# Chapter 10: Practice 4: Detect And Send iMessages

## 10.1 iMessage

iMessage is an IM service that Apple implemented seamlessly into stock Messages App (hereafter referrer to as MobileSMS). It was born in iOS 5 and grows in iOS 8. Whether it’s plain text, or image, audio, 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 iMessages are the 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 I wrote 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 so 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 in iPhone 5, iOS 8.1.

## 10.2 Detect if a number or email address supports iMessage

按照惯例，在使用工具开始逆向工程之前，要先分析一下抽象的目标，并将其具体化，然后制定这次逆向工程的思路，再贯彻思想实地执行。

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 Observer MobileSMS and look for cut-in points

### 观察MobileSMS界面元素的变化，寻找逆向切入点

使用MobileSMS的朋友都会注意到，在发送一条信息的整个过程中，苹果都会通过文字以及颜色的变化，来提示用户当前发送的是一条短信（以下简称SMS）还是一条iMessage，具体表现在：

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:

1. 当你开始编写一条信息，刚输入完对方的地址，还没有输入信息内容时，如果iOS检测到对方支持iMessage，信息输入框处的占位符（placeholder）就会由“Text Message”变成“iMessage”，如图10-1所示。
2. 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

1. 当你输入信息内容时，如果对方仅支持SMS，则输入框旁的“Send”字样是绿色的；如果对方支持iMessage，则“Send”是蓝色的；
2. When you start to input message texts, 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.
3. 当你点击“Send”发送此条信息时，如果这是一条SMS，信息气泡的颜色是绿色的；如果是一条iMessage，气泡是蓝色的。
4. 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.

这3种现象会依次出现，不过，因为检测iMessage的操作在第一个现象里已经出现了，所以仅把这一个现象作为切入点来分析就已经能够达到本节的目标了。下面会把火力集中在现象A上。

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 phenomenon A from now on.

确定了切入点之后，跟笔者一起思考，怎么把这个现象给具象化成逆向工程的思路：

After locating the cut-in point, let’s think together to concretize the phenomenon into a reverse engineering idea.

我们能够观察到的是发生在UI上的现象，即“Text Message”变成“iMessage”。我们知道，UI上显示的内容不是凭空生成的，它显示的是其数据源的值——那么就可以根据这个现象，用Cycript找到UI的数据源，即placeholder。

What we can observe is visible on UI, i.e. “Text Message” changes into “iMessage”. As we’ve already known, visualizations on UI don’t come from nowhere but the data source, hence by referring to the visualizations, we can find the data source, i.e. placeholder, using Cycript.

placeholder也不是凭空生成的，它的值也来自它的数据源——它之所以发生改变，是因为它的数据源（的数据源的数据源……以下简称N重数据源）发生了改变，类似于下面的伪代码：

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 data source = ?;

id a = function(data source);

id b = function(a);

id c = function(b);

…

id z = function(y);

NSString \*placeholder = function(z);

从上面的伪代码可以知晓，原始数据源是data source，data source发生了变化，间接导致placeholder变化。可以理解吧？那原始数据源是什么呢？在现象A里，我们唯一输入的就是收件人地址，因此原始数据源当然就是收件人地址啊！对于“检测iMessage”来说，MobileSMS中应该存在一个data source转换为placeholder的过程，这个过程就是“检测iMessage”的确切含义，也就是本节的目标，如图10-2所示。

From the above snippet we can know that the original data source is data source, its change in turn results in the change of placeholder. Well, what’s the original data source? In phenomenon A, 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 data source 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.

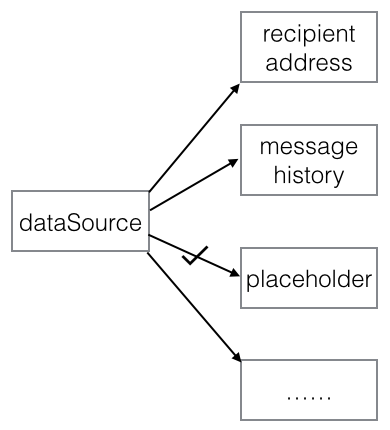


Figure 10- 2 Conversion from data source to placeholder

你可能会想，图10-2这么直观，既然data source已知，那就直接从它入手，找到placeholder，不就达到目的了？但是现实往往没有这么美好——我们没有源代码，进程流程一般也没有这么简单，在大多数时候，data source转换为placeholder的过程如图10-3所示。

You may wonder, since figure 10-2 is so straightforward and data source is already known, why don’t we start from it directly and track 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.

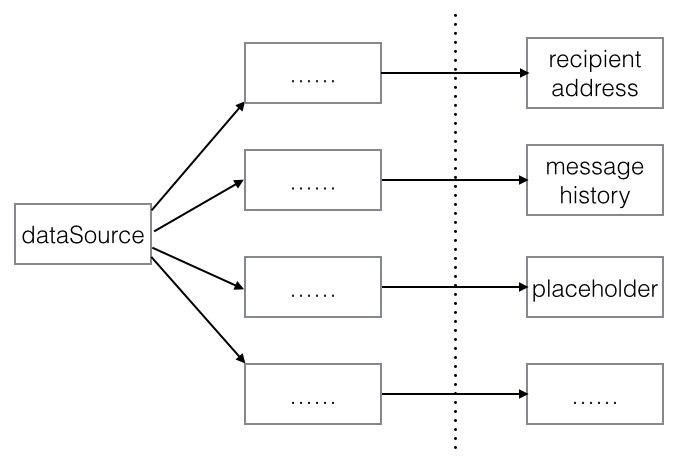


Figure 10- 3 Real conversion from data source to placeholder

data source要经过多重转换才能生成placeholder，它们之间的关系非常复杂。如果从data source入手，我们怎么知道它要走4条路中的哪条路才能到达placeholder？在这种情况下，因为placeholder是唯一的，所以从placeholder入手，用终点倒推起点，来还原过程，是更高效更可行的做法。

data source must be converted multiple times to become placeholder, their relationship is very intricate. If we start from data source, 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 track back to data source to reproduce the whole process.

总结一下，逆向工程的思路是这样的：先用Cycript定位placeholder，然后通过IDA和LLDB寻找placeholder的N重数据源，直到找到data source，最后还原data source生成placeholder的过程。看起来很简单是吧？实际操作起来你就知道有多复杂了。这就开始吧。

In conclusion, the ideas of this practice are: first use Cycript to locate placeholder, then track the Nth data source of placeholder using IDA and LLDB, until we get data source. Finally reproduce the process of how data source 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 Cycript

### 用Cycript找出placeholder

打开MobileSMS，新建一条信息，在地址输入框中填写“bbs.iosre.com”，然后点击“return”结束输入，如图10-4所示。

Open MobileSMS, 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.

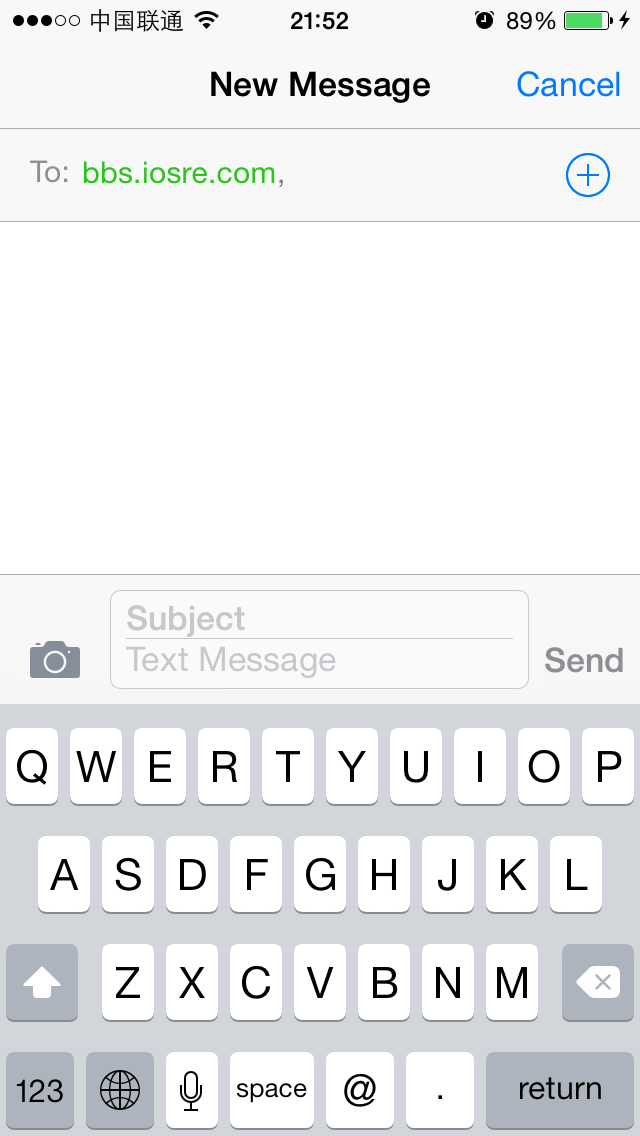


Figure 10- 4 Create a new message

既然要用Cycript找placeholder，那就用Cycript来找找显示placeholder当前值“Text Message”的是哪个view，placeholder一定跟那个view密切相关，对吧？走起！

Since we are using Cycript 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# cycript -p MobileSMS

cy# ?expand

expand == true

cy# [[UIApp keyWindow] recursiveDescription]

上面代码执行之后，Cycript会打印出keyWindow的视图结构，内容很多，就不贴在这里了。在输出里搜索“Text Message”，发现一个匹配也搜不到。这是怎么回事？相信你也想到了——“Text Message”不在keyWindow里。为了验证我们的猜想，来看看当前这个界面有几个window，如下：

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 aren’t any matches. 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>>"]

可以看到，每一个以“#”开头的都是一个window，这里一共有4个window，其中第一个是keyWindow。那么哪一个是“Text Message”所在的window呢？从关键字上也能猜测出，带“Text”字样的第2和第4个window可能是我们的目标，而第4个window的hidden属性是YES，它压根儿没有显示在界面上，因此肯定不是它。可见，第2个window很可能就是我们的目标，用Cycript测测看：

As we can see, each item starting with “#” is a window, there are 4 of them, and the first one is keyWindows. Well, which one contains “Text Message”? As the names suggest, the 2nd and 4th windows with the keyword “Text” may be our targets. However, the 4th window is even invisible because its hidden property. This leaves us with the 2nd window, let’s test it out in Cycript.

cy# [#0x1579ab70 setHidden:YES]

回车之后发现，不只是信息输入框，整个键盘都被隐藏起来了，如图10-5所示。

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

从而可以得出结论，“Text Message”就位于这个window中，继续通过Cycript来定位它。

Now we can confirm that “Text Message” is located right in this window. Keep looking for it using cycript.

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>>"]

上面代码中又有3个subview，哪个是“Text Message”的所在？用排除法来过一遍：

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]

上面语句执行之后变成了图10-6所示的界面，说明这个view只是键盘背景而已。

After the above command, the view looks like figure 10-6, indicating that this view is just keyboard background.

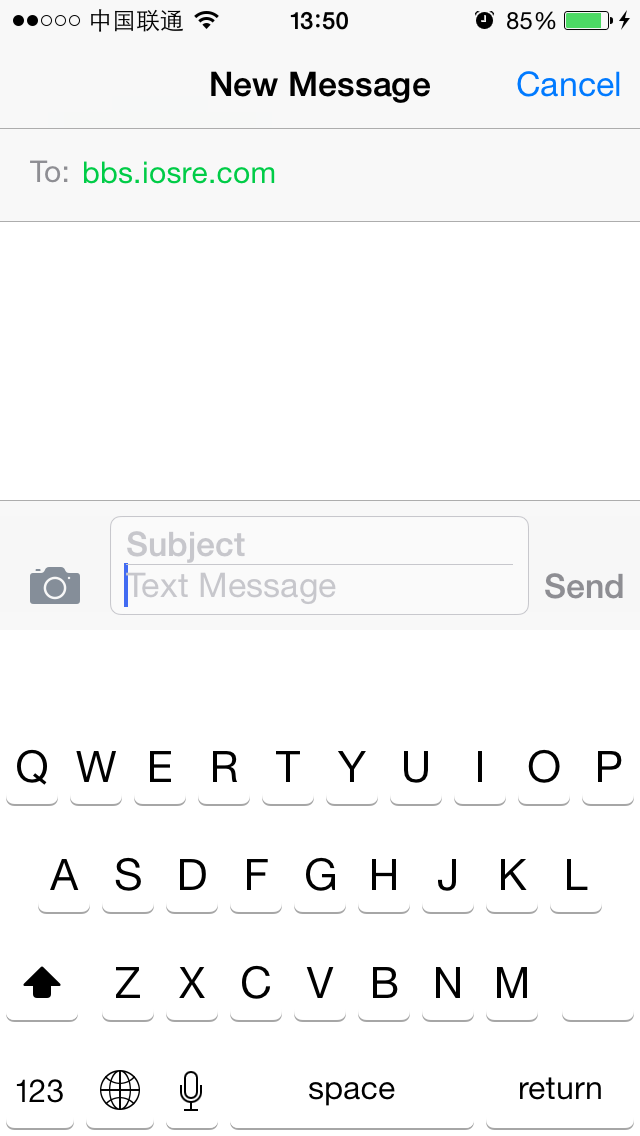


Figure 10- 6 Keyboard background is hidden

cy# [#0x16827620 setHidden:NO]

cy# [#0x157b88d0 setHidden:YES]

这两个语句执行之后界面变成了图10-7所示的状态。

After these 2 commands, the view changes to figure 10-7.

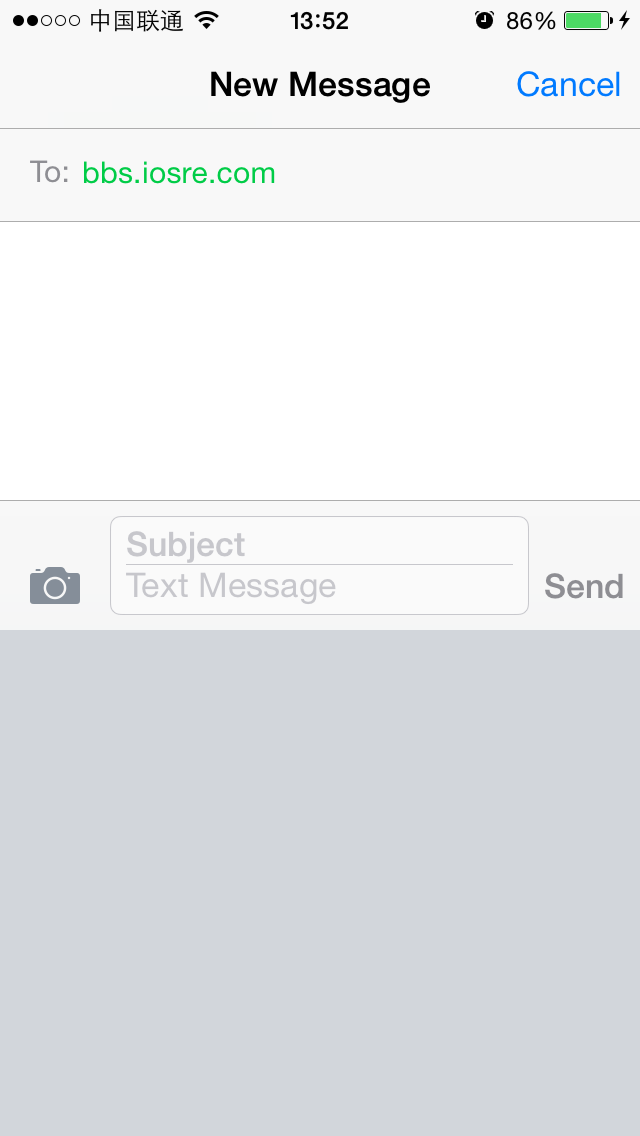


Figure 10- 7 Keyboard is hidden

说明这个view就是键盘本身。也就是说，UIKBInputBackdropView和UIKBCompatInputView共同构成了一个键盘的view，这种官方设计模式可以供第三方键盘开发者或者键盘主题制作者参考。

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 theme makers.

现在只剩最后一个subview了，CKMessageEntryView这个名字的意义其实也很明显，还是验证一下吧：

Now that there is the last subview with an explicit name “CKMessageEntryView”, waiting for our test:

cy# [#0x157b88d0 setHidden:NO]

cy# [#0x1682ca50 setHidden:YES]

执行后界面如图10-8所示。

The view looks like figure 10-8 after the above commands.

