unnamed_function425$$IMCore + 1358,
queue = 'com.apple.main-thread, stop reason = instruction step over
    frame #0: 0x30230d72 IMCore`___lldb_unnamed_function425$$IMCore + 1358
IMCore`___lldb_unnamed_function425$$IMCore + 1358:
-> 0x30230d72: blx   r6
    0x30230d74: movw  r0, #26972
    0x30230d78: movt  r0, #2081
    0x30230d7c: add   r0, pc
(lldb) si
Process 37477 stopped
* thread #1: tid = 0x9265, 0x2db76444 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke, queue = 'com.apple.main-
thread, stop reason = instruction step into
    frame #0: 0x2db76444 ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke:
-> 0x2db76444: push  {r4, r5, r6, r7, lr}
    0x2db76446: add  r7, sp, #12
    0x2db76448: str  r8, [sp, #-4]!
    0x2db7644c: sub  sp, #4
(lldb) ni
.....
Process 37477 stopped
* thread #1: tid = 0x9265, 0x2db7645c ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 24, queue =
'com.apple.main-thread, stop reason = instruction step over
    frame #0: 0x2db7645c ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 24
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 24:
-> 0x2db7645c: cmp    r0, #2
    0x2db7645e: bne   0x2db7647a                ; __71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 54
    0x2db76460: movw  r0, #19376
    0x2db76464: movt  r0, #2535
(lldb) p $r0
(unsigned int) $29 = 0
(lldb) ni
.....
Process 37477 stopped

```

```

* thread #1: tid = 0x9265, 0x2db7647e ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 58, queue =
'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x2db7647e ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 58
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_invoke + 58:
-> 0x2db7647e: tst.w  r1, #255
    0x2db76482: movt  r0, #2535
    0x2db76486: add   r0, pc
    0x2db76488: ldr   r0, [r0]
(lldb) p $r1
(unsigned int) $30 = 1

```

The output verifies our analysis, the 14th data source A is 0 and 12th data source B is 1.

Next, we need to focus on IMCore to keep looking for 15th data source A and 13th data source B. Let's get started from the 15th data source A.

The 15th data source A is presented in figure 10-40 intuitively.

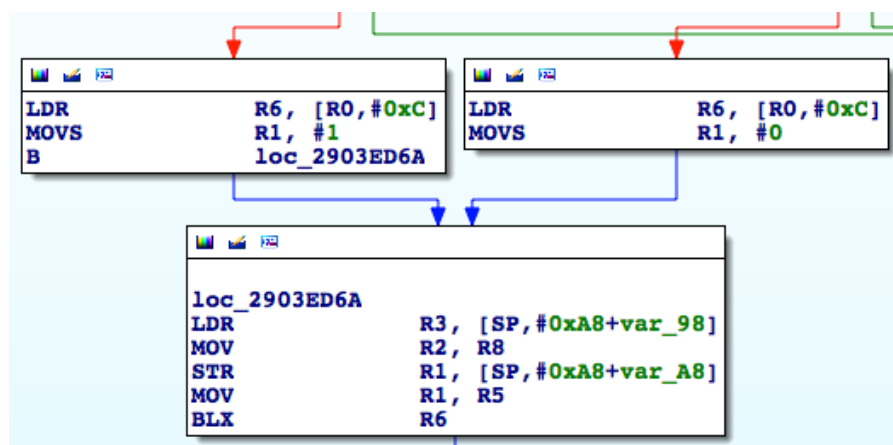


Figure 10- 40 15th data source A

It comes either from "MOVS R1, #1" or "MOVS R1, #0". In other words, the 15th data source A is either 0 or 1. Things are getting interesting.

If I remember correctly, since the 11th data source A, the value of data source A has never changed, the values of 11th, 12th, 13th, 14th and 15th data source A are all the same, which are either 0 or 1. However, the previous pseudo code is like this:

```

- (BOOL)support+IMessage
{
    if (11thDataSourceA == 2 || 11thDataSourceB != 0) return YES;
    return NO;
}

```

Because the 11th data source A is either 0 or 1, under no circumstance can it be 2. In that case, data source A becomes meaningless in our tracing, right? Hence the pseudo code can be simplified as follows:

```

- (BOOL)support+IMessage

```

```

{
    if (11thDataSourceB != 0) return YES;
    return NO;
}

```

As a result, we can ignore data source A and concentrate on the finding of the 13th data source B, hereafter referred to as the 13th data source. Since the 12th data source B is R5, we can confirm that 13th data source must be written into R5 by a certain instruction, right? Click R5 and IDA will highlight all R5s as yellow to make it more convenient for tracing in the sea of ARM assembly. Keep reversing to find where R5 is written.

When we're searching upward to locate the 13th data source, we see there're 4 branches to loc_2903EAE0, as shown in figure 10-41.

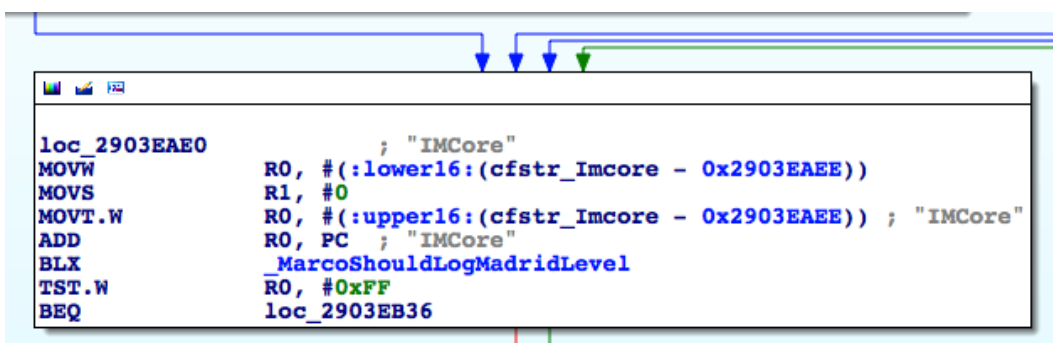


Figure 10- 41 loc_2903EAE0

In figure 10-41, the left 3 branches all contain a "MOVS R5, #0", which contradicts the result of R5 = 1, so loc_2903EAE0 must be reached via the rightmost branch, and the 13th data source should be located in this branch. Follow this branch for R5.

When we trace into loc_2903EA3E, the situation is similar to loc_2903EAE0. Although there are 3 branches upon it, the 1st and 2nd branches both contain a "MOVS R5, #0" as shown in figure 10-42, so they can be excluded for now.

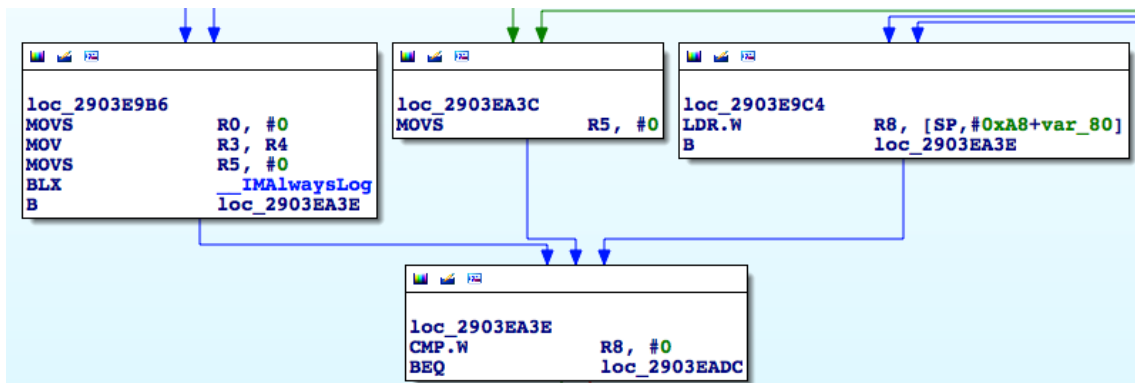


Figure 10- 42 loc_2903EA3E

As a result, the actual upstream is the 3rd branch, i.e. loc_2903E9C4, which has 2 branches

upon it. Now that both branches contain “MOVS R5, #1”, which is the actual one?

Reinput the address and set breakpoints on both branches. Then press “return” to see which breakpoint will be triggered, that’s our answer. Here, I’ll leave the LLDB operation to you, please finish it independently. After you’ve done, you will have a deeper understanding and find that the left branch is the actual one MobileSMS chose, as shown in figure 10-43.

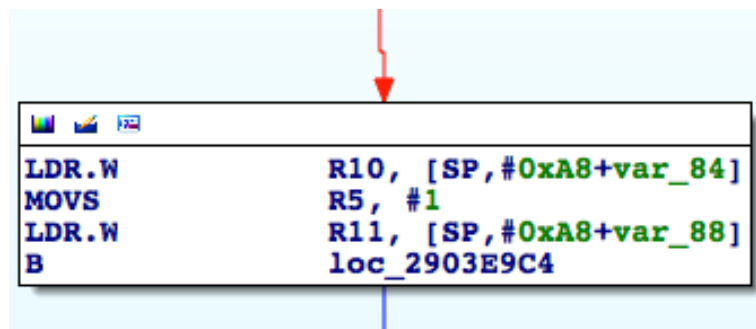


Figure 10- 43 The left branch

Now, we have found the 13th data source, it’s a constant with value 1. You may wonder, if 13th data source is a constant, does 14th data source still exist? The data source clues seem to be interrupted, what should we do next? Good point.

In the previous figures, there’re several “MOVS R5, #0”. Although the 13th data source comes from “MOVS R5, #1”, which seems to be a constant, according to programmatic paradigm, there should be a conditional branch to determine whether “MOVS R5, #0” or “MOVS R5, #1” gets executed, just like the pseudo code below.

```
if (iMessageIsAvailable) R5 = 1;
else R5 = 0;
```

To represent in our familiar IDA graph view, it looks like figure 10-44.

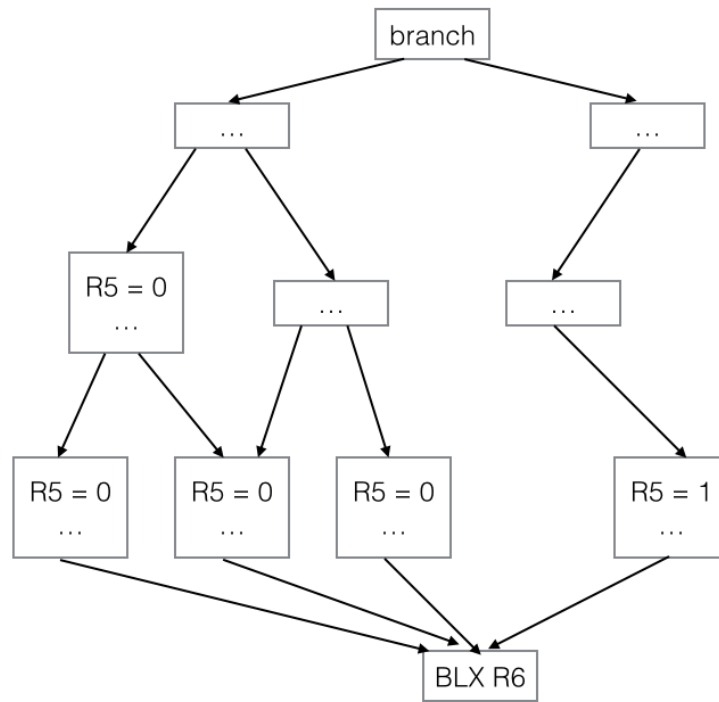


Figure 10- 44 Pseudo IDA graph view

From a macro point of view, this conditional branch is actually the 14th data source, right? I bet you've realized that the above pseudo code can be rewritten as below:

```
R5 = iMessageIsAvailable;
```

If you can understand this, then our next task is to keep tracing back to analyze every branch we meet. If different branches result in writing different values into R5, we need to figure out what's the branch condition, and this condition is our target data source. Let's head to figure 10-45 and start from here.

```

LDR    R1, [SP, #0xA8+var_9C]
MOVS   R0, #0x10
ADD    R2, SP, #0xA8+var_7C
STR    R0, [SP, #0xA8+var_A8]
MOV    R0, R5
ADD    R3, SP, #0xA8+var_5C
BLX    _objc_msgSend
MOV    R8, R0
CMP.W  R8, #0
BNE    loc_2903E8E2

```

Figure 10- 45 Branch

If the process branches left, R5 is possibly to be set 0. Since the branch condition is the

return value of objc_msgSend, let's set a breakpoint here and see what method it is:

```
Process 132234 stopped
* thread #1: tid = 0x2048a, 0x331f092e IMCore`___lldb_unnamed_function425$$IMCore + 266,
queue = 'com.apple.main-thread, stop reason = breakpoint 5.1
  frame #0: 0x331f092e IMCore`___lldb_unnamed_function425$$IMCore + 266
IMCore`___lldb_unnamed_function425$$IMCore + 266:
-> 0x331f092e: blx    0x332603b0          ; symbol stub for: objc_msgSend
   0x331f0932: mov    r8, r0
   0x331f0934: cmp.w  r8, #0
   0x331f0938: bne    0x331f08e2          ; ___lldb_unnamed_function425$$IMCore +
190
(lldb) p (char *)$r1
(char *) $6 = 0x2f7d81d9 "countByEnumeratingWithState:objects:count:"
(lldb) po $r0
<__NSArrayI 0x16706930>(  
mailto:snakeninny@gmail.com  
)
```

As we can see, this method returns the count of the recipient array. If the array is not empty, MobileSMS will branch right. Actually, the recipient array is not empty, therefore this branch condition is not met, MobileSMS will branch right, which doesn't change R5. OK, search upward for the next branch, as shown in figure 10-46.

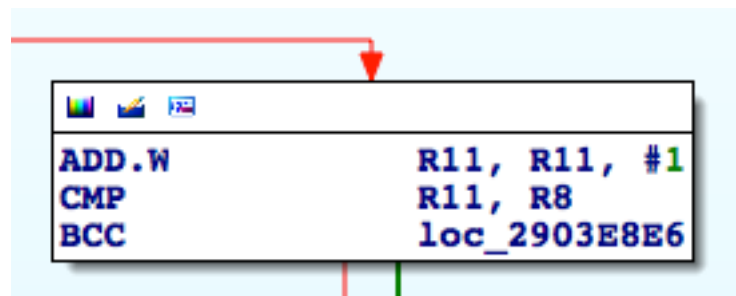


Figure 10- 46 Branch

In figure 10-46, what are R11 and R8 respectively? We can get a straightforward answer from IDA that R11 is from figure 10-47.

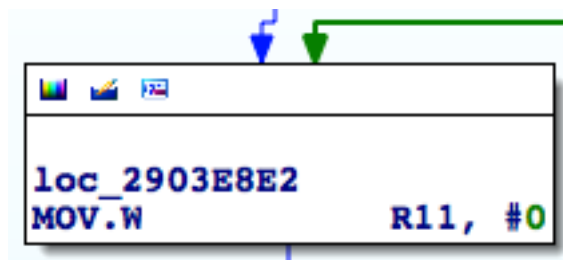


Figure 10- 47 loc_2903e8e2

The initial value of R11 is 0. Each time before executing “CMP R11, R8”, R11 will increase by 1. In this way, R11 plays the role of a counter. “CMP” performs subtraction operation, if there's borrow, then carry flag will be set 0, otherwise carry flag will be set 1. The branch instruction here is “BCC”, in which “CC” means “Carry Clear”, i.e. “if carry flag is 0”.

Therefore, if R11 - R8 produces borrow, i.e. R8 is greater than R11, then MobileSMS will branch right, otherwise it will branch left. So the key here is R8, as shown in figure 10-48.

