# Chapter 10 Practice 4: Detect And Send iMessages

## 10.1 iMessage

iMessage is an IM service that Apple implements seamlessly into the stock Messages App (hereafter referrered to as MobileSMS). It was born in iOS 5 and became better ever since. Whether it’s plain text, image, audio, or even video, iMessage can handle them with high speed, security and efficiency. We all love iMessage!

Among all functions of iMessage, detecting if an address supports iMessage, and sending an iMessage are 2 most interesting functions without doubt. Surprisingly, there are even companies that make profit from sending spam iMessages, and that’s one of the main reasons that I developed the Cydia tweak “SMSNinja”. You can’t understand how to defense without knowing how to attack. In this chapter, we will combine all knowledge points we’ve studied by far and start from scratch to reverse the functions of detecting and sending iMessages, as sublimation of the book. All the following operations are finished on iPhone 5, iOS 8.1.

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

As usual, before using tools to start reverse engineering, let’s analyze the abstract target and concretize it, then form the idea and carry it out.

### 10.2.1 Observe MobileSMS and look for cut-in points

As MobileSMS users, we will notice that during the process of sending a message, Apple will show us if we’re currently sending an SMS or iMessage through the changes of texts and colors, say:

1. 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. When you start to input message body, if the address only supports SMS, the “Send” button beside the input box will be green; if it supports iMessage, the button will be blue.
2. When you hit the “Send” button to send this message, if this is an SMS, the message bubble will be green, otherwise it will be blue.

These 3 phenomena will appear one after another. Since the process of detecting iMessage has already happened in the 1st phenomenon, it is enough to act as the cut-in point. We’ll focus on phenomenon A from now on.

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

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

Placeholder doesn’t come from nowhere but its data source either. The reason why placeholder changes is that its data source (data source’s data source, and so on. Hereafter referred to as the Nth data source) changes, like the following pseudo code presents:

id dataSource = ?;

id a = function(dataSource);

id b = function(a);

id c = function(b);

…

id z = function(y);

NSString \*placeholder = function(z);

From the above snippet we can know that the original data source is dataSource, its change in turn results in the change of placeholder. Well, what’s the original data source? In 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 dataSource to placeholder, and this conversion process is the actual meaning of “detecting iMessages” as well our target in this section, as shown in figure 10-2.

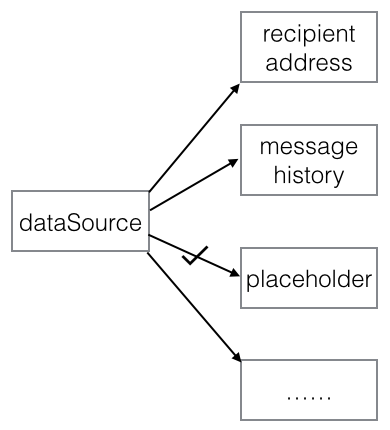


Figure 10- 2 Conversion from dataSource to placeholder

You may wonder, since figure 10-2 is so straightforward and dataSource is already known, why don’t we start from it directly and trace placeholder? Then we can reproduce the process and achieve our goal. Actually, we’re not living in a fairy tale, the real world is usually not idealized. For one thing, we don’t have the source code of MobileSMS; for the other thing, in general cases, the conversion is much more complex, as can be illustrated in figure 10-3.

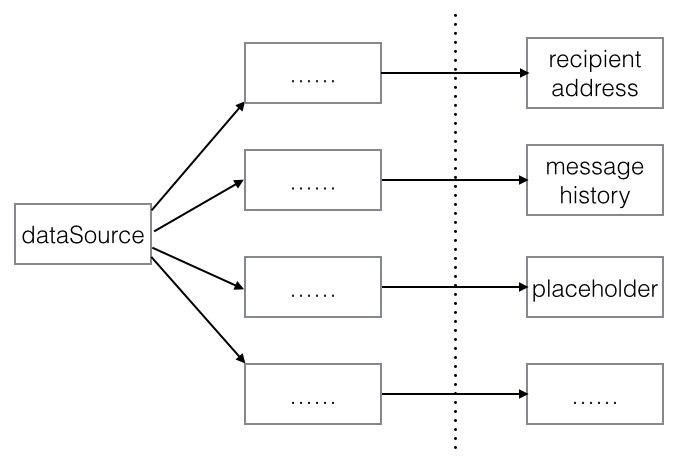


Figure 10- 3 Real conversion from dataSource to placeholder

dataSource must be converted multiple times to become placeholder, their relationship is very intricate. If we start from dataSource, how can we know which of the 4 routines leads to placeholder? Under such circumstance, because there is only one placeholder, it’s more efficient and doable to start from placeholder and trace back to dataSource to reproduce the whole process.

In conclusion, the ideas of this practice are: first use Cycript to locate placeholder, then trace the Nth data source of placeholder using IDA and LLDB, until we get dataSource. Finally reproduce the process of how dataSource becomes placeholder. Looks as easy as a regular 3-step job? Actions not only speak louder than words, but also implement harder than words, you’ll feel it soon.

### 10.2.2 Find placeholder using Cycript

Open MobileSMS and create a new message; enter “bbs.iosre.com” as the address and then tap “return” on keyboard to end editing, as shown in figure 10-4.

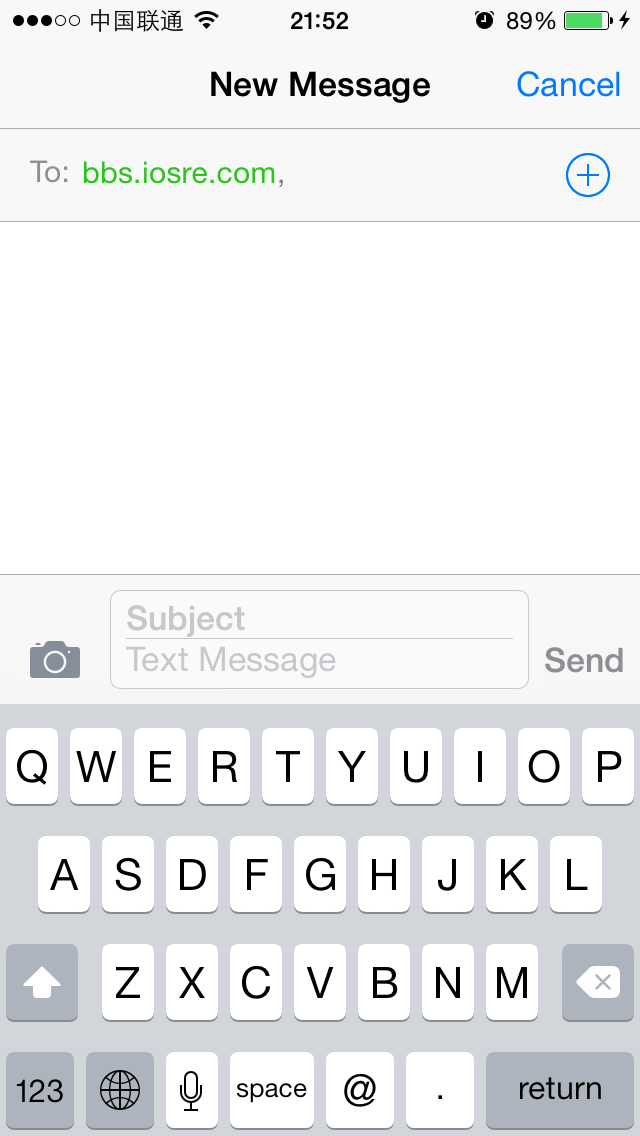


Figure 10- 4 Create a new message

Since we are using 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]

The view hierarchy of keyWindow is quite rich in content, so we’re not pasting it here. If you search “Text Message” in the output, you will find that there isn't any match. Why? Maybe you’ve already guessed the answer: “Text Message” isn’t in keyWindow. For verification, let’s see how many windows are there in the current view:

cy# [UIApp windows]

@[#"<UIWindow: 0x1575ca10; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x15629c60>; layer = <UIWindowLayer: 0x156e36f0>>",#"<UITextEffectsWindow: 0x1579ab70; frame = (0 0; 320 568); opaque = NO; autoresize = W+H; gestureRecognizers = <NSArray: 0x1579b300>; layer = <UIWindowLayer: 0x1579adf0>>",#"<CKJoystickWindow: 0x1552bf90; baseClass = UIAutoRotatingWindow; frame = (0 0; 320 568); hidden = YES; gestureRecognizers = <NSArray: 0x1552b730>; layer = <UIWindowLayer: 0x1552bdc0>>",#"<UITextEffectsWindow: 0x1683a2e0; frame = (0 0; 320 568); hidden = YES; gestureRecognizers = <NSArray: 0x1688b9e0>; layer = <UIWindowLayer: 0x168b9ad0>>"]

As we can see, each item starting with “#” is a window, there are 4 of them, and the 1st is keyWindow. Well, which one contains “Text Message”? As the names suggest, the 2nd and 4th windows with the keyword “Text” in their names may be our targets. However, the 4th window is even invisible because of its hidden property. This leaves us with the 2nd window only, let’s test it out in Cycript.

cy# [#0x1579ab70 setHidden:YES]

After this command, not only the input box but also the whole keyboard are hidden, as shown in figure 10-5:



Figure 10- 5 The bottom half is hidden

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

There are 3 subviews in the above code, which one does “Text Message” reside? Let’s test them one by one.

cy# [#0x16827620 setHidden:YES]

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

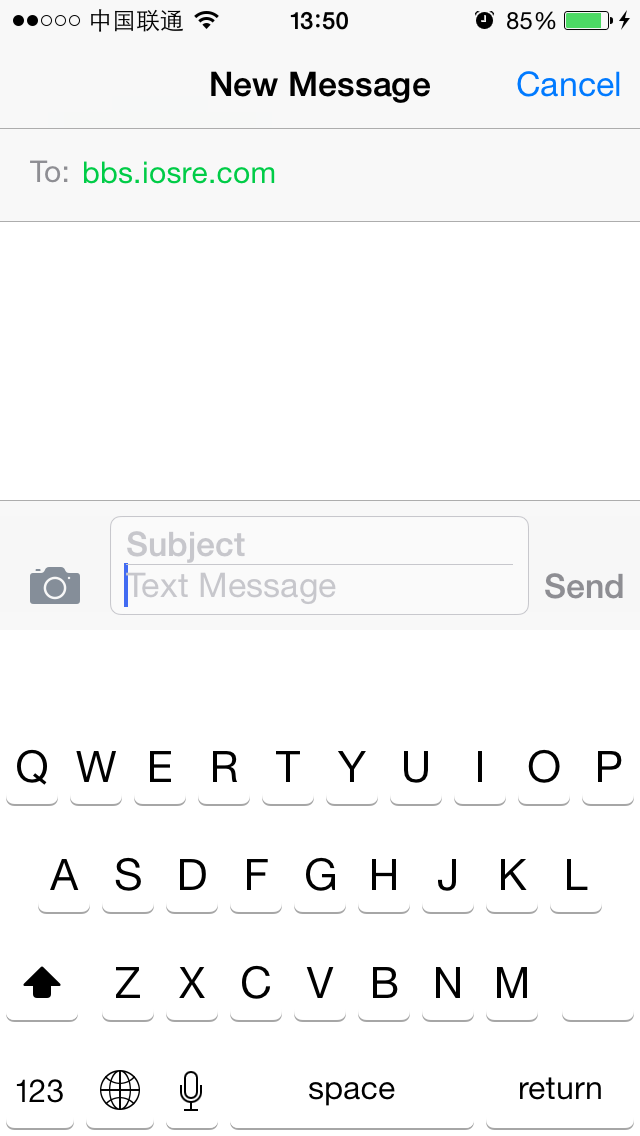


Figure 10- 6 Keyboard background is hidden

cy# [#0x16827620 setHidden:NO]

cy# [#0x157b88d0 setHidden:YES]