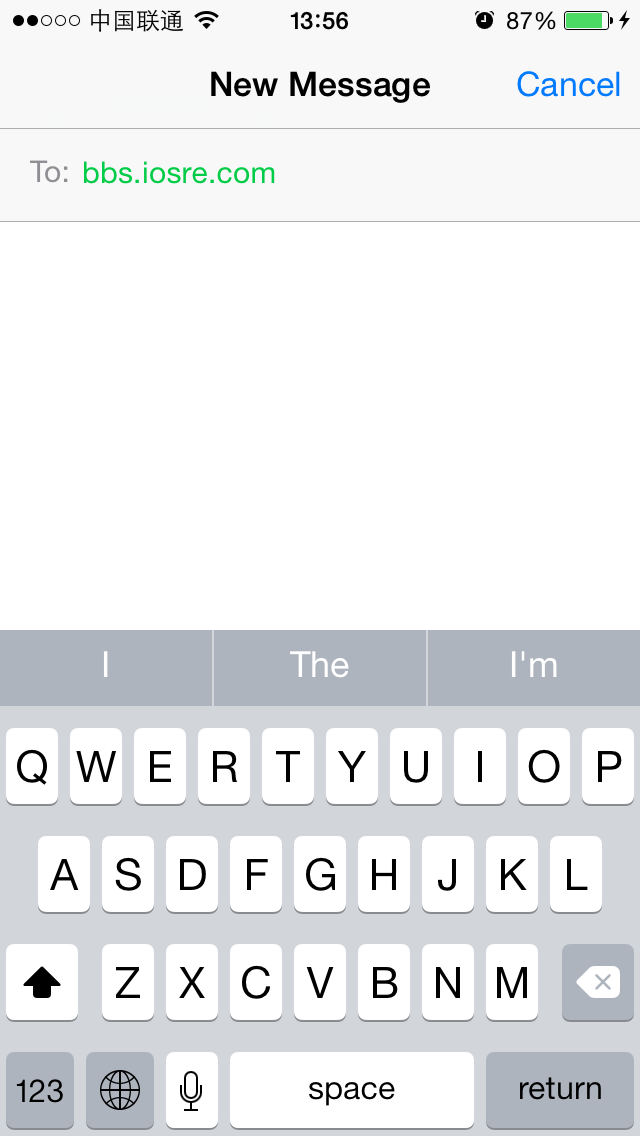


Figure 10- 8 Message entry view is hidden

通过验证，说明“Text Message”在CKMessageEntryView中。继续缩小范围：

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>>"]

还是使用排除法寻找“Text Message”所在的view，这里的过程就不再重复了，留给读者自己练习。在定位到了“UIView: 0x168dcf10”（注意，是第二个UIView对象）之后，继续查看它的subview。

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}>"]

上面代码中只有一个subview，继续查看这个subview的subview：

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); text = ''; clipsToBounds = YES; opaque = NO; gestureRecognizers = <NSArray: 0x1578e600>; layer = <CALayer: 0x157dcff0>; contentOffset: {0, 0}; contentSize: {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>>"]

还是用排除法寻找“Text Message”所在的view，我们发现，当执行“[#0x16295200 setHidden:YES]”时，只有“Text Message”被隐藏了，界面上的其他控件并未受影响，如图10-9所示。

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.

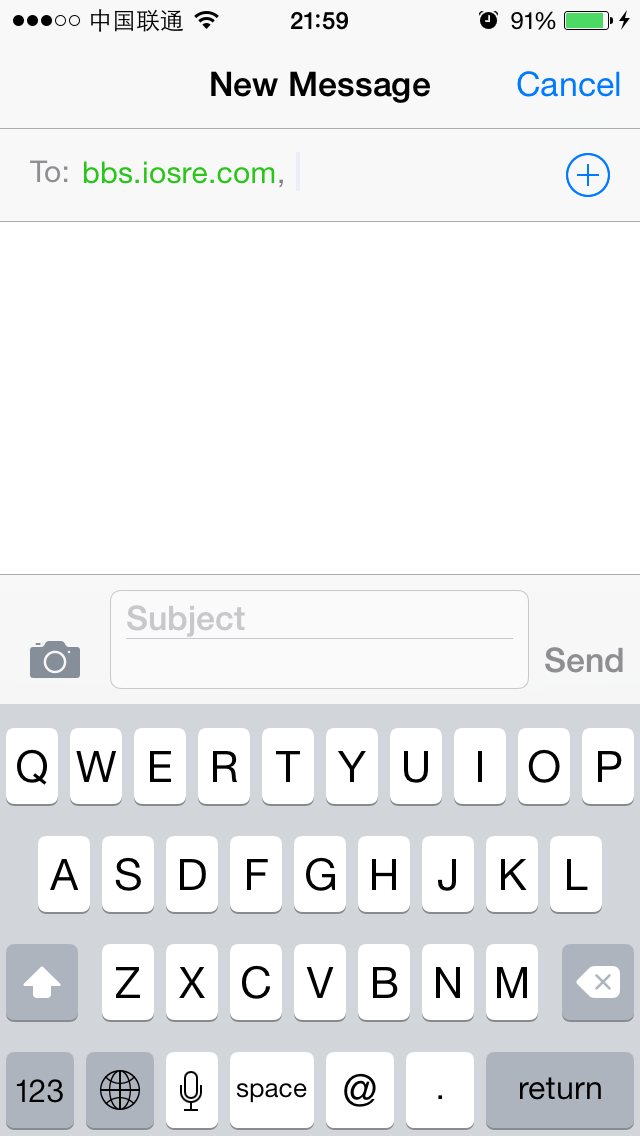


Figure 10- 9 placeholder is hidden

这说明CKMessageEntryRichTextView就是我们想要定位的view。打开CKMessageEntryRichTextView.h，看看能不能找到placeholder的踪影，如图10-10所示。

It means that CKMessageEntryRichTextView is our target view. Open CKMessageEntryRichTextView.h and see if there’s any “placeholder”, as shown in figure 10-10.

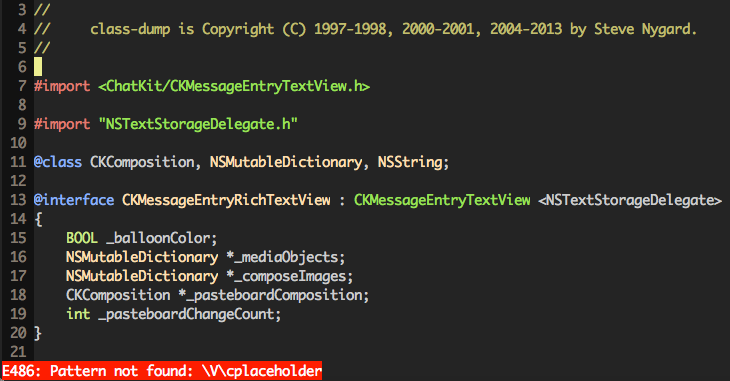


Figure 10- 10 CKMessageEntryRichTextView.h

可是，在CKMessageEntryRichTextView中搜不到placeholder。难道是我们的推理出了差错？不要着急，转战它的父类CKMessageEntryTextView里去看看，如图10-11所示。

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.

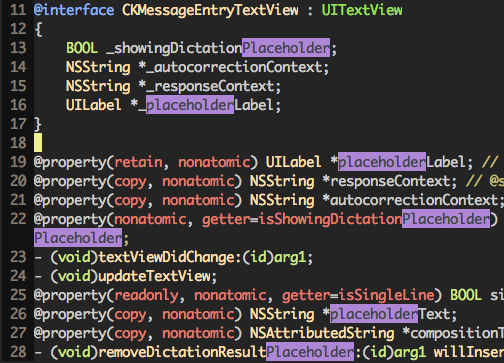


Figure 10- 11 CKMessageEntryTextView.h

可以看到，满屏的placeholder字眼。其中，placeholderLabel和placeholderText这2个property很可疑，难道它们就是我们在找的placeholder？用 Cycript验证一下，执行如下命令：

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:@"iOSRE"]

此时，界面变成了这个样子（如图10-12所示）。

Now, the view looks like figure 10-12.

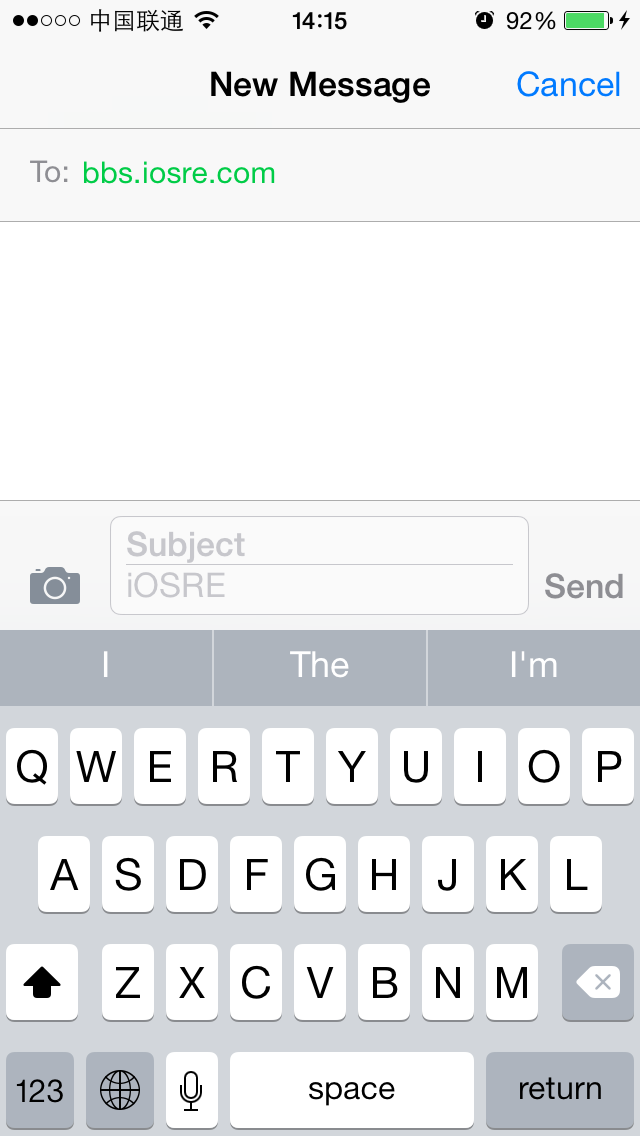


Figure 10- 12 Change placeholder to “iOSRE”

不错，placeholderText就是我们要找的placeholder，自此它们两者等同，为了避免混淆，下文统一使用placeholderText。旗开得胜，我们跨出了万里长征的第一步！

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 用IDA和LLDB找出placeholderText的一重数据源

### Find the 1st data source of placeholderText using IDA and LLDB

placeholderText是一个property，而要改变一个property，笔者的第一反应就是它的setter。在通过调用setPlaceholderText:函数，把placeholderText从“Text Message”改为“iOSRE”的同时，不妨大胆假设一下，MobileSMS会不会也是通过调用这个setter来改变placeholderText的呢？实际验证一下是最好不过的，接下来轮到IDA和LLDB上场了。

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.

因为最后定位到的CKMessageEntryTextView类来自ChatKit，所以接下来的目标是分析MobileSMS这个可执行文件中的ChatKit库，可以理解吧？好的，把ChatKit的二进制文件丢到IDA中，初始分析结束后，定位到[CKMessageEntryTextView setPlaceholderText:]，如图10-13所示。

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.

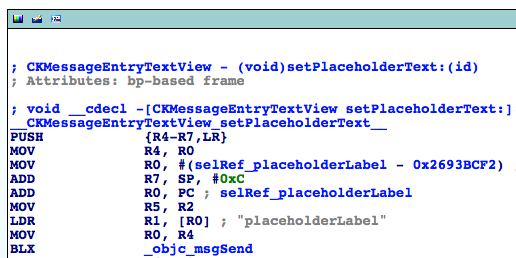


Figure 10- 13 [CKMessageEntryTextView setPlaceholderText:]

然后用LLDB附加MobileSMS，待初始化完成后执行“c”命令，让MobileSMS运行起来，如下：

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

之后，再看看ChatKit的ASLR偏移，如下：

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

这个偏移值是0x1eac000。有了这个值，就可以在[CKMessageEntryTextView setPlaceholderText:]中下个断点，看看它有没有被调用，如果被调用，调用者又是谁。为了把断点下在这个函数的开头位置，要先在IDA中看看这个函数的基地址，如图10-14所示，可以看到，是0x2693BCE0。

The ASLR offset is 0x1eac000. With this offset, we can set a breakpoint on [CKMessageEntryTextView setPlaceholderText:] to check whether it is called or not; 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.

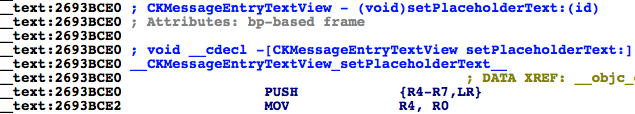


Figure 10- 14 [CKMessageEntryTextView setPlaceholderText:]

所以断点地址是0x1eac000 + 0x2693BCE0 = 0x287E7CE0。

So the breakpoint should be set at 0x1eac000 + 0x2693BCE0 = 0x287E7CE0.

(lldb) br s -a 0x287E7CE0

Breakpoint 1: where = ChatKit`-[CKMessageEntryTextView setPlaceholderText:], address = 0x287e7ce0

接着，把地址输入框中的“bbs.iosre.com”换成一个支持iMessage的地址“snakeninny@gmail.com”，看看进程会不会停在断点上。这时，你会发现，在实际操作中，随着地址的变动，进程会多次停在断点上，这说明[CKMessageEntryTextView setPlaceholderText:]被调用了很多次。那么我们怎么知道是在哪一次调用中，placeholderText从“Text Message”变成“iMessage”呢？此刻，之前介绍的comm命令就派上用场了：

Next, let’s change “bbs.iosre.com” to “[snakeninny@gmail.com](mailto: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

这个命令的意思很简单，就是在断点触发时打印R2的“Objective-C description”，即setPlaceholdeText:的参数；然后以十六进制格式打印LR的值，即[CKMessageEntryTextView setPlaceholderText:]的返回地址。如果R2的值是“iMessage”，说明系统自身也是通过setPlaceholderText:来改变placeholderText的，这个函数的参数就是placeholderText的一重数据源；同时因为对应LR的值位于调用者的地址范围内，所以可以找到setPlaceholderText:的调用者，继续跟踪参数的数据源，即placeholderText的二重数据源。清空地址输入框，再重新输入“snakeninny@gmail.com”，看看LLDB什么时候会打印“iMessage”。

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 track the 2nd data source from inside the caller. Clear the address entry and enter “[snakeninny@gmail.com](mailto: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.

可以看到，当placeholder变成“iMessage”时，LR的值是0x28768b33。根据第4章“偏移后指令基地址 = 偏移前指令基地址 + 指令所在模块的ASLR偏移”公式，0x28768b33 - 0x1eac000 = 0x268BCB33。可见，这个地址位于ChatKit内，如图10-15所示。

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.

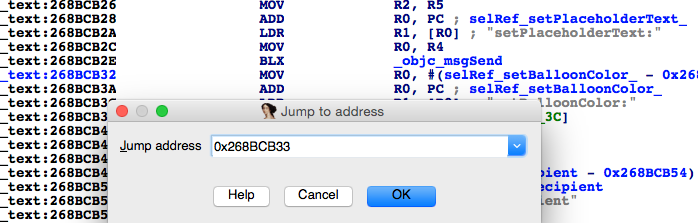


Figure 10- 15 Jump to 0x268BCB33

说明MobileSMS确实就是通过setPlaceholder:函数来改变placeholder的，函数的参数就是placeholder的一重数据源，同时二重数据源的线索也有了着落。万里长征第二步走完，有惊无险！

This address is located in ChatKit. OK, we've found the 1st data source of placeholder, which is the argument of setPlaceholder:, and get on the way to find the 2nd data source. What an uneventful achievement, meh.

### 10.2.4 用IDA和LLDB找出placeholderText的N重数据源

### Find the Nth data source of placeholderText using IDA and LLDB

前面在操作中多次编辑地址时，不知大家有没有注意到一个现象，那就是当我们正在编辑时，placeholderText是空白的；只有在我们点击了键盘上的“return”结束编辑后，placeholderText才会显示“Text Message”或者“iMessage”。也就是说，当地址编辑结束时，iOS才会检测当前地址是否支持iMessage，出于节省性能的考虑，这样的设计合情合理。也正是基于这样的设计，在接下来的调试中，可以先把目标地址编辑好，然后设置断点，最后点击“return”，这样断点如果被触发，则说明进程停在了iMessage检测的过程中。下面把IDA往上拉，看看[CKMessageEntryTextView setPlaceholderText:]的调用者是谁，如图10-16所示。

I don’t know if you've noticed that placeholderText was blank during address editing. Not until we press "return" button on the keyword will the placeholderText show “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 the process 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.

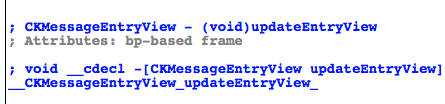


Figure 10- 16 Caller of [CKMessageEntryTextView setPlaceholderText:]

虽然在“更新输入视图”的时候“设置占位符”，说得过去，但是，[CKMessageEntryView updateEntryView]没有参数，它怎么知道placeholderText应该设置成“Text Message”还是“iMessage”呢？那只有一种可能，就是它的内部进行了判断，得出了该地址支持iMessage的结论，改变了placeholderText的二重数据源。回到IDA，看看二重数据源来自哪里，如图10-17所示。

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.

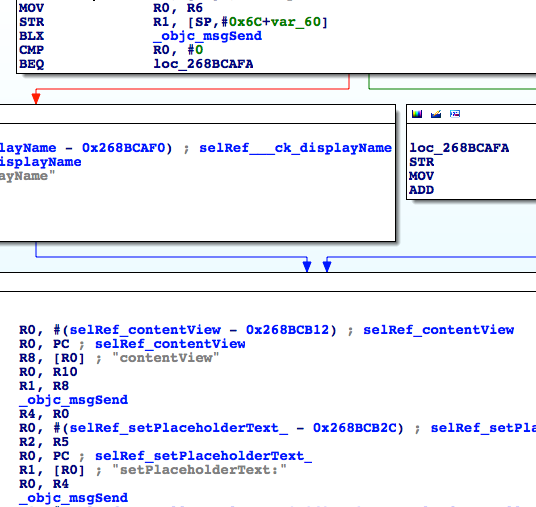


Figure 10- 17 Look for the 2nd data source

R2即函数参数，也就是一重数据源；而R2来自于R5，因此R5就是二重数据源。R5又是来自于哪里呢？这里出现了分支，我们看看分支的跳转条件，如图10-18所示。

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.