```

LDR      R6, [R0] ; "countByEnumeratingWithState:objects:cou"...
MOVS    R0, #0x10
STR     R0, [SP, #0xA8+var_A8]
MOV     R0, R5
MOV     R1, R6
BLX    _objc_msgSend
MOV     R8, R0

```

Figure 10- 48 Where R8 comes

R8 comes from [NSArray countByEnumeratingWithState:objects:count:]. Reinput the address, set the breakpoint and press “return”, let’s see what NSArray is:

```

(lldb) br s -a 0x3023089C
Breakpoint 2: where = IMCore`___lldb_unnamed_function425$$IMCore + 120, address =
0x3023089c
Process 102482 stopped
* thread #1: tid = 0x19052, 0x3023089c IMCore`___lldb_unnamed_function425$$IMCore + 120,
queue = 'com.apple.main-thread, stop reason = breakpoint 2.1
   frame #0: 0x3023089c IMCore`___lldb_unnamed_function425$$IMCore + 120
IMCore`___lldb_unnamed_function425$$IMCore + 120:
-> 0x3023089c: blx    0x302a03b0          ; symbol stub for: objc_msgSend
   0x302308a0: mov    r8, r0
   0x302308a2: cmp.w r8, #0
   0x302308a6: beq.w 0x302309c2          ; ___lldb_unnamed_function425$$IMCore +
414
(lldb) p (char *)$r1
(char *) $5 = 0x2c8181d9 "countByEnumeratingWithState:objects:count:"
(lldb) po $r0
<__NSArrayI 0x178d6b20>(
mailto:snakeninny@gmail.com
)

```

NSArray is an array of recipients, thus R8 is the recipient count. If there’s more than 1 recipients, then since R11 is 1 when “CMP R11, R8” gets executed for the first time, we can know that R8 is greater than R11 and MobileSMS will branch right, as shown in figure 10-49.

```

loc_2903E8E6
LDR      R0, [SP, #0xA8+var_74]
LDR      R0, [R0]
CMP      R0, R10
ITT     NE
MOVNE   R0, R5
BLXNE   _objc_enumerationMutation
LDR      R0, [SP, #0xA8+var_78]
MOV     R1, R6
LDR.W   R4, [R0, R11, LSL#2]
LDR     R0, [SP, #0xA8+var_88]
MOV     R2, R4
BLX     _objc_msgSend
LDR     R0, [SP, #0xA8+var_80]
MOV     R2, R4
LDR     R1, [SP, #0xA8+var_94]
BLX     _objc_msgSend
LDR     R1, [SP, #0xA8+var_90]
BLX     _objc_msgSend
CBZ     R0, loc_2903E946

```

Figure 10- 49 Branch

The branch condition inside loc_2903E8E6 is R0. If $R0 == 0$, then branch left, meaning this address doesn't support iMessage. Otherwise branch right and reach figure 10-50.

```

CMP      R0, #2
BEQ     loc_2903E9CA

```

Figure 10- 50 Branch

The branch condition in figure 10-50 is still R0. If $R0 == 2$ then branch left, iMessage is not supported. Otherwise branch right and go back to figure 10-46. Note, these 3 blocks of code don't change the value of R8. As a result, R0 at the bottom of loc_2903E8E6 is very important; as long as $R0 != 0 \ \&\& \ R0 != 2$, the branch in figure 10-46 is useless. That's because R11 keeps increasing while R8 stays the same, MobileSMS will eventually branch left and come to the conclusion that iMessage is supported. So judging from all information above, we can think of R0 as the essential branch condition in this loop. Do you still remember what I've just said? "If different branches result in writing different values into R5, we need to figure out what's the branch condition, and this condition is our target data source". Thus, R0 is the 14th data source.

Next, let's check with LLDB what are these objc_msgSends in figure 10-49, as well the

source of R0:

```
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0900 IMCore`___lldb_unnamed_function425$$IMCore + 220,
queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
  frame #0: 0x331f0900 IMCore`___lldb_unnamed_function425$$IMCore + 220
IMCore`___lldb_unnamed_function425$$IMCore + 220:
-> 0x331f0900: blx    0x332603b0          ; symbol stub for: objc_msgSend
   0x331f0904: ldr    r0, [sp, #40]
   0x331f0906: mov    r2, r4
   0x331f0908: ldr    r1, [sp, #20]
(lldb) p (char *)$r1
(char *) $7 = 0x2f7d897a "removeObject:"
(lldb) po $r0
<__NSArrayM 0x170ec120>(  
mailto:snakeninny@gmail.com  
)

(lldb) po $r2
mailto:snakeninny@gmail.com
(lldb) ni
.....
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f090a IMCore`___lldb_unnamed_function425$$IMCore + 230,
queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x331f090a IMCore`___lldb_unnamed_function425$$IMCore + 230
IMCore`___lldb_unnamed_function425$$IMCore + 230:
-> 0x331f090a: blx    0x332603b0          ; symbol stub for: objc_msgSend
   0x331f090e: ldr    r1, [sp, #24]
   0x331f0910: blx    0x332603b0          ; symbol stub for: objc_msgSend
   0x331f0914: cbz    r0, 0x331f0946      ; ___lldb_unnamed_function425$$IMCore +
290
(lldb) p (char *)$r1
(char *) $10 = 0x2f7d8113 "valueForKey:"
(lldb) po $r2
mailto:snakeninny@gmail.com
(lldb) po $r0
{
    "mailto:snakeninny@gmail.com" = 1;
}
(lldb) po [$r0 class]
__NSCFDictionary
(lldb) ni
.....
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0910 IMCore`___lldb_unnamed_function425$$IMCore + 236,
queue = 'com.apple.main-thread, stop reason = instruction step over
  frame #0: 0x331f0910 IMCore`___lldb_unnamed_function425$$IMCore + 236
IMCore`___lldb_unnamed_function425$$IMCore + 236:
-> 0x331f0910: blx    0x332603b0          ; symbol stub for: objc_msgSend
   0x331f0914: cbz    r0, 0x331f0946      ; ___lldb_unnamed_function425$$IMCore +
290
   0x331f0916: cmp    r0, #2
   0x331f0918: beq    0x331f09ca          ; ___lldb_unnamed_function425$$IMCore +
422
(lldb) p (char *)$r1
(char *) $14 = 0x2f7de6f3 "integerValue"
(lldb) po $r0
1
(lldb) po [$r0 class]
```

```
__NSCFNumber  
(lldb) c
```

Reproduce these 3 objc_msgSends into Objective-C methods, they are [NSArray removeObject:@"mailto:snakeninny@gmail.com"], [NSDictionary valueForKey:@"mailto:snakeninny@gmail.com"] and [NSNumber integerValue] respectively. Among them, R0 of the 2nd objc_msgSend deserves our special attention. It is the key-value pair in this R0 (an NSDictionary) that determines the 14th data source. Therefore, this NSDictionary is the 15th data source. According to figure 10-49, we can know that it comes from [SP,#0xA8+var_80], which means [SP,#0xA8+var_80] is the 16th data source. Here comes our familiar operation to trace the 17th data source; inspect the cross references to var_80 as shown in figure 10-51.

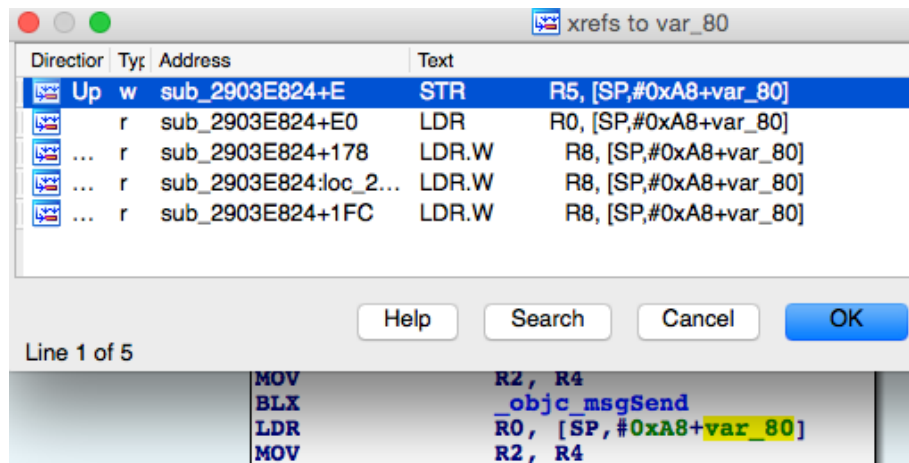


Figure 10- 51 Inspect cross references

As we can see, only one instruction writes into this address. Double click this instruction to jump to the beginning of sub_2903E824, as shown in figure 10-52.

```

sub_2903E824
var_A8= -0xA8
var_A0= -0xA0
var_9C= -0x9C
var_98= -0x98
var_94= -0x94
var_90= -0x90
var_8C= -0x8C
var_88= -0x88
var_84= -0x84
var_80= -0x80
var_7C= -0x7C
var_78= -0x78
var_74= -0x74
var_5C= -0x5C
var_1C= -0x1C

PUSH      {R4-R7,LR}
ADD       R7, SP, #0xC
PUSH.W   {R8,R10,R11}
SUB       SP, SP, #0x90
MOV       R5, R1
STR       R2, [SP,#0xA8+var_98]
STR       R5, [SP,#0xA8+var_80]
STR       R0, [SP,#0xA8+var_8C]

```

Figure 10- 52 sub_2903E824

The 16th data source comes from R5, which is the 17th data source. The 17th data source is from R1, which is the 18th data source, and it is read without being written, meaning R1 comes from the caller of sub_2903E824, right? Let’s take a look at the subroutine’s cross references, as shown in figure 10-53.



Figure 10- 53 Inspect cross references

“Calculate service for sending new compose”, as the name suggests, its function is quite clear. Double click the first cross reference to check its caller, as shown in figure 10-54.

```

LDR.W      R6, [R9] ; "sharedInstance"
LDR        R0, [R1] ; _OBJC_CLASS_$_IDSIDQueryController
MOV        R1, R6
BLX       _objc_msgSend
MOVW      R1, #(:lower16:(_IDSServiceNameiMessage_ptr - 0x2903E64C))
MOV       R11, R4
MOVT.W   R1, #(:upper16:(_IDSServiceNameiMessage_ptr - 0x2903E64C))
MOVW      R2, #(:lower16:(selRef__currentIDStatusForDestinations_service_list
ADD       R1, PC ; _IDSServiceNameiMessage_ptr
MOVT.W   R2, #(:upper16:(selRef__currentIDStatusForDestinations_service_list
LDR       R3, [R1]
ADD       R2, PC ; selRef__currentIDStatusForDestinations_service_listenerID
MOVW      R5, #(:lower16:(cfstr__kimchatservi - 0x2903E662)) ; "__kIMChatSer
LDR       R1, [R2] ; "_currentIDStatusForDestinations:service"...
MOVT.W   R5, #(:upper16:(cfstr__kimchatservi - 0x2903E662)) ; "__kIMChatSer
MOV       R2, R4
ADD       R5, PC ; "__kIMChatServiceForSendingIDSQueryControllerListenerID"
LDR.W    R10, [R3]
STR      R5, [SP, #0x64+var_64]
MOV      R3, R10
BLX     _objc_msgSend
MOV     R5, R0
ADD    R0, SP, #0x64+var_38
MOV    R1, R5
MOVS   R2, #0
MOV    R8, R0
BL     sub_2903E824

```

Figure 10- 54 Caller of sub_2903E824

To avoid any implicit calling, let's first make sure the caller of sub_2903E824 is actually IMChatCalculateServiceForSendingNewCompose. Reinput the address, set a breakpoint at the first instruction of sub_2903E824 and then press "return" to trigger the breakpoint:

```

Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0824 IMCore`___lldb_unnamed_function425$$IMCore, queue
= 'com.apple.main-thread, stop reason = breakpoint 2.1
  frame #0: 0x331f0824 IMCore`___lldb_unnamed_function425$$IMCore
IMCore`___lldb_unnamed_function425$$IMCore:
-> 0x331f0824: push   {r4, r5, r6, r7, lr}
0x331f0826: add    r7, sp, #12
0x331f0828: push.w {r8, r10, r11}
0x331f082c: sub    sp, #144
(lldb) p/x $lr
(unsigned int) $17 = 0x331f067b
(lldb)

```

The ASLR offset is 0xa1b2000, so LR without offset is 0x2903E67B, which is exactly inside IMChatCalculateServiceForSendingNewCompose. OK, since the 18th data source is from R5, then R5 is the 19th data source. Further, the 19th data source is from the return value of objc_msgSend, so this return value is the 20th data source. With everything ready, let's reveal this mysterious objc_msgSend:

```

Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0668 IMCore`IMChatCalculateServiceForSendingNewCompose
+ 688, queue = 'com.apple.main-thread, stop reason = breakpoint 3.1
  frame #0: 0x331f0668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688
IMCore`IMChatCalculateServiceForSendingNewCompose + 688:
-> 0x331f0668: blx    0x332603b0 ; symbol stub for: objc_msgSend
0x331f066c: mov    r5, r0
0x331f066e: add    r0, sp, #44
0x331f0670: mov    r1, r5
(lldb) p (char *)$r1

```

```

(char *) $18 = 0x33274340 "_currentIDStatusForDestinations:service:listenerID:"
(lldb) po $r0
<IDSIDQueryController: 0x15dcb010>
(lldb) po $r2
<__NSArrayM 0x170e7900>(  
mailto:snakeninny@gmail.com  
)

(lldb) po $r3
com.apple.madrid
(lldb) po [$r3 class]
__NSCFConstantString
(lldb) x/10 $sp
0x001e4548: 0x3b3f52b8 0x001e459c 0x3b4227b4 0x3c01b05c  
0x001e4558: 0x00000001 0x00000000 0x170828d0 0x001e4594  
0x001e4568: 0x2baac821 0x00000000
(lldb) po 0x3b3f52b8
__kIMChatServiceForSendingIDSQueryControllerListenerID
(lldb) po [0x3b3f52b8 class]
__NSCFConstantString
(lldb) c

```

Success belongs to the persevering. This `objc_msgSend` is restored to `[[IDSIDQueryController sharedInstance] _currentIDStatusForDestinations:@[@"mailto:snakeninny@gmail.com"] service:@"com.apple.madrid" listenerID:@"__kIMChatServiceForSendingIDSQueryControllerListenerID"]`. Since the last 2 arguments are constants, the only variable argument is the first array, i.e. the recipient array. What a long journey! We've finally tracked down the original data source!

I know, I know, this section is so hard that you're already dizzy now. Stay up for a while, we're almost done with this task.

10.2.5 Restore the process of the original data source becoming `placeholderText`

Now that we have found the core method, seems we can detect whether an address supports `iMessage` by modifying the first argument, i.e. the `NSArray` of recipients. As long as the key (an address) associated value (an integer) in the return value (an `NSDictionary`) is neither 0 nor 2, we can confirm that this address supports `iMessage`; otherwise it only supports `SMS`. Is that so? As we already know, the format of email addresses is "mailto:email@address", how about phone number format? Let's set a breakpoint on `_currentIDStatusForDestinations:service:listenerID` and take a look:

```
Process 102482 stopped
```



```

* thread #1: tid = 0x19052, 0x30230668 IMCore`IMChatCalculateServiceForSendingNewCompose
+ 688, queue = 'com.apple.main-thread, stop reason = breakpoint 6.1
  frame #0: 0x30230668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688
IMCore`IMChatCalculateServiceForSendingNewCompose + 688:
-> 0x30230668: blx    0x302a03b0          ; symbol stub for: objc_msgSend
   0x3023066c: mov    r5, r0
   0x3023066e: add   r0, sp, #44
   0x30230670: mov   r1, r5
(lldb) po $r2
<__NSArrayM 0x17820560>(  
tel:+86PhoneNumber  
)

```

OK, we can now turn back to Cycrypt to verify our assumption:

```

FunMaker-5:~ root# cycrypt -p MobileSMS
cy# [[IDSIDQueryController sharedInstance]
_currentIDStatusForDestinations:@[@"mailto:snakeninny@gmail.com",
@"mailto:snakeninny@icloud.com", @"tel:bbs.iosre.com", @"mailto:bbs.iosre.com",
@"tel:911", @"tel:+86PhoneNumber"] service:@"com.apple.madrid"
listenerID:@"__kIMChatServiceForSendingIDSQueryControllerListenerID"]
@{"tel:bbs.iosre.com":2,"mailto:snakeninny@gmail.com":1,"tel:911":2,"mailto:bbs.iosre.co
m":2,"mailto:snakeninny@icloud.com":1,"tel:+86PhoneNumber":1}

```

Aha, the output clearly supports our statements: 2 iMessage supportive emails and 1 iMessage supportive phone number all return 1, while the other 3 iMessage unsupportive addresses return 2. What’s more, we know the code name of iMessage is “Madrid”. Mission complete! Cheers!

10.3 Send iMessages

Through the baptism of section 10.2, I believe many of you may share the same feeling with me: debugging with LLDB step by step is of course rigorous and precise, but the workload along with it is overwhelmingly heavy. Reverse engineering is full of error checks, don’t be afraid of making mistakes. In this section, we’ll jump out and step up with wild guesses to achieve our goal; we’ll try to avoid massive analysis with LLDB, instead make use of class-dump to filter suspicious methods, and test them with IDA and Cycrypt to finally achieve our goal of sending iMessages.

10.3.1 Observe MobileSMS and look for cut-in points

In comparison with detecting iMessages, cut-in point of sending iMessages is more noticeable. In figure 10-55, the bold blue “Send” button is Apple’s gift for this section.

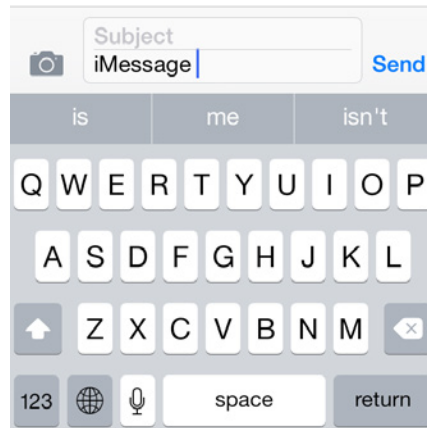
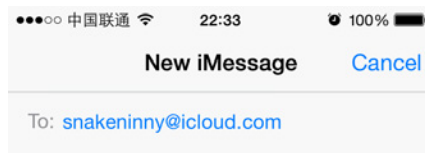


Figure 10- 55 “Send” button

We can send an iMessage by pressing “Send”, and the whole process will be animated on UI. Like what we did in section 10.2, let’s consider how to turn clues on UI into ideas in reverse engineering:

“Send” button is supposed to be a UIView object, or more specifically and possibly, a UIButton object; we press this button to call its response method; overall response actions include refreshing UI, sending the iMessage, adding a sending record and so on. That’s to say, the action of sending iMessages is only a subset of all response actions.

In “New Message” view, our inputs include recipient addresses and message contents, they’re the original data source. Since we can get all response actions, and the action of sending iMessages is supposed to take the original data source as arguments, so they can be references for us to filter the action of sending iMessages out of all response actions. Unlike what we’ve done in the last section, which was tracing back from tail to head, in the following sections, we’re tracing from head to tail, showing you another common scenario of iOS reverse engineering.

In a nutshell, our thoughts are: first uncover response method of “Send” button with Cycript, then overview all response actions with IDA and class-dump, as well filter those suspicious methods out. Finally, test the filtered methods and locate our target.

10.3.2 Find response method of “Send” button using Cycrypt

Since we’ve already known that the superview of “Send” button is a CKMessageEntryView object in section 10.2, we can repeat what we’ve done in section 10.2.2 and get the superview without further tests:

```
FunMaker-5:~ root# cycrypt -p MobileSMS
cy# ?expand
expand == true
cy# [UIApp windows]
@[#"<UIWindow: 0x14e12fa0; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x14e11f50>; layer = <UIWindowLayer: 0x14ee4570>>",#"<UITextEffectsWindow: 0x14fa6000; frame = (0 0; 320 568); opaque = NO; gestureRecognizers = <NSArray: 0x14fa66d0>; layer = <UIWindowLayer: 0x14fa5fc0>>",#"<CKJoystickWindow: 0x14d22310; baseClass = UIAutoRotatingWindow; frame = (0 0; 320 568); hidden = YES; gestureRecognizers = <NSArray: 0x14d21ab0>; layer = <UIWindowLayer: 0x14d22140>>"]
cy# [#0x14fa6000 subviews]
@[#"<UIInputSetContainerView: 0x14d03930; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer: 0x14d03770>>"]
cy# [#0x14d03930 subviews]
@[#"<UIInputSetHostView: 0x14d033f0; frame = (0 250; 320 318); layer = <CALayer: 0x14d03290>>"]
cy# [#0x14d033f0 subviews]
@[#"<UIKBInputBackdropView: 0x160441a0; frame = (0 65; 320 253); userInteractionEnabled = NO; layer = <CALayer: 0x16043b60>>",#"<_UIKBCompatInputView: 0x14f78a20; frame = (0 65; 320 253); layer = <CALayer: 0x14f78920>>",#"<CKMessageEntryView: 0x160c6180; frame = (0 0; 320 65); opaque = NO; autoresize = W; layer = <CALayer: 0x16089920>>"]
cy# [#0x160c6180 subviews]
@[#"<_UIBackdropView: 0x16069d40; frame = (0 0; 320 65); opaque = NO; autoresize = W+H; userInteractionEnabled = NO; layer = <_UIBackdropViewLayer: 0x14d627c0>>",#"<UIView: 0x16052920; frame = (0 0; 320 0.5); layer = <CALayer: 0x160529d0>>",#"<UIButton: 0x1605a8b0; frame = (266 27; 53 33); opaque = NO; layer = <CALayer: 0x16052a00>>",#"<UIButton: 0x14d0b2c0; frame = (266 30; 53 26); hidden = YES; opaque = NO; gestureRecognizers = <NSArray: 0x160f9800>; layer = <CALayer: 0x1605a140>>",#"<UIButton: 0x1606f040; frame = (15 33.5; 25 18.5); opaque = NO; gestureRecognizers = <NSArray: 0x14d07970>; layer = <CALayer: 0x1605aaa0>>",#"<UITextFieldRoundedRectBackgroundViewNeue: 0x160e5ed0; frame = (55 8; 209.5 49.5); opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x160d3a10>>",#"<UIView: 0x160a3390; frame = (55 8; 209.5 49.5); clipsToBounds = YES; opaque = NO; layer = <CALayer: 0x160b8ab0>>",#"<CKMessageEntryWaveformView: 0x160c4750; frame = (15 25.5; 251 35); alpha = 0; opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x160c47e0>>"]
```

Among these views, “UIView: 0x16052920” is where “iMessage” resides, do you remember? As a result, the following 2 UIButtons are quite suspicious, my intuition tells me “Send” is one of them. Meanwhile, the hidden property of the 2nd UIButton is set to YES, indicating its invisibility. Well, let’s test the 1st UIButton, “UIButton: 0x1605a8b0” with Cycrypt:

```
cy# [#0x1605a8b0 setHidden:YES]
```

The view changed to figure 10-56 after the above command:

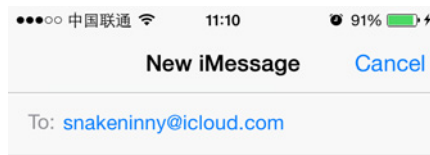


Figure 10- 56 Hide “Send”

Neat. After pressing this UIButton, an iMessage will be sent; a UIButton and its response action are always associated with [UIButton addTarget:action:forControlEvents:]. Since UIControl offers another method actionsForTarget:forControlEvents: to find its own response method, let’s see what method will get called after pressing “Send” with this method:

```
cy# [#0x1605a8b0 setHidden:NO]
cy# button = #0x1605a8b0
#"<UIButton: 0x1605a8b0; frame = (266 27; 53 33); hidden = YES; opaque = NO; layer =
<CALayer: 0x16052a00>>"
cy# [button allTargets]
[NSSet setWithArray:@[#"<CKMessageEntryView: 0x160c6180; frame = (0 0; 320 65); opaque =
NO; autoresize = W; layer = <CALayer: 0x16089920>>"]]
cy# [button allControlEvents]
64
cy# [button actionsForTarget:#0x160c6180 forControlEvents:64]
@["touchUpInside:"]
```

As we can see, the response method is [CKMessageEntryView touchUpInsideSendButton:]. Now let’s turn to IDA and LLDB for deeper analysis.

10.3.3 Find suspicious sending action in response method

[CKMessageEntryView touchUpInsideSendButton:] doesn’t do much, as shown in figure 10-57.

```

; CKMessageEntryView - (void)touchUpInsideSendButton:(id)
; Attributes: bp-based frame

; void __cdecl -[CKMessageEntryView touchUpInsideSendButton:]
__CKMessageEntryView_touchUpInsideSendButton__
PUSH    {R4,R7,LR}
MOV     R4, R0
MOV     R0, #(selRef_delegate - 0x268BC7B6) ; selRef_
ADD     R7, SP, #4
ADD     R0, PC ; selRef_delegate
LDR     R1, [R0] ; "delegate"
MOV     R0, R4
BLX    __objc_msgSend
MOVW   R1, #(:lower16:(selRef_messageEntryViewSendBu
MOV     R2, R4
MOVT.W R1, #(:upper16:(selRef_messageEntryViewSendBu
ADD     R1, PC ; selRef_messageEntryViewSendButtonHit
LDR     R1, [R1] ; "messageEntryViewSendButtonHit:"
BLX    __objc_msgSend
MOV     R0, #(selRef_updateEntryView - 0x268BC7DA) ;
ADD     R0, PC ; selRef_updateEntryView
LDR     R1, [R0] ; "updateEntryView"
MOV     R0, R4
POP.W  {R4,R7,LR}
B.W    j__objc_msgSend
; End of function -[CKMessageEntryView touchUpInsideSendButto

```

Figure 10- 57 [CKMessageEntryView touchUpInsideSendButton:button]

It first calls [[self delegate] messageEntryViewSendButtonHit:self] then calls [self updateEntryView]. As their names suggest, the latter method simply refreshes UI; so sending action should come from the former one. Use Cycrypt to find out what's [self delegate]:

```

cy# [#0x160c6180 delegate]
#"<CKTranscriptController: 0x15537200>"

```

Go to [CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView] in IDA. This is a pretty simple method, as shown in figure 10-58.

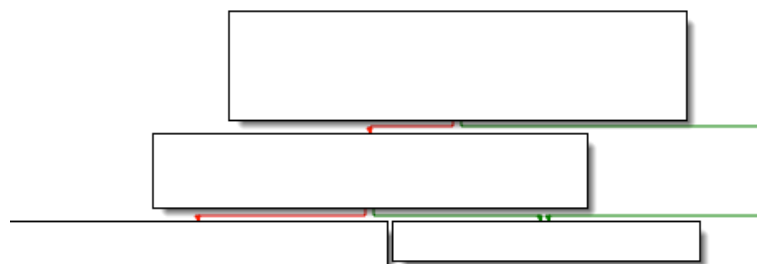


Figure 10- 58 [CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView]

By overviewing this method, I bet you can easily locate the actual sending action in [self sendComposition:[CKMessageEntryView compositionWithAcceptedAutocorrection]]. Let's see what's [self compositionWithAcceptedAutocorrection] in Cycrypt:

```

cy# [#0x160c6180 compositionWithAcceptedAutocorrection]

```

```
#"<CKComposition: 0x160b79d0> text:'iMessage {\n}' subject:'(null)'"
```

It's an object of CKComposition, which clearly contains message text and subject. Keep digging into sendComposition:, as shown in figure 10-59.

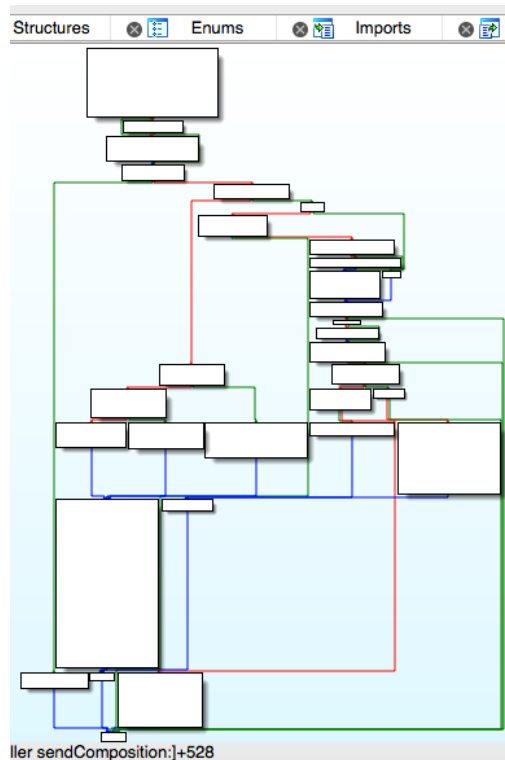


Figure 10- 59 [self sendComposition:]

The implementation is rather complicated. As we said earlier in this section, we'll try to avoid massive use of LLDB, thus let's first go over all branches in this method to glimpse the possible execution flows, then debug the uncertain ones with LLDB. We start from loc_268D427C, as shown in figure 10-60.

```
loc_268D427C
MOV      R0, #(selRef_hasContent - 0x268D4288) ; selRef_hasContent
ADD      R0, PC ; selRef_hasContent
LDR      R5, [R0] ; "hasContent"
MOV      R0, R8
MOV      R1, R5
BLX      _objc_msgSend
TST.W   R0, #0xFF
BEQ      loc_268D4350
```

Figure 10- 60 loc_268D427C

If the iMessage “hasContent”, branches right. According to figure 10-56, our content is “iMessage”, so branch right and arrive at figure 10-61.

```

MOVW    R0, #(:lower16:(selRef_nextMediaObjectToTrimInComposition_ - 0x268D42A4))
MOV     R2, R8
MOVT.W  R0, #(:upper16:(selRef_nextMediaObjectToTrimInComposition_ - 0x268D42A4))
ADD     R0, PC ; selRef_nextMediaObjectToTrimInComposition_
LDR     R1, [R0] ; "nextMediaObjectToTrimInComposition_"
MOV     R0, R4
BLX     _objc_msgSend
CMP     R0, #0
BEQ     loc_268D4366

```

Figure 10- 61 branch

“nextMediaObjectToTrimInComposition:”? Is “media object” referring to image, audio or video kind of things? Since we’re sending plain text, there’s no media at all. Branch right and arrive at figure 10-62.

```

loc_268D4366
LDRB    R0, [R4, R6]
TST.W   R0, #4
BEQ.W   loc_268D4602

```

Figure 10- 62 Branch

What’s R0? Get back to the beginning of sendComposition:, as shown in figure 10-63.

```

BLX     _objc_msgSend
MOV     R0, #(_OBJC_IVAR_$_CKTranscript
ADD     R0, PC ; int _newRecipient:1;
LDR     R6, [R0] ; int _newRecipient:1;
LDRB    R0, [R4, R6]
TST.W   R0, #4
BNE     loc_268D424C

```

Figure 10- 63 Trace R0

R0 turns out to be self->_newRecipient, let’s print its value in Cypriat:

```

cy# #0x15537200->_newRecipient
1

```

So the result of “TST.W R0, #4” is 0, branch right and arrive at loc_268D4604, as shown in figure 10-64.

```

loc_268D4604
MOV      RO, #(selRef_isSendingMessage - 0x268D4610) ; selRef_isSendingMessage
ADD      RO, PC ; selRef_isSendingMessage
LDR      R1, [RO] ; "isSendingMessage"
MOV      RO, R4
BLX      _objc_msgSend
TST.W    RO, #0xFF
BNE.W    loc_268D47C6

```

Figure 10- 64 loc_268D4604

Whether iOS “isSendingMessage”? We don’t know if the timing is before or after pressing “Send” button, so let’s test them both. Before pressing “Send”:

```

cy# [#0x15537200 isSendingMessage]
0

```

And after pressing “Send”:

```

cy# [#0x15537200 isSendingMessage]
0

```

So, [self isSendingMessage] returns 0 anyway. Branch left and arrive at loc_268D4636, as shown in figure 10-65.