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

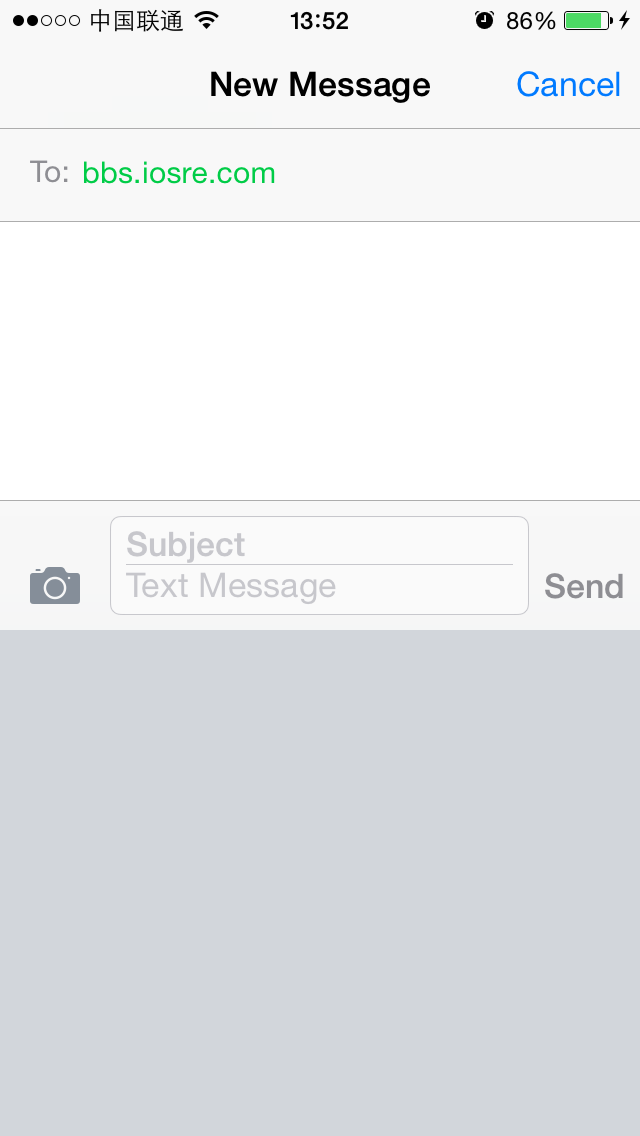


Figure 10- 7 Keyboard is hidden

OK, this view is keyboard itself. Thus, we can infer that UIKBInputBackdropView and UIKBCompatInputView work together to form a keyboard’s view. This official design mode can be a good reference for 3rd-party keyboard developers and theme makers.

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

cy# [#0x157b88d0 setHidden:NO]

cy# [#0x1682ca50 setHidden:YES]

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

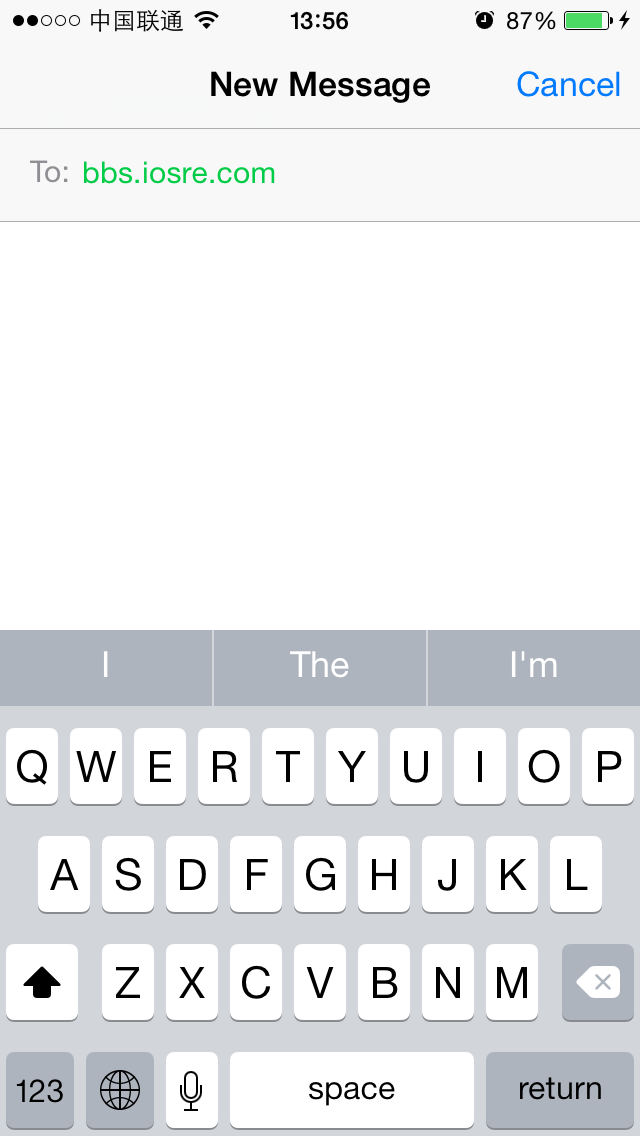


Figure 10- 8 Message entry view is hidden

According to the result, we know that “Text Message” is inside CKMessageEntryView. Go on.

cy# [#0x1682ca50 setHidden:NO]

cy# [#0x1682ca50 subviews]

@[#"<\_UIBackdropView: 0x168ce210; frame = (0 0; 320 65); opaque = NO; autoresize = W+H; userInteractionEnabled = NO; layer = <\_UIBackdropViewLayer: 0x168f5300>>",#"<UIView: 0x168d2b70; frame = (0 0; 320 0.5); layer = <CALayer: 0x168d2be0>>",#"<UIButton: 0x1684b240; frame = (266 27; 53 33); opaque = NO; layer = <CALayer: 0x168d64b0>>",#"<UIButton: 0x168b88b0; frame = (266 30; 53 26); hidden = YES; opaque = NO; gestureRecognizers = <NSArray: 0x16840030>; layer = <CALayer: 0x16858420>>",#"<UIButton: 0x16833ac0; frame = (15 33.5; 25 18.5); opaque = NO; gestureRecognizers = <NSArray: 0x1682d9b0>; layer = <CALayer: 0x16838780>>",#"<\_UITextFieldRoundedRectBackgroundViewNeue: 0x168fba00; frame = (55 8; 209.5 49.5); opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x1682da50>>",#"<UIView: 0x168dcf10; frame = (55 8; 209.5 49.5); clipsToBounds = YES; opaque = NO; layer = <CALayer: 0x168e4170>>",#"<CKMessageEntryWaveformView: 0x1571b710; frame = (15 25.5; 251 35); alpha = 0; opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x1578fc90>>"]

Again, let’s hide these views one by one to locate “Text Message”, and I’ll leave the work to you as an exercise. After locating “UIView: 0x168dcf10” (Notice, it’s the 2nd UIView object) as the target, let’s continue with its subviews.

cy# [#0x168dcf10 subviews]

@[#"<CKMessageEntryContentView: 0x16389000; baseClass = UIScrollView; frame = (3 -4; 203.5 57.5); clipsToBounds = YES; opaque = NO; gestureRecognizers = <NSArray: 0x168f0730>; layer = <CALayer: 0x168e41a0>; contentOffset: {0, 0}; contentSize: {203.5, 57}>"]

There is only one subview, keep digging.

cy# [#0x16389000 subviews]

@[#"<CKMessageEntryRichTextView: 0x16295200; baseClass = UITextView; frame = (0 20.5; 203.5 36.5); text = ''; clipsToBounds = YES; opaque = NO; gestureRecognizers = <NSArray: 0x168f5a60>; layer = <CALayer: 0x168f59c0>; contentOffset: {0, 0}; contentSize: {203.5, 36.5}>",#"<CKMessageEntryTextView: 0x15ad2a00; baseClass = UITextView; frame = (0 0; 203.5 36.5); 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>>"]

By hiding these views one by one, we can find that when executing “[#0x16295200 setHidden:YES]”, only “Text Message” is hidden, other control objects are not affected, as shown in figure 10-9.

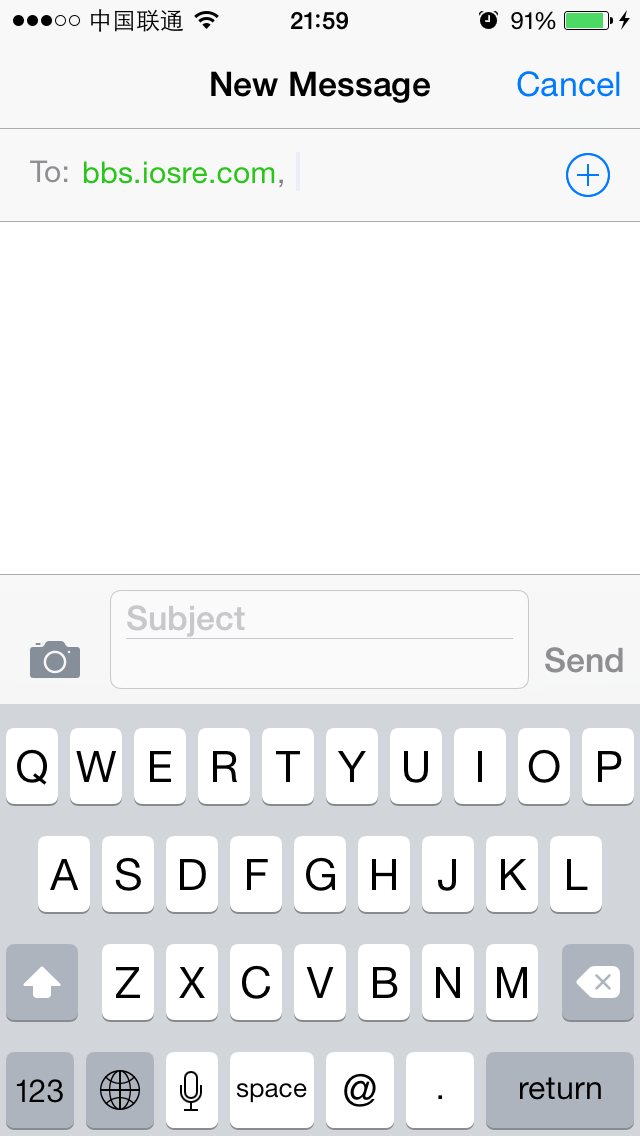


Figure 10- 9 placeholder is hidden

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

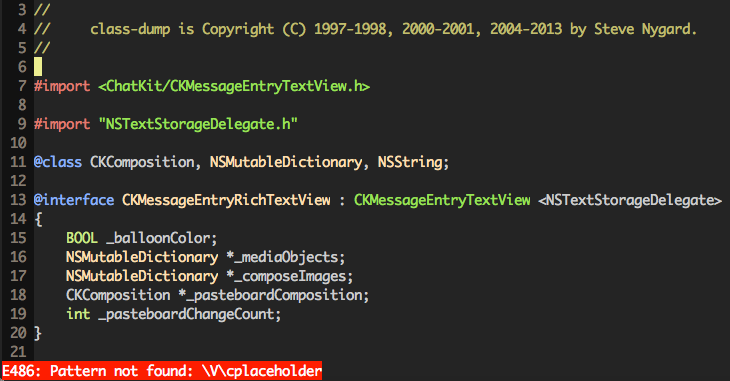


Figure 10- 10 CKMessageEntryRichTextView.h

Unluckily, we cannot find placeholder in CKMessageEntryRichTextView.h. Was there something wrong in our deduction? Not really. Let’s have a look at its superclass, i.e. CKMessageEntryTextView, as shown in figure 10-11.