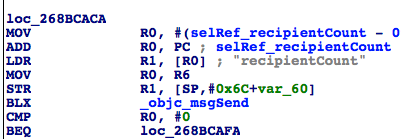


Figure 10- 18 Condition of branch

可以看到，跳转条件是“[$r0 recipientCount] == 0”。“recipient”的字面意思很明显，就是指信息的收件人，当收件人数为0，即没有收件人时，进程走右边，否则走左边。因为这里已经有一个收件人了，收件人数不为0，所以进程很有可能走左边。要验证也很简单，先清空地址输入框，然后在地址输入框中输入“[snakeninny@gmail.com](mailto:snakeninny@gmail.com)”，接着在右边的分支尾部下一个断点，最后点击键盘上的“return”。此时，会发现断点并没有得到触发，因此可以肯定地说，R5来自于左边的[$r8 \_\_ck\_displayName]，即[$r8 \_\_ck\_displayName]是三重数据源。但R8又来自哪里呢？往上拉IDA，在[CKMessageEntryView updateEntryView]的开始部分，可以看到R8来自[[self conversation] sendingService]，如图10-19所示。

We can see that the branch condition is "[$r0 recipientCount] == 0". The meaning of "recipient" is very obvious that it represents the receiver of message. When the recipient count is 0, namely no recipient, the process will branch right, otherwise left. In this example, because there is already one recipient, the process will probably branch left. It’s very simple to verify our assumption: input "snakeninny@gmail.com" in the address entry, then set a breakpoint on any instruction in the right branch and at last press "return" to finish editing. We can see that the breakpoint is not triggered; as a result, we can confirm that R5 comes from [$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.

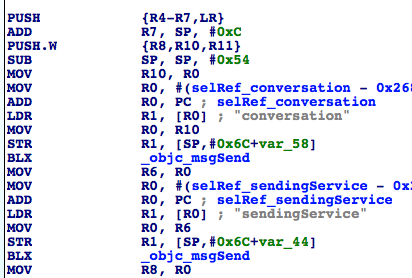


Figure 10- 19 Look for 4th data source

因此，[[self conversation] sendingService]是四重数据源。再用LLDB来验证一下我们的判断：清空地址输入框，输入“[snakeninny@gmail.com](mailto:snakeninny@gmail.com)”，然后在图10-19的“MOV R8, R0”处下一个断点，接着点击“return”。待断点触发时执行“po [$r0 \_\_ck\_displayName]”，看看是否会输出“iMessage”：

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.

从上面代码来看，断点被触发两次，第二次被触发时iOS检测到“[snakeninny@gmail.com](mailto:snakeninny@gmail.com)”支持iMessage。既然“iMessage”来自于[[[self conversation] sendingService] \_\_ck\_displayName]，那么[self conversation]是个什么类型的对象？[[self conversation] sendingService]呢？别急，下面一个一个来看。

From the output, we know that the breakpoint has been triggered twice, and iMessage support was detected in the 2ndtime. 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.

重新输入地址，然后在[CKMessageEntryView updateEntryView]的前两个“objc\_msgSend”上各下一个断点，最后点击“return”，现在来看看这里的对象类型：

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)

可见，[self conversation]返回的是一个CKPendingConversation类型的对象。接着看下一个：

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

显然，[CKPendingConversation sendingService]返回的是一个IMSerciceImpl对象，且其值为“IMService[SMS]”（第2次被触发时就会变成“IMService[iMessage]”）。因此，四重数据源就是“[CKPendingConversation sendingService]”，没问题吧？

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 IMSerciceImpl. Therefore, the 4th data source is [CKPendingConversation sendingService]. Get you still follow?

分析到这里，再回到IDA，定位[CKPendingConversation sendingService]，看看它的内部是如何工作的，如图10-20所示。

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.

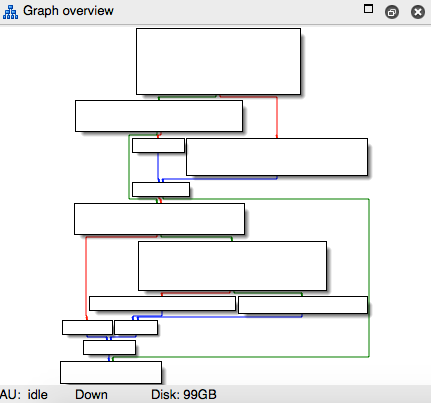


Figure 10- 20 [CKPendingConversation sendingService]

其中的逻辑并不算太复杂，但在若干分支里，进程实际走的是哪一条路呢？用LLDB调试一下（下断点前先重新输入地址），在所有条件跳转指令上留意一下跳转条件的值和下一条指令的地址：

The execution flow is not too complicated. But there are several branches so that we can’t make sure which one the process actually goes. Debug again with LLDB and pay attention to every branch condition and 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}

进程的执行流程就显而易见了。这里一共有3次条件跳转，分别是CBZ、CBZ和CBNZ，而此时的R0分别是0、341691792和341691792，由此可知进程的执行流程是绿线、红线、蓝线、绿线，如图10-21所示。

The execution flow of the process is very evident now. There are 3 conditional branches, which are CBZ, CBZ and CBNZ respectively. And 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.

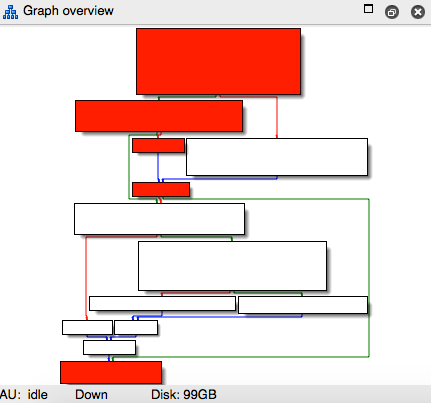


Figure 10- 21 Execution flow

那么[CKPendingConversation sendingService]的值实际来自[CKPendingConversation composeSendingService]，它是五重数据源，没问题吧？现在，在IDA中定位到了新的函数（如图10-22所示），继续分析。

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.

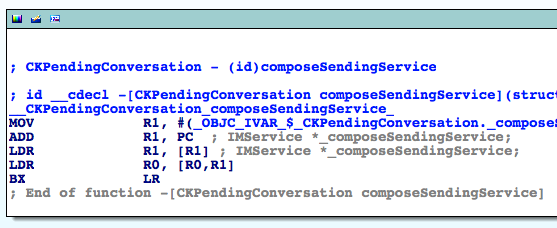


Figure 10- 22 [CKPendingConversation composeSendingService]

很明显，[CKPendingConversation composeSendingService]只是读取了实例变量\_composeSendingService的值，也就是说，\_composeSendingService是六重数据源。既然如此，我们只要找到写入此实例变量的位置，就找到了七重数据源了。

Obviously, [CKPendingConversation composeSendingService] just 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.

单击\_OBJC\_IVAR\_$\_CKPendingConversation.\_composeSendingService，让光标停留在其上，然后按下“x”，打开此变量的xrefs窗口，如图10-23所示。

Click \_OBJC\_IVAR\_$\_CKPendingConversation.\_composeSendingService to focus the cursor on it. Then press "x" to browse xrefs to this variable, as shown in figure 10-23.

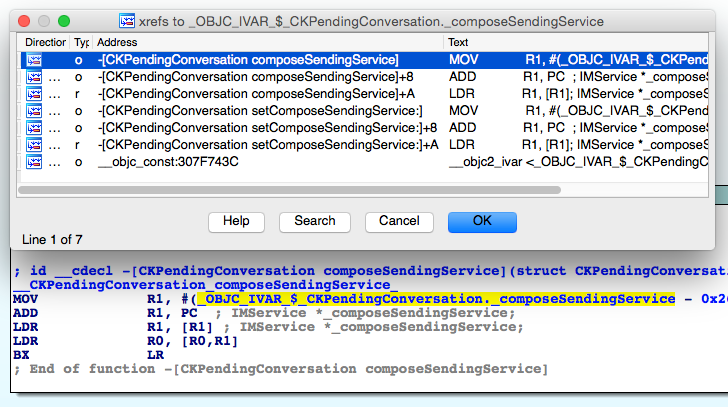


Figure 10- 23 See cross references

在这里，显式调用\_composeSendingService的正好是一个getter和一个setter，因此composeSendingService很可能是一个property，打开CKPendingConversation.h来验证一下，如图10-24所示。

Here, we can find there are 2 methods calling \_composeSendingService explicitly, 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.

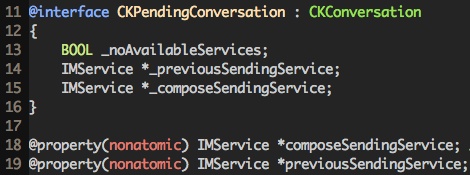


Figure 10- 24 CKPendingConversation.h

在Objective-C中，对property的写操作一般是通过setter完成的，那么要寻找七重数据源，就要在[CKPendingConversation setComposeSendingService:]上下一个断点，然后查看其调用者的情况。操作流程跟前面完全一样，先重新输入地址，然后在[CKPendingConversation setComposeSendingService:]的开始处下一个断点，最后点击键盘上的“return”，触发断点：

In Objective-C, write operation to 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're the callers. 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

用这里的LR减去ChatKit的ASLR偏移得到0x2698456D，得出其偏移前的值。再在IDA中跳到这个值所在的地址，如图10-25所示。

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.

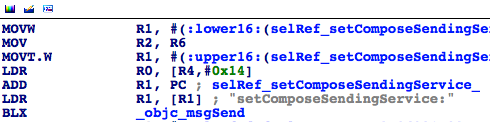


Figure 10- 25 Jump to 0x2698456D

[CKPendingConversation setComposeSendingService:]的参数R2就是七重数据源。R2来自R6，因此R6是八重数据源。再往上看看R6是什么，如图10-26所示。

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.

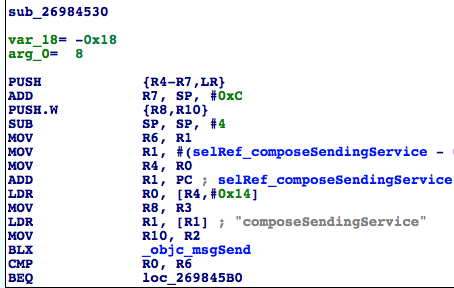


Figure 10- 26 Look for the 9th data source

R6来自R1，可见R1是九重数据源。那么R1又是哪里来的？因为我们现在位于sub\_26984530的内部，而R1没有被赋值就直接取值了，说明R1来自sub\_26984530的调用者，对吧？那就去看看sub\_26984530的交叉引用信息，寻找它的调用者信息，如图10-27所示。

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 set, then R1 comes from the caller of sub\_26984530, right? So let’s take a look at the cross references of sub\_26984530 to look for its callers, as shown in figure 10-27.

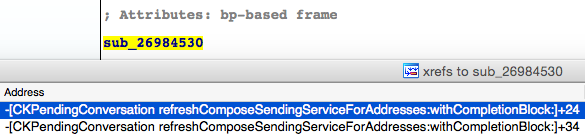


Figure 10- 27 See cross references

刷新发送服务？这个名字有点“暧昧”啊！到图10-28中所示的[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]中看看，sub\_26984530明显是refreshStatusForAddresses:withCompletionBlock:的第二个参数，即completionBlock，如图10-28所示。

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. As we can see, sub\_26984530 is obviously the 2nd argument of refreshStatusForAddresses:withCompletionBlock:, namely the completionBlock, as shown in figure 10-28.

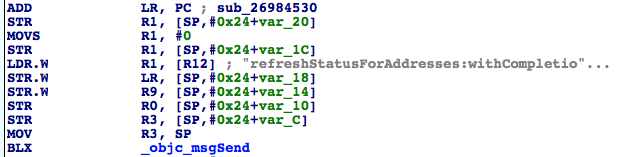


Figure 10- 28 [CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]

这里虽然显式用到了sub\_26984530，但只是作为参数传递给objc\_msgSend的，并没有直接调用。那么它的调用者到底是谁呢？这个套路前面已经反复演练过了：重新输入地址，在sub\_26984530的开始处下断点，点击键盘上的“return”，触发断点：

Although sub\_26984530 appears in this method, it is just an argument of objc\_msgSend and it isn’t called directly. Well, who is the 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偏移前的值是0x30b364bb - 0xa1b2000 = 0x269844BB。定位到IDA中，如图10-29所示。

LR without offset is 0x30b364bb - 0xa1b2000 = 0x269844BB. Locate it in IDA, as shown in figure 10-29.

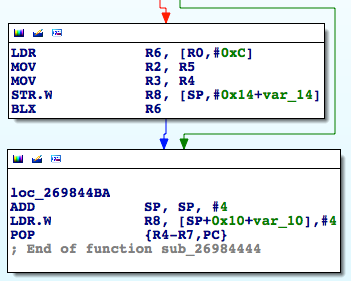


Figure 10- 29 Caller of sub\_26984530

可见，sub\_26984530并没有被显式调用，而是在sub\_26984444内部把函数的地址存放到了R6中，然后跳转过去隐式调用。那么自然，九重数据源就来自于sub\_26984444，再往上看看它的来源，如图10-30所示。

As we can see, sub\_26984530 isn’t called explicitly. Instead, its address is stored in R6 to where the execution flow jumps, 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.

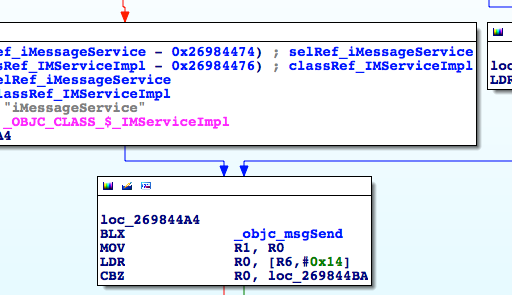


Figure 10- 30 Look for the 9th data source

这个subroutine内部进行了多次判断，才决定是把[IMServiceImpl smsService]还是把[IMServiceImpl iMessageService]赋给R1。我们看看判断条件是什么（如图10-31所示）。

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 condition, as shown in figure 10-31.

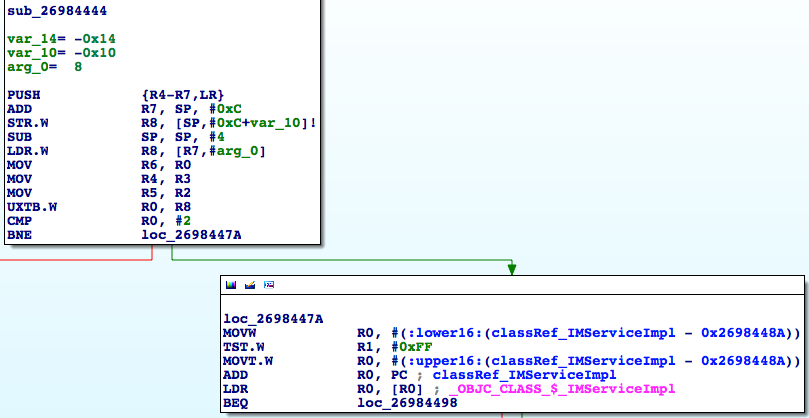


Figure 10- 31 Look for the 10th data source

当R0为2时，[IMServiceImpl iMessageService]是十重数据源，否则还要判断R1的值，若其值是0，则十重数据源是[IMServiceImpl smsService]，否则是[IMServiceImpl iMessageService]。用伪代码表示如下：

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 R0 is 0, then [IMServiceImpl smsService] is the 10th datasouce, 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;

}

也就是说，这里的R0与R1共同决定十重数据源的值，它们俩共同担当起了十一重数据源的重任，我们分别称它们为十一重数据源A和十一重数据源B，上面的伪代码也可以写作：

That is to say, the value of the 10th data source is determined by both 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;

}

回到图10-31，再看看十一重数据源是哪里来的——R0来自于“UXTB.W R0, R8”。

Get back to figure 10-31 to track the 11th data source; R0 comes from "UXTB.W R0, R8".

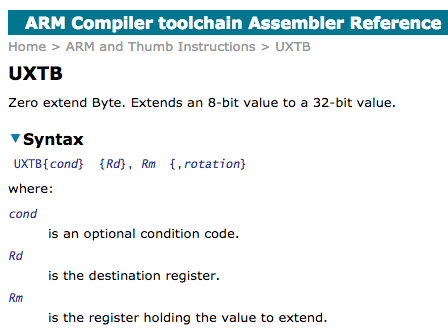


Figure 10- 32 UXTB

依图10-32所示的ARM官方文档可知，UXTB的作用是给R8中存放的8位数值高位填充0，将其扩展到32位，然后放到R0里去（R0是32位寄存器），也就是说，R0来自于R8，R8是十二重数据源A。而arg\_0 = 0x8，R8 = \*(R7 + arg\_0) = \*(R7 + 0x8)，R7 = SP + 0xc，因此R8 = \*(SP + 0x14)，即\*(SP + 0x14)是十三重数据源A。\*(SP + 0x14)又是哪里来的呢？它不是凭空产生的，因此在“LDR.W R8, [R7,#8]”之前，一定有一条指令往\*(SP + 0x14)写了数据，对吧？那里就是十四重数据源A的所在之处，我们要倒推回去找到这个往\*(SP + 0x14)写数据的地方。

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 (R0 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 arg\_0 = 0x8, R8 = \*(R7 + 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 definitively 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).

但是事情远没有说起来这么简单——因为PUSH和POP等操作会改变SP的值，所以现在的\*(SP + 0x14)在其他的指令中可能会由于SP发生变化而变成\*(SP’ + offset)，而offset的值现在还没法确定！这下子麻烦了，我们必须找出“LDR.W R8, [R7,#8]”之前每一个往\*(SP’ + offset)写数据的操作，然后检查(SP + 0x14)和(SP’ + offset)是否相等。而SP的变化又比较频繁，这就成了难点。请大家一定跟紧了，现在，要从“LDR.W R8, [R7,#8]”开始，倒推并检查每一个往\*(SP’ + offset)写数据的操作。

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 appears in the form of \*(SP’ + offset) in other instructions due to the change of SP. And what’s even worse is that the value of offset is undetermined statically. Sounds like we're getting into troubles! From now on, we have to find every single operation that writes into \*(SP’ + offset) before "LDR.W R8, [R7,#8]", and then check whether (SP + 0x14) equals to (SP’ + offset). Thanks to the frequent and irregular change 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’ + offset).

在sub\_26984444，“LDR.W R8, [R7,#8]”前的4条指令全都跟SP相关，我们把当前指令还未执行时SP的值用SP1~SP4标注到指令上，如图10-33所示。

In sub\_26984444, the first 4 instructions before "LDR.W R8, [R7,#8]" all SP related. We use SP1~SP4 to mark the value of SP before the execution of the current instruction, as shown in figure 10-33.