```

loc_268D4636
MOVW     R1, #(:lower16:(selRef_canSendToRecipients_alertIfUnable_ - 0x268D4648))
MOV      R2, R6
MOVT.W   R1, #(:upper16:(selRef_canSendToRecipients_alertIfUnable_ - 0x268D4648))
LDR.W    RO, [R4,R10]
ADD      R1, PC ; selRef_canSendToRecipients_alertIfUnable_
MOVS     R3, #1
LDR      R1, [R1] ; "canSendToRecipients:alertIfUnable:"
BLX      _objc_msgSend
TST.W    RO, #0xFF
BEQ.W    loc_268D47C6

```

Figure 10- 65 loc_268D4636

Can we send the iMessage to the recipient? Since the recipient is a valid iMessage account, of course we can! Branch left and arrive at figure 10-66.

```

MOVW     RO, #0
ADD      R3, SP, #0x3C+var_38
STR      RO, [SP,#0x3C+var_38]
MOVW     R1, #(:lower16:(selRef_canSendComposition_error_ - 0x268D466E))
MOV      R2, R8
MOVT.W   R1, #(:upper16:(selRef_canSendComposition_error_ - 0x268D466E))
LDR.W    RO, [R4,R10]
ADD      R1, PC ; selRef_canSendComposition_error_
LDR      R1, [R1] ; "canSendComposition:error:"
BLX      _objc_msgSend
TST.W    RO, #0xFF
BEQ      loc_268D471E

```

Figure 10- 66 Branch

Can we send the composition? Since we’ve already printed the CKComposition object, there doesn’t seem to be any problems. Branch left and arrive at figure 10-67.


```

MOVW    R0, #(:lower16:(selRef_setSendingMessage_ - 0x268D4686))
MOVS    R2, #1
MOVT.W  R0, #(:upper16:(selRef_setSendingMessage_ - 0x268D4686))
ADD     R0, PC ; selRef_setSendingMessage_
LDR.W   R10, [R0] ; "setSendingMessage:"
MOV     R0, R4
MOV     R1, R10
BLX     _objc_msgSend
MOV     R0, R8
MOV     R1, R5
BLX     _objc_msgSend
TST.W   R0, #0xFF
BEQ.W   loc_268D47CE

```

Figure 10- 67 Branch

The branch condition R0 comes from the return value of the 2nd objc_msgSend. Search upwards, we can find R5 in figure 10-60; it's determining if the iMessage "hasContent" again. Therefore, branch right and arrive at figure 10-68.

```

MOV     R0, #(selRef_activeKeyboard - 0x268D46B4) ; selRef_activeKeyboard
MOV     R2, #(classRef_UIKeyboard - 0x268D46B6) ; classRef_UIKeyboard
ADD     R0, PC ; selRef_activeKeyboard
ADD     R2, PC ; classRef_UIKeyboard
LDR     R1, [R0] ; "activeKeyboard"
LDR     R0, [R2] ; _OBJC_CLASS_$_UIKeyboard
BLX     _objc_msgSend
MOV     R1, #(selRef_removeAutocorrectPrompt - 0x268D46C8) ; selRef_removeAutocorrectPrompt
ADD     R1, PC ; selRef_removeAutocorrectPrompt
LDR     R1, [R1] ; "removeAutocorrectPrompt"
BLX     _objc_msgSend
MOV     R0, R4
MOV     R1, R11
BLX     _objc_msgSend
MOV     R1, #(selRef_view - 0x268D46E0) ; selRef_view
ADD     R1, PC ; selRef_view
LDR     R1, [R1] ; "view"
BLX     _objc_msgSend
MOVW    R1, #(:lower16:(selRef_setUserInteractionEnabled_ - 0x268D46F2))
MOVS    R2, #0
MOVT.W  R1, #(:upper16:(selRef_setUserInteractionEnabled_ - 0x268D46F2))
ADD     R1, PC ; selRef_setUserInteractionEnabled_
LDR     R1, [R1] ; "setUserInteractionEnabled:"
BLX     _objc_msgSend
MOV     R0, #(selRef_updateNavigationButtons - 0x268D4702) ; selRef_updateNavigationButtons
ADD     R0, PC ; selRef_updateNavigationButtons
LDR     R1, [R0] ; "updateNavigationButtons"
MOV     R0, R4
BLX     _objc_msgSend
MOVW    R0, #(:lower16:(selRef_sendMessage_ - 0x268D4716))
MOV     R2, R8
MOVT.W  R0, #(:upper16:(selRef_sendMessage_ - 0x268D4716))
ADD     R0, PC ; selRef_sendMessage_
LDR     R1, [R0] ; "sendMessage:"
MOV     R0, R4
BLX     _objc_msgSend
B       loc_268D47C6

```

Figure 10- 68 Branch

This is an informative figure. If you look close, you'll discover that most objc_msgSends are just refreshing UI, making the last objc_msgSend, i.e. [R4 sendMessage:R2] more eye-catching. What's R4 and R2? Look upwards, you'll see they're CKTranscriptController and the argument of [self sendComposition:], respectively. Let's continue analyzing from [CKTranscriptController

sendMessage:], as shown in figure 10-69.

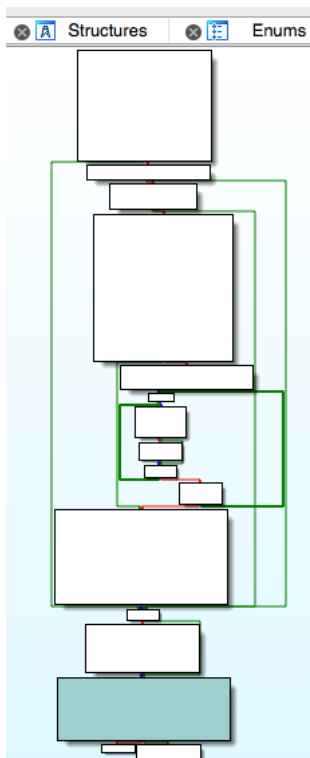


Figure 10- 69 [CKTranscriptController sendMessage:]

Another method full of branches. But after giving a glimpse to the possible execution flows just like what we did to sendComposition:, we can find that most branches are just making preparations, only “_startCreatingNewMessageForSending:” looks promising. Let’s take a look at its implementation, as shown in figure 10-70.

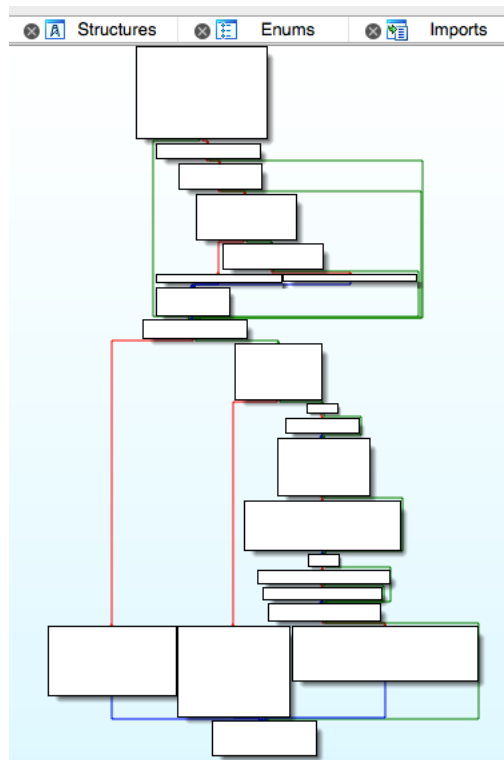


Figure 10- 70 [CKTranscriptController _startCreatingNewMessageForSending:]

Again, it's a method full of branches. Overview the implementation, I think you'll notice the method "sendMessage:newComposition:" just like me. The method occurs twice in [CKTranscriptController _startCreatingNewMessageForSending:], as shown in the 2 red blocks in figure 10-71.

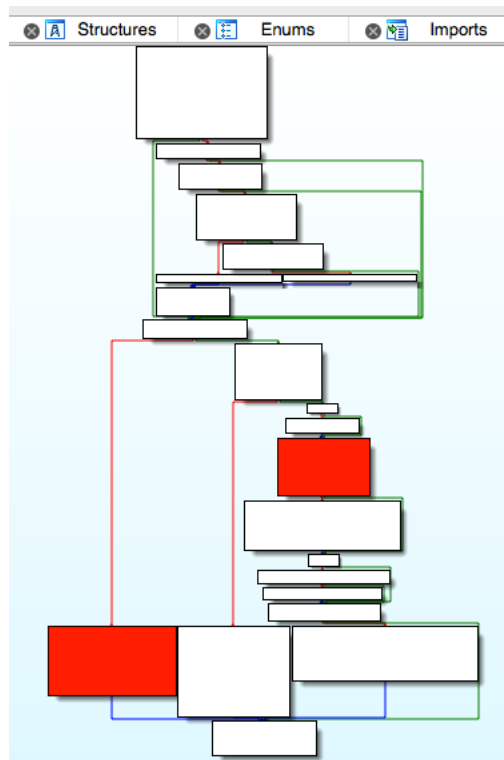


Figure 10- 71 [CKTranscriptController_startCreatingNewMessageForSending:]

Take a look at the implementation of this method, as shown in figure 10-72.

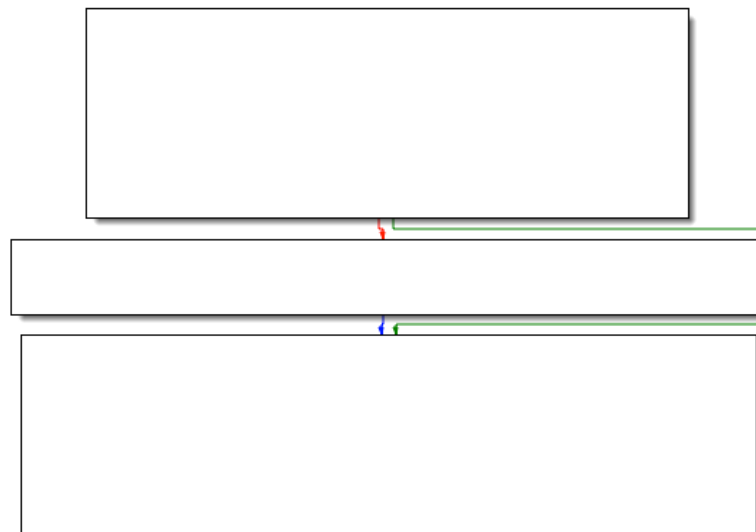


Figure 10- 72 [CKConversation sendMessage:newComposition:]

It further calls “sendMessage:onService:newComposition:”, so proceed to this method, as shown in figure 10-73.

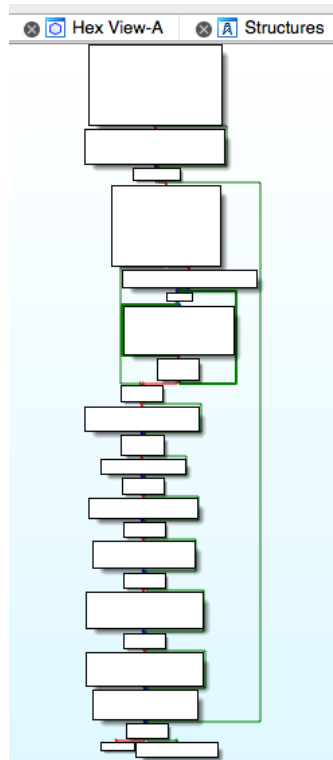


Figure 10- 73 [CKConversation sendMessage:onService:newComposition:]

The execution flow of this method is more straightforward than the previous ones. Skim it briefly, we can see phrases like “Sending message with guid: %@”, “ => Sending account: %@” and “=> Recipients: [%@]”, most of which are arguments of _CKLogExternal. If MobileSMS has already started recording these into syslog, doesn’t it prove that “send iMessage” is happening? What’s more, we’ve seen the suspicious keyword “sendMessage:” again in figure 10-74:

```

loc_2691F836
MOVW    R0, #(:lower16:(selRef_sendMessage_ - 0x2691F844))
MOV     R2, R6
MOVT.W R0, #(:upper16:(selRef_sendMessage_ - 0x2691F844))
ADD     R0, PC ; selRef_sendMessage_
LDR     R1, [R0] ; "sendMessage:"
MOV     R0, R5
BLX     _objc_msgSend
MOV     R0, #(selRef_recordRecentContact - 0x2691F856) ; selRef_recordRecentContact
ADD     R0, PC ; selRef_recordRecentContact
LDR     R1, [R0] ; "_recordRecentContact"
MOV     R0, R10
BLX     _objc_msgSend
  
```

Figure 10- 74 loc_2691f836

What’s the receiver and arguments of “sendMessage:”? Let’s find them in IDA; the receiver, R0, comes from R5. Where does R5 come from? Keep looking upwards until loc_2691F726, as shown in figure 10-75.

```

loc_2691F726
MOVS      R0, #0x12
BL        __CKShouldLogExternal
LDR.W     R8, [SP, #0xA4+var_90]
TST.W     R0, #0xFF
LDR.W     R10, [SP, #0xA4+var_94]
LDR       R5, [SP, #0xA4+var_98]
BEQ       loc_2691F74E

```

Figure 10- 75 loc_2691f726

The instruction “LDR R5, [SP,#0xA4+var_98]” decides R5. Well, what’s [SP,#0xA4+var_98]? Do you remember how we’ve solved this kind of problems in section 10.2? Place the cursor on var_98 and press “x” to view its cross references, as shown in figure 10-76.

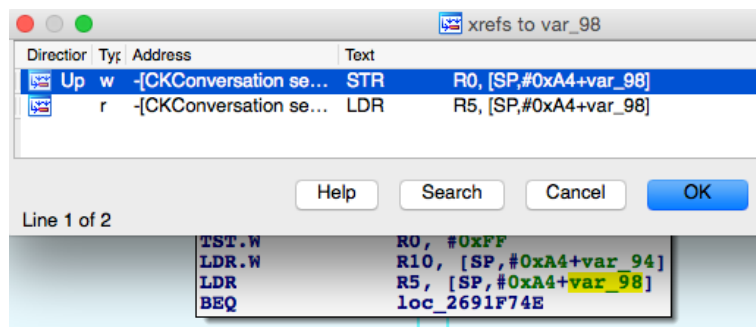


Figure 10- 76 Inspect cross references

Double click the first xref to jump to “STR R0, [SP,#0xA4+var_98]”. Around here, R0 comes from [R6 chat]; R6 first appears in the beginning of [CKConversation sendMessage:onService:newComposition:], it’s “self”; so the receiver of “sendMessage:” is [self chat]. Back to figure 10-74, we can see the argument of “sendMessage:” is from R6. Go a little upwards to loc_2691F6F4, R6 is set in “LDR R6, [SP,#0xA4+var_80]”, as shown in figure 10-77.