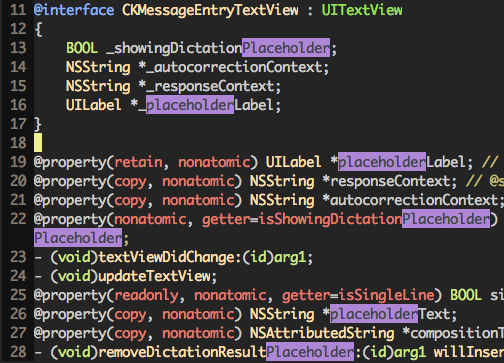


Figure 10- 11 CKMessageEntryTextView.h

Aha, there are lots of placeholders in this file. Among them placeholderLabel and placeholderText are quite noticeable, is anyone of them our target placeholder? Let’s verify with Cycript:

cy# [#0x16295200 setPlaceholderText:@"iOSRE"]

Now, the view looks like figure 10-12.

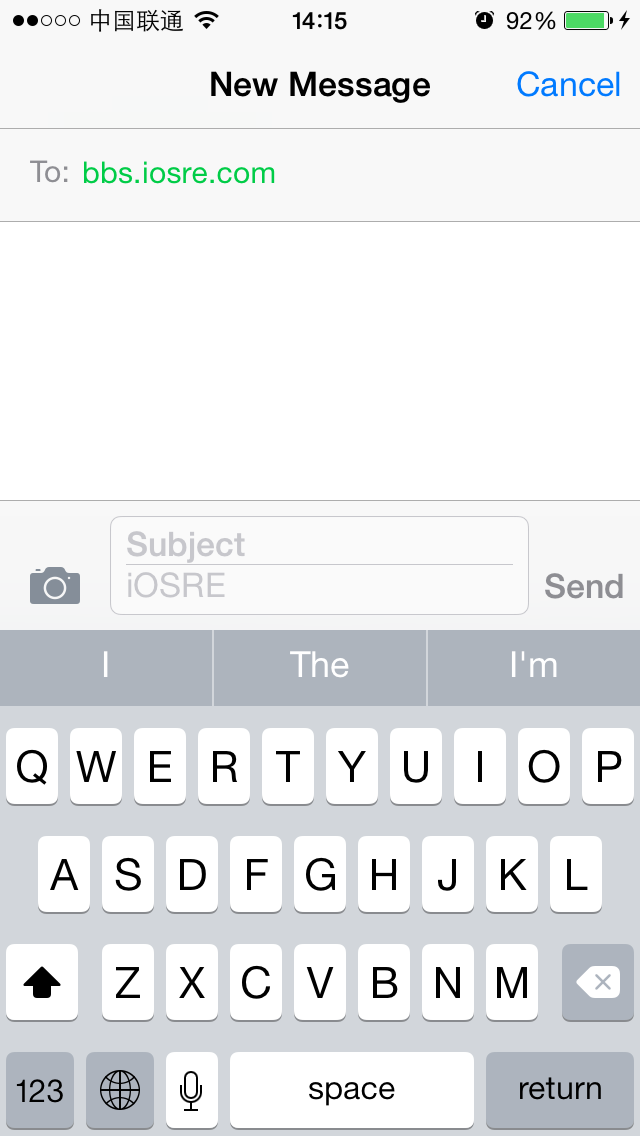


Figure 10- 12 Change placeholder to “iOSRE”

Great! placeholderText is exactly the placeholder we’re looking for. To avoid confusion, hereafter we will refer to placeholder as placeholderText. So far, we have taken the first step in a long march. Well done!

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

placeholderText is a property. To modify a property, our first reaction is to use its setter. We have already changed placeholderText from “Text Message” to “iOSRE” by calling setPlaceholderText:, does MobileSMS also call this setter to change placeholderText? To verify our guesses, we need the help of IDA and LLDB.

Since CKMessageEntryTextView comes from ChatKit, our next focus should turn to framework ChatKit in process MobileSMS, can you get it? OK, drag and drop ChatKit into IDA. After the initial analysis, locate to [CKMessageEntryTextView setPlaceholderText:], as shown in figure 10-13.

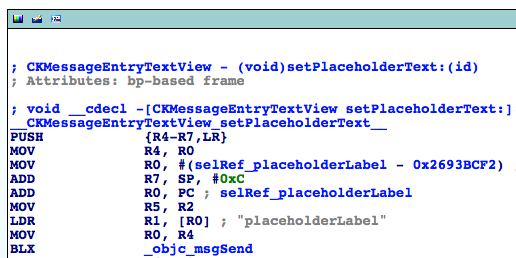


Figure 10- 13 [CKMessageEntryTextView setPlaceholderText:]

Attach LLDB to MobileSMS and continue the process as follows:

(lldb) process connect connect://iOSIP:1234

Process 200596 stopped

\* thread #1: tid = 0x30f94, 0x316554f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20, queue = 'com.apple.main-thread, stop reason = signal SIGSTOP

frame #0: 0x316554f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20

libsystem\_kernel.dylib`mach\_msg\_trap + 20:

-> 0x316554f0: pop {r4, r5, r6, r8}

0x316554f4: bx lr

libsystem\_kernel.dylib`mach\_msg\_overwrite\_trap:

0x316554f8: mov r12, sp

0x316554fc: push {r4, r5, r6, r8}

(lldb) c

Process 200596 resuming

Then check the ASLR offset of ChatKit as follows:

(lldb) image list -o -f

[ 0] 0x00079000 /private/var/db/stash/\_.29LMeZ/Applications/MobileSMS.app/MobileSMS(0x000000000007d000)

[ 1] 0x0019c000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x000000000019c000)

[ 2] 0x01eac000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/Frameworks/Foundation.framework/Foundation

……

[ 9] 0x01eac000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PrivateFrameworks/ChatKit.framework/ChatKit

The ASLR offset is 0x1eac000. With this offset, we can set a breakpoint on [CKMessageEntryTextView setPlaceholderText:] to check whether it is called or not, and if it’s called, who’s the caller. The base address of this method is shown in figure 10-14, as we can see, it’s 0x2693BCE0.

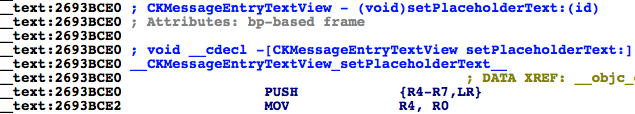


Figure 10- 14 [CKMessageEntryTextView setPlaceholderText:]

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

(lldb) br s -a 0x287E7CE0

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

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

This command is very straightforward; when the breakpoint gets triggered, LLDB prints the Objective-C description of R2, i.e. the only argument of setPlaceholderText:, then prints LR in hexadecimal, i.e. the return address of [CKMessageEntryTextView setPlaceholderText:]. If R2 is “iMessage”, it indicates that the argument is the 1st data source. Meanwhile, since LR is inside the caller, we can trace the 2nd data source from inside the caller. Clear the address entry and enter “[snakeninny@gmail.com](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.

As we can see, when placeholderText turns to “iMessage”, LR’s value is 0x28768b33. 0x28768b33 - 0x1eac000 = 0x268BCB33, let’s jump to this address, as shown in figure 10-15.

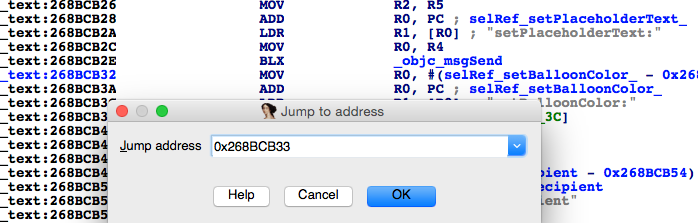


Figure 10- 15 Jump to 0x268BCB33

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

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

I don’t know if you've noticed that placeholderText was blank during address editing. Not until we've pressed "return" on the keyword that the placeholderText became “Text Message” or “iMessage”. In other words, iOS will not detect whether current address supports iMessage until editing is over; from the perspective of energy saving, this makes sense. Based on this design, we can firstly edit the recipient's address, then set a breakpoint and at last press "return" to finish editing. If the breakpoint gets triggered under such circumstance, we can say that MobileSMS is stopped during the process of detecting iMessage. Now, let’s search upward from figure 10-15 to see who is the caller of [CKMessageEntryTextView setPlaceholderText:], as shown in figure 10-16.

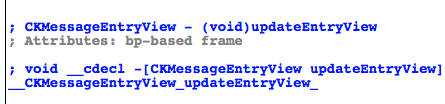


Figure 10- 16 Caller of [CKMessageEntryTextView setPlaceholderText:]

Set placeholder text when updating entry view, this is rather reasonable. However, without any argument, how does [CKMessageEntryView updateEntryView] know whether it should set placeholderText to "Text Message" or "iMessage"? Judging from this, we can say that [CKMessageEntryView updateEntryView] must have conducted some internal judges to get the conclusion that the address supports iMessage, hence changed the 2nd data source. Let’s get back to IDA to see where the 2nd data source comes from, as shown in figure 10-17.

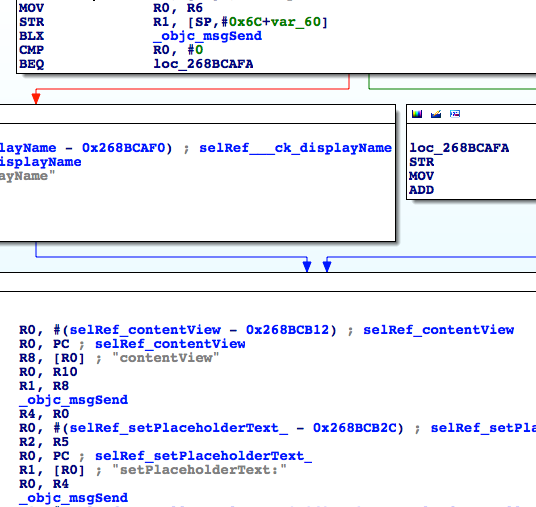


Figure 10- 17 Look for the 2nd data source

R2 is the argument of setPlaceholderText:, which is also the 1st data source. And R2 comes from R5, therefore R5 is the 2nd data source. Where does R5 come from? There is a branch here, so let’s take a look at its condition, as shown in figure 10-18.

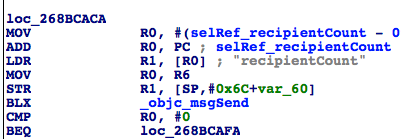


Figure 10- 18 Branch condition

We can see that the branch condition is "[$r0 recipientCount] == 0". The meaning of "recipient" is very obvious that it represents the receiver of message. When the recipient count is 0, namely there's no recipient, MobileSMS will branch right, otherwise left. In the current case, because there is already one recipient, MobileSMS will probably branch left. It’s very simple to verify our assumption: input "snakeninny@gmail.com" in the address entry, then set a breakpoint on any instruction in the right branch and at last press "return" to finish editing. We can see that the breakpoint is not triggered; as a result, we can confirm that R5 comes from [$r8 \_\_ck\_displayName] in the left branch. In other words, [$r8 \_\_ck\_displayName] is the 3rd data source. Where does R8 come from? Scroll up in IDA, we can find that R8 is from [[self conversation] sendingService] at the beginning of [CKMessageEntryView updateEntryView], as shown in figure 10-19.