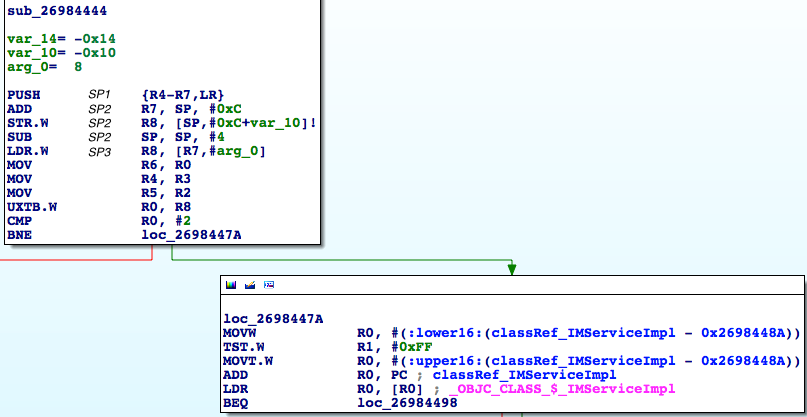


Figure 10- 33 Mark different SPs

当“PUSH {R4-R7,LR}”还未执行时，SP的值是SP1，执行之后变成SP2，可以理解吧？下面一条条指令推导一下这里的SP是怎么变化的：

Before and after the execution of "PUSH {R4-R7,LR}", the value of SP is SP1 and SP2 respectively, can you understand? Next, we will try to deduce how SP changes instruction by instruction.

“PUSH {R4-R7,LR}”将R4、R5、R6、R7和LR共5个寄存器入栈，每个寄存器都是32位即4字节，而ARM的栈是满递减的，因此，SP2 = SP1 - 5 \* 0x4 = SP1 - 0x14；“ADD R7, SP, #0xC”即R7 = SP2 + 0xc，没有影响SP的值；“STR.W R8, [SP,#0xC+var\_10]!”中的var\_10 = -0x10，因此这条指令等价于“STR.W R8, [SP,#-4]!”，即\*(SP2 – 0x4) = R8，且此条指令仍未影响SP的值；“SUB SP, SP, #4”即SP3 = SP2 - 0x4。按照这种表达方式，十三重数据源A是\*(SP2 + 0x14)，在通过“LDR.W R8, [R7,#8]”取值时尚未在sub\_26984444内赋值，因此\*(SP2 + 0x14)的值一定来自sub\_26984444的调用者。类似地，在sub\_26984444内部R1未被赋值即已取值，它也一定来自sub\_26984444的调用者，可以理解吗？如果你还不理解，就把这一段重看一遍，一定要先想清楚，再继续看下面的内容。

"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.

好了，十三重数据源A和十一重数据源B都来自sub\_26984444的调用者，下一步的任务已经很明确了：去sub\_26984444的调用者寻找十四重数据源A和十二重数据源B。

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.

重输地址，在sub\_26984444的开始处下断点，点击键盘上的“return”，触发断点：

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偏移前的值是0x331f0d75 – 0xa1b2000 = 0x2903ED75，并不在ChatKit中。这时候该怎么定位0x2903ED75位于哪一个模块呢？方法之前也说过了，那就是在sub\_26984444的末尾下断点，然后“ni”到调用者的内部，看看它在哪个模块就好了，如下：

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 10, which is simply set a breakpoint at the end of sub\_26984444 and keep executing "ni" to enter the internal of caller and identify it. 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]

我们来到了IMCore的内部。刚才已经计算出了sub\_26984444执行完成后的返回地址是0x2903ED75，因此把IMCore拖进IDA，待分析完毕后跳转到0x2903ED75，如图10-34所示。

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.

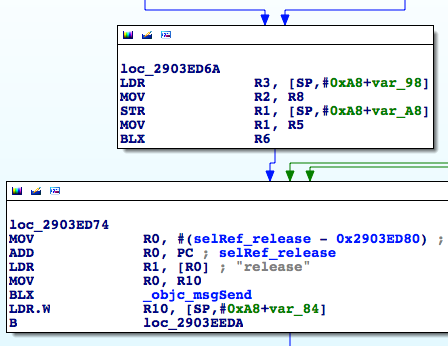


Figure 10- 34 Caller of sub\_26984444

又是一个来自sub\_2903E824的隐式调用，且它上面的4条指令又有2条跟SP相关。为方便阅读，下面把“BLX R6”前后两个模块的指令给拼到一张图里，拼接前后的效果如图10-35和图10-36所示。

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 result is shown in figure 10-35 and figure 10-36.

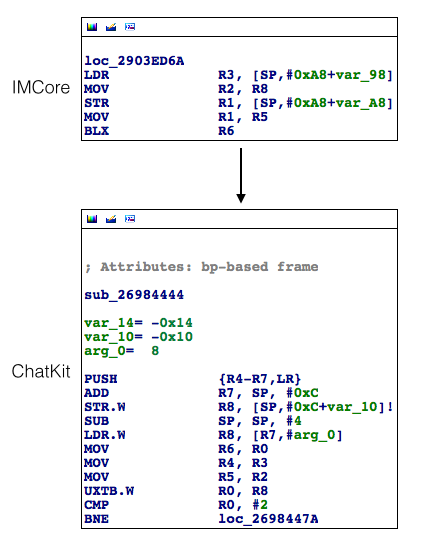


Figure 10- 35 Before instructions of 2 images are put together

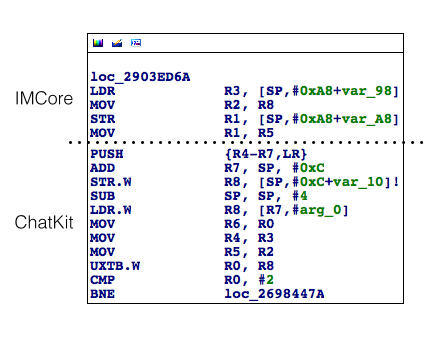


Figure 10- 36 After instructions of 2 images are put together

这里先寻找十四重数据源A，它被写入了\*(SP2 + 0x14)，还记得吗？模仿上面的步骤，把loc\_2903ED6A里的SP标号，如图10-37所示。

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.

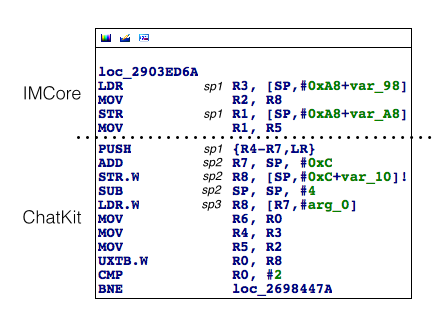


Figure 10- 37 Mark SPs

然后从loc\_2903ED6A的第一条指令开始，看看这里的SP是怎么变化的：

Then we should go through loc\_2903ED6A from its 1st instruction to check how SP changes here.

“LDR R3, [SP,#0xA8+var\_98]”，即R3 = \*(SP1 + 0xA8 + var\_98)，而var\_98 = -0x98（如图10-38所示）。

"LDR R3, [SP,#0xA8+var\_98]" equals to R3 = \*(SP1 + 0xA8 + var\_98). And var\_98 = -0x98 (as shown in figure 10-38).

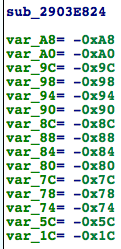


Figure 10- 38 sub\_2903e824

因此R3 = \*(SP1 + 0x10)，且本条指令不影响SP的值；“MOV R2, R8”跟SP无关；“STR R1, [SP,#0xA8+var\_A8]”中的var\_A8 = -0xA8，即\*SP1 = R1，且本条指令也不影响SP的值；“MOV R1, R5”跟SP无关。这么多SP可能会把你绕晕，再梳理一下其中的逻辑：

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.

目的：找到写入\*(SP2 + 0x14)的地方

Our goal is to find where \*(SP2 + 0x14) is written to.

因为SP2 = SP1 - 0x14

Since SP2 = SP1 - 0x14 and \*SP1 = R1,

且\*SP1 = R1

所以，“STR R1, [SP,#0xA8+var\_A8]”就是写入\*(SP2 + 0x14)的地方，十四重数据源A就是这条指令中的R1！而十二重数据源B，显然就是“MOV R1, R5”中的R5。从十三重数据源A到十四重数据源A，从十一重数据源B到十二重数据源B的追踪跨模块，逻辑比较复杂，用图10-39来展示会较为直观，建议大家对照着这张图，把这个跨模块的地方弄清楚。

Therefore, "STR R1, [SP,#0xA8+var\_A8]" is the place where \*(SP2 + 0x14) is written to, 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 logic of tracing from 13th data source A to 14th data source A and from 11th data source B to 12th data source B crosses images, which brings 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.

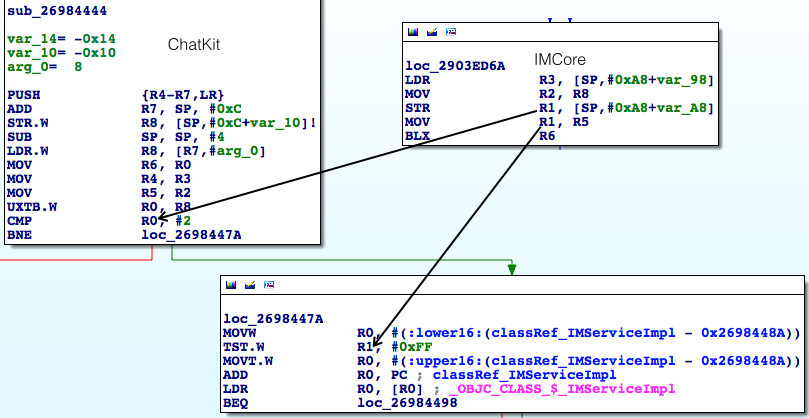


Figure 10- 39 How data sources evolve

在继续分析之前，先用LLDB来验证一下到此为止的判断：重输地址，然后在“STR R1, [SP,#0xA8+var\_A8]”上设置断点，打印R1，即十四重数据源A；执行“ni”命令到“MOV R1, R5”，打印R5，即十二重数据源B；接着要经历一次模块切换，执行“si”命令到“CMP R0, #2”，打印R0，即十三重数据源A；执行“ni”命令到“TST.W R1, #0xFF”，打印R1，即十一重数据源B。点击“return”，触发断点后，看看它们的值是不是像图10-39里示意的那样两两相等：

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

打印结果验证了我们的分析结果，且十四重数据源A的值是0，十二重数据源B的值是1。接下来把火力集中在IMCore上，继续寻找十五重数据源A和十三重数据源B，先从前者下手。

The output verifies our analysis, and the 14th data source A is 0 and 12th data source B is 1. Next, we need to focus on IMCore to keep finding 15th data source A and 13th data source B. Let’s get started from the 15th data source A.

十五重数据源A已经直观表现在了图10-40中。

The 15th data source A has been presented in figure 10-40 intuitively.

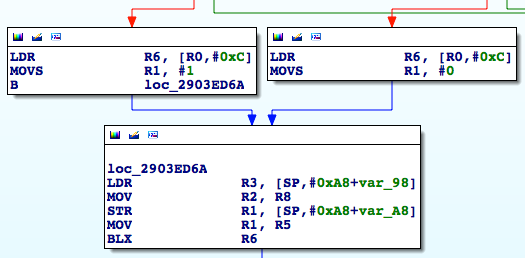


Figure 10- 40 15th data source A

它要么来自“MOVS R1, #1”，要么来自“MOVS R1, #0”，也就是说十五重数据源的值要么是0，要么是1。事情变得有意思起来了……

It comes either from "MOVS R1, #1" or "MOVS R1, #0". In other words, the 15th data source is either 0 or 1. Things are getting interesting.

不知道大家有没有注意到，从十一重数据源A开始，数据源A的值就是一脉相承的，即十一重 = 十二重 = 十三重 = 十四重 = 十五重数据源A = 0或1。但是，我们之前分析的伪代码是这样的：

IIRC, since the 11th data source A, 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)supportIMessage

{

if (11thDataSourceA == 2 || 11thDataSourceB!= 0) return YES;

return NO;

}

而十一重数据源A的值非0即1，是不可能等于2的。这样的话，数据源A已经没有意义了，不是吗？伪代码可以改成：

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)supportIMessage

{

if (11thDataSourceB != 0) return YES;

return NO;

}

因此，可以把重点放在寻找十三重数据源B上来，以下简称十三重数据源。因为十二重数据源B是R5，所以十三重数据源一定被某条指令写入R5了，对吧？单击R5，IDA会贴心地帮我们将其标记为黄色，方便从大量的汇编代码中定位R5。继续逆向，看看R5是什么时候被写入的。

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 tracking in the sea of ARM assembly. Keep reversing to find where R5 is written to.

当我们向上寻找十三重数据源，跟踪到loc\_2903EAE0时，会发现它的上游有4条分支（如图10-41所示）。

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.

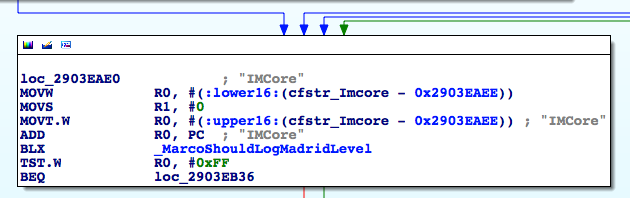


Figure 10- 41 loc\_2903EAE0

在图10-41中，从左到右的3条分支中均含有“MOVS R5, #0”的操作，但这跟R5 = 1的结果是矛盾的，因此loc\_2903EAE0一定是经由最右边的那一条线路到达的，十三重数据源就位于这条线路上，顺着这条线路继续向上寻找R5的踪影。

In figure 10-41, the left 3 branches all contain a "MOVS R5, #0", which is contradictory with the result of R5 = 1, so loc\_2903EAE0 must be reached via the rightmost branch, and the 13th data source should be right in this branch. Follow this branch for R5.

跟踪到loc\_2903EA3E时，出现的情况似曾相识。它的上游虽然有3条分支，但左1和左2分支均含有“MOVS R5, #0”的操作（如图10-42所示），因此可以直接排除。

When we track 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.

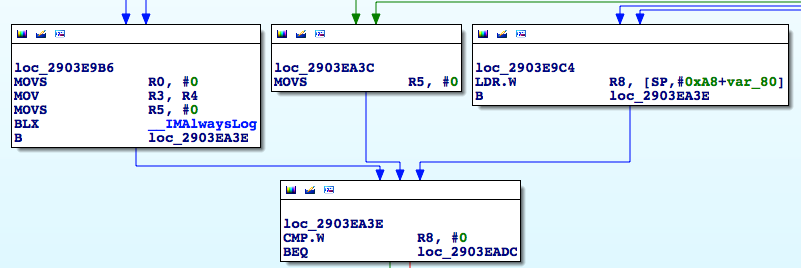


Figure 10- 42 loc\_2903EA3E

它的实际上游是左3分支，即loc\_2903E9C4；而loc\_2903E9C4的上游有2条分支，其中均含有“MOVS R5, #1”的操作，那么进程的实际执行流程是从这2条分支中的哪一条到达loc\_2903E9C4的呢？重输地址，在2个分支的跳转处各下一个断点，点击键盘上的“return”，看看会触发哪个断点，这样就一清二楚了。这里的操作流程笔者就省略掉了，请读者独立完成，相信在简单的操作之后，你也会发现，左1分支才是进程实际执行的流程，即图10-43。

As a result, the real upstream is the 3rd branch, i.e. loc\_2903E9C4, and there are 2 branches upon it. But both of 2 branches contain "MOVS R5, #1", which is the actual one?

Reinput the address and set breakpoints on each of these 2 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 the process chose, as shown in figure 10-43.

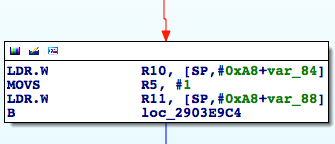


Figure 10- 43 The left branch

现在，找到了十三重数据源，它是常量1。你可能会问，十三重数据源是常量的话，十四重数据源还存在吗？数据源的线索看似中断了，下一步该怎么办？问得好！

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 really exist? The data source clues seem to be interrupted, what should we to do next? Good point.

在刚才的代码中，我们看到了几处“MOVS R5, #0”的操作，而十三重数据源来自“MOVS R5, #1”，看似数据源是常量，但按照程序设计的思想，到底是往R5里写0还是写1，应该是由一个条件判断来决定，就像下面的伪代码：

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;

用我们熟悉的IDA流程图表示，就是图10-44所示的样子。

To represent in our familiar IDA graph view, it looks like figure 10-44.

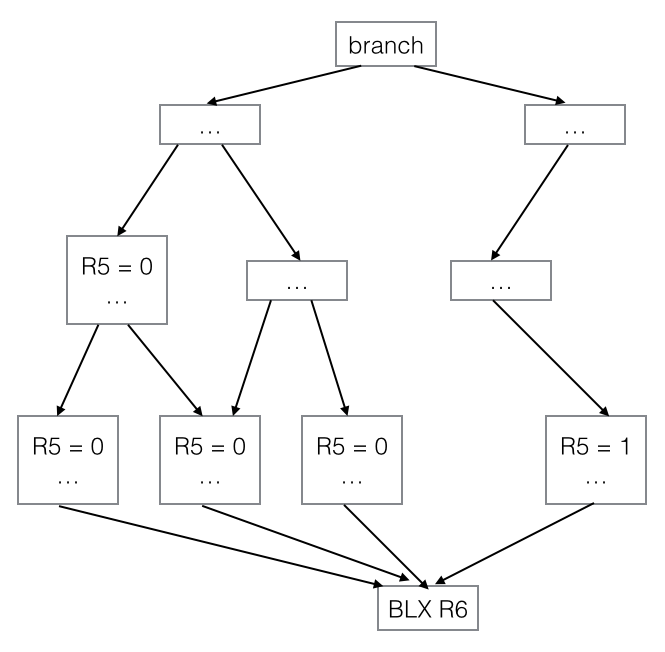


Figure 10- 44 Pseudo IDA graph view

从宏观角度来讲，其实这个条件判断就是十四重数据源，不是吗？相信你也反应过来了，上面的伪代码其实就是：

From a macro point of view, this conditional branch is actually the 14th data sources, right? I bet you've realized that the above pseudo code can be rewritten as below:

R5 = iMessageIsAvailable;

弄清了这个概念，接下来的任务就是继续回溯，分析每个出现分支的地方，如果它的不同分支会往R5里写不同的值，就要搞清楚分支条件是什么，而这个分支条件就是我们要找的数据源。到图10-45所示的代码段里去看看。

After understanding this concept, our next task is to continue going back to analyze each branch. If different branch result in the different value of R5, we have to figure it out what’s branch condition. And the branch condition is the data source we look for. So let’s take a look at the code snippet in figure 10-45.