```

loc_2691F6F4
MOVS      R0, #0x12
BL        __CKShouldLogExternal
LDR       R6, [SP, #0xA4+var_80]
TST.W     R0, #0xFF
BEQ       loc_2691F726

```

Figure 10- 77 loc_2691f6f4

What’s next? We’ve performed the same operation just now, so I’ll leave some figures (from

10-78 to 10-80) rather than texts as references for you to follow:

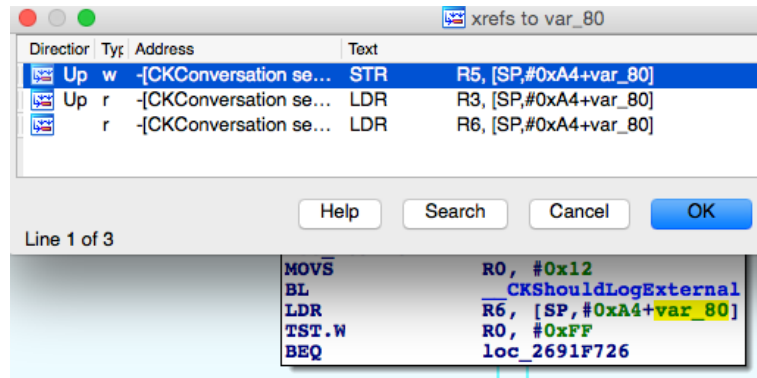


Figure 10- 78 Inspect cross references

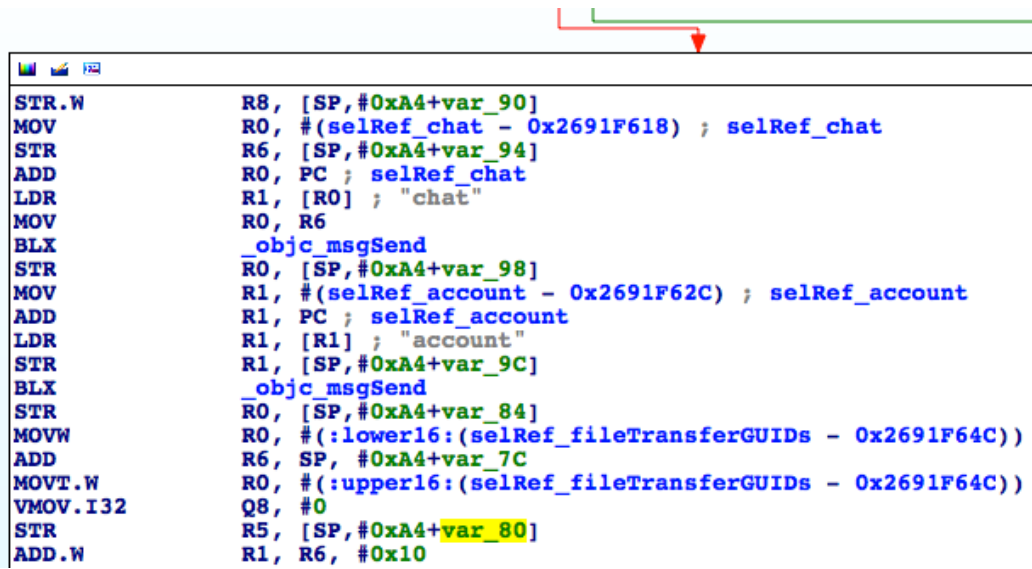


Figure 10- 79 [CKConversation setChat:]

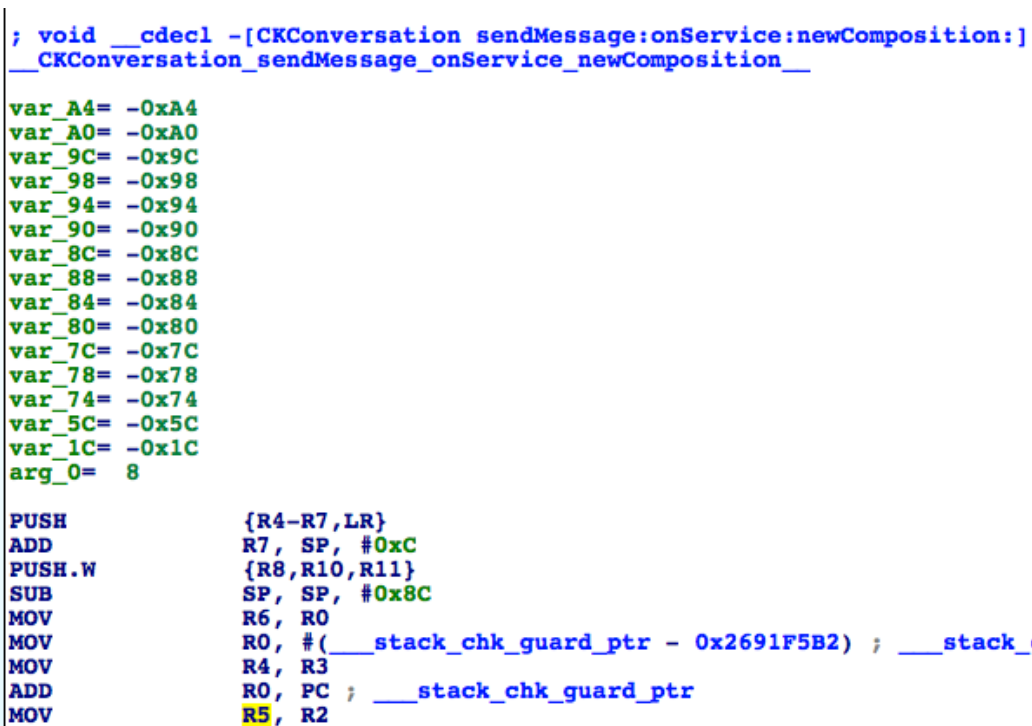


Figure 10- 80 [CKConversation sendMessage:onService:newComposition:]

So the argument of `[[self chat] sendMessage:]` is exactly the first argument of `[self sendMessage:onService:newComposition:]`. Well, what're the types and values of `[self chat]` and the argument? We've gone out of clue in IDA, so it's time to bring out LLDB.

First compose an `iMessage`, then set a breakpoint on the `objc_msgSend` right under "sendMessage:" in figure 10-74, which is at the end of `[CKConversation sendMessage:onService:newComposition:]`. After that, press "Send" button to trigger the breakpoint:

```
Process 233590 stopped
* thread #1: tid = 0x39076, 0x30ad1846 ChatKit`-[CKConversation
sendMessage:onService:newComposition:] + 686, queue = 'com.apple.main-thread, stop
reason = breakpoint 1.1
  frame #0: 0x30ad1846 ChatKit`-[CKConversation sendMessage:onService:newComposition:]
+ 686
ChatKit`-[CKConversation sendMessage:onService:newComposition:] + 686:
-> 0x30ad1846: blx    0x30b3bf44                ; symbol stub for:
MarcoShouldLogMadridLevel$shim
  0x30ad184a: movw   r0, #49322
  0x30ad184e: movt   r0, #2541
  0x30ad1852: add    r0, pc
(lldb) p (char *)$r1
(char *) $0 = 0x32b26146 "sendMessage:"
(lldb) po $r0
<IMChat 0x5ef2ce0> [Identifier: snakeninny@icloud.com  GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com  Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363  Style: -  State: 3  Participants: 1  Room Name:
(null)  Display Name: (null)  Last Addressed: (null)  Group ID: F399B0B5-800F-47A4-A66C-
72C43ACC0428  Unread Count: 0  Failure Count: 0]
(lldb) po $r2
IMessage[from=(null); msg-subject=(null); account:(null); flags=100005; subject='<<
Message Not Loggable >>' text='<< Message Not Loggable >>' messageID: 0 GUID:'966C2CD6-
3710-4D0F-BCEF-BCFEE8E60FE9' date:'437730968.559627' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO finished: YES sent: NO read: NO
delivered: NO audio: NO played: NO from-me: YES emote: NO dd-results: NO dd-scanned: YES
error: (null)]
(lldb) ni
```

The output contains exactly what we want: `[IMChat sendMessage:IMMessage]`. There's one thing to mention: after printing out all necessary information, I've executed an extra "ni" command and heard a familiar "message sent" text tone. This phenomenon indicates that the operation of "send iMessage" is indeed performed inside `[IMChat sendMessage:IMMessage]`. Because the prefixes of `IMChat` and `IMMessage` are both `IM`, they come from a library other than `ChatKit`; the lowest level "send iMessage" function in `ChatKit` stops at `[CKConversation sendMessage:onService:newComposition:]`. We can confirm for now that if we're able to construct an `IMChat` object and an `IMMessage` object, we can successfully send an `iMessage`. Old problems solved, new problems occur: how do we compose these 2 objects? Let's see if

there're any clues in class-dump headers.

To compose objects of IMChat and IMessage from scratch, we need to see if there're any constructors or initializers in their headers. Let's start from IMChat.h and search for methods with the name "init":

```
- (id)_initWithDictionaryRepresentation:(id)arg1 items:(id)arg2
participantsHint:(id)arg3 accountHint:(id)arg4;
- (id)init;
- (id)_initWithGUID:(id)arg1 account:(id)arg2 style:(unsigned char)arg3
roomName:(id)arg4 displayName:(id)arg5 items:(id)arg6 participants:(id)arg7;
```

Although they seem to be initializers, there're various arguments, which we don't know how to compose. The clues break, what's next?

Do you still remember how we managed to find the receiver of "sendMessage:"? Yes, it's [self chat]; self is a CKConversation object. Since [CKConversation chat] returns an IMChat object, let's analyze this method in IDA to see if there's any clue, as shown in figure 10-81.

```
; CKConversation - (id)chat
; id __cdecl -[CKConversation chat](struct CKConversation *self, SEL)
CKConversation chat
MOV     R1, #(_OBJC_IVAR_$_CKConversation._chat - 0x26920778); IMChat *_chat;
ADD     R1, PC; IMChat *_chat;
LDR     R1, [R1]; IMChat *_chat;
LDR     R0, [R0, R1]
BX      LR
; End of function -[CKConversation chat]
```

Figure 10- 81 [CKConversation chat]

[CKConversation chat] simply returns the instance variable _chat. This scenario is quite familiar, isn't it? We've met a similar situation analyzing _composeSendingService in figure 10-22. Once again, we have to let LLDB take the job for now. Delete this iMessage conversation (to delete this CKConversation object) and create a new iMessage (to create a new CKConversation object), then set a breakpoint on [CKConversation setChat:]. Press "Send" to trigger the breakpoint:

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30ad277c ChatKit`-[CKConversation setChat:], queue =
'com.apple.main-thread, stop reason = breakpoint 13.1
  frame #0: 0x30ad277c ChatKit`-[CKConversation setChat:]
ChatKit`-[CKConversation setChat:]
-> 0x30ad277c: movw   r3, #55168
      0x30ad2780: movt   r3, #2541
      0x30ad2784: add    r3, pc
      0x30ad2786: ldr    r3, [r3]
(lldb) po $r2
```

```

<IMChat 0x1594f7e0> [Identifier: snakeninny@icloud.com GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $20 = 0x30acf625

```

LR without offset is $0x30acf625 - 0xa1b2000 = 0x2691d625$, it's inside `[CKConversation initWithChat:]`. Since `IMChat` is the argument, to trace its source, we have to find out the method caller. Repeat the previous operations to recreate a new `iMessage`, then set a breakpoint at the beginning of `[CKConversation initWithChat:]` and trigger it:

```

Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30acf5ec ChatKit`-[CKConversation initWithChat:], queue =
'com.apple.main-thread, stop reason = breakpoint 14.1
  frame #0: 0x30acf5ec ChatKit`-[CKConversation initWithChat:]
ChatKit`-[CKConversation initWithChat:]
-> 0x30acf5ec: push    {r4, r5, r6, r7, lr}
   0x30acf5ee: add     r7, sp, #12
   0x30acf5f0: push.w {r8, r10, r11}
   0x30acf5f4: sub     sp, #8
(lldb) po $r2
<IMChat 0x1470a520> [Identifier: snakeninny@icloud.com GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $22 = 0x30a8d131

```

LR without offset is $0x30a8d131 - 0xa1b2000 = 0x268db131$, which is inside `[CKConversationList _beginTrackingConversationWithChat:]`. Again, it's the argument, so let's continue tracing the method caller:

```

Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30a8d09c ChatKit`-[CKConversationList
_beginTrackingConversationWithChat:], queue = 'com.apple.main-thread, stop reason =
breakpoint 15.1
  frame #0: 0x30a8d09c ChatKit`-[CKConversationList
_beginTrackingConversationWithChat:]
ChatKit`-[CKConversationList _beginTrackingConversationWithChat:]
-> 0x30a8d09c: push    {r4, r5, r6, r7, lr}
   0x30a8d09e: mov     r5, r0
   0x30a8d0a0: movs   r0, #25
   0x30a8d0a2: add    r7, sp, #12
(lldb) po $r2
<IMChat 0x15a326a0> [Identifier: snakeninny@icloud.com GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $24 = 0x30a8d4f1

```

LR without offset is $0x30a8d4f1 - 0xa1b2000 = 0x268db4f1$, which is inside

[CKConversationList _handleRegistryDidRegisterChatNotification:]; you'll see in your IDA that this time IMChat is from [notification object] instead of the argument, which is a notification. Since this IMChat object is passed through a notification, to trace its source, we have to find the poster of this notification instead of the caller of [CKConversationList _handleRegistryDidRegisterChatNotification:]. Let's set a breakpoint on the base address of this method and take a look at the structure of notification:

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30a8d4ac ChatKit`-[CKConversationList
_handleRegistryDidRegisterChatNotification:], queue = 'com.apple.main-thread, stop
reason = breakpoint 16.1
    frame #0: 0x30a8d4ac ChatKit`-[CKConversationList
_handleRegistryDidRegisterChatNotification:]
ChatKit`-[CKConversationList _handleRegistryDidRegisterChatNotification:]
-> 0x30a8d4ac: push    {r4, r5, r6, r7, lr}
    0x30a8d4ae: add     r7, sp, #12
    0x30a8d4b0: push.w {r8, r10, r11}
    0x30a8d4b4: sub.w  r4, sp, #64
(lldb) po $r2
NSConcreteNotification 0x15934340 {name = __kIMChatRegistryDidRegisterChatNotification;
object = <IMChat 0x147c39f0> [Identifier: snakeninny@icloud.com GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]}
```

The name of the notification is “__kIMChatRegistryDidRegisterChatNotification”. To find out its poster, a good solution is to grep the whole filesystem and see what binaries contain the notification name, as shown below:

```
FunMaker-5:~ root# grep -r _handleRegistryDidRegisterChatNotification: /System/
Binary file /System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7s matches
grep: /System/Library/Caches/com.apple.dyld/enable-dylibs-to-override-cache: No such
file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGCorePDF.dylib: No
such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltin.dylib:
No such file or directory
grep: /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMaps.dylib: No
such file or directory
grep: /System/Library/Frameworks/System.framework/System: No such file or directory
```

The keyword appears in the cache. Naturally, let's grep those decached files:

```
snakeninnys-MacBook:~ snakeninny$ grep -r __kIMChatRegistryDidRegisterChatNotification
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//dyld_shared_cache_armv7s matches
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//System/Library/Caches/com.apple.xp
c/sdk.dylib: Too many levels of symbolic links
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//System/Library/Frameworks/OpenGL
ES.framework/libLLVMContainer.dylib: Too many levels of symbolic links
```

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

You may have already guessed from the results that both IMCore and ChatKit are in charge of iMessage related operations, but IMCore is lower level than ChatKit; ChatKit receives the commands from the user and hands them to IMCore for processing, then IMCore passes the result back to ChatKit for UI animation. By way of analogy, you can consider MobileSMS as a restaurant, ChatKit as the waiter and IMCore as the cook. Can you get it?

Naturally, drag and drop IMCore into IDA and search for “__kIMChatRegistryDidRegisterChatNotification” globally, the results are shown in figure 10-82.

Address	Function
__text:2908426A	-[IMChatRegistry _registerChatDictionary:forChat:isIncoming:newGUID:]
__text:29084278	-[IMChatRegistry _registerChatDictionary:forChat:isIncoming:newGUID:]
__cstring:290BABE2	
__const:31241FD0	
__cstring:312472E8	

Figure 10- 82 Occurrences of “__kIMChatRegistryDidRegisterChatNotification” in IDA

Good. Let’s double click the first row and take a look at its context, as shown in figure 10-83.