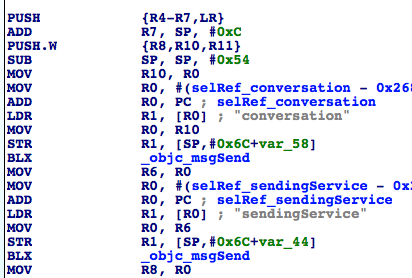


Figure 10- 19 Look for 4th data source

Therefore, [[self conversation] sendingService] is the 4th data source. Let's verify our analysis so far with LLDB: input "snakeninny@gmail.com" in the address entry, then set a breakpoint on "MOV R8, R0" in figure 10-19 and at last press "return" to finish editing. Execute "po [$r0 \_\_ck\_displayName]" when the breakpoint gets triggered and then see whether LLDB outputs “iMessage”:

(lldb) br s -a 0x28768962

Breakpoint 14: where = ChatKit`-[CKMessageEntryView updateEntryView] + 54, address = 0x28768962

(lldb) br com add 14

Enter your debugger command(s). Type 'DONE' to end.

> po [$r0 \_\_ck\_displayName]

> c

> DONE

Text Message

Process 200596 resuming

Command #2 'c' continued the target.

iMessage

Process 200596 resuming

Command #2 'c' continued the target.

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

Reinput the address and set 2 breakpoints on the first 2 objc\_msgSends in [CKMessageEntryView updateEntryView] respectively. Then press "return" to trigger the breakpoints:

Process 14235 stopped

\* thread #1: tid = 0x379b, 0x2b528948 ChatKit`-[CKMessageEntryView updateEntryView] + 28, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x2b528948 ChatKit`-[CKMessageEntryView updateEntryView] + 28

ChatKit`-[CKMessageEntryView updateEntryView] + 28:

-> 0x2b528948: blx 0x2b5f5f44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

0x2b52894c: mov r6, r0

0x2b52894e: movw r0, #51162

0x2b528952: movt r0, #2547

(lldb) p (char \*)$r1

(char \*) $6 = 0x2b60cc16 "conversation"

(lldb) ni

Process 14235 stopped

\* thread #1: tid = 0x379b, 0x2b52894c ChatKit`-[CKMessageEntryView updateEntryView] + 32, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b52894c ChatKit`-[CKMessageEntryView updateEntryView] + 32

ChatKit`-[CKMessageEntryView updateEntryView] + 32:

-> 0x2b52894c: mov r6, r0

0x2b52894e: movw r0, #51162

0x2b528952: movt r0, #2547

0x2b528956: add r0, pc

(lldb) po $r0

CKPendingConversation<0x1587e870>{identifier:'(null)' guid:'(null)'}(null)

The return value of [self conversation] is a CKPendingConversation object. OK, now look at the next one:

(lldb) c

Process 14235 resuming

Process 14235 stopped

\* thread #1: tid = 0x379b, 0x2b52895e ChatKit`-[CKMessageEntryView updateEntryView] + 50, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x2b52895e ChatKit`-[CKMessageEntryView updateEntryView] + 50

ChatKit`-[CKMessageEntryView updateEntryView] + 50:

-> 0x2b52895e: blx 0x2b5f5f44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

0x2b528962: mov r8, r0

0x2b528964: movw r0, #52792

0x2b528968: movt r0, #2547

(lldb) p (char \*)$r1

(char \*) $8 = 0x2b6105e1 "sendingService"

(lldb) ni

Process 14235 stopped

\* thread #1: tid = 0x379b, 0x2b528962 ChatKit`-[CKMessageEntryView updateEntryView] + 54, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b528962 ChatKit`-[CKMessageEntryView updateEntryView] + 54

ChatKit`-[CKMessageEntryView updateEntryView] + 54:

-> 0x2b528962: mov r8, r0

0x2b528964: movw r0, #52792

0x2b528968: movt r0, #2547

0x2b52896c: add r0, pc

(lldb) po $r0

IMService[SMS]

(lldb) po [$r0 class]

IMServiceImpl

Obviously, the return value of [CKPendingConversation sendingService] is IMService[SMS] (the value becomes IMService[iMessage] when this breakpoint gets triggered the 2nd time), whose type is IMSerciceImpl. Therefore, the 4th data source is [CKPendingConversation sendingService]. Can you still follow?

Till now, we have already got a lot of useful information. So let’s turn back to IDA, locate [CKPendingConversation sendingService] and find out how it works internally, as shown in figure 10-20.

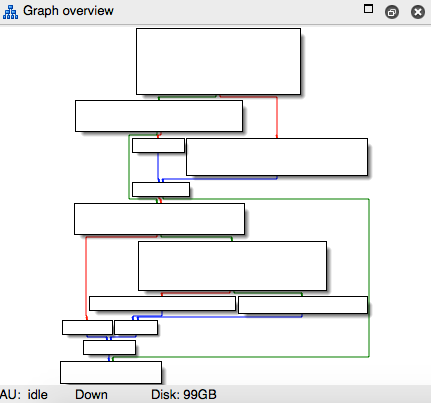


Figure 10- 20 [CKPendingConversation sendingService]

The implementation logic is not too complicated. But there are several branches so that we can’t make sure which one MobileSMS actually goes. Debug again with LLDB and pay attention to every branch condition as well as the address of the next instruction.

Process 14235 stopped

\* thread #1: tid = 0x379b, 0x2b5f0264 ChatKit`-[CKPendingConversation sendingService], queue = 'com.apple.main-thread, stop reason = breakpoint 3.1

frame #0: 0x2b5f0264 ChatKit`-[CKPendingConversation sendingService]

ChatKit`-[CKPendingConversation sendingService]:

-> 0x2b5f0264: push {r4, r5, r7, lr}

0x2b5f0266: add r7, sp, #8

0x2b5f0268: sub sp, #8

0x2b5f026a: mov r4, r0

(lldb) ni

Process 14235 stopped

……

\* thread #1: tid = 0x379b, 0x2b5f027e ChatKit`-[CKPendingConversation sendingService] + 26, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b5f027e ChatKit`-[CKPendingConversation sendingService] + 26

ChatKit`-[CKPendingConversation sendingService] + 26:

-> 0x2b5f027e: cbz r0, 0x2b5f02a4 ; -[CKPendingConversation sendingService] + 64

0x2b5f0280: movw r0, #38082

0x2b5f0284: movt r0, #2535

0x2b5f0288: str r4, [sp]

(lldb) p $r0

(unsigned int) $11 = 0

(lldb) ni

Process 14235 stopped

……

\* thread #1: tid = 0x379b, 0x2b5f02b8 ChatKit`-[CKPendingConversation sendingService] + 84, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b5f02b8 ChatKit`-[CKPendingConversation sendingService] + 84

ChatKit`-[CKPendingConversation sendingService] + 84:

-> 0x2b5f02b8: cbz r0, 0x2b5f02c4 ; -[CKPendingConversation sendingService] + 96

0x2b5f02ba: mov r0, r4

0x2b5f02bc: mov r1, r5

0x2b5f02be: blx 0x2b5f5f44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

(lldb) p $r0

(unsigned int) $12 = 341691792

(lldb) ni

Process 14235 stopped

……

\* thread #1: tid = 0x379b, 0x2b5f02c2 ChatKit`-[CKPendingConversation sendingService] + 94, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b5f02c2 ChatKit`-[CKPendingConversation sendingService] + 94

ChatKit`-[CKPendingConversation sendingService] + 94:

-> 0x2b5f02c2: cbnz r0, 0x2b5f032c ; -[CKPendingConversation sendingService] + 200

0x2b5f02c4: movw r0, #35464

0x2b5f02c8: movt r0, #2535

0x2b5f02cc: add r0, pc

(lldb) p $r0

(unsigned int) $13 = 341691792

(lldb) ni

Process 14235 stopped

……

\* thread #1: tid = 0x379b, 0x2b5f032e ChatKit`-[CKPendingConversation sendingService] + 202, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2b5f032e ChatKit`-[CKPendingConversation sendingService] + 202

ChatKit`-[CKPendingConversation sendingService] + 202:

-> 0x2b5f032e: pop {r4, r5, r7, pc}

ChatKit`-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]:

0x2b5f0330: push {r4, r5, r6, r7, lr}

0x2b5f0332: add r7, sp, #12

0x2b5f0334: push.w {r8, r10, r11}

The execution flow of MobileSMS is very evident now. There are 3 conditional branches, which are CBZ, CBZ and CBNZ respectively. At each time, the value of R0 is 0, 341691792 and 341691792 respectively. As a result, we can know that the execution flow is shown in figure 10-21.

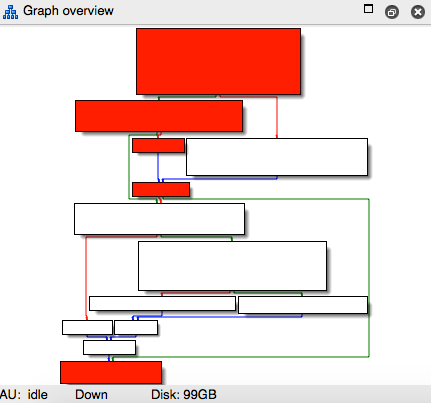


Figure 10- 21 Execution flow

So the value of [CKPendingConversation sendingService] actually comes from [CKPendingConversation composeSendingService], which is the 5th data source, right? OK, let's proceed to the new method in IDA, as shown in figure 10-22.

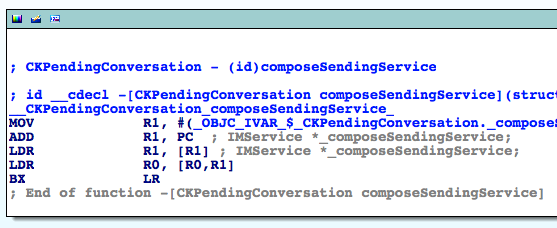


Figure 10- 22 [CKPendingConversation composeSendingService]

Obviously, [CKPendingConversation composeSendingService] merely returns the value of instance variable \_composeSendingService. In other words, \_composeSendingService is the 6th data source. In that case, we just need to find where this instance variable is written and there comes the 7th data sources.

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

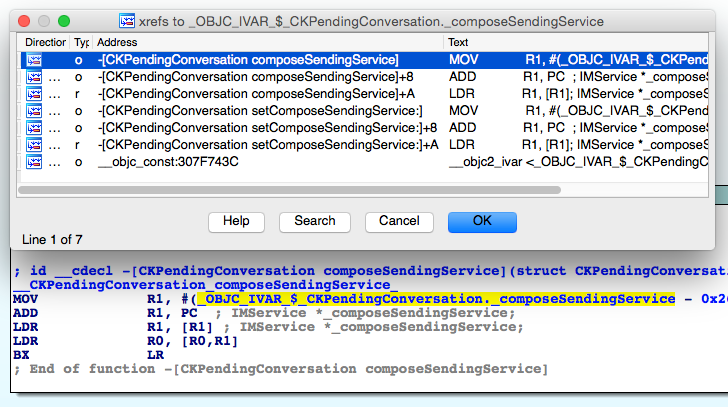


Figure 10- 23 Inspect cross references

Here, we can find 2 methods explicitly accessing \_composeSendingService, which happens to be one setter and one getter respectively. Naturally, we guess that \_composeSendingService is a property. Open CKPendingConversation.h and verify our assumption, as shown in figure 10-24.