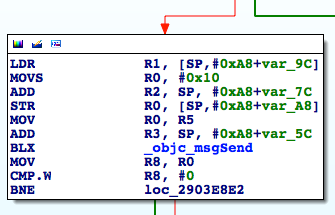


图10- 45分支

如果分支走了左边，R5是有可能被置0的。由于分支条件是objc\_msgSend的返回值，因此来下个断点看看执行的到底是什么函数：

If the process choose the left branch, R5 can be set to zero. 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

)

可以看到，是一个对收件人队列的遍历函数，如果收件人队列不为空，分支就会走右边。实际上我们的收件人队列肯定不会为空，因此这个分支条件不会成立，类似于已弃用的数据源A，是不会产生分支的。继续往上，寻找上一个分支发生的地方，如图10-46所示。

We can see that, it is a method traversing the queue of receivers. If the queue is not empty, the process choose to go rightside. Actually, the receiver queue will not be empty and therefore this branch condition cannot be true. Just like the deprecated data sourceA, there is only one workflow here. Ok, go upward to find another place where branch condition happens, as shown in figure 10-46.

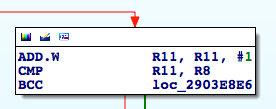


图10- 46 分支

在图10-46中，R11和R8各是什么？在IDA里可以很直观地看到，R11来自图10-47。

In figure 10-46, what is R11 and R8? We can get a straightforward answer from IDA. R11 is from figure 10-47.

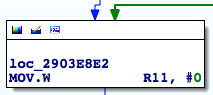


图10- 47 loc\_2903e8e2

R11的初始值是0，每次在执行“CMP R11, R8”之前，R11都递增1。这么看起来，R11充当了计数器的作用。“CMP”执行了减法操作，如果产生借位，则将Carry置0，否则Carry置1。这里的分支指令是“BCC”，“CC”代表“Carry Clear”，也就是Carry位为0。也就是说，如果R11– R8产生借位，即R8大于R11，则分支走右边，否则走左边。我们看看R8是什么，如图10-48所示。

The initial value of R11 is 0. Each time before executing CMP R11, R8, R11 will increment by 1. In this way, we can regard R11 as a counter. "CMP" performs the subtraction operation, if it produces a borrow, the carry will be set 0, otherwise the carry will be 1. Branch instruction here is BCC. CC means carry clear, in other words, carry is 0. Therefore, if R11 – R8 produces a borrow, namely R8 is bigger than R11, then the process will go through right one. Otherwise it will choose the left one.So let’s take a look at what R8 is, as shown in figure 10-48.



图10- 48 R8来源

R8来自[NSArray countByEnumeratingWithState:objects:count:]。重输地址，下断点，点击“return，”看看NSArray是什么：

R8 comes from [NSArray countByEnumeratingWithState:objects:count:]. Re-input the address, set the breakpoint and press the return button to 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是收件人队列，因此R8是收件人个数，如果收件人个数大于1，在第一次执行“CMP R11, R8”时R11是1，则R8大于R11，分支走右边，到达图10-49。

NSArray is a queue of receivers. Thus R8 is the count of receiver. If the count is greater than 1 and R11 is 1 when CMP R11, R8 get executed for the first time, we can know that R8 is greater than R11 and the process choose the right branch, as shown in figure 10-49.

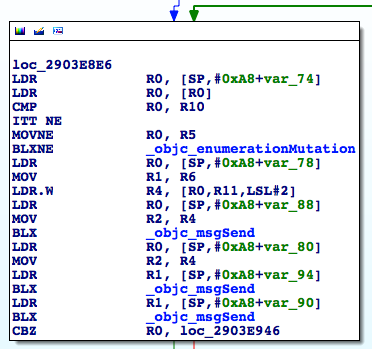


图10- 49 分支

loc\_2903E8E6的分支条件是R0，如果R0 == 0则走左边，不支持iMessage；否则走右边，到达图10-50。

The branch condition for loc\_2903E8E6 is RO. If R0 == 0, then the left branch get executed, which means this address doesn’t support iMessage. Otherwise, the right side will be executed and go to the instruction shown in figure 10-50.

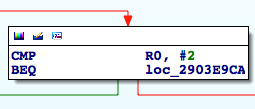


图10- 50 分支

图10-50的分支条件仍是R0，如果R0 == 2则走左边，不支持iMessage；否则走右边，回到图10-46。这3段代码循环没有更改R8的值，只要loc\_2903E8E6最下面的R0 != 0 && R0 != 2，图10-46的分支就没有意义——R11一直递增，而R8不变，这个分支早晚会走左边，得出支持iMessage的结论；也就是说，在这个循环里，本质的分支条件是R0的值。还记得我们刚刚得出的结论吗——“分析每个出现分支的地方，如果它的不同分支会往R5里写不同的值，就要搞清楚分支条件是什么，而这个分支条件就是我们要找的数据源。”——R0就是十四重数据源。

The branch condition in figure 10-50 is still R0. If RO == 2, then left side is executed, in other words, iMessage is not supported. Otherwise the process choose the right side and turn back to figure 10-46. These three code loops don’t change the value of R8.As long as R0 != 0 && R0 != 2 at the bottom of loc\_2903E8E6 is true, the branch in figure 10-46 is useless. That’s because R11 is keeping increased while R8 always keeps the same, the branch will eventually choose the left side and find iMessage is supported. So judging from all information above, we can think of the essential branch condition in this loop is R0. Do you still remember the conclusion we just got? “We need to analyze every single branch. If different branch results in the different value of R5, we have to figure out the branch condition and the branch condition is the data source we look for”. Thus, we can say R0 is the 14th data source.

下面用LLDB看看这里的几个objc\_msgSend都是什么，R0是怎么来的：

Next, use LLDB to check what are these objc\_msgSend and where is R0 from.

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

将这3个objc\_msgSend还原成ObjC函数，分别是[NSArray removeObject:@ "mailto:snakeninny@gmail.com"]、[NSDictionary valueForKey: @"mailto:snakeninny@gmail.com"]和[NSNumber integerValue]，其中第二个objc\_msgSend的R0值得关注，正是它（NSDictionary）中包含的键值对，决定了十四重数据源；因此，这个NSDictionary就是十五重数据源。由图10-49可知，它来自于[SP,#0xA8+var\_80]，因此[SP,#0xA8+var\_80]是十六重数据源。接下来的套路已经做过好几遍了：把光标放在var\_80上，然后按下“x”，看看它的交叉引用，如图10-51所示。

Restore these three objc\_msgSend into Objc methods, they are [NSArray removeObject:@ "mailto:snakeninny@gmail.com"]、[NSDictionary valueForKey: @"mailto:snakeninny@gmail.com"]和[NSNumber integerValue] respectively. Among them, the R0 of the second objc\_msgSend attracts our special attention. It is the key-value pari of its NSDictionary determines the 14th data source. Therefore, this NSDictionary is the 15th data source. From figure 10-49, we can know that it comes from [SP,#0xA8+var\_80], which means the [SP,#0xA8+var\_80] is the 16th data source. After that, we follow the same pattern as we did before, put cursor on var\_80 and press x to check it cross reference, as shown in figure 10-51.

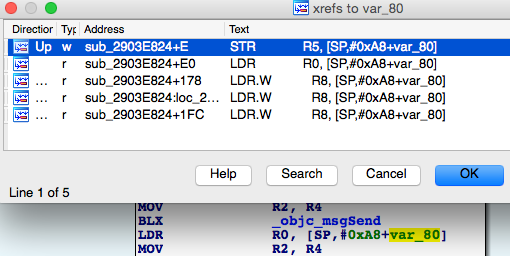


图10- 51 查看交叉引用

可以看到，只有一处指令写入了这个地址，双击这条指令，直接跳转到了sub\_2903E824的头部，如图10-52所示。

As we can see, only one instruction writes into this address. Double click this instruction to jump to the head of sub\_2903E824, as shown in figure 10-52.

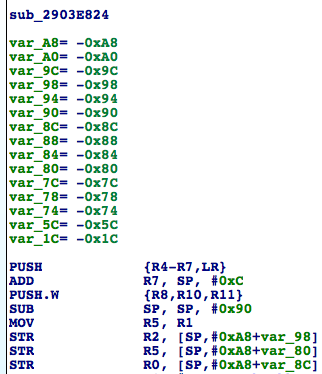


图10- 52 sub\_2903E824

十六重数据源来自于R5，故R5是十七重数据源；十七重数据源又来自于R1，因此R1是十八重数据源，而它没有被赋值就直接取值了，说明R1来自sub\_2903E824的调用者，对吧？我们看看它的交叉引用，如图10-53所示。

The 16th data source comes from R5. Therefore, R5 is the 17th data source. And we also know the 17th data source is from R1. So R1 is the 18th data source and its value is fetched without being assigned, which means R1 comes from the callee of sub\_2903E824, right? Let’s take a look at its cross references, as shown in figure 10-53.

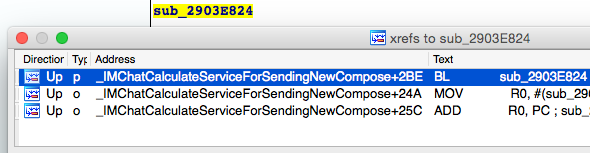


图10- 53 查看交叉引用

“为发送新的信息计算服务”这个名字的含义已经很明显了，双击第一条交叉引用，去它的显式调用者那瞅瞅，如图10-54所示。

ServiceForSendingNewCompose is very clear for us to get its meaning. Double click on the first cross reference and check its callee, as shown in figure 10-54.

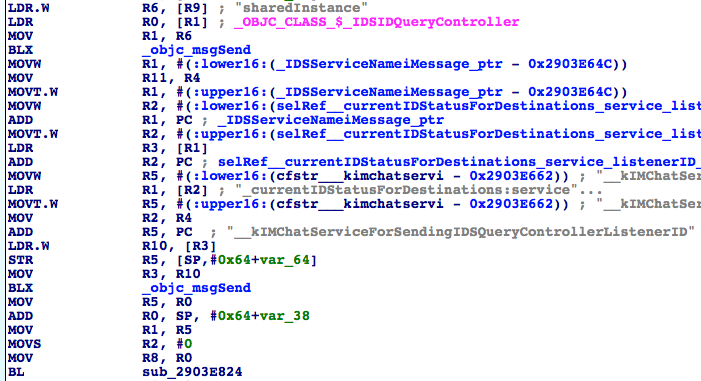


图10- 54 sub\_2903E824的调用者

这里要先确认一下sub\_2903E824的调用者是不是来自“IMChatCalculateServiceForSendingNewCompose”——清空地址输入框，输入地址，在sub\_2903E824的第一条指令上下一个断点，点击键盘上的“return”，触发断点：

Here, we first need to make sure whether the callee of sub\_2903E824 is from IMChatCalculateServiceForSendingNewCompose. Clear the address input and input a new address. Set a breakpoint at the first instruction of sub\_2903E824 and then press the 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)

这里的ASLR偏移是0xa1b2000，所以调用者的实际地址是0x2903E67B，正是来自“IMChatCalculateServiceForSendingNewCompose”——十八重数据源来自R5，因此R5是十九重数据源；而十九重数据源来自objc\_msgSend的返回值，故而该返回值是二十重数据源。万事俱备，只欠东风，我们看看这个神秘的objc\_msgSend到底做了些什么：

The ASLR offset is 0xa1b2000, so the real address of callee is 0x2903E67B, which is the same as the return value of IMChatCalculateServiceForSendingNewCompose. Ok, We have figured out the 18th data source. Since 18th data source is from R5, R5 is the 19th data source. Further, 19th data source is from the return value of objc\_msgSend, so this return value is the 20th data source. Confusing? Relax, we have get everything done and the only thing left is to check what’s inside this 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

锲而不舍，终有斩获。这个objc\_msgSend还原之后，是[[IDSIDQueryController sharedInstance] \_currentIDStatusForDestinations:@[@”mailto:snakeninny@gmail.com”] service:@”com.apple.madrid” listenerID:@”\_\_kIMChatServiceForSendingIDSQueryControllerListenerID”]，因为后两个参数是常量，所以可变参数只有第一个数组，也就是收件人数组，我们终于跟踪到了原始数据源！

Success comes after perseverance. After we restore the objc\_msgSend, we find it is [[IDSIDQueryController sharedInstance] \_currentIDStatusForDestinations:@[@”mailto:snakeninny@gmail.com”] service:@”com.apple.madrid” listenerID:@”\_\_kIMChatServiceForSendingIDSQueryControllerListenerID”]. Since the last two parameters are constant, the only variable is the first array, namely the queue of receivers. Ok, we finally find the original data source!

笔者知道本节的内容很难，你可能已经被绕晕了，但行百里者半九十，还差最后一步了，打起精神来！

I know the knowledge of this chapter is very hard and may make you confused. But remember, we only have one step left. So keep it up!

### 10.2.5 还原原始数据源生成placeholderText的过程

The process of restoring data source into placeholderText.

函数都被我们找到了，貌似可以通过更改第一个NSArray参数，来达到检测任意目标地址是否支持iMessage的效果，只要它的返回值（NSDictionary）中key所对应的value非0且非2，则key支持iMessage，否则key仅支持SMS。真的是这样吗？我们已经知道，对于邮件地址来说，参数格式为“<mailto:>”，那电话号码的参数格式呢？在\_currentIDStatusForDestinations:service:listenerID:上下个断点看一看：

Since we have found all methods, it seems that we can check whether an address supports iMessage by modifying the first NSArray parameter. As long as value associated with the key in the returned NSDictionary is neither 0 nor 2, we can confirm that this address supports iMessage. Otherwise it only supports SMS. Is this true? I know the format of email address is mailto:, how about the one of phone number? Let’s set a breakpoint \_currentIDStatusForDestinations:service:listenerID.

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

)

LLDB和debugserver可以暂时休息一下了。回到Cycript中，实际验证一下我们的猜测：

Ok, we can now turn back to Cycript to verify our assumption.

FunMaker-5:~ root# cycript -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.com":2,"mailto:snakeninny@icloud.com":1,"tel:+86PhoneNumber ":1}

哈哈，输出的结果再清楚不过了，2个支持iMessage的邮箱和1个手机号均返回了1，而另3个不支持iMessage的地址返回了2，实验结果验证了我们的分析，而且还知道了iMessage的内部称呼为“Madrid”。任务完成，万岁！

Aha, the return result clearly support our assumpition. Two iMessage emails and one cell phone number all return 1. And the other 3 non-iMessage supported address return 2. Also, from the above the result, we know the iMessage is called Madrid internally. mission complete! cheers!