```
loc_2908423E
MOV     R0, #(selRef_defaultCenter - 0x29084252) ; selRef_defaultCenter
MOV     R2, #(classRef_NSNotificationCenter - 0x29084254) ; classRef_NSNot
ADD     R0, PC ; selRef_defaultCenter
ADD     R2, PC ; classRef_NSNotificationCenter
LDR     R1, [R0] ; "defaultCenter"
LDR     R0, [R2] ; _OBJC_CLASS_$_NSNotificationCenter
MOVS   R2, #1
STR     R2, [SP,#0x98+var_48]
BLX    _objc_msgSend
MOVW   R1, #(:lower16:(selRef___mainThreadPostNotificationName_object_use
MOVS   R5, #2
MOVT.W R1, #(:upper16:(selRef___mainThreadPostNotificationName_object_use
LDR     R3, [SP,#0x98+var_60]
MOV     R2, #(cfstr___kimchatregis - 0x2908427C) ; "__kIMChatRegistryDidRe
ADD     R1, PC ; selRef___mainThreadPostNotificationName_object_userInfo_
LDR     R6, [SP,#0x98+var_50]
LDR     R1, [R1] ; "___mainThreadPostNotificationName:object"...
ADD     R2, PC ; "___kIMChatRegistryDidRegisterChatNotification"
STR     R5, [SP,#0x98+var_48]
STR     R6, [SP,#0x98+var_98]
BLX    _objc_msgSend
```

Figure 10- 83 loc_2908423E

After seeing the keyword “PostNotification”, we know the notification that ChatKit received is right from here. Since IMChat is the 2nd argument, i.e. R3, and R3 comes from [SP, #0x98+var_60]. You know what to do by referring to figure 10-84 and figure 10-85.

Director	Ty	Address	Text
Up	w	-[IMChatRegistry_re...	STR R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R5, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR.W R8, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R1, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry_re...	LDR R2, [SP,#0x98+var_60]
...	r	-[IMChatRegistry_re...	LDR R3, [SP,#0x98+var_60]
...	r	-[IMChatRegistry_re...	LDR R0, [SP,#0x98+var_60]

Figure 10- 84 Inspect cross references

```

; void __cdecl -[IMChatRegistry_registerChatDictionary:forChat:isIncoming:newGUID:]
__IMChatRegistry_registerChatDictionary_forChat_isIncoming_newGUID__
var_98= -0x98
var_94= -0x94
var_90= -0x90
var_70= -0x70
var_6C= -0x6C
var_68= -0x68
var_64= -0x64
var_60= -0x60
var_5C= -0x5C
var_58= -0x58
var_54= -0x54
var_50= -0x50
var_4C= -0x4C
var_48= -0x48
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_18= -0x18
arg_0= 8
arg_4= 0xC

PUSH {R4-R7,LR}
ADD R7, SP, #0xC
PUSH.W {R8,R10,R11}
SUB.W R4, SP, #0x40
BIC.W R4, R4, #0xF
MOV SP, R4
VST1.64 {D8-D11}, [R4@128]!
VST1.64 {D12-D15}, [R4@128]
SUB SP, SP, #0x80
MOV R5, R0
MOV R0, #(selRef_shouldRegisterChat - 0x29083972) ; selRef_shouldRegis
STR R3, [SP,#0x98+var_60]

```

Figure 10- 85 [IMChatRegistry_registerChatDictionary:forChat:isIncoming:newGUID:]

According to the above figures, IMChat comes from the 2nd argument of [IMChatRegistry_registerChatDictionary:forChat:isIncoming:newGUID:], whose caller is:

```

Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x33235944 IMCore`___lldb_unnamed_function2048$$IMCore,
queue = 'com.apple.main-thread, stop reason = breakpoint 17.1

```

```

    frame #0: 0x33235944 IMCore`___lldb_unnamed_function2048$$IMCore
IMCore`___lldb_unnamed_function2048$$IMCore:
-> 0x33235944: push   {r4, r5, r6, r7, lr}
    0x33235946: add    r7, sp, #12
    0x33235948: push.w {r8, r10, r11}
    0x3323594c: sub.w  r4, sp, #64
(lldb) po $r3
<IMChat 0x147c7f30> [Identifier: snakeninny@icloud.com  GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com  Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363  Style: -  State: 0  Participants: 1  Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $27 = 0x3323646f

```

LR without offset is $0x3323646f - 0xa1b2000 = 0x2908446F$, which is located inside

[IMChatRegistry _registerChat:isIncoming:guid:]. Keep tracing the caller:

```

Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x3323644c IMCore`___lldb_unnamed_function2049$$IMCore,
queue = 'com.apple.main-thread, stop reason = breakpoint 20.1
    frame #0: 0x3323644c IMCore`___lldb_unnamed_function2049$$IMCore
IMCore`___lldb_unnamed_function2049$$IMCore:
-> 0x3323644c: push   {r4, r5, r7, lr}
    0x3323644e: add    r7, sp, #8
    0x33236450: sub    sp, #8
    0x33236452: movw  r1, #9840
(lldb) po $r2
<IMChat 0x15972f20> [Identifier: snakeninny@icloud.com  GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com  Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363  Style: -  State: 0  Participants: 1  Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0
Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $30 = 0x33237173

```

LR without offset is $0x33237173 - 0xa1b2000 = 0x29085173$, which is located inside

[IMChatRegistry chatForIMHandle:]. Meanwhile, the 1st argument of [IMChatRegistry _registerChat:isIncoming:guid:], i.e. IMChat, is from R5; at the end of [IMChatRegistry chatForIMHandle:], R5 appears as the return value. In other words, [IMChatRegistry chatForIMHandle:] returns an IMChat object! Further more, as the name suggests, IMChatRegistry is a class for registering chats, so getting an IMChat object from this class is quite reasonable. Old questions go, new questions come: How do we get an IMChatRegistry object and the argument of chatForIMHandle:? Let's get to them one by one, starting from IMChatRegistry.


```

13 @interface IMChatRegistry : NSObject <NSFastEnumeration>
14 {
15     NSMutableArray *_allChats;
16     NSMutableDictionary *_chatGUIDToCurrentThreadMap;
17     NSMutableDictionary *_chatGUIDToInfoMap;
18     NSMutableDictionary *_chatGUIDToChatMap;
19     NSMutableDictionary *_threadNameToChatMap;
20     NSMutableDictionary *_chatGUIDToMessageSentOrReceivedMap;
21     NSMutableArray *_allChatsInThreadNameMap;
22     NSMutableArray *_pendingQueries;
23     NSMutableArray *_waitingForQueries;
24     NSString *_historyModificationStamp;
25     IMTimer *_markAsReadTimer;
26     double _timerStartTimeInterval;
27     BOOL _firstLoad;
28     BOOL _loading;
29     BOOL _daemonHadTerminated;
30     BOOL _wantsHistoryReload;
31     BOOL _postMessageSentNotifications;
32     unsigned int _defaultNumberOfMessagesToLoad;
33     unsigned int _daemonUnreadCount;
34     long long _daemonLastFailedMessageID;
35     NSUserActivity *_userActivity;
36 }
37
38 + (Class)messageClass;
39 + (void)setMessageClass:(Class)arg1;
40 + (Class)chatClass;
41 + (void)setChatClass:(Class)arg1;
42 + (Class)chatRegistryClass;
43 + (void)setChatRegistryClass:(Class)arg1;
44 + (id)sharedInstance;

```

Figure 10- 86 IMChatRegistry.h

According to line 44, we know that IMChatRegistry is a singleton, we can get the registry by calling [IMChatRegistry sharedInstance]. So easy!

Next question, where does the argument of chatForIMHandle: come from? It definitely comes from its caller. It's LLDB's show time again.

```

Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x33236d8c IMCore`___lldb_unnamed_function2054$$IMCore,
queue = 'com.apple.main-thread, stop reason = breakpoint 21.1
  frame #0: 0x33236d8c IMCore`___lldb_unnamed_function2054$$IMCore
IMCore`___lldb_unnamed_function2054$$IMCore:
-> 0x33236d8c: push   {r4, r5, r6, r7, lr}
0x33236d8e: add   r7, sp, #12
0x33236d90: str   r11, [sp, #-4]!
0x33236d94: sub   sp, #20
(lldb) po $r2
[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account:
P:+86PhoneNumber]
(lldb) p/x $lr
(unsigned int) $32 = 0x30a8dca5

```

LR without offset is $0x30a8dca5 - 0xa1b2000 = 0x268dbca5$, which is not located inside

IMCore anymore. Like we've just said, we're jumping between IMCore and ChatKit, and ChatKit's ASLR offset happens to be 0xa1b2000 too, so let's head to ChatKit to see if 0x268dbca5 is there:

```

__text:268DBCA0          BLX          _objc_msgSend
__text:268DBCA4          MOV          R2, R0
__text:268DBCA6          MOV          R0, #(selRef_conversationForChat_ - 0x268DBC2) ; selRef_co
__text:268DBCAE          ADD          R0, PC ; selRef_conversationForChat_
__text:268DBC0           LDR          R1, [R0] ; "_conversationForChat:"
__text:268DBC2           MOV          R0, R6
__text:268DBC4           ADD          SP, SP, #8
__text:268DBC6           POP.W       {R4-R7,LR}
__text:268DBC8           B.W         j__objc_msgSend
__text:268DBCBA ; End of function -[CKConversationList conversationForHandles:displayName:joinedChatsOnly:cre

```

Figure 10- 87 [CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]

0x268dbca5 is inside [CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:], whose 1st argument is the source of the argument of chatForIMHandle:. Keep tracing the caller:

```

Process 292950 stopped
* thread #1: tid = 0x47856, 0x30a8dc60 ChatKit`-[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create:], queue = 'com.apple.main-
thread, stop reason = breakpoint 1.1
    frame #0: 0x30a8dc60 ChatKit`-[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create:]
ChatKit`-[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create:]
-> 0x30a8dc60: push   {r4, r5, r6, r7, lr}
    0x30a8dc62: add    r7, sp, #12
    0x30a8dc64: sub    sp, #8
    0x30a8dc66: mov    r6, r0
(lldb) po $r2
<__NSArrayM 0x178d2290>(
[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account:
P:+86PhoneNumber]
)

(lldb) p/x $lr
(unsigned int) $1 = 0x30a84efd

```

LR without offset is 0x30a84efd - 0xa1b2000 = 0x268d2efd, which is located inside [CKTranscriptController sendMessage:]. Can you believe it? We've walked through a big circle and returned to our starting point, which brings us a mixed feeling. Keep calm and carry on, let's see how this NSArray is composed, as shown in figure 10-88.


```

MOVSW          R2, #1
MOVT.W        R1, #(:upper16:(selRef_conversationForHandles_displa
LDR           R6, [SP,#0xA8+var_80]
MOVSW          R3, #0
ADD           R1, PC ; selRef_conversationForHandles_displayName_j
STR           R2, [SP,#0xA8+var_A8]
LDR           R1, [R1] ; "conversationForHandles:displayName:join"
STR           R2, [SP,#0xA8+var_A4]
MOV           R2, R6
BLX          _objc_msgSend

```

Figure 10- 88 Tracing the NSArray

R2 comes from R6, and R6 comes from [SP, #0xA8+var_80]. The same pattern has reappeared, so as usual, I'll replace text illustration with figure references, as shown in figure 10-89 and 10-90.

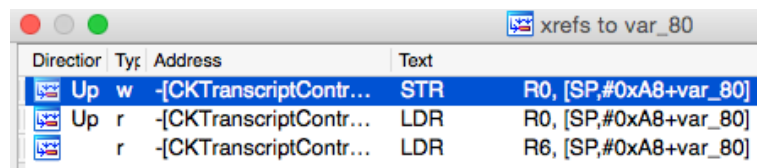


Figure 10- 89 Inspect cross references

```

LDR           R1, [R0] ; "alloc"
LDR           R0, [R2] ; _OBJC_CLASS_$_NSMutableArray
BLX          _objc_msgSend
MOV           R6, R0
MOV           R0, #(selRef_count - 0x268D2DA6) ; selRef_count
ADD           R0, PC ; selRef_count
LDR           R1, [R0] ; "count"
MOV           R0, R8
BLX          _objc_msgSend
MOV           R2, R0
MOV           R0, #(selRef_initWithCapacity_ - 0x268D2DBA) ;
ADD           R0, PC ; selRef_initWithCapacity_
LDR           R1, [R0] ; "initWithCapacity:"
MOV           R0, R6
BLX          _objc_msgSend
STR           R0, [SP,#0xA8+var_80]

```

Figure 10- 90 [CKTranscriptController sendMessage:]

You may have already found that things are getting a little bit different. “STR R0, [SP,#0xA8+var_80]” is just storing an initialized NSMutableArray into [SP, #0xA8+var_80], it doesn't contain any IMHandle yet. Hehe, since it's an NSMutableArray, it can be extended by addObject:, which could happen in the 2nd “LDR R0, [SP,#0xA8+var_80]” of figure 10-89. Let's jump there for a look, as shown in figure 10-91.

```

LDR      R0, [SP,#0xA8+var_84]
MOV      R1, R8
MOVS     R3, #0
BLX      _objc_msgSend
MOV      R2, R0
LDR      R0, [SP,#0xA8+var_80]
MOV      R1, R11
BLX      _objc_msgSend

```

Figure 10- 91 Trace IMHandle

You'll find it is indeed an addObject:, and by its context, you'll see its argument comes from imHandleWithID:alreadyCanonical:. As the name suggests, it returns an IMHandle object. It's getting closer, let's set a breakpoint on the first objc_msgSend in figure 10-91 to reconstruct the prototype of imHandleWithID:alreadyCanonical:.

```

Process 343388 stopped
* thread #1: tid = 0x53d5c, 0x30a84e98 ChatKit`-[CKTranscriptController sendMessage:] +
516, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1
  frame #0: 0x30a84e98 ChatKit`-[CKTranscriptController sendMessage:] + 516
ChatKit`-[CKTranscriptController sendMessage:] + 516:
-> 0x30a84e98: blx    0x30b3bf44          ; symbol stub for:
MarcoShouldLogMadridLevel$shim
  0x30a84e9c: mov    r2, r0
  0x30a84e9e: ldr    r0, [sp, #40]
  0x30a84ea0: mov    r1, r11
(lldb) p (char *)$r1
(char *) $0 = 0x30b55fb4 "imHandleWithID:alreadyCanonical:"
(lldb) po $r0
IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Service:
IMService[iMessage] Login: P:+86PhoneNumber Active: YES LoginStatus: Connected]
(lldb) po $r2
snakeninny@icloud.com
(lldb) p $r3
(unsigned int) $3 = 0

```

Both arguments are revealed; the 1st is my iMessage address, the 2nd is 0, i.e. NO in BOOL. The receiver is an IMAccount object, where is it from? As shown in figure 10-91, R0 comes from [SP, #0xA8+var_84], so according to figure 10-92 and 10-93, IMAccount comes from [[IMAccountController sharedInstance] __ck_defaultAccountForService:[CKConversation sendingService]].

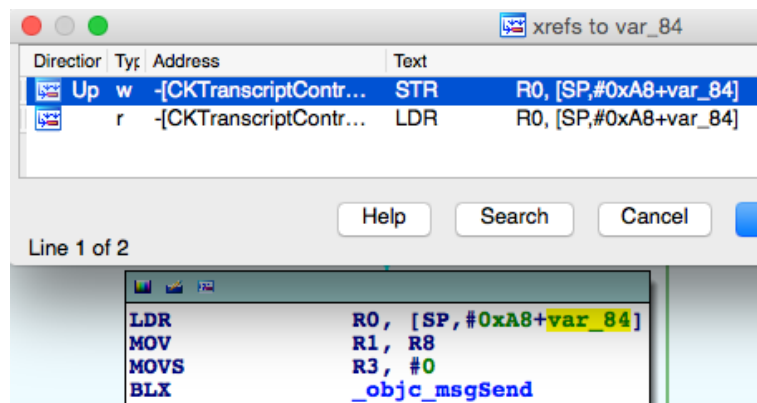


Figure 10- 92 Inspect cross references

```

LDR      R1, [R0] ; "sharedInstance"
LDR      R0, [R2] ; _OBJC_CLASS_$_IMAccountController
BLX      _objc_msgSend
MOVW     R1, #(:lower16:(selRef__ck_defaultAccountForService_
MOV      R2, R4
MOVT.W   R1, #(:upper16:(selRef__ck_defaultAccountForService_
ADD      R1, PC ; selRef__ck_defaultAccountForService_
LDR      R1, [R1] ; "__ck_defaultAccountForService:"
BLX      _objc_msgSend
STR      R0, [SP,#0xA8+var_84]

```

Figure 10- 93 [CKTranscriptController sendMessage:]

OK, let's figure out what's [CKConversation sendingService]. Set a breakpoint on the 2nd objc_msgSend of figure 10-93 and trigger it:

```

Process 343388 stopped
* thread #1: tid = 0x53d5c, 0x30a84e08 ChatKit`-[CKTranscriptController sendMessage:] +
372, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1
  frame #0: 0x30a84e08 ChatKit`-[CKTranscriptController sendMessage:] + 372
ChatKit`-[CKTranscriptController sendMessage:] + 372:
-> 0x30a84e08: blx    0x30b3bf44          ; symbol stub for:
MarcoShouldLogMadridLevel$shim
0x30a84e0c: str    r0, [sp, #36]
0x30a84e0e: movw  r0, #23756
0x30a84e12: add   r2, sp, #44
(lldb) p (char *)$r1
(char *) $4 = 0x30b55f95 "__ck_defaultAccountForService:"
(lldb) po $r2
IMService[iMessage]
(lldb) po [$r2 class]
IMServiceImpl

```

So it's an IMServiceImpl object. How do we get such an object? In fact, we've already done this in section 10.2. Open IMServiceImpl.h, as shown in figure 10-94.