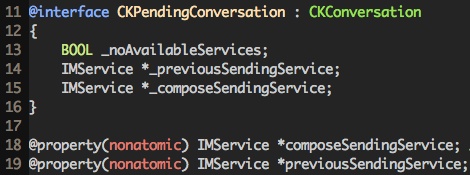


Figure 10- 24 CKPendingConversation.h

In Objective-C, write operation of a property is often carried out through its setter. Thus to find the 7th data source, we should set a breakpoint on [CKPendingConversation setComposeSendingService:] and check out who's the caller. Repeat our previous operations: reinput the address, set breakpoint at the beginning of [CKPendingConversation setComposeSendingService:], and then press "return" to trigger the breakpoint:

Process 30928 stopped

\* thread #1: tid = 0x78d0, 0x30b3665c ChatKit`-[CKPendingConversation setComposeSendingService:], queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x30b3665c ChatKit`-[CKPendingConversation setComposeSendingService:]

ChatKit`-[CKPendingConversation setComposeSendingService:]:

-> 0x30b3665c: movw r1, #41004

0x30b36660: movt r1, #2535

0x30b36664: add r1, pc

0x30b36666: ldr r1, [r1]

(lldb) p/x $lr

(unsigned int) $0 = 0x30b3656d

By subtracting ASLR offset of ChatKit from LR here, we get 0x2698456D, which is LR without offset. Then jump to this address in IDA, as shown in figure 10-25.

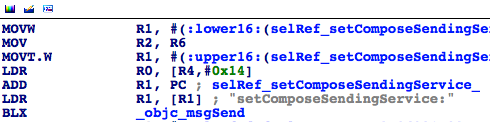


Figure 10- 25 Jump to 0x2698456D

The argument of [CKPendingConversation setComposeSendingService:], i.e. R2, is the 7th data source. R2 comes from R6, therefore R6 is the 8th data source. Search upwards to find R6's source, as shown in figure 10-26.

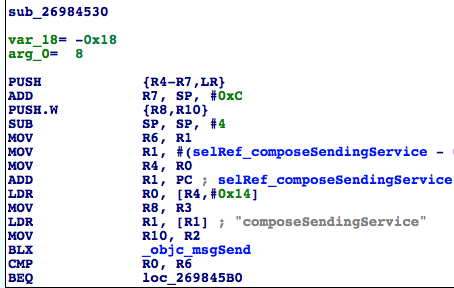


Figure 10- 26 Look for the 9th data source

R6 is from R1, so R1 is the 9th data source. And where does R1 come from? Since we are inside sub\_26984530 and R1 is read without being written, so R1 comes from the caller of sub\_26984530, right? Let’s take a look at the cross references to sub\_26984530 to look for its possible callers, as shown in figure 10-27.

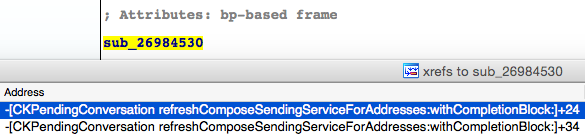


Figure 10- 27 Inspect cross references

Refresh sending service? This name is very informative. Let's head directly to [CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:] as shown in figure 10-28 for more details. In this method, sub\_26984530 is obviously the 2nd argument of refreshStatusForAddresses:withCompletionBlock:, namely the completionBlock, as shown in figure 10-28.

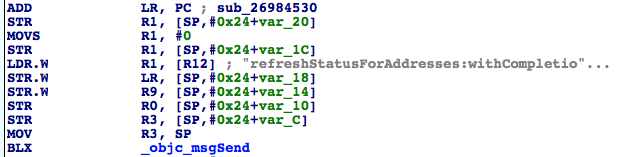


Figure 10- 28 [CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]

Although sub\_26984530 appears in this method, it just acts as an argument of objc\_msgSend, hence is not called directly. Well, who is the direct caller on earth? Actually, we've already mastered the solution of such problems: reinput the address, set a breakpoint at the beginning of sub\_26984530 and then press "return" to trigger the breakpoint.

Process 30928 stopped

\* thread #1: tid = 0x78d0, 0x30b36530 ChatKit`\_\_86-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]\_block\_invoke, queue = 'com.apple.main-thread, stop reason = breakpoint 6.1

frame #0: 0x30b36530 ChatKit`\_\_86-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]\_block\_invoke

ChatKit`\_\_86-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]\_block\_invoke:

-> 0x30b36530: push {r4, r5, r6, r7, lr}

0x30b36532: add r7, sp, #12

0x30b36534: push.w {r8, r10}

0x30b36538: sub sp, #4

(lldb) p/x $lr

(unsigned int) $38 = 0x30b364bb

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

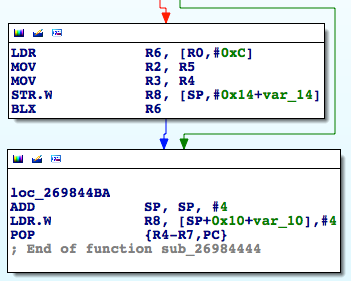


Figure 10- 29 Caller of sub\_26984530

As we can see, sub\_26984530 isn’t called explicitly. Instead, its address is stored in R6 to where the execution flow jumps, and then sub\_26984530 is called implicitly. As a result, the 9th data source comes from sub\_26984444. Well done! We have achieved a lot so far. Let’s keep searching for the occurrences of the 9th data source, as shown in figure 10-30.

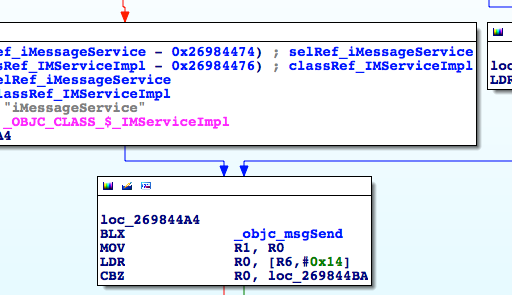


Figure 10- 30 Look for the 9th data source

There are several branches inside this subroutine to determine whether it should assign [IMServiceImpl smsService] or [IMServiceImpl iMessageService] to R1. Let’s figure out the branch conditions, starting from figure 10-31.

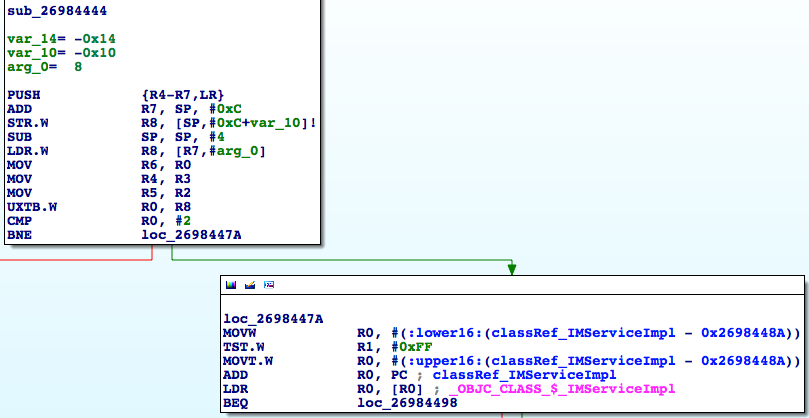


Figure 10- 31 Look for the 10th data source

If the value of R0 is 2, [IMServiceImpl iMessageService] is the 10th data source, otherwise we have to further check the value of R1. If R1 is 0, then [IMServiceImpl smsService] is the 10th 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;

}

That is to say, the value of the 10th data source is determined by the combination of R0 and R1, both of whom assume the responsibility of being the 11th data source, hereafter referred to as 11th data source A and 11th data source B respectively. At the same time, the above pseudo code can also be written as the following:

- (BOOL)supportIMessage

{

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

return NO;

}

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

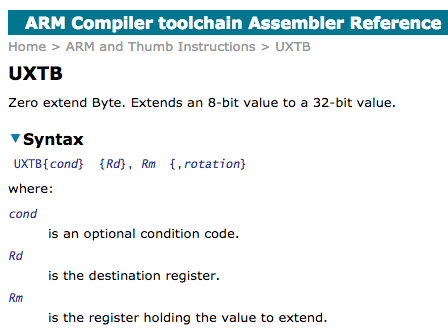


Figure 10- 32 UXTB

According to the ARM official document in figure 10-32, UXTB is used to zero extend the 8-bit value in R8 to a 32-bit value and then put it into R0, who is a 32-bit register. In other words, R0 comes from R8, so R8 is the 12th data source A; and from the facts that 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).

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 yet. 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 changes of SP, the following section is the hardest part of this book. So please stay very close! Let's start from “LDR.W R8, [R7,#8]" and trace back every single operation that writes into \*(SP’ + offset) for now.

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

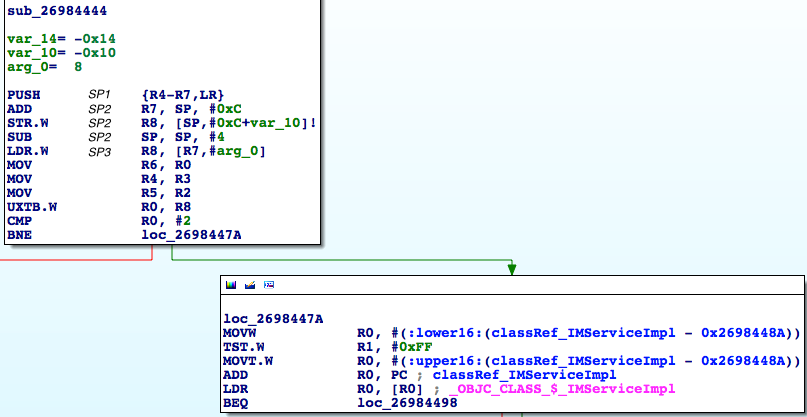


Figure 10- 33 Mark different SPs

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

"PUSH {R4-R7,LR}" pushes 5 registers, i.e. R4, R5, R6, R7 and LR into stack. Every register is 32-bit i.e. 4 bytes. Since the ARM stack is full descending, therefore SP2 = SP1 - 5 \* 0x4 = SP1 - 0x14. "ADD R7, SP, #0xC" is equivalent to R7 = SP2 + 0xC, which has no influence on SP. The value of var\_10 in "STR.W R8, [SP,#0xC+var\_10]!" is -0x10, so this instruction equals to “STR.W R8, [SP,#-4]", i.e. \*(SP2 - 0x4) = R8 and this instruction doesn’t have impact on SP either. "SUB SP, SP, #4" equals to SP3 = SP2 - 0x4. According to our marking rules, 13th data source A is \*(SP2 + 0x14). No instruction inside sub\_26984444 has written to this address before "LDR.W R8, [R7,#8]", so the value of \*(SP2 + 0x14) must come from the caller of sub\_26984444. Similarly, R1 is read without being written inside sub\_26984444, it must also come from the caller of sub\_26984444, right? If you are still confused, please review this paragraph until you understand it clearly, and then you're allowed to continue.

Alright, both the 13th data source A and the 11th data source B come from the caller of sub\_26984444. So our next specific task is to find the 14th data source A and the 12th data source B in the caller of sub\_26984444.

Reinput the recipient's address, set a breakpoint at the beginning of sub\_26984444, then press "return" to trigger the breakpoint:

Process 30928 stopped

\* thread #1: tid = 0x78d0, 0x30b36444 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke, queue = 'com.apple.main-thread, stop reason = breakpoint 7.1

frame #0: 0x30b36444 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke

ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke:

-> 0x30b36444: push {r4, r5, r6, r7, lr}

0x30b36446: add r7, sp, #12

0x30b36448: str r8, [sp, #-4]!