## 10.3 发送iMessage

经过10.2节的洗礼，相信部分读者会产生跟笔者相同的感觉：一步一步用LLDB调试虽然准确严谨，但工作量巨大，容易让人感到厌倦。逆向工程就是要勇于试错，不走寻常路，用跳跃的思维和大胆的猜想来达到我们的目的。本节就会采用这种方式——尽量少地使用LLDB，尽量多地通过IDA和class-dump中看到的关键词，并用Cycript配合我们的联想来达到发送iMessage的目的。

### 10.3.1 从MobileSMS界面元素寻找逆向切入点

相对于检测iMessage，发送iMessage的切入点就要明显得多，在图10-55所示的iOS截图上，这个大大的“Send”按钮，不就是苹果送给我们的大礼么？

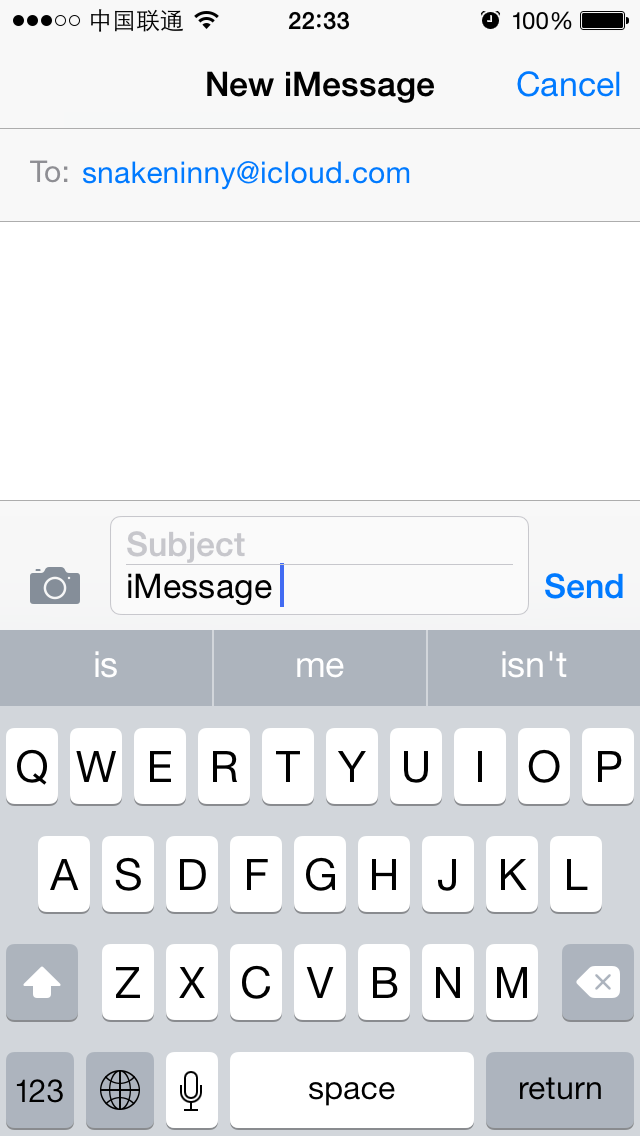


图10- 55 显眼的“Send”

按下“Send”，发送一条iMessage，这就是发送iMessage最直观的表现。跟10.2节一样，先想想怎么把这个现象给具象化成逆向工程：

“Send”按钮是一个UIView，具体地说，可能是一个UIButton；点击这个UIButton，调用UIButton的响应动作；全套响应动作包括更新界面，发送信息，添加已发送记录，等等，也就是说，发送信息的操作只是全套响应动作的子集。

在MobileSMS发送界面中，我们的输入只有收件人地址和信息内容，它们是原始数据源。因为可以拿到全套响应动作，而发送信息的操作一定要以原始数据源为参数，所以可以根据这2个条件，在全套响应动作里筛选出发送信息的操作。与上节的终点倒推起点不同，这次将从起点到达终点，演示逆向工程的另一种思路。

总结一下，逆向工程的思路是这样的：先用Cycript定位“Send”按钮的响应函数，然后用IDA纵览全套响应动作，结合LLDB和数据源，寻找可疑的发送操作。

### 10.3.2 用Cycript找出“Send”按钮的响应函数

因为前面在10.2节中已经找到了“Send”所在的view是一个CKMessageEntryView，所以这里就可以直接重复10.2.2节的分析思路，得出下面的结果：

FunMaker-5:~ root# cycript -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>>"]

其中，“UIView: 0x16052920”就是“iMessage”所在的view，还记得吧？那么，紧随其后的2个UIButton就显得十分可疑了，直觉告诉笔者，“Send”就是它俩其中之一。同时我们注意到，第三个UIButton的hidden属性是YES，也就是说这个按钮是隐藏的，那么可见的“Send”肯定就是“UIButton: 0x1605a8b0”了。还是用Cycript来确认一下：

cy# [#0x1605a8b0 setHidden:YES]

执行之后，界面变成了图10-56的样子：

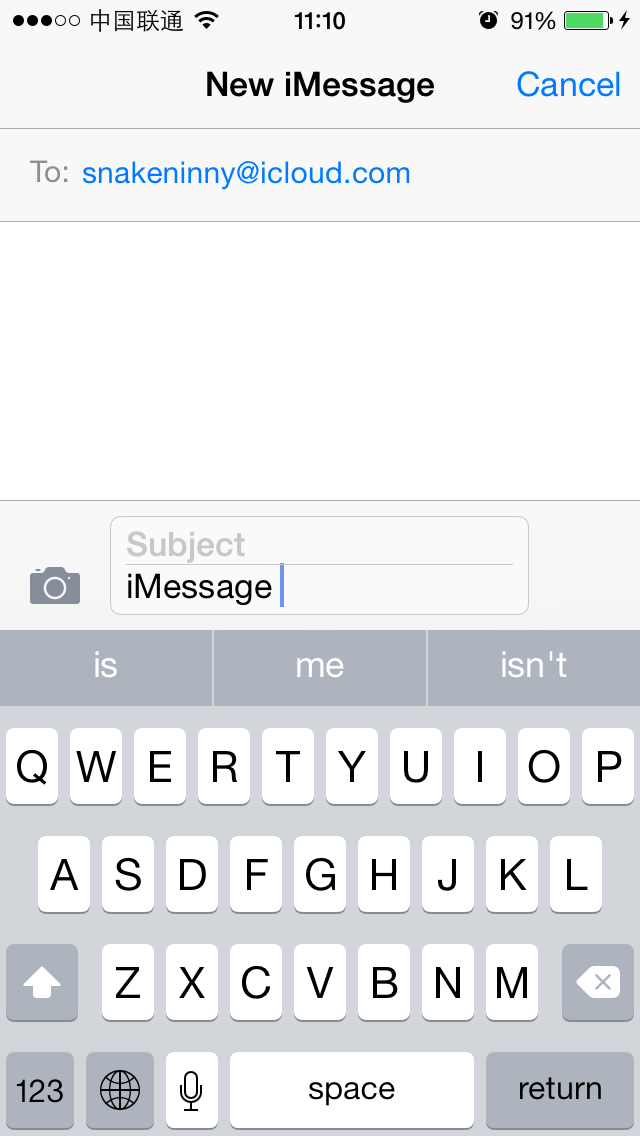


图10- 56 隐藏“Send”

准确无误。按下这个UIButton后，发送一条iMessage；UIButton与其点击之后的动作一般是通过addTarget:action:forControlEvents:函数来关联的，这是UIButton的父类UIControl中的一个函数。而UIControl类本身就提供了一个actionsForTarget:forControlEvent:来反查UIControl对象的动作。可以利用这个函数，来看看按下“Send”之后会触发什么动作：

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 forControlEvent:64]

@["touchUpInsideSendButton:"]

可以看到，触发的函数是[CKMessageEntryView touchUpInsideSendButton:button]。现在，转战到IDA和LLDB上，看看这个函数的内部实现。

### 10.3.3 在响应函数中寻找可疑的发送操作

[CKMessageEntryView touchUpInsideSendButton:button]的实现很简单，如图10-57所示。



图10- 57 [CKMessageEntryView touchUpInsideSendButton:button]

先[[self delegate] messageEntryViewSendButtonHit:self]，然后[self updateEntryView]。看名字就知道后者是简单地更新视图，那发送的动作应该就包含在前者内。下面先用Cycript看看[self delegate]是什么：

cy# [#0x160c6180 delegate]

#"<CKTranscriptController: 0x15537200>"

在IDA中前往[CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView]。这个函数的逻辑比较简单，如图10-58所示。

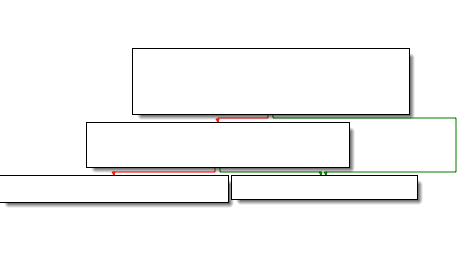


图10- 58 [CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView]

相信你用肉眼就能看出，实际的发送操作暗藏在[self sendComposition:[CKMessageEntryView compositionWithAcceptedAutocorrection]]中。接下来在Cycript中看看[self compositionWithAcceptedAutocorrection]是什么：

cy# [#0x160c6180 compositionWithAcceptedAutocorrection]

#"<CKComposition: 0x160b79d0> text:'iMessage {\n}' subject:'(null)'"

它是一个CKComposition对象，且明明白白地显示了要发送的标题和内容。继续看sendComposition:的内部实现，如图10-59所示。

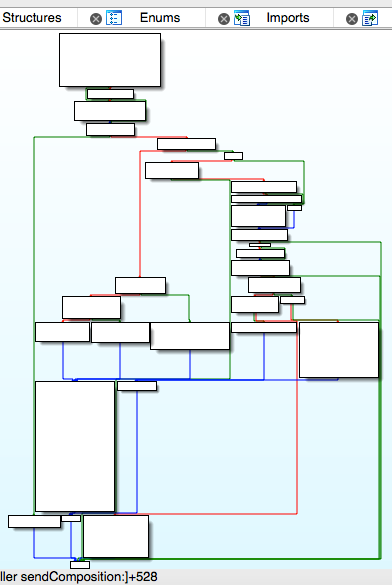


图10- 59 [self sendComposition:]

其实现很复杂，有必要在到LLDB上一步一步调试之前，先大概地过一下进程流程中的几个分支，看看其逻辑走向。先来到loc\_268D427C中，如图10-60所示。

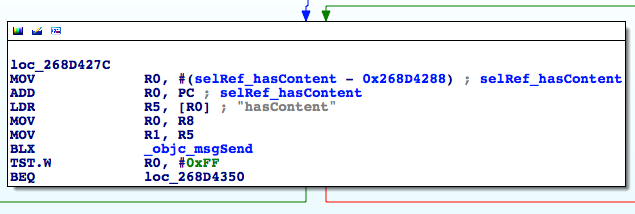


图10- 60 loc\_268D427C

如果“有内容”就走右边，我们发送的内容是“iMessage”，当然算是“有内容”，走右边，到达图10-61。

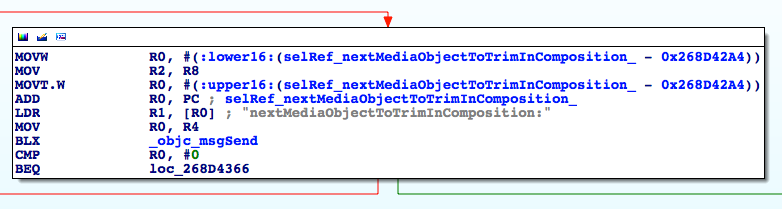


图10- 61 分支

“下一个待调整的媒体对象”？难道是指的图片、语音、视频这类东西？我们要发的iMessage是纯文字，应该不涉及这些东西，走右边，到达图10-62。

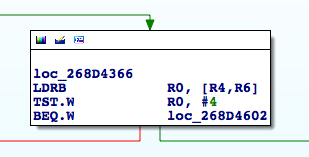


图10- 62 分支

R0是个啥？回到sendComposition:的开始部分，如图10-63所示。

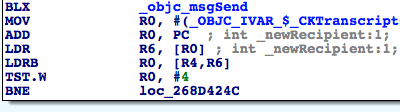


图10- 63 追踪R0的值

R0原来是self->\_newRecipient，在Cycript中看看它的值是多少：

cy# #0x15537200->\_newRecipient

1

因此“TST.W R0, #4”的结果是0，走右边到达loc\_268D4604，如图10-64所示。

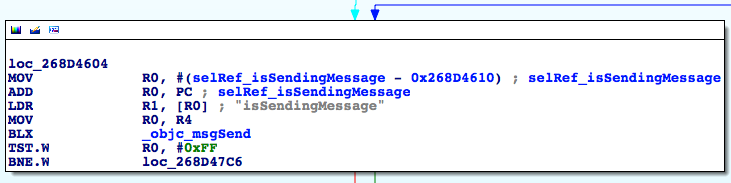


图10- 64 loc\_268D4604

“正在发送信息”？按下“Send”之后才发送信息，那这里的“正在”是指的按下“Send”之前还是之后呢？分别测试一下好了：

cy# [#0x15537200 isSendingMessage]

0

然后按下“Send”，再测一次：

cy# [#0x15537200 isSendingMessage]

0

可见，不管是按下“Send”之前还是之后，[self isSendingMessage]的返回值都是0，走左边那条路，继续寻找下一个分支，如图10-65所示。

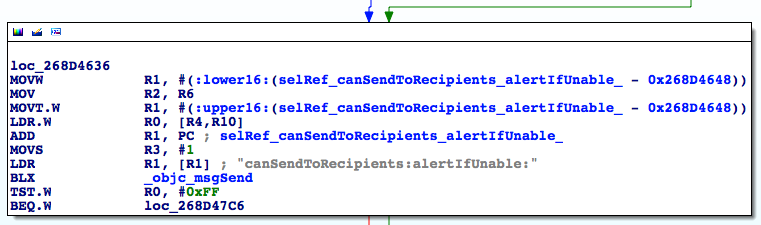


图10- 65 分支

“能发送给收件人吗？”我们的目标地址是一个有效的iMessage账号，当然能了！走左边，到达图10-66。

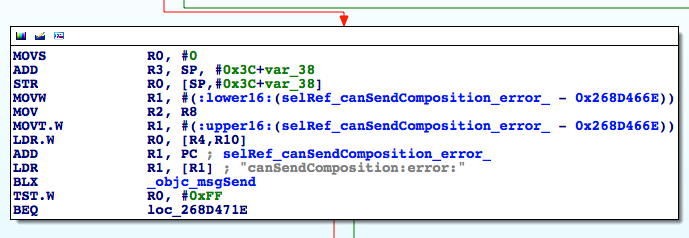


图10- 66 分支

“能发送写好的内容吗？”因为刚才都已经把CKComposition的内容打印出来了，所以这里也没问题，走左边，到达图10-67。

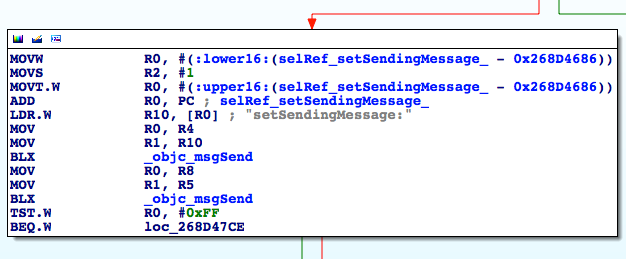


图10- 67 分支

这里又是一个分支。往上看看，就可以在图10-60中找到R5的值，可见，这里是再次判断我们发送的信息是否“有内容”了。走右边，到达图10-68。



图10- 68 分支

图10-68这张图的信息量略大，但仔细看看，你就会发现前面做的一系列动作都是UI层面的刷新操作，只有最后一个sendMessage:十分可疑。这个函数的参数是什么？往上回溯，可以看到其实就是[self sendComposition:]的参数，即一个CKComposition对象。继续分析[CKTranscriptController sendMessage:]的实现，如图10-69所示。

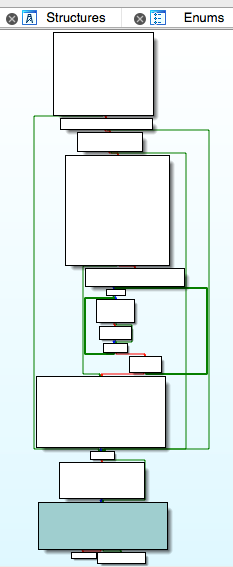


图10- 69 [CKTranscriptController sendMessage:]

这个函数的流程看起来分支众多，但在浏览一遍（思路跟浏览sendComposition:时一样）后就会发现，分支里都只是做了一些准备工作，“\_startCreatingNewMessageForSending:”才是可能发出信息的地方。去它的实现里看看，如图10-70所示。

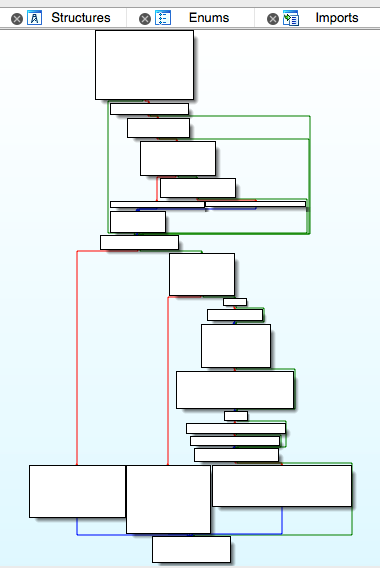


图10- 70 [CKTranscriptController \_startCreatingNewMessageForSending:]

这里的逻辑略纠结。按照上面描述过的思路，大致浏览一遍。相信你在浏览关键词的时候，也会同笔者一样注意到“sendMessage:newComposition:”，且它一共出现了2次，即图10-71所示的2个深色方块。

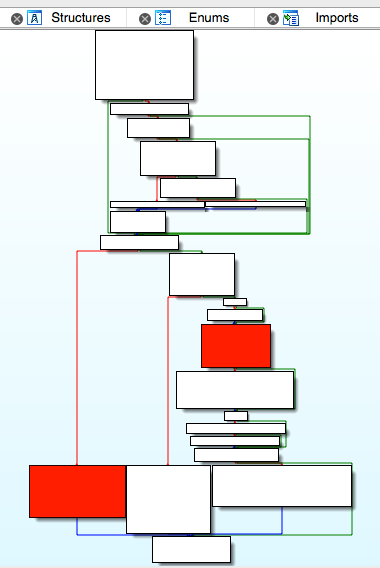


图10- 71 [CKTranscriptController \_startCreatingNewMessageForSending:]

下面来看看这个函数的实现，如图10-72所示。

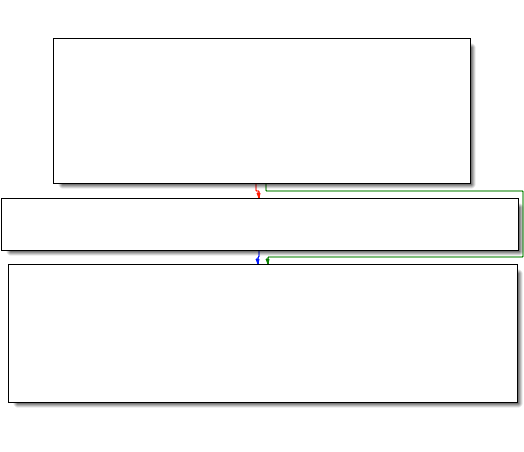


图10- 72 [CKConversation sendMessage:newComposition:]

它进一步调用了“sendMessage:onService:newComposition:”，继续跟过去，如图10-73所示。

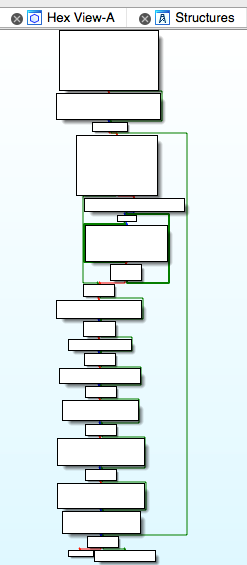


图10- 73 [CKConversation sendMessage:onService:newComposition:]

流程比较简洁，大致浏览一下，就会看到诸如“Sending message with guid: %@”、“ => Sending account: %@”、“ => Recipients: [%@]”等的字眼，且它们大都是\_CKLogExternal的参数——都已经开始记录这些字眼了，不恰恰说明正在发生“发送信息”这件事吗？进一步，在图10-74中又看到了可疑的字眼“sendMessage:”。