```

6
7 #import <IMCore/IMService.h>
8
9 @class IMAccount, NSArray, NSData, NSDictionary, NSMutableDictionary, NSString;
10
11 @interface IMServiceImpl : IMService

```

Figure 10- 94 IMServiceImpl.h

It inherits from IMService, and IMService.h is shown in figure 10-95.

```

26 + (id)smsService;
27 + (id)iMessageService;
28 + (id)facetimeService;
29 + (id)callService;
30 + (id)jabberService;
31 + (id)subnetService;
32 + (id)aimService;

```

Figure 10- 95 IMService.h

[IMServiceImpl iMessageService], that's it. Reconfirm with Cycript:

```

cy# [IMServiceImpl iMessageService]
#"IMService[iMessage]"

```

By far, we've completely reversed the generation of an IMChat object. Let's try it out in Cycript:

```

FunMaker-5:~ root# cycript -p MobileSMS
cy# service = [IMServiceImpl iMessageService]
#"IMService[iMessage]"
cy# account = [[IMAccountController sharedInstance]
__ck_defaultAccountForService:service]
#"IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Service:
IMService[iMessage] Login: P:+86PhoneNumber Active: YES LoginStatus: Connected]"
cy# handle = [account imHandleWithID:@"snakeninny@icloud.com" alreadyCanonical:NO]
#"[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account: P:+86
MyPhoneNumber]"
cy# chat = [[IMChatRegistry sharedInstance] chatForIMHandle:handle]
#"<IMChat 0x15809000> [Identifier: snakeninny@icloud.com GUID:
iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account:
26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 3 Participants: 1 Room Name:
(null) Display Name: (null) Last Addressed: (null) Group ID: 6592DD84-4B34-4D54-BB40-
E2AB17B2FC67 Unread Count: 0 Failure Count: 0]"

```

Gorgeous! To finally make it, we need to construct an IMessage object for sending. Let's move it now.

Open IMessage.h, as shown in figure 10-96.

```

13 @interface IMessage : NSObject <NSCopying>
14 {
15     IMHandle *_sender;
16     IMHandle *_subject;
17     NSAttributedString *_text;
18     NSString *_plainBody;
19     NSDate *_time;
20     NSDate *_timeDelivered;
21     NSDate *_timeRead;
22     NSDate *_timePlayed;
23     NSString *_guid;
24     NSAttributedString *_messageSubject;
25     NSArray *_fileTransferGUIDs;
26     NSError *_error;
27     unsigned long long _flags;
28     BOOL _isInvitationMessage;
29     long long _messageID;
30 }
31
32 + (id)messageFromIMMessageItemDictionary:(id)arg1 sender:(id)arg2 subject:(id)arg3;
33 + (id)messageFromIMMessageItem:(id)arg1 sender:(id)arg2 subject:(id)arg3;
34 + (id)fromMeIMHandle:(id)arg1 withText:(id)arg2 fileTransferGUIDs:(id)arg3 flags:(unsigned long long)arg4;
35 + (id)instantMessageWithText:(id)arg1 messageSubject:(id)arg2 fileTransferGUIDs:(id)arg3 flags:(unsigned long long)arg4;
36 + (id)instantMessageWithText:(id)arg1 messageSubject:(id)arg2 flags:(unsigned long long)arg3;
37 + (id)instantMessageWithText:(id)arg1 flags:(unsigned long long)arg2;
38 + (id)defaultInvitationMessageFromSender:(id)arg1 flags:(unsigned long long)arg2;
39 + (id)locatingMessageWithGuid:(id)arg1 error:(id)arg2;
40 + (id)messageWithLocation:(id)arg1 flags:(unsigned long long)arg2 error:(id)arg3 guid:(id)arg4;
41 + (id)vCardDataWithCLLocation:(id)arg1;

```

Figure 10- 96 IMessage.h

There’re lots of class methods, among which “instantMessageWithText:flags:” gets our attention. Seems it returns an IMessage object, but what’re the 2 arguments? The 1st may be an NSString object, what about “flag”? I don’t know if you still remember that earlier in this section, when we were locating [IMChat sendMessage:IMMessage], we’ve “po”ed an IMessage object in LLDB:

```

(lldb) po $r2
IMessage[from=(null); msg-subject=(null); account:(null); flags=100005; subject='<<
Message Not Loggable >>' text='<< Message Not Loggable >>' messageID: 0 GUID:'966C2CD6-
3710-4D0F-BCEF-BCFEE8E60FE9' date:'437730968.559627' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO finished: YES sent: NO read: NO
delivered: NO audio: NO played: NO from-me: YES emote: NO dd-results: NO dd-scanned: YES
error: (null)]

```

Although the “text” is “not loggable”, the “flag” is 100005. Let’s try it out in Cycript:

```

cy# [IMessage instantMessageWithText:@"iOSRE test" flags:100005]
-[__NSCFString string]: unrecognized selector sent to instance 0x1468c140

```

Cycript reminds us that NSString failed to respond to @selector(string). In other words, the 1st argument is not an NSString object, but instead some class that can respond to @selector(string). Let’s try to get some clues in figure 10-96, do you see “NSAttributedString *_text” in line 17? According to the official documents, NSAttributedString does have a property named “string”, whose getter is “-(NSString *)string”, as shown in figure 10-97.

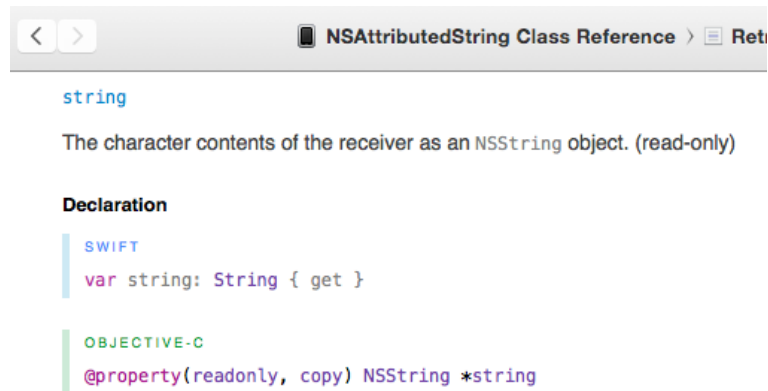


Figure 10- 97 [NSAttributedString string]

Let's test "instantMessageWithText:flags:" with an NSAttributedString object:

```
cy# attributedString = [[NSAttributedString alloc] initWithString:@"iOSRE test"]
#"iOSRE test{\n}"
cy# message = [IMessage instantMessageWithText:attributedString flags:100005]
#"IMessage[from=(null); msg-subject=(null); account:(null); flags=186a5; subject='<<
Message Not Loggable >>' text='<< Message Not Loggable >>' messageID: 0 GUID:'00A8C645-
D207-4F93-9739-07AAC94E7465' date:'437812476.099226' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO finished: YES sent: YES read: NO
delivered: NO audio: NO played: NO from-me: YES emote: NO dd-results: YES dd-scanned: NO
error: (null)]"
cy# [attributedString release]
```

An IMessage object appears, but as you can see, the value of flags is presented in hexadecimal instead of decimal. Only a tiny fix is needed to make it correct:

```
cy# message = [IMessage instantMessageWithText:attributedString flags:1048581]
#"IMessage[from=(null); msg-subject=(null); account:(null); flags=100005; subject='<<
Message Not Loggable >>' text='<< Message Not Loggable >>' messageID: 0 GUID:'61012DF3-
1C0F-4DED-9451-975E5771D493' date:'447412682.028256' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO finished: YES sent: NO read: NO
delivered: NO audio: NO played: NO from-me: YES emote: NO dd-results: NO dd-scanned: YES
error: (null)]"
```

Everything should be ready by now. Last but not least:

```
cy# [chat sendMessage:message]
```

The effect is shown in figure 10-98. Let's call it a day.

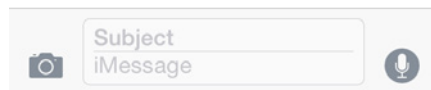


Figure 10- 98 iMessage delivered

10.4 Result Interpretation

Compared to previous practices, the reverse engineering methodology used in this chapter doesn't change much, but the overall workload has increased tremendously; As for difficulty, this chapter is way harder than chapter 7 and 8, though they're all targeting system Apps. To reverse the functions of detecting and sending iMessages, our general thoughts are as follows.

1. Cut into the code via UI

The changing from "Text Message" to "iMessage", green color to blue color, and "Send" button itself are all UI visualizations produced by programs. As long as we can describe what we see on UI, we can cut into the App from there. In this chapter, our cut-in points are message placeholder and "Send" button. Their UI functions can be easily located with Cycrypt, and are helpful in further analysis.

2. Browse and test class-dump headers to find interesting dots

Objective-C headers are clearly organized, methods are explicitly named. Their high readability is the perfect place for us to look for reverse engineering clues. Testing private methods, properties and instance variables with Cycrypt can be really helpful when we want to

know a certain private class better. In this chapter, when we came across some suspicious variables, we didn't strictly analyze them with IDA and LLDB, but by only browsing corresponding headers, guessing their prototypes and usages, then testing with Cycript to achieve our goals. The famous leader in my country Deng Xiaoping once said: "It doesn't matter whether a cat is white or black, as long as it catches mice", which applies to iOS reverse engineering too.

3. Analyze functions in IDA to connect the dots and form a plane

As to inspect the implementation of a function, IDA is one of the most handy tools. Cross references, addresses jumping, global search and whatever, they help us quickly locate what we're interested in, as well browse the context to form an overall understanding. In detecting iMessages, we've straightened out the relationships of [CKMessageEntryView updateEntryView], [CKPendingConversation sendingService], [CKPendingConversation composeSendingService], IMChatCalculateServiceForSendingNewCompose and so on; among them IMChatCalculateServiceForSendingNewCompose is a C function, hence is immune to class-dump. In sending iMessages, we've traced from the high level method [CKTranscriptController sendComposition:CKComposition], through [CKTranscriptController _startCreatingNewMessageForSending:], [CKConversation sendMessage:newComposition:] and [CKConversation sendMessage:onService:newComposition:], to the low level method [IMChat sendMessage:IMMessage]. All these operations are picking call chains from a plane according to keywords and clues provided by IDA. That's a lot of handwork, but thanks to the assistance of IDA, the workload is totally acceptable.

4. Pick out the exact line, i.e. call chain from the plane with LLDB

LLDB plays a significant role throughout the whole chapter. Although we've tried to limit its usage in section 10.3, we have to bring it out when tracing function callers and dynamically analyzing arguments. Compared with GDB, LLDB is more iOS supportive, there're rare crashes and bugs; it works great on Objective-C objects, making our debugging much smoother. When we were working on the detecting and sending of iMessages, LLDB helped us clarify great amounts of details; based on the careful analysis of tightly related data sources, we've abstracted a short piece of the working principles and designing ideas of iMessage: MobileSMS plays the

role of a post office; its building materials, office equipments and clerks are all from ChatKit, while IMCore is the postman. When I have a letter to send, I'll go to the post office and put the letter in the mailbox. Then a clerk will sort the letters out and hand them to the postman; later the postman will give feedback of the delivery progress and result to the clerk, who are in charge of informing me what's happening to my letter. This kind of closed-loop service is very Apple-ish; MobileSMS, ChatKit and IMCore play different roles, bringing Apple fans terrific user experiences. If we can learn how Apple design and implement all kinds of services via iOS reverse engineering, put them together and make them our own, it'll bring dramatically improvement to the elegance, design and robustness of our products, which is unattainable by only reading the official documents.

10.5 Tweak writing

After prototyping the tweak with Cycrypt, coding with Theos is just physical labor without much thinking. We'll add 2 methods to SMSApplication in MobileSMS, namely “-(int)madridStatusForAddress:(NSString *)address” and “-(void)sendMadridMessageToAddress:(NSString *)address withText:(NSString *)text” then test them with Cycrypt. Let's move it!

10.5.1 Create tweak project “iOSREMadridMessenger” using Theos

The Theos commands are as follows:

```
snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
-----
[1.] iphone/application
[2.] iphone/cydyget
[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): iOSREMadridMessenger
Package Name [com.yourcompany.iosremadridmessenger]: com.iosre.iosremadridmessenger
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.apple.MobileSMS
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: MobileSMS
Instantiating iphone/tweak in iosremadridmessenger/...
```

Done.

10.5.2 Compose iOSREMadridMessenger.h

We've made use of multiple private classes and methods in previous sections, so we need to provide their definitions to avoid any compiler warning or error. Of course, the contents of iOSREMadridMessenger.h don't come from nowhere, it's composed by picking snippets from other class-dump headers, forming a "select header". The finalized iOSREMadridMessenger.h looks like this:

```
@interface IDSIDQueryController
+ (instancetype)sharedInstance;
- (NSDictionary *)_currentIDStatusForDestinations:(NSArray *)arg1 service:(NSString *)arg2 listenerID:(NSString *)arg3;
@end

@interface IMServiceImpl : NSObject
+ (instancetype)iMessageService;
@end

@class IMHandle;

@interface IMAccount : NSObject
- (IMHandle *)imHandleWithID:(NSString *)arg1 alreadyCanonical:(BOOL)arg2;
@end

@interface IMAccountController : NSObject
+ (instancetype)sharedInstance;
- (IMAccount *)__ck_defaultAccountForService:(IMServiceImpl *)arg1;
@end

@interface IMessage : NSObject
+ (instancetype)instantMessageWithText:(NSAttributedString *)arg1 flags:(unsigned long long)arg2;
@end

@interface IMChat : NSObject
- (void)sendMessage:(IMessage *)arg1;
@end

@interface IMChatRegistry : NSObject
+ (instancetype)sharedInstance;
- (IMChat *)chatForIMHandle:(IMHandle *)arg1;
@end
```

10.5.3 Edit Tweak.xm

The finalized Tweak.xm looks like this:

```
#import "iOSREMadridMessenger.h"
%hook SMSApplication
%new
- (int)madridStatusForAddress:(NSString *)address
{
    NSString *formattedAddress = nil;
```

```

        if ([address rangeOfString:@"@"].location != NSNotFound) formattedAddress =
        [@"mailto:" stringByAppendingString:address];
        else formattedAddress = [@"tel:" stringByAppendingString:address];
        NSDictionary *status = [[IDSIDQueryController sharedInstance]
        _currentIDStatusForDestinations:@[formattedAddress] service:@"com.apple.madrid"
        listenerID:@"__kIMChatServiceForSendingIDSQueryControllerListenerID"];
        return [status[formattedAddress] intValue];
    }

%new
- (void)sendMadridMessageToAddress:(NSString *)address withText:(NSString *)text
{
    IMServiceImpl *service = [IMServiceImpl iMessageService];
    IMAccount *account = [[IMAccountController sharedInstance]
    __ck_defaultAccountForService:service];
    IMHandle *handle = [account imHandleWithID:address alreadyCanonical:NO];
    IMChat *chat = [[IMChatRegistry sharedInstance] chatForIMHandle:handle];
    NSAttributedString *attributedString = [[NSAttributedString alloc]
    initWithString:text];
    IMessage *message = [IMessage instantMessageWithText:attributedString
    flags:1048581];
    [chat sendMessage:message];
    [attributedString release];
}
%end

```

10.5.4 Edit Makefile and control files

The finalized 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 = iOSREMadridMessenger
iOSREMadridMessenger_FILES = Tweak.xm
iOSREMadridMessenger_PRIVATE_FRAMEWORKS = IDS ChatKit IMCore

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 MobileSMS"

```

The finalized control looks like this:

```

Package: com.iosre.iosremadridmessenger
Name: iOSREMadridMessenger
Depends: mobilessubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Detect and send iMessage example
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com

```

10.5.5 Test with Ccrypt

Compile and install the tweak, then ssh into iOS and execute the following commands:

```
FunMaker-5:~ root# cccrypt -p MobileSMS
cy# [UIApp madridStatusForAddress:@"snakeninny@icloud.com"]
1
cy# [UIApp sendMadridMessageToAddress:@"snakeninny@icloud.com" withText:@"Sent from
iOSREMadridMessenger"]
```

The madrid status of “snakeninny@icloud.com” is 1, indicating it supports iMessage; plus the iMessage is sent and delivered like silk, as shown in figure 10-99.