0x30b3644c: sub sp, #4

(lldb) p/x $lr

(unsigned int) $39 = 0x331f0d75

LR without offset is 0x331f0d75 – 0xa1b2000 = 0x2903ED75, which is outside ChatKit. Under such circumstance, how can we locate the image where 0x2903ED75 is? We've talked about the solution in chapter 6, which is simply set a breakpoint at the end of sub\_26984444 and keep executing "ni" to enter the internal of caller and identify the image. The commands are as follows:

Process 30928 stopped

\* thread #1: tid = 0x78d0, 0x30b364c0 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 124, queue = 'com.apple.main-thread, stop reason = breakpoint 8.1

frame #0: 0x30b364c0 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 124

ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 124:

-> 0x30b364c0: pop {r4, r5, r6, r7, pc}

0x30b364c2: nop

ChatKit`\_\_copy\_helper\_block\_:

0x30b364c4: ldr r1, [r1, #20]

0x30b364c6: adds r0, #20

(lldb) ni

Process 30928 stopped

\* thread #1: tid = 0x78d0, 0x331f0d74 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1360, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x331f0d74 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1360

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1360:

-> 0x331f0d74: movw r0, #26972

0x331f0d78: movt r0, #2081

0x331f0d7c: add r0, pc

0x331f0d7e: ldr r1, [r0]

We're inside IMCore now. Since we have just calculated the value of LR without offset to be 0x2903ED75, as well IMCore shares the same ASLR offset with ChatKit, so just drag and drop IMCore into IDA and jump to 0x2903ED75 when the initial analysis has been finished, as shown in figure 10-34.

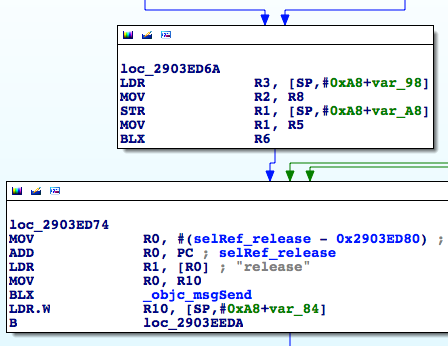


Figure 10- 34 Caller of sub\_26984444

See, another implicit call from sub\_2903E824, and 2 of 4 instructions before "BLX R6" has relation with SP. To make it more convenient for reading, I'll take instructions before and after calling "BLX R6" from their respective images and put them together into one figure. The process and result is shown in figure 10-35 and figure 10-36.

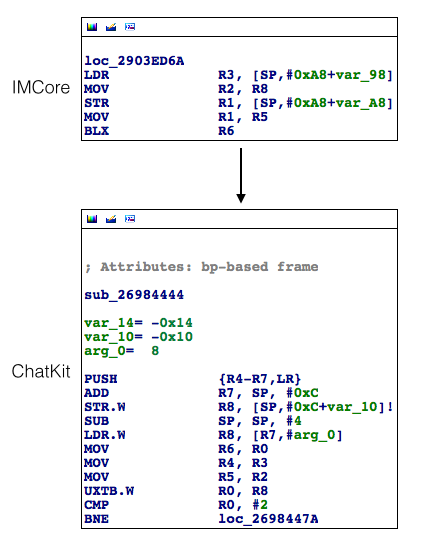


Figure 10- 35 Before instructions of 2 images are put together

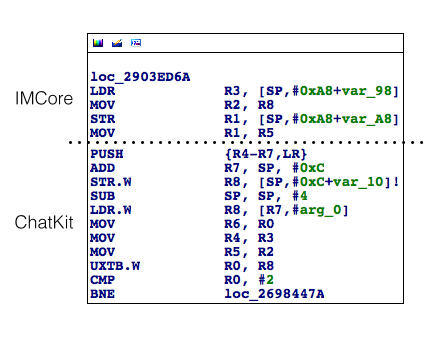


Figure 10- 36 After instructions of 2 images are put together

Let’s keep looking for the 14th data source A, which has been written into \*(SP2 + 0x14), do you still remember? OK, mark the SPs in loc\_2903ED6A just like what we've done, as shown in figure 10-37.

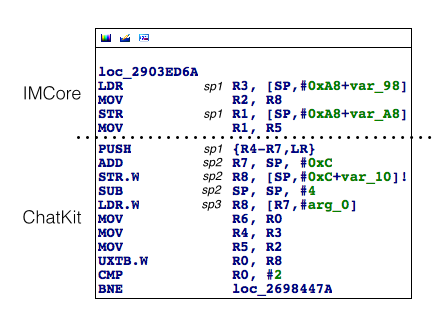


Figure 10- 37 Mark SPs

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

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

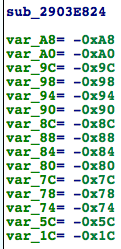


Figure 10- 38 sub\_2903e824

As a result, R3 = \*(SP1 + 0x10) and this instruction has no influence on the value of SP. “MOV R2, R8” has nothing to do with SP; the value of var\_A8 in “STR R1, [SP,#0xA8+var\_A8]” is -0xA8, so \*SP1 = R1, which doesn’t influence SP too; “MOV R1, R5” has nothing to do with SP either. These SPs are really confusing for sure, so take a break and let me summarize it.

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

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

Therefore, "STR R1, [SP,#0xA8+var\_A8]" is the place where \*(SP2 + 0x14) is written, and R1 in this instruction is the 14th data source A! Also, we can easily find that R5 in "MOV R1, R5" is the 12th data source B. The logics of tracing from 13th data source A to 14th data source A and from 11th data source B to 12th data source B go across images, bringing high complexity. With the illustration of figure 10-39, I hope everything is more intuitive. We strongly suggest you comb through everything by referring to this figure before moving on to the next paragraph.

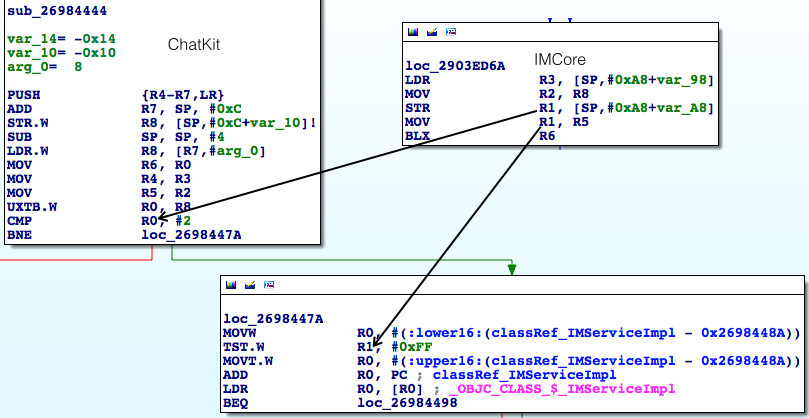


Figure 10- 39 How data sources evolve

Before we continue our analysis, let's verify our deduction so far with LLDB: reinput the address and set the breakpoint on “STR R1, [SP,#0xA8+var\_A8]" to print R1, i.e. the 14th data source A. Next, execute "ni" until we reach "MOV R1, R5", print R5 i.e. the 12th data source B. Then we'll experience an image switch from IMCore to ChatKit, so execute "si" to reach "CMP R0, #2" and print R0, i.e. the 13th data source A. Finally, we execute "ni" until "TST.W R1, #0xFF" to print R1, i.e. the 11th data source B. Press "return" to trigger the breakpoint and follow the above steps to check whether their values equal to each other like figure 10-39 shows.

(lldb) br s -a 0x30230D6E

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x30230d6e IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1354, queue = 'com.apple.main-thread, stop reason = breakpoint 11.1

frame #0: 0x30230d6e IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1354

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1354:

-> 0x30230d6e: str r1, [sp]

0x30230d70: mov r1, r5

0x30230d72: blx r6

0x30230d74: movw r0, #26972

(lldb) p $r1

(unsigned int) $27 = 0

(lldb) ni

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x30230d70 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1356, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x30230d70 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1356

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1356:

-> 0x30230d70: mov r1, r5

0x30230d72: blx r6

0x30230d74: movw r0, #26972

0x30230d78: movt r0, #2081

(lldb) p $r5

(unsigned int) $28 = 1

(lldb) ni

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x30230d72 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1358, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x30230d72 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1358

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 1358:

-> 0x30230d72: blx r6

0x30230d74: movw r0, #26972

0x30230d78: movt r0, #2081

0x30230d7c: add r0, pc

(lldb) si

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x2db76444 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke, queue = 'com.apple.main-thread, stop reason = instruction step into

frame #0: 0x2db76444 ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke

ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke:

-> 0x2db76444: push {r4, r5, r6, r7, lr}

0x2db76446: add r7, sp, #12

0x2db76448: str r8, [sp, #-4]!

0x2db7644c: sub sp, #4

(lldb) ni

……

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x2db7645c ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 24, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2db7645c ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 24

ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 24:

-> 0x2db7645c: cmp r0, #2

0x2db7645e: bne 0x2db7647a ; \_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 54

0x2db76460: movw r0, #19376

0x2db76464: movt r0, #2535

(lldb) p $r0

(unsigned int) $29 = 0

(lldb) ni

……

Process 37477 stopped

\* thread #1: tid = 0x9265, 0x2db7647e ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 58, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2db7647e ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 58

ChatKit`\_\_71-[CKPendingConversation refreshStatusForAddresses:withCompletionBlock:]\_block\_invoke + 58:

-> 0x2db7647e: tst.w r1, #255

0x2db76482: movt r0, #2535

0x2db76486: add r0, pc

0x2db76488: ldr r0, [r0]

(lldb) p $r1

(unsigned int) $30 = 1

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

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

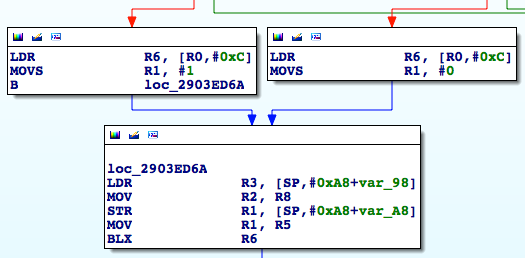


Figure 10- 40 15th data source A

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

IIRC, since the 11th data source A, the value of data source A has never changed, the values of 11th, 12th, 13th, 14th and 15th data source A are all the same, which are either 0 or 1. However, the previous pseudo code is like this:

- (BOOL)supportIMessage

{

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

return NO;

}

Because the 11th data source A is either 0 or 1, under no circumstance can it be 2. In that case, data source A becomes meaningless in our tracing, right? Hence the pseudo code can be simplified as follows:

- (BOOL)supportIMessage

{

if (11thDataSourceB != 0) return YES;

return NO;

}

As a result, we can ignore data source A and concentrate on the finding of the 13th data source B, hereafter referred to as the 13th data source. Since the 12th data source B is R5, we can confirm that 13th data source must be written into R5 by a certain instruction, right? Click R5 and IDA will highlight all R5s as yellow to make it more convenient for tracing in the sea of ARM assembly. Keep reversing to find where R5 is written.

When we're searching upward to locate the 13th data source, we see there're 4 branches to loc\_2903EAE0, as shown in figure 10-41.

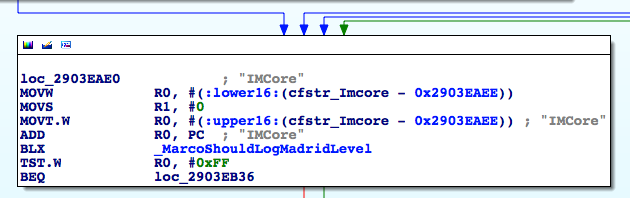


Figure 10- 41 loc\_2903EAE0

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

When we trace into loc\_2903EA3E, the situation is similar to loc\_2903EAE0. Although there are 3 branches upon it, the 1st and 2nd branches both contain a "MOVS R5, #0" as shown in figure 10-42, so they can be excluded for now.