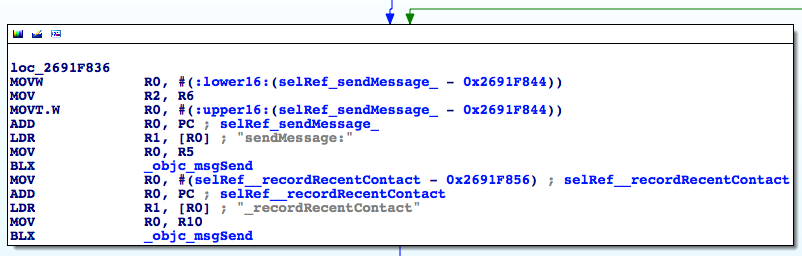


图10- 74 loc\_2691f836

它的调用者和参数是什么？还是直接用IDA把它们给找出来。先看调用者R0，它来自R5。R5又来自哪里呢？往上走，往上走，到loc\_2691F726处，如图10-75所示。

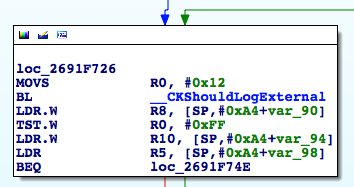


图10- 75 loc\_2691f726

其中，“LDR R5, [SP,#0xA4+var\_98]”决定了R5的值，那[SP,#0xA4+var\_98]是什么呢？还记得在10.2节中是如何处理这类问题的吗？把光标放在var\_98上，然后按下“x”，查看其交叉引用，如图10-76所示。

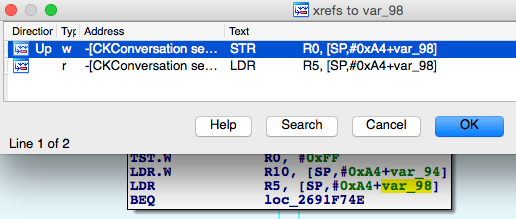


图10- 76 查看交叉引用

双击第一条交叉引用，跳转至“STR R0, [SP,#0xA4+var\_98]”，R0来自[R6 chat]。R6在[CKConversation sendMessage:onService:newComposition:]的开始部分第一次出现，很明显是self，所以“sendMessage:”的调用者是[self chat]。接着回到图10-74中，可看到它的参数R2来自R6。往上拉一点，可以看到R6来自loc\_2691F6F4中的“LDR R6, [SP,#0xA4+var\_80]”，如图10-77所示。

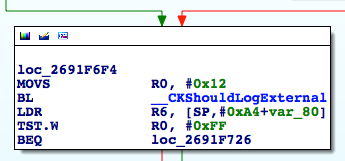


图10- 77 loc\_2691f6f4

接下来该怎么办？我们在1分钟前刚完成了一次类似的操作，这里就不再用文字描述了，只用几张图片（如图10-78至图10-80所示）作为小提示，由读者自己来完成这个操作。

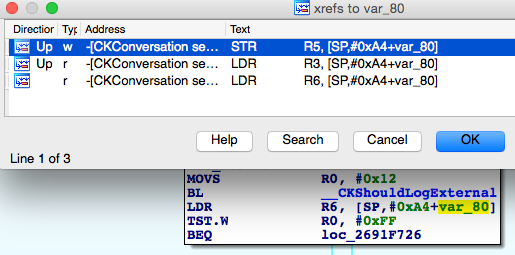


图10- 78 查看交叉引用

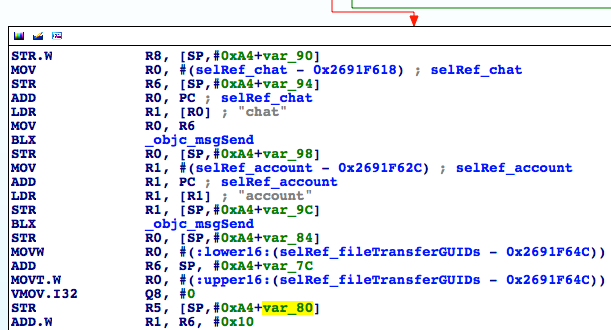


图10- 79 [CKConversation setChat:]

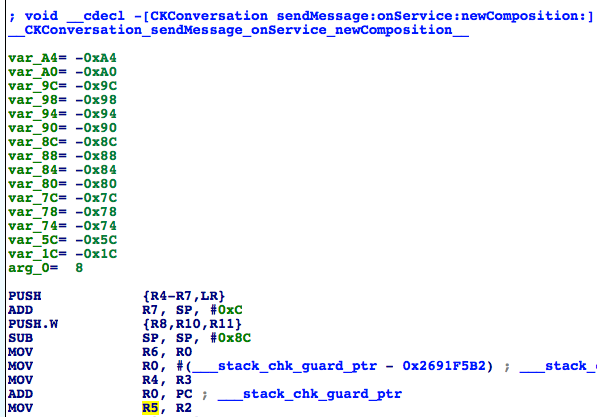


图10- 80 [CKConversation sendMessage:onService:newComposition:]

所以[[self chat] sendMessage:]的参数与[self sendMessage:onService:newComposition:]的第一个参数一脉相承。那么[self chat]是什么类型，参数又是什么类型呢？在上面的静态分析中，我们并没有在IDA中找到什么明显的线索，因此，是时候让LLDB出来热热身了。

先编写一条iMessage，然后在[CKConversation sendMessage:onService:newComposition:]尾部“sendMessage:”下方的那个objc\_msgSend上下个断点，接着按下“Send”，触发断点：

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

IMMessage[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

结果不能再明显了，[IMChat sendMessage:IMMessage]就是我们要找的答案。注意，笔者在打印完所有需要的信息后执行了“ni”命令，然后听到iPhone发出了一个熟悉的“信息已发送”的提示音。这从侧面说明，实际的发送操作正是在[IMChat sendMessage:IMMessage]内完成的。因为IMChat和IMMessage的前缀均为IM，所以它们来自ChatKit以外的库，ChatKit所提供的最底层的发送信息函数到[CKConversation sendMessage:onService:newComposition:]为止。此时，可以肯定，只要能够构造我们自己的IMChat和IMMessage，就可以实现发送iMessage的功能了。那么问题来了，怎么构造这2个类的对象呢？化繁为简，在class-dump的头文件里找找看有没有线索。

要构造IMChat和IMMessage，就先看看它们的头文件里有没有什么明显的构造方法。先打开IMChat.h，看看有没有包含“init”字眼的关键词：

- (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;

上面的代码中参数众多，如何一个个构造它们，我们一点头绪都没有。那接下来该怎么办呢？

还记得我们是如何找到“sendMessage:”调用者的吗？对了，使用[self chat]——self是一个CKConversation对象，看看[CKConversation chat]是怎么来的，不就知道IMChat是如何生成的了吗？在IDA里定位到[CKConversation chat]，如图10-81所示。

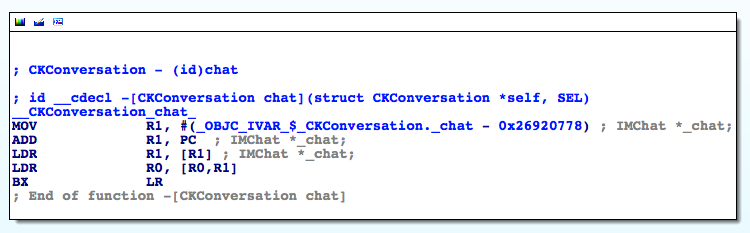


图10- 81 [CKConversation chat]

[CKConversation chat]就是实例变量\_chat的值，这个场景似曾相识啊——还记得大明湖畔的夏雨荷，哦不，是图10-22里的\_composeSendingService吗？此处的想法与操作与那里完全一致，LLDB又要在我们的逆向推导中派上大用场了！删掉已发送的iMessage对话（即删掉这个CKConversation），新建一条iMessage（新建一条CKConversation），然后在[CKConversation setChat:]上下一个断点，点击“Send”，触发断点：

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偏移前的值是0x30acf625 – 0xa1b2000 = 0x2691d625，这个地址位于[CKConversation initWithChat:]中。不过，[CKConversation initWithChat:]的调用者又是谁呢？如法炮制（注意，每次下断点前都要删掉已发送的iMessage对话，再新建一条iMessage，下同）：

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偏移前的值是0x30a8d131 – 0xa1b2000 = 0x268db131，这个地址位于[CKConversationList \_beginTrackingConversationWithChat:]中。继续：

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偏移前的值是0x30a8d4f1 – 0xa1b2000 = 0x268db131，这个地址位于[CKConversationList \_handleRegistryDidRegisterChatNotification:]中，且这里的IMChat对象来自于[notification object]。因为这里的IMChat对象是通过notification传播的，所以我们的下一个目标不是找到[CKConversationList \_handleRegistryDidRegisterChatNotification:]的调用者，而是找到这条notification的发布者，它才是“罪魁祸首”。在这个函数的第一条指令上下一个断点，看看这条notification的结构：

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]}

这条notification的name是“\_\_kIMChatRegistryDidRegisterChatNotification”，object是一个IMChat对象，因此这个IMChat对象一定是在发布（post）这条notification之前就已经生成了。要找出这条notification的发布者，最好的方法，就是执行grep命令搜索一遍系统文件，看看“\_\_kIMChatRegistryDidRegisterChatNotification”的关键字都会在哪些文件里出现，如下：

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

因为它在cache里，所以下面grep一遍decache出的文件：

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.xpc/sdk.dylib: Too many levels of symbolic links

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

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

其实到这里，相信你也能猜到，IMCore与ChatKit都负责与信息相关的操作，但IMCore比ChatKit要更底层，ChatKit接到的命令都会交给IMCore完成，IMCore完成的结果再交给ChatKit展示给用户——MobileSMS是餐馆，ChatKit是服务员，IMCore是厨师，这么比喻，我想就好理解多了吧。

接下来，在IDA中打开IMCore，全文搜索“\_\_kIMChatRegistryDidRegisterChatNotification”，如图10-82所示。

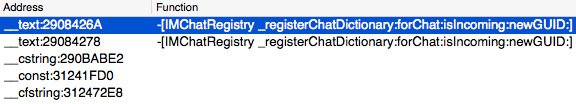


图10- 82 在IDA里搜索“\_\_kIMChatRegistryDidRegisterChatNotification”

很好，我们直接双击第一条搜索结果，看看它的上下文，如图10-83所示。

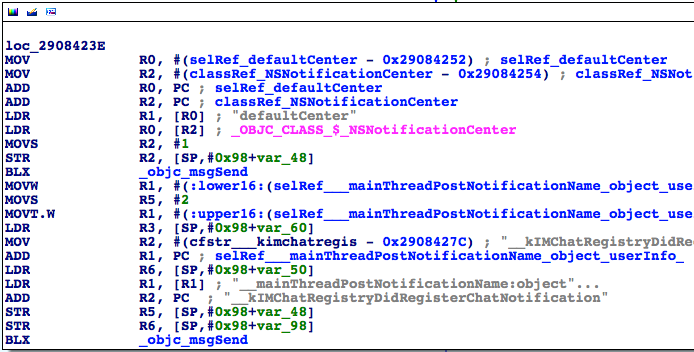


图10- 83 查看搜索结果

看到“PostNotification”字眼，我们就知道了，ChatKit收到的那条notification正是来自于此。IMChat对象是第二个参数，即R3，而R3来自[SP, #0x98+var\_60]。还记得怎么操作吗？还是以提示图（如图10-84与图10-85所示）代替文字，请读者自己来摸索着操作吧。

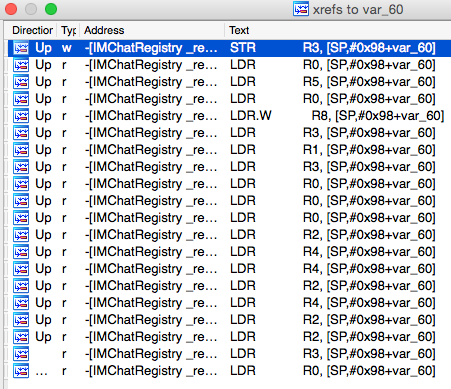


图10- 84 查看交叉引用



图10- 85 [IMChatRegistry \_registerChatDictionary:forChat:isIncoming:newGUID:]

通过上面的分析可知，IMChat对象来自于[IMChatRegistry \_registerChatDictionary:forChat:isIncoming:newGUID:]的第二个参数。这个函数的调用者是：

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偏移前的值是0x3323646f – 0xa1b2000 = 0x2908446F，这个地址位于[IMChatRegistry \_registerChat:isIncoming:guid:]中。继续：

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偏移前的值是0x33237173 – 0xa1b2000 = 0x29085173，这个地址位于[IMChatRegistry chatForIMHandle:]中，且[IMChatRegistry \_registerChat:isIncoming:guid:]的第一个参数，即IMChat对象来自于R5，在[IMChatRegistry chatForIMHandle:]的最后阶段，R5是以返回值的形象出现的。也就是说，[IMChatRegistry chatForIMHandle:]这个函数返回了一个IMChat！且从IMChatRegistry的名字来看，就知道这个类负责注册登记IMChat，一个IMChat对象由此而来，合情合理。但是，解决了1个老问题，带来了2个新问题——IMChatRegistry和chatForIMHandle:的参数从哪里来？饭要一口一口地吃，逆向要一步一步地来。打开IMChatRegistry.h（如图10-86所示），先从IMChatRegistry下手。

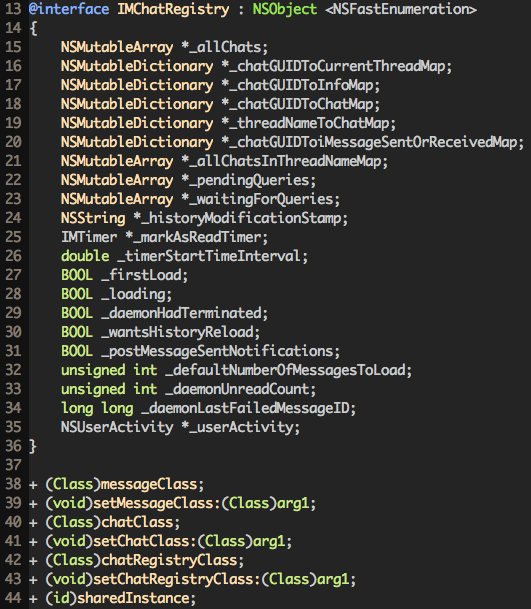


图10- 86 IMChatRegistry.h

其中，第44行的sharedInstance，说明IMChatRegistry是一个单例，通过调用[IMChatRegistry sharedInstance]就可以获取到它的实例。So easy！

那chatForIMHandle:的参数从哪里来？自然是从它的调用者那里来，继续用LLDB追踪吧！

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偏移前的值是0x30a8dca5 – 0xa1b2000 = 0x268dbca5，这个地址已经不再位于IMCore的地址范围内了。刚才也说了，现在正不断地在IMCore和ChatKit间徘徊，正好ChatKit的ASLR偏移也是0xa1b2000，那就去ChatKit看看0x268dbca5在不在它里面，如图10-87所示。

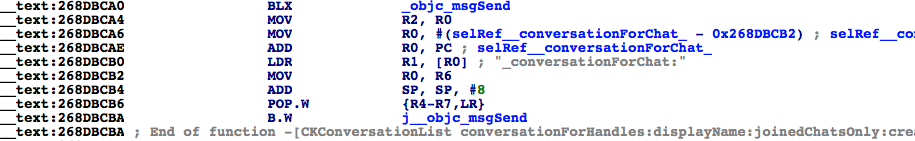


图10- 87 [CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]

0x268dbca5位于[CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]内部，chatForIMHandle:的参数也是来自于[CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]的第一个参数。继续回溯：

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偏移前的值是0x30a84efd – 0xa1b2000 = 0x268d2efd，这个地址位于[CKTranscriptController sendMessage:]中！你！敢！信！吗！我们绕了一大圈，又回到了原点，让人不禁感叹，缘，妙不可言。擦干喜悦的泪珠，来看看，这个IMHandle数组到底是怎么来的，如图10-88所示。

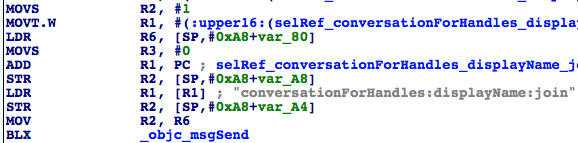


图10- 88 IMHandle数组的来源

R2来自R6，R6来自[SP, #0xA8+var\_80]。我们熟悉的套路又回来了，下面还是只给出提示图（如图10-89和图10-90所示），请读者自行分析。

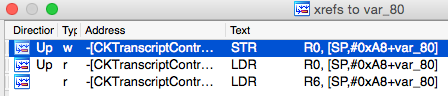


图10- 89 查看交叉引用

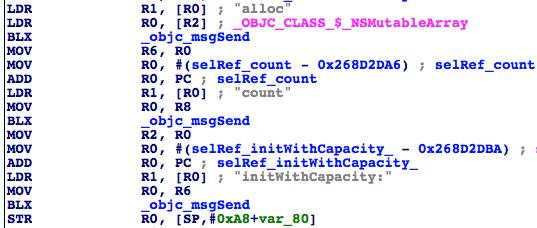


图10- 90 [CKTranscriptController sendMessage:]

你可能也会发现，这里的情况跟前几次不一样了——“STR R0, [SP,#0xA8+var\_80]”貌似只是往[SP, #0xA8+var\_80]里存了一个初始化过的NSMutableArray而已啊！说好的IMHandle呢？嘿嘿，既然是一个NSMutableArray，那么就可能调用addObject:往里面加东西，所以图10-89里中间的那一个“LDR R0, [SP,#0xA8+var\_80]”……双击跳转过去看看，如图10-91所示。

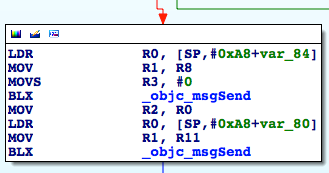


图10- 91 寻找IMHandle

果不其然，它是一个addObject:，而且通过简单观察上下文你就会发现，addObject:的参数来自于imHandleWithID:alreadyCanonical:，并且从这个函数的名字就可以看出来，它返回了一个IMHandle。看来，我们的IMHandle有着落了！在图10-91所示的第一个objc\_msgSend上下一个断点，看看imHandleWithID:alreadyCanonical:的调用者和参数：