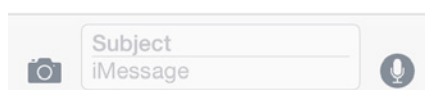
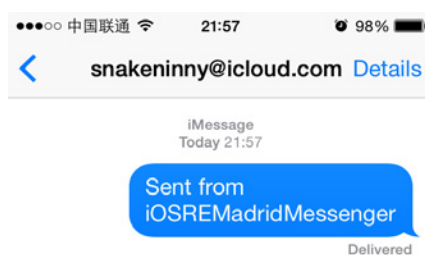


Figure 10- 99 iMessage sent

If you've been so far, feel free to iMessage me at “snakeninny@gmail.com” with iOSREMadridMessenger and share your joy :)

10.6 Conclusion

Being one of the key services since iOS 5, iMessage is greatly enhanced in iOS 8. Whether it's plain text, image, audio, or even video, iMessage can handle them all. Although detecting and sending iMessages is only a tip of iceberg in all iMessage operations, we've switched among IDS, ChatKit and IMCore, as well felt the high complexity of the entire iMessage service. According to our analysis, the class in charge of iMessage accounts is IMAccountController; the class of iMessage accounts is IMAccount; recipient class is IMHandle; a message conversation is

an IMChat or CKConversation object; IMChatRegistry is responsible for managing all IMChats; an iMessage is an IMessage or CKComposition object. For those IM developers, the design of iMessage can be a precious reference.

If you're still unsatisfied with this chapter and want to dig deeper into iMessage, try the following bonuses:

- Send SMS programmatically (Tip: just replace IMServiceImpl would be alright).
- Send iMessage with ChatKit (Tip: you can get a CKConversationfrom object from an IMChat object).
- Send iMessage inside SpringBoard (Tip: calling [IMChat sendMessage:IMMessage] inside SpringBoard fails to send messages, because SpringBoard lacks certain "capabilities").

If you can digest the contents of this chapter inside-out, and prototype the tweak without referring to the book, congratulations, you're a fairly outstanding iOS reverse engineer now, you are qualified and encouraged to step toward a higher goal, say, jailbreak. Before you begin the new journey, share your knowledge and experiences with us on <http://bbs.iosre.com> to help build the jailbreak community. Thank you!

Jailbreaking for Developers, An Overview

Much has been said about Apple's closed approach to selling devices and running an app platform. But what few know is that behind closed doors there's a massive ecosystem of libraries and hardware features waiting to be unlocked by developers. All of the APIs Apple uses internally to build their services, applications, and widgets are available once the locks are broken via a process called jailbreaking. Most of them are written in Objective-C, a dynamic language that provides very rich introspection capabilities and has a culture of self-describing code. Further tearing down barriers, most people install something called CydiaSubstrate shortly after jailbreaking, which allows running custom code inside every existing process on the device. This is very powerful—not only have we broken out of the walled garden into the rest of the forest, all of the footpaths are already labeled. Building code that targets jailbroken iOS devices involves unique ways of inspecting APIs, injecting code into processes, and writing code that modifies existing classes and finalized behaviors of the system.

The APIs implemented on iOS can be divided into four categories: framework-level Objective-C APIs, app-level Objective-C classes, C-accessible APIs and JavaScript-accessible APIs. Objective-C frameworks are the most easily accessible. Normally the structure of a component is only accessible to the programmer and those the source code or documentation have been made available to, but compiled Objective-C binaries include method tables describing all of the classes, protocols, methods and instance variables contained in the binary. An entire family of “class-dump” tools exists to take these method tables and convert them to header-like output for easy consumption by adventurous programmers. Calling these APIs is as simple as adding the generated headers to your project and linking with the framework or library. The second category of app internal classes may be inspected via the same tools, but are not linkable via standard tools. To get to those classes, one has to have code injected into the app in question and use the Objective-C runtime function `objc_getClass` to get a reference to the class; from there, one can call APIs via the headers generated by the tool. Inspecting C-level

functions are more difficult. No information about what the parameters or data structures are baked into the binaries, only the names of exported functions. The developer tools that ship with OS X come with a disassembler named “otool” which can dump the instructions used to implement the code in the device. Paired with knowledge of ARM assembly, the type information can be reconstructed by hand with much effort. This is much more cumbersome than with Objective-C code. Luckily, some of the components implemented in C are shared with OS X and have headers available in the OS X SDK, or are available as open-source from Apple. JavaScript-level APIs are most often facades over Objective-C level APIs to make additional functionality accessible to web pages hosted inside the iTunes, App Store, iCloud and iAd sections of the operating system.

Putting the APIs one has uncovered to use often requires having code run inside the process where their implementations are present. This can be done using the `DYLD_INSERT_LIBRARIES` environment variable on systems that use dyld, but this facility offers very few provisions for crash protection and can easily leave a device in a state where a restore is necessary. Instead, the gold standard on iOS devices is a system known as Cydia Substrate, a package that standardizes process injection and offers safety features to limit the damage testing new code can do. Once Cydia Substrate is installed, one needs only to drop a dynamic library compiled for the device in `/Library/MobileSubstrate/DynamicLibraries`, and substrate will load it automatically in every process on the device. Filtering to only a specific process can be achieved by dropping a property list of the same name alongside it with details on which process or grouping of processes to filter to. Once inside, one can register for events, call system APIs and perform any of the same behaviors that the process normally could. This applies to apps that come preinstalled on the device, apps available from the App Store, the window manager known as SpringBoard, UI services that apps can make use of such as the mail composer, and background services such as the media decoder daemon. It is important to note that any state that the injected code has will be unique to the process it's injected into and to share state mandates use inter-process communication techniques such as sockets, fifos, mach ports and shared memory.

Modifying existing code is where it really starts to get powerful and allows tweaking existing functionality of the device in simple or even radical ways. Because Objective-C method lookup is all done at runtime and the runtime offers APIs to modify methods and classes, it is really

straightforward to replace the implementations of existing methods with new ones that add new functionality, suppress the original behavior or both. This is known as method hooking and in Objective-C is done through a complicated dance of calls to the `class_addMethod`, `class_getInstanceMethod`, `method_getImplementation` and `method_setImplementation` runtime functions. This is very unwieldy; tools to automate this have been built. The simplest is Cydia Substrate's own `MSHookMessage` function. It takes a class, the name of the method you want to replace, the new implementation, and gives back the original implementation of the function so that the replacement can perform the original behavior if necessary. This has been further automated in the Logos Objective-C preprocessor tool, which introduces syntax specifically for method hooking and is what most tweaks are now written in. Writing Logos code is as simple as writing what would normally be an Objective-C method implementation, and sticking it inside of a `%hook ClassName ... %end` block instead of an `@implementation` `ClassName ... %end` block, and calling `%orig()` instead of `[super ...]`. Simple tweaks to how the system behaves can often be done by replacing a single method with a different implementation, but complicated adjustments often require assembling numerous method hooks. Since most of iOS is implemented in Objective-C, the vast majority of tweaks need only these building blocks to apply the modifications they require. For the lower levels of the system that are written in C, a more complicated hooking approach is required. The lowest level and most compatible way of doing so is to simply rewrite the assembly instructions of the victim function. This is very dangerous and does not compose well when many developers are modifying the same parts of the system. Again, CydiaSubstrate introduces an API to automate this in form of `MSHookFunction`. Just like `MSHookMessage`, one needs only to pass in the target function, new replacement implementation function, and it applies the hook and returns the old implementation that the new replacement can call if necessary. With the tools the community has made available, the details of the very complex mechanics of hooking have been abstracted and simplified to the point where they're hidden from view and a developer can concentrate on what new features they're adding.

Combining these techniques unique to the jailbreak scene, with those present in the standard iOS and OS X development communities yields a very flexible and powerful tool chest for building features and experiences that the world hasn't seen yet.

Ryan Petrich

Evading the Sandbox

As a security measure and to keep apps on the device from sharing data or interfering with each other, iOS includes a security system known as the sandbox. The sandbox blocks access to files, network sockets, bootstrap service names, and the ability to spawn subprocesses. Part of the jailbreaking process involves modifying the sandbox so that all processes can load Cydia Substrate, but much of the sandbox is left intact to respect the security and privacy of the user's data.

With each new release, Apple further improves the sandbox to improve privacy and security. When building extensions or tweaks that need to share information across processes or persist data to disk, this can be restrictive. One approach is to survey the sandbox restrictions that exist on the processes where the extension is to be run, and choose file paths and names based on them. This is common, but can leave oneself stranded when Apple tightens the tourniquet and as of iOS 8 there is no location that all processes can read and write successfully. A better approach is to do all of the interesting work inside a privileged process such as SpringBoard, backboardd or even a manually created launch daemon of your own. Child processes can then send work to the privileged service. This ensures that as the sandbox tightens, your extension will still behave properly as long as it can communicate with the service.

Oddly enough, as of iOS 8 Apple has also decided to limit which services an app store process may query. This makes nearly all forms of inter-process communication ineffective on iOS, outside of the well-defined static services that Apple has designated. RocketBootstrap was created as a way around this that simultaneously allows additional services to be registered and respects the security and privacy of the user's data. Services registered with RocketBootstrap are made globally accessible even in spite of very restrictive sandbox rules and it will serve as a

single project that needs updating as the rules change.

Ryan Petrich
Code Wrangler, Father of Activator

Tweaking is the new-age hacking

I am not a prolific programmer by any means. I have a programmer's mind, and I have proven in my days I am capable of writing working solutions. I have a few tweaks in my name, and more ideas to be realized. Creating more has been about having more free time. However, my time has been spent becoming familiar with iOS-internals, because I find that I am a good learner. I have a fair understanding due to the tools we have available, made by great programmers before our time, and from documentation and examples shared by the community. Because of the nature of Cocoa and Objective-C, we can take a great adventure and introspection into the workings of third-party software, and Apple's operating system. This provides a foundation and skills for making tweaks. We want to encourage tweak making because it has been the driving initiative behind the audience that wants to have jailbroken devices, besides for the groups that wish to only have a jailbreak for pirating apps and games. The growth of this jailbreak ecosystem has gone with the proliferation of new tweaks, ever pushing the boundaries of modification while maintaining a safe environment for the end-users.

The jailbreak development scene has given a unique opportunity to developers to express themselves in a new way. In the days before CydiaSubstrate, apps and games were not tweaked. This is a new concept; examining and debugging existing software and then rewriting portions of it with the least invasive tools available, the changes are nonpermanent and for the most part free of worry for breaking something with any lasting effect. Tweaks allow for a redefining of how software works and behaves. We do this with tweaks, and there has really been nothing like it before in the world of programming, even on the PC. There were opportunities throughout previous decades to make game patches, hacks and so forth, but it's only with the emergence of the audience of jailbreakers and iOS that we find our unique situation. Only recently has it become feasible to make small adjustments to existing UI and modify how things work without requiring the replacement of whole parts of the code - CydiaSubstrate allows

careful targeting of methods and functions.

It's a lot of fun to discover how things work, and tweak making is the embodiment of that fun time for developers. One of the challenges for tweak making is coming up with new ideas to create, and sometimes these ideas only arise after studying the internals in some detail. If you make tweaks as a hobby, and not as a profession, you're free to do as you wish and to focus on projects that interest you. For new tweak makers, there're quite a lot of existing projects to learn from, but a lot of the easier projects have already been realized. Creating new original ideas that are unique is a task of being familiar with the available tweaks on Cydia, and then going to work discovering how the internal parts work, debugging and testing until you have a diagram or picture in your mind how it's put together. When you reach a near complete understanding, you are primed to tackle whatever challenge you make for yourself.

Some of our greatest tools and resources are free: Apple's own documentation is excellent, and for tweak makers we have a wiki and the opportunity to use class-dump to examine what methods are exposed for hooking inside the target app or process. Debugging and disassembly tools that vary from free to paid, all can be great assets for tweak makers. A well-studied programmer with some prior experience with standard projects will be in a good position to continue learning from these materials. To the contrary, a newcomer programmer, even a person with some good ideas will struggle at first with the learning curve. We recommend a core understanding of Objective-C and Cocoa principles for aspiring young tweak makers; this can be a significant investment of time, but it is really a hurdle for new tweak makers that haven't a clue where to start. To the uninitiated, the object-oriented nature of the programming involved can be a daunting thing to realize. Generating tweak ideas can be a task for amateurs, but the writing of the code for the tweak implementation is often the result of planning and research and testing for a significant time. We find that many young new programmers are impatient because their ideas for new tweaks do take more time to materialize than they were willing to invest. Patience is a virtue of course, and the best-made tweaks are all products of careful programmers.

The greatest tweak is Activator (libactivator). Based on a commonsense idea of having more triggers system-wide, activator is also a graciously open-sourced project; the product of many months and years of work by our most senior tweak maker, Ryan Petrich. His dedication and expertise shines through in Activator, which doubles as a platform for third-parties to harness

the powerful triggers from anyplace to use in their own projects. It represents a lot of research and understanding of the most obscure internals on iOS: SpringBoard and backboard. If there is one shining example to point to as a goal for a tweak maker to show how much research and careful planning can go into a tweak, that is the example to look towards. It's a lofty project that none should consider as being trivial to do, however. For some aspiring developers it can be a great encouragement to see what is possible. Kudos to Ryan Petrich for making it, and for all he does to further the jailbreak development community.

As the repo maintainer for TheBigBoss, I have a job description for myself. Doing my job has given tremendous opportunity to be an influence or guidance for new tweak makers. Often their first experience with another member of the jailbreak development community is with myself when they first contact me or submit to the repo. We wish that all developers can be involved in the social channels of this scene: chat, forums, twitter et al., however, it's not uncommon that some developers work in relative isolation from these social groups. My involvement then can be seen as important: I may be the only other voice that the programmer will hear, and I will give an opinion on the technical merits of new tweak projects; often this first encounter is invaluable because those developers that work in isolation are not wise to many of the caveats and conventions we hold as important in this community. Our documentation and wikis have improved to make these details more available, but still I am often the first time a developer has some interaction with someone with a greater expertise than their own. I try to give my wisdom and guidance to the developers because its in our best interest to support, if not groom, newcomer developers so they feel as part of the group of jailbreak developers, and they can be pointed towards ways to avoid some of the pitfalls that many newcomers make. I take some pride in doing this and helping in part to strengthen the developer community that is based around the tweak-making culture. I want the jailbreak platform to continue to grow and mature by the great ideas that are envisioned and the expertise to realize them.

Do not be discouraged when the task seems difficult. We have some developers with years and decades of programming experience, and we also have some with only a few weeks or months. I come from the school of thought that it should be well made and well tested, and not rushed or forced. If you have a goal, it should not be merely to have something of yours published on a Cydia repo, but to give something to the public, which will enrich their jailbreak experiences - that is for hobbyists like myself. If you have some commercial interest in Cydia,

and for making an selling tweaks, do wide testing with users and alongside other tweaks to help assure a product that works for many users and their combinations of tweaks; your duty as a responsible tweak maker is to be careful while you modify the insides of others' programs or apps, and to be thorough in testing compatibility with others' tweaks.

Tweak making is the new-age hacking. There're already enough reasons for you to get started with tweak development, and we need tweaks to keep the jailbreak community in bloom. Join us, learn from others, work hard, be patient, and have fun.

Optimo

Administrator at TheBigBoss repo