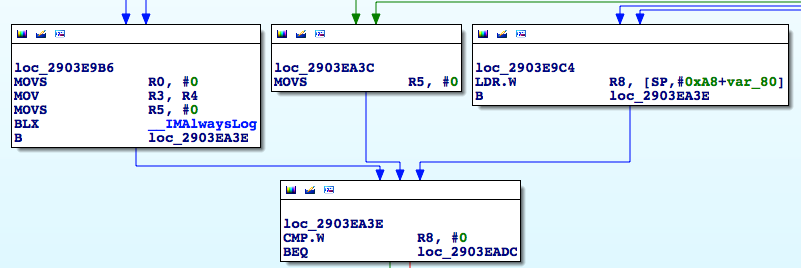


Figure 10- 42 loc\_2903EA3E

As a result, the actual upstream is the 3rd branch, i.e. loc\_2903E9C4, which has 2 branches upon it. Now that both branches contain "MOVS R5, #1", which is the actual one?

Reinput the address and set breakpoints on 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 MobileSMS chose, as shown in figure 10-43.

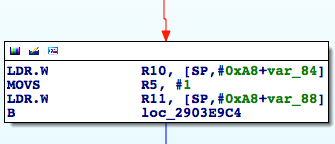


Figure 10- 43 The left branch

Now, we have found the 13th data source, it’s a constant with value 1. You may wonder, if 13th data source is a constant, does 14th data source still exist? The data source clues seem to be interrupted, what should we to do next? Good point.

In the previous figures, there're several "MOVS R5, #0". Although the 13th data source comes from "MOVS R5, #1", which seems to be a constant, according to programmatic paradigm, there should be a conditional branch to determine whether " MOVS R5, #0" or " MOVS R5, #1" gets executed, just like the pseudo code below.

if (iMessageIsAvailable) R5 = 1;

else R5 = 0;

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

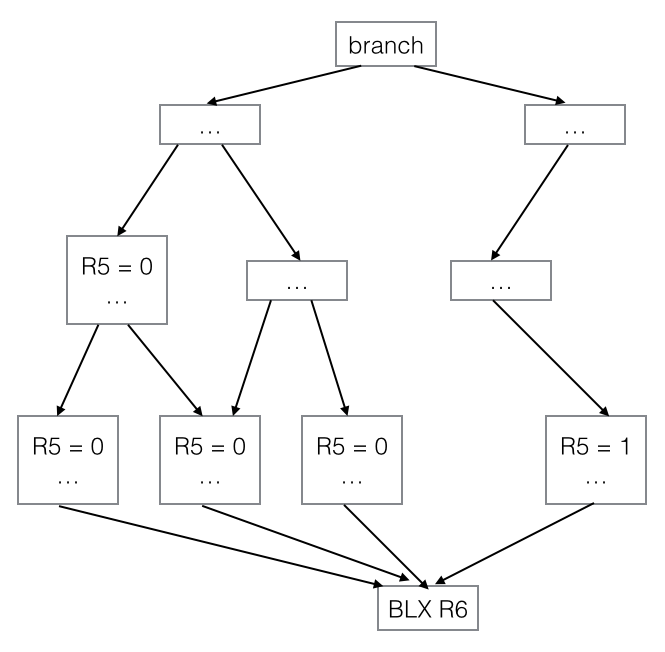


Figure 10- 44 Pseudo IDA graph view

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

R5 = iMessageIsAvailable;

If you can understand this, then our next task is to keep tracing back to analyze every branch we meet. If different branches result in writing different values into R5, we need to figure out what’s the branch condition, and this condition is our target data source. Let’s head to figure 10-45 and start from here.

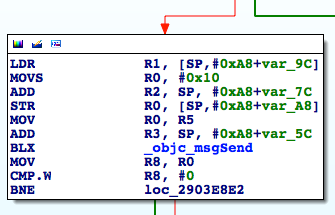


Figure 10- 45 Branch

If the process branches left, R5 is possibly to be set 0. Since the branch condition is the return value of objc\_msgSend, let’s set a breakpoint here and see what method it is:

Process 132234 stopped

\* thread #1: tid = 0x2048a, 0x331f092e IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 266, queue = 'com.apple.main-thread, stop reason = breakpoint 5.1

frame #0: 0x331f092e IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 266

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 266:

-> 0x331f092e: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f0932: mov r8, r0

0x331f0934: cmp.w r8, #0

0x331f0938: bne 0x331f08e2 ; \_\_\_lldb\_unnamed\_function425$$IMCore + 190

(lldb) p (char \*)$r1

(char \*) $6 = 0x2f7d81d9 "countByEnumeratingWithState:objects:count:"

(lldb) po $r0

<\_\_NSArrayI 0x16706930>(

mailto:snakeninny@gmail.com

)

As we can see, this method returns the count of the recipient array. If the array is not empty, MobileSMS will branch right. Actually, the recipient array is not empty, therefore this branch condition is not met, MobileSMS will branch right, which doesn't change R5. OK, search upward for the next branch, as shown in figure 10-46.

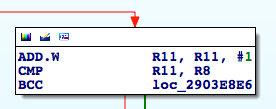


Figure 10- 46 Branch

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

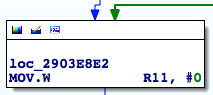


Figure 10- 47 loc\_2903e8e2

The initial value of R11 is 0. Each time before executing "CMP R11, R8", R11 will increase by 1. In this way, R11 plays the role of a counter. "CMP" performs subtraction operation, if there's borrow, then carry flag will be set 0, otherwise carry flag will be set 1. The branch instruction here is "BCC", in which "CC" means "Carry Clear", i.e. "if carry flag is 0". Therefore, if R11 - R8 produces borrow, i.e. R8 is greater than R11, then MobileSMS will branch right, otherwise it will branch left. So the key here is R8, as shown in figure 10-48.



Figure 10- 48 Where R8 comes

R8 comes from [NSArray countByEnumeratingWithState:objects:count:]. Reinput the address, set the breakpoint and press "return", let's see what NSArray is:

(lldb) br s -a 0x3023089C

Breakpoint 2: where = IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 120, address = 0x3023089c

Process 102482 stopped

\* thread #1: tid = 0x19052, 0x3023089c IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 120, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x3023089c IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 120

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 120:

-> 0x3023089c: blx 0x302a03b0 ; symbol stub for: objc\_msgSend

0x302308a0: mov r8, r0

0x302308a2: cmp.w r8, #0

0x302308a6: beq.w 0x302309c2 ; \_\_\_lldb\_unnamed\_function425$$IMCore + 414

(lldb) p (char \*)$r1

(char \*) $5 = 0x2c8181d9 "countByEnumeratingWithState:objects:count:"

(lldb) po $r0

<\_\_NSArrayI 0x178d6b20>(

mailto:snakeninny@gmail.com

)

NSArray is an array of recipients, thus R8 is the recipient count. If there's more than 1 recipients, then since R11 is 1 when "CMP R11, R8" gets executed for the first time, we can know that R8 is greater than R11 and MobileSMS will branch right, as shown in figure 10-49.

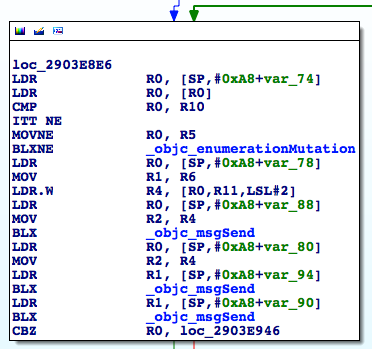


Figure 10- 49 Branch

The branch condition inside loc\_2903E8E6 is R0. If R0 == 0, then branch left, meaning this address doesn’t support iMessage. Otherwise branch right and reach figure 10-50.

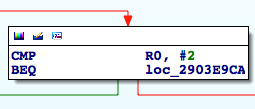


Figure 10- 50 Branch

The branch condition in figure 10-50 is still R0. If R0 == 2 then branch left, iMessage is not supported. Otherwise branch right and go back to figure 10-46. Note, these 3 blocks of code don’t change the value of R8. As a result, R0 at the bottom of loc\_2903E8E6 is very import; as long as R0 != 0 && R0 != 2, the branch in figure 10-46 is useless. That’s because R11 keeps increasing while R8 stays the same, MobileSMS will eventually branch left and come to the conclusion that iMessage is supported. So judging from all information above, we can think of R0 as the essential branch condition in this loop. Do you still remember what I've just said? “If different branches result in writing different values into R5, we need to figure out what’s the branch condition, and this condition is our target data source”. Thus, R0 is the 14th data source.

Next, let's check with LLDB what are these objc\_msgSends in figure 10-49, as well the source of R0:

Process 154446 stopped

\* thread #1: tid = 0x25b4e, 0x331f0900 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 220, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x331f0900 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 220

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 220:

-> 0x331f0900: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f0904: ldr r0, [sp, #40]

0x331f0906: mov r2, r4

0x331f0908: ldr r1, [sp, #20]

(lldb) p (char \*)$r1

(char \*) $7 = 0x2f7d897a "removeObject:"

(lldb) po $r0

<\_\_NSArrayM 0x170ec120>(

mailto:snakeninny@gmail.com

)

(lldb) po $r2

mailto:snakeninny@gmail.com

(lldb) ni

……

Process 154446 stopped

\* thread #1: tid = 0x25b4e, 0x331f090a IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 230, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x331f090a IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 230

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 230:

-> 0x331f090a: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f090e: ldr r1, [sp, #24]

0x331f0910: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f0914: cbz r0, 0x331f0946 ; \_\_\_lldb\_unnamed\_function425$$IMCore + 290

(lldb) p (char \*)$r1

(char \*) $10 = 0x2f7d8113 "valueForKey:"

(lldb) po $r2

mailto:snakeninny@gmail.com

(lldb) po $r0

{

"mailto:snakeninny@gmail.com" = 1;

}

(lldb) po [$r0 class]

\_\_NSCFDictionary

(lldb) ni

……

Process 154446 stopped

\* thread #1: tid = 0x25b4e, 0x331f0910 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 236, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x331f0910 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 236

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore + 236:

-> 0x331f0910: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f0914: cbz r0, 0x331f0946 ; \_\_\_lldb\_unnamed\_function425$$IMCore + 290

0x331f0916: cmp r0, #2

0x331f0918: beq 0x331f09ca ; \_\_\_lldb\_unnamed\_function425$$IMCore + 422

(lldb) p (char \*)$r1

(char \*) $14 = 0x2f7de6f3 "integerValue"

(lldb) po $r0

1

(lldb) po [$r0 class]

\_\_NSCFNumber

(lldb) c

Restore these 3 objc\_msgSends into Objective-C methods, they are [NSArray removeObject:@ "mailto:snakeninny@gmail.com"], [NSDictionary valueForKey: @"mailto:snakeninny@gmail.com"] and [NSNumber integerValue] respectively. Among them, R0 of the 2nd objc\_msgSend deserves our special attention. It is the key-value pair in this R0 (an NSDictionary) that determines the 14th data source. Therefore, this NSDictionary is the 15th data source. According to figure 10-49, we can know that it comes from [SP,#0xA8+var\_80], which means [SP,#0xA8+var\_80] is the 16th data source. Here comes our familiar operation to trace the 17th data source; inspect the cross references to var\_80 as shown in figure 10-51.