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

2个参数都搞定了，第1个就是iMessage地址，第2个是0（即BOOL的NO），那调用者，这个IMAccount对象是哪里来的呢？如图10-91所示，R0来自[SP, #0xA8+var\_84]，所以根据提示图10-92和图10-93可知，IMAccount对象来自[[IMAccountController sharedInstance] \_\_ck\_defaultAccountForService:[CKConversation sendingService]]。

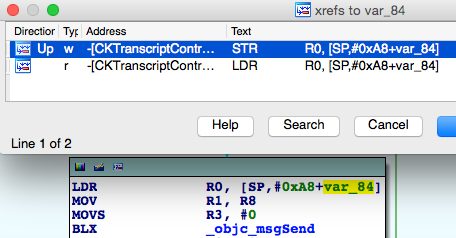


图10- 92 查看交叉引用

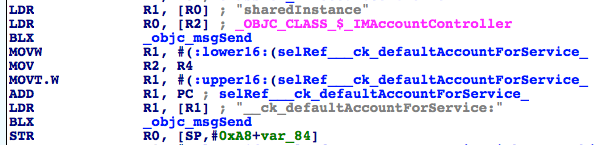


图10- 93 [CKTranscriptController sendMessage:]

趁热打铁，在图10-93的第2个objc\_msgSend上下一个断点，看看[CKConversation sendingService]是什么：

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

可见，它是一个IMServiceImpl对象，那在我们自己的代码中，该如何得到这样一个IMServiceImpl对象呢？其实在10.2节中已经拿到这个类的对象了，打开IMServiceImpl.h，如图10-94。

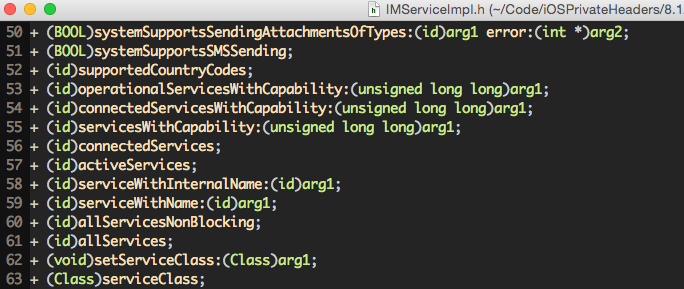


图10- 94 IMServiceImpl.h

其中的[IMServiceImpl iMessageService]就是了。用Cycript再次确认：

cy# [IMServiceImpl iMessageService]

#"IMService[iMessage]"

到此为止，一个可用的IMChat类是如何生成的，被我们完整地逆向了出来。下面在Cycript里验证一下可行性：

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]"

完美！最后的任务，就是构造一个可用的IMMessage对象，这样就可以实现iMessage的发送了，这就行动起来！

打开IMMessage.h，如图10-95所示。

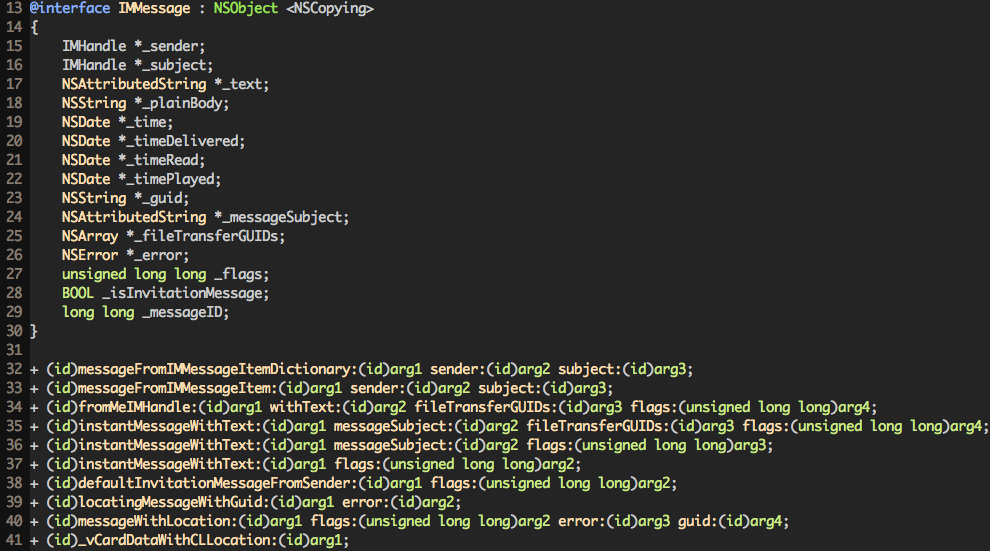


图10- 95 IMMessage.h

又是一堆类方法，其中“instantMessageWithText:flags:”引起了我们的注意，这2个参数应该各传什么？针对第一个“text”传一个NSString进去试试，但第二个“flags”呢？不知道你还有没有印象，在本节的前面，当我们找到[IMChat sendMessage: IMMessage]时，曾在LLDB中打印出了一个IMMessage的结构：

(lldb) po $r2

IMMessage[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)]

这里的“text”无法显示，而“flags”是100005。在Cycript中试试：

cy# [IMMessage instantMessageWithText:@"iOSRE test" flags:100005]

-[\_\_NSCFString string]: unrecognized selector sent to instance 0x1468c140

Cycript告诉我们，NSString不能响应@selector(string)，也就是说，第一个参数并不是一个NSString对象，正确类型的参数应该是可以响应这个@selector(string)的。重新审视图10-95，看看能不能找出一些蛛丝马迹。注意到第17行的“NSAttributedString \*\_text”了吗？查阅苹果官方提供的文档，NSAttributedString确实有一个“- (NSString \*)string”方法，如图10-96所示。

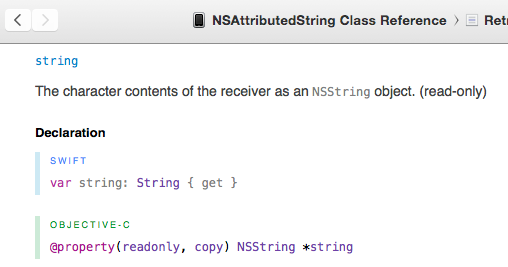


图10- 96 [NSAttributedString string]

重新尝试传一个NSAttributedString对象进去试试：

cy# attributedString = [[NSAttributedString alloc] initWithString:@"iOSRE test"]

#"iOSRE test{\n}"

cy# message = [IMMessage instantMessageWithText:attributedString flags:100005]

#"IMMessage[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]

这里成功构造了一个IMMessage对象。接下来，“这是我生命中美好的时刻，我要完成我最喜欢的测试，在这美丽的月光下在这美丽的Cycript里”：

cy# [chat sendMessage:message]

效果如图10-97所示。打完收工！

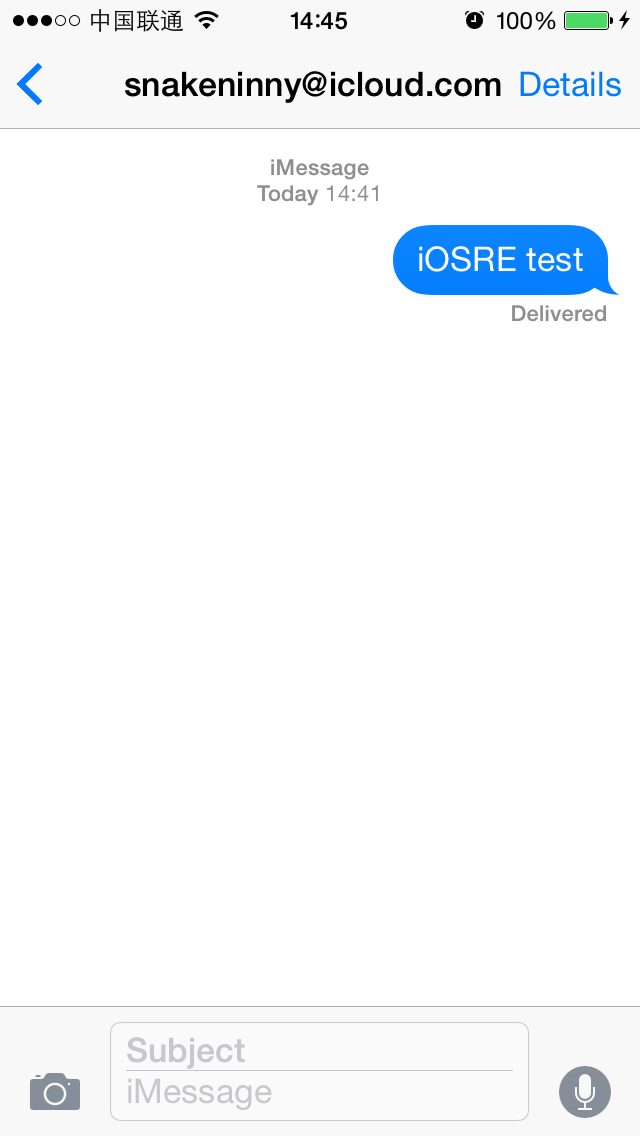


图10- 97 成功发送iMessage

## 10.4 逆向结果整理

相对于前几章的示例来说，本章展示的逆向工程的整体思路虽未变，但复杂程度增大了不少，尤其是与同为系统App的第7章Notes相比，两者的难度相差了若干数量级。为了逆向出看似很简单的iMessage检测和发送操作，大致的思路是这样的：

（1）从表面现象入手

“Text Message”变成“iMessage”，绿色变成蓝色，“Send”按钮，这些表层现象都来自于底层代码。只要我们能描述出所观察到的东西，就可以以此入手，开始逆向分析。本章，就是从信息输入框的占位符和“Send”按钮入手，通过Cycript定位到其实现代码，并切入代码层的。

（2）浏览class-dump出的头文件，找到感兴趣的点

Objective-C头文件结构清晰，函数名的含义明确，可读性强，是寻找蛛丝马迹的理想场所。用Cycript对那些简单的函数、属性、实例变量做测试，有助于我们对类的功能有大体了解。本章，在获取一些重要变量的时候，并没有严谨地借助IDA和LLDB这2样大杀器，而是仅靠阅读头文件，通过函数名猜测参数的类型与用法，配合Cycript进行测试，最终得出了理想的结果。黑猫白猫，抓到老鼠的就是好猫。

（3）在IDA中查看函数是如何形成一个面的

在查看函数内部实现时，IDA无疑是最好用的神器之一。不管是通过交叉引用、地址跳转，还是全局搜索，都可以快速定位关键词，并方便地浏览上下文，对关键词的前因后果有准确的把握。在检测iMessage时，我们用IDA理顺了[CKMessageEntryView updateEntryView]、[CKPendingConversation sendingService]、[CKPendingConversation composeSendingService]和IMChatCalculateServiceForSendingNewCompose等函数的调用关系，其中IMChatCalculateServiceForSendingNewCompose是一个C函数，对class-dump免疫；在发送iMessage时，我们从上层的[CKTranscriptController sendComposition:CKComposition]，一路经过[CKTranscriptController \_startCreatingNewMessageForSending:]、[CKConversation sendMessage:newComposition:]、[CKConversation sendMessage:onService:newComposition:]，追踪到了底层的[IMChat sendMessage:IMMessage]，都是依靠IDA提供的关键词及关联性从一个面中，手工地把那条线给挑出来的。虽然没有机器自动完成方便，但工作量也完全在我们可以接受的范围内，这都要归功于IDA提供的强大分析结果。

（4）用LLDB确认唯一的那条线

LLDB的使用贯穿本章的始终，即使是在有意“克制”的10.3节，我们也在寻找函数调用者、动态查看参数的时候“不得不”惊动LLDB它“老人家”。相对于GDB，LLDB对iOS的支持要好得多，基本不会出现崩溃等Bug，对于Objective-C的支持也很到位，让我们可以专注在调试本身上。在进行iMessage的检测及发送的环节中，用LLDB澄清了大量细节，通过对数据源一环扣一环的分析，基本厘清了iOS发送iMessage的一小段流程。由小见大，从中也可以窥探出苹果的设计思路：MobileSMS是一间邮局，邮局的建筑材料、办公设备和工作人员均来自ChatKit，而充当邮差工作的是IMCore。用户去邮局发一封信，他把信件放在邮筒里，由工作人员整理后交付给邮差，送信的进度和结果再由邮差反馈给工作人员，工作人员再通知用户，这样就完成了整个的服务流程闭环。三者各司其职，为果粉带来良好的用户体验；我们通过逆向工程学习到的这种设计思路，如果能融会贯通，运用到自己的产品设计里去，给产品所带来的优雅度、设计感、健壮性都将是仅仅阅读开发文档所无法企及的。

## 10.5 编写tweak

在使用Cycript完成核心功能的测试后，用Theos编写代码就仅仅是简单的体力劳动了。本节，就用最直观的代码，给MobileSMS中的SMSApplication类添加2个实例函数，分别是“- (int)madridStatusForAddress:(NSString \*)address”和“- (void)sendMadridMessageToAddress:(NSString \*)address withText:(NSString \*)text”，然后用Cycript测试这2个类函数的有效性。开始行动！

### 10.5.1 用Theos新建tweak工程“iOSREMadridMessenger”

新建iOSREMadridMessenger工程的命令如下：

snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl

NIC 2.0 - New Instance Creator

------------------------------

[1.] iphone/application

[2.] iphone/cydget

[3.] iphone/framework

[4.] iphone/library

[5.] iphone/notification\_center\_widget

[6.] iphone/preference\_bundle

[7.] iphone/sbsettingstoggle

[8.] iphone/tool

[9.] iphone/tweak

[10.] iphone/xpc\_service

Choose a Template (required): 9

Project Name (required): 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 构造iOSREMadridMessenger.h

在10.2节的检测以及10.3节的发送中，用到了私有框架IDS、ChatKit和IMCore中的多个私有类和私有函数，我们必须给出它们的定义，才能避免编译器报错或警告。当然，iOSREMadridMessenger.h的内容并不是凭空构造出来的，所有的定义均来自于class-dump出的头文件，我们只是把用到的东西挑选出来，再整合到一个头文件里而已——我们构造的只是一个“精选头文件”。编辑后的iOSREMadridMessenger.h内容如下：

@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 IMMessage : NSObject

+ (instancetype)instantMessageWithText:(NSAttributedString \*)arg1 flags:(unsigned long long)arg2;

@end

@interface IMChat : NSObject

- (void)sendMessage:(IMMessage \*)arg1;

@end

@interface IMChatRegistry : NSObject

+ (instancetype)sharedInstance;

- (IMChat \*)chatForIMHandle:(IMHandle \*)arg1;

@end

### 10.5.3 编辑Tweak.xm

编辑后的Tweak.xm内容如下：

#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];

IMMessage \*message = [IMMessage instantMessageWithText:attributedString flags:100005];

[chat sendMessage:message];

[attributedString release];

}

%end

### 10.5.4 编辑Makefile以及control

编辑后的Makefile内容如下：

THEOS\_DEVICE\_IP = iOSIP

ARCHS = armv7 arm64

TARGET = iphone: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"

编辑后的control内容如下：

Package: com.iosre.iosremadridmessenger

Name: iOSREMadridMessenger

Depends: mobilesubstrate, 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 用Cycript测试

将写好的tweak编译打包安装到iOS后， 执行ssh命令连接到iOS中，然后执行如下代码：

FunMaker-5:~ root# cycript -p MobileSMS

cy# [UIApp madridStatusForAddress:@"snakeninny@icloud.com"]

1

cy# [UIApp sendMadridMessageToAddress:@"snakeninny@icloud.com" withText:@"Sent from iOSREMadridMessenger"]

“[snakeninny@icloud.com](mailto:snakeninny@icloud.com)”的检测结果是1，支持iMessage，且此条iMessage被成功发送，如图10-98所示。



图10- 98 成功发送iMessage

如果你按照上面的思路和方法成功搞定了iMessage的检测和发送，就给“[snakeninny@gmail.com](mailto:snakeninny@gmail.com)”发一条iMessage吧！

## 10.6 小结

作为苹果在iOS 5之后重点打造的核心服务之一，iMessage的功能在iOS 8中得到了大幅度增强，不管是单纯的文字，还是多媒体照片、语音，甚至是视频，iMessage都能完美地hold住。本章的iMessage检测与发送虽然仅仅是所有iMessage操作的冰山一角，但都已经要在IDS、ChatKit和IMCore这3个模块间来回切换了，可见整个iMessage系统的复杂度之高。从上面的分析过程中可知道，负责管理iMessage发件人的是IMAccountController，作为发送人的我们是一个个IMAccount；收件人是一个IMHandle；一条对话就是一个IMChat或者CKConversation；IMChatRegistry管理所有的对话；一条iMessage就是一个IMMessage或者CKComposition。对于那些有意涉足即时通信的开发者来说，iMessage的这些设计方式非常值得借鉴。如果你对iMessage很感兴趣，觉得本章的内容仍意犹未尽，不妨尝试分析笔者留下的3个“隐藏关卡”，它们由易到难，运用本章的所用到的逆向思路和技巧，就可以各个击破：

* 搞定SMS的发送（提示：只要更换IMServiceImpl对象即可）；
* 用ChatKit类搞定iMessage发送（提示：CKConversation对象可以由IMChat对象生成）；
* 把发送iMessage的操作移植到SpringBoard进程中（提示：在SpringBoard中调用[IMChat sendMessage:IMMessage]之所以无效，是因为SpringBoard缺少某种“capabilities”）。

如果本章的内容你能完全吃透，并“脱稿”完成，那么恭喜你，你已经是一名优秀的iOS逆向工程师了，可以朝着更高的目标（比如越狱？）迈进了。在开始新的征程前，先来我们的论坛<http://bbs.iosre.com>，与各位同好分享这份喜悦吧！