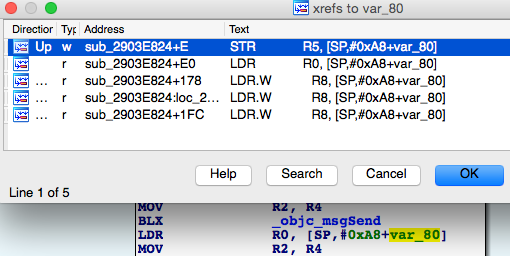


Figure 10- 51 Inspect cross references

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

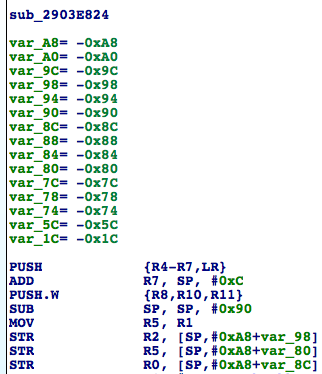


Figure 10- 52 sub\_2903E824

The 16th data source comes from R5, which is the 17th data source. The 17th data source is from R1, which is the 18th data source, and it is read without being written, meaning R1 comes from the caller of sub\_2903E824, right? Let’s take a look at the subroutine's cross references, as shown in figure 10-53.

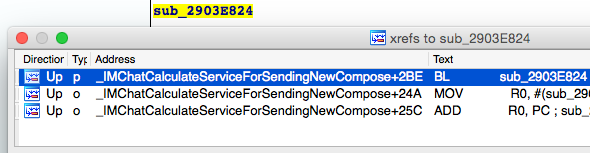


Figure 10- 53 Inspect cross references

"Calculate service for sending new compose", as the name suggests, its function is quite clear. Double click the first cross reference to check its caller, as shown in figure 10-54.

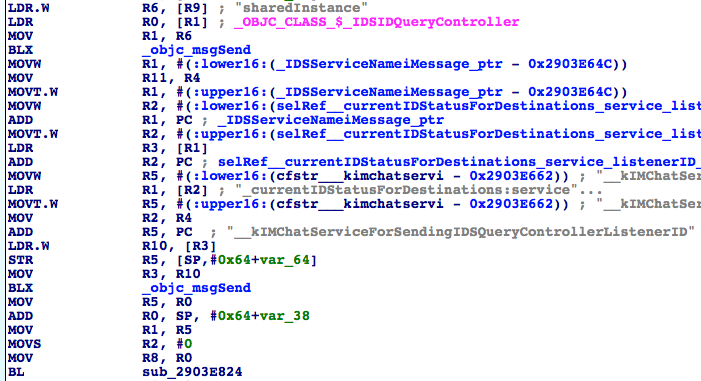


Figure 10- 54 Caller of sub\_2903E824

To avoid any implicit calling, let's first make sure the caller of sub\_2903E824 is actually IMChatCalculateServiceForSendingNewCompose. Reinput the address, set a breakpoint at the first instruction of sub\_2903E824 and then press "return" to trigger the breakpoint:

Process 154446 stopped

\* thread #1: tid = 0x25b4e, 0x331f0824 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x331f0824 IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore

IMCore`\_\_\_lldb\_unnamed\_function425$$IMCore:

-> 0x331f0824: push {r4, r5, r6, r7, lr}

0x331f0826: add r7, sp, #12

0x331f0828: push.w {r8, r10, r11}

0x331f082c: sub sp, #144

(lldb) p/x $lr

(unsigned int) $17 = 0x331f067b

(lldb)

The ASLR offset is 0xa1b2000, so LR without offset is 0x2903E67B, which is exactly inside IMChatCalculateServiceForSendingNewCompose. OK, since the 18th data source is from R5, then R5 is the 19th data source. Further, the 19th data source is from the return value of objc\_msgSend, so this return value is the 20th data source. With everything ready, let's reveal this mysterious objc\_msgSend:

Process 154446 stopped

\* thread #1: tid = 0x25b4e, 0x331f0668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688, queue = 'com.apple.main-thread, stop reason = breakpoint 3.1

frame #0: 0x331f0668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688

IMCore`IMChatCalculateServiceForSendingNewCompose + 688:

-> 0x331f0668: blx 0x332603b0 ; symbol stub for: objc\_msgSend

0x331f066c: mov r5, r0

0x331f066e: add r0, sp, #44

0x331f0670: mov r1, r5

(lldb) p (char \*)$r1

(char \*) $18 = 0x33274340 "\_currentIDStatusForDestinations:service:listenerID:"

(lldb) po $r0

<IDSIDQueryController: 0x15dcb010>

(lldb) po $r2

<\_\_NSArrayM 0x170e7900>(

mailto:snakeninny@gmail.com

)

(lldb) po $r3

com.apple.madrid

(lldb) po [$r3 class]

\_\_NSCFConstantString

(lldb) x/10 $sp

0x001e4548: 0x3b3f52b8 0x001e459c 0x3b4227b4 0x3c01b05c

0x001e4558: 0x00000001 0x00000000 0x170828d0 0x001e4594

0x001e4568: 0x2baac821 0x00000000

(lldb) po 0x3b3f52b8

\_\_kIMChatServiceForSendingIDSQueryControllerListenerID

(lldb) po [0x3b3f52b8 class]

\_\_NSCFConstantString

(lldb) c

Success belongs to the persevering. This objc\_msgSend is restored to [[IDSIDQueryController sharedInstance] \_currentIDStatusForDestinations:@[@"mailto:snakeninny@gmail.com"] service:@"com.apple.madrid" listenerID:@"\_\_kIMChatServiceForSendingIDSQueryControllerListenerID"]. Since the last 2 arguments are constants, the only variable argument is the first array, i.e. the recipient array. What a long journey! We've finally tracked down the original data source!

I know, I know, this section is so hard that you're already dizzy now. Stay up for a while, we're almost done with this task.

### 10.2.5 Restore the process of the original data source becoming placeholderText

Now that we have found the core method, seems we can detect whether an address supports iMessage by modifying the first argument, i.e. the NSArray of recipients. As long as the key (an address) associated value (an integer) in the return value (an NSDictionary) is neither 0 nor 2, we can confirm that this address supports iMessage; otherwise it only supports SMS. Is that so? As we already know, the format of email addresses is "mailto:email@address", how about phone number format? Let’s set a breakpoint on \_currentIDStatusForDestinations:service:listenerID and take a look:

Process 102482 stopped

\* thread #1: tid = 0x19052, 0x30230668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688, queue = 'com.apple.main-thread, stop reason = breakpoint 6.1

frame #0: 0x30230668 IMCore`IMChatCalculateServiceForSendingNewCompose + 688

IMCore`IMChatCalculateServiceForSendingNewCompose + 688:

-> 0x30230668: blx 0x302a03b0 ; symbol stub for: objc\_msgSend

0x3023066c: mov r5, r0

0x3023066e: add r0, sp, #44

0x30230670: mov r1, r5

(lldb) po $r2

<\_\_NSArrayM 0x17820560>(

tel:+86PhoneNumber

)

OK, we can now turn back to 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}

Aha, the output clearly supports our statements: 2 iMessage supportive emails and 1 iMessage supportive phone number all return 1, while the other 3 iMessage unsupportive addresses return 2. What's more, we know the code name of iMessage is "Madrid". Mission complete! Cheers!

10.3 Send iMessages

Through the baptism of section 10.2, I believe many of you may share the same feeling with me: debugging with LLDB step by step is of course rigorous and precise, but the workload along with it is overwhelmingly heavy. Reverse engineering is full of error checks, don’t be afraid of making mistakes. In this section, we’ll jump out and step up with wild guesses to achieve our goal; we’ll try to avoid massive analysis with LLDB, instead make use of class-dump to filter suspicious methods, and test them with IDA and Cycript to finally achieve our goal of sending iMessages.

10.3.1 Observe MobileSMS and look for cut-in points

In comparison with detecting iMessages, cut-in point of sending iMessages is more noticeable. In figure 10-55, the bold blue “Send” button is Apple’s gift for this section.

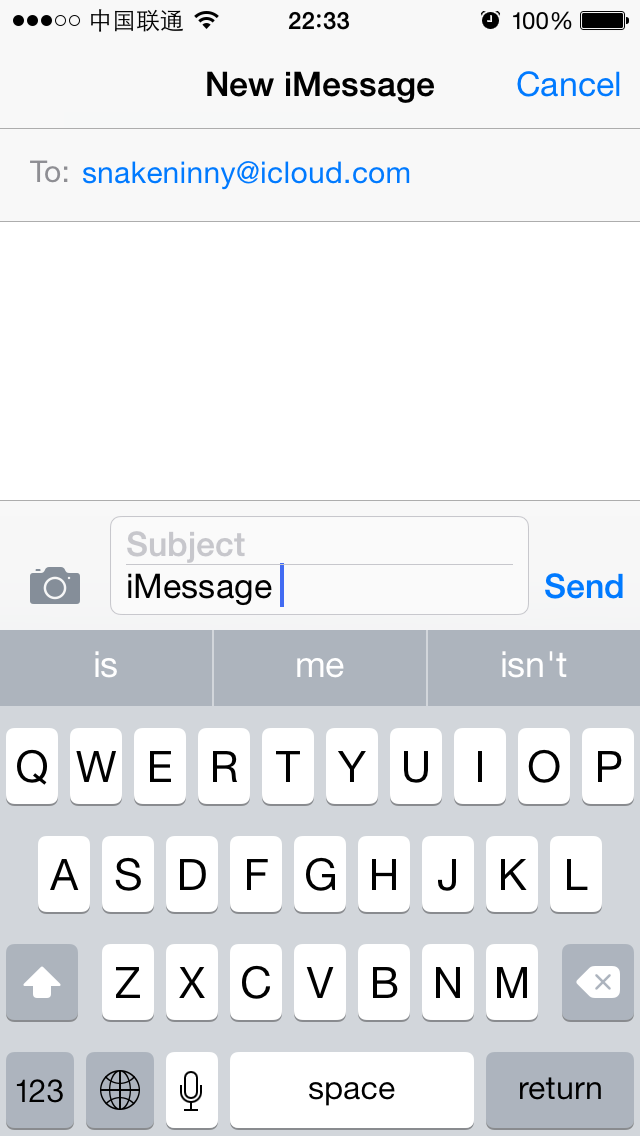


Figure 10- 55 “Send” button

We can send an iMessage by pressing “Send”, and the whole process will be animated on UI. Like what we did in section 10.2, let’s consider how to turn clues on UI into ideas in reverse engineering:

“Send” button is supposed to be a UIView object, or more specifically and possibly, a UIButton object; we press this button to call its response method; overall response actions include refreshing UI, sending the iMessage, adding a sending record and so on. That’s to say, the action of sending iMessages is only a subset of all response actions.

In “New Message” view, our inputs include recipient addresses and message contents, they’re the original data source. Since we can get all response actions, and the action of sending iMessages is supposed to take the original data source as arguments, so they can be references for us to filter the action of sending iMessages out of all response actions. Unlike what we’ve done in the last section, which was tracing back from tail to head, in the following sections, we’re tracing from head to tail, showing you another common scenario of iOS reverse engineering.

In a nutshell, our thoughts are: first uncover response method of “Send” button with Cycript, then overview all response actions with IDA and class-dump, as well filter those suspicious methods out. Finally, test the filtered methods and locate our target.

10.3.2 Find response method of “Send” button using Cycript

Since we’ve already known that the superview of “Send” button is a CKMessageEntryView object in section 10.2, we can repeat what we’ve done in section 10.2.2 and get the superview without further tests:

FunMaker-5:~ root# 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>>"]

Among these views, “UIView: 0x16052920” is where “iMessage” resides, do you remember? As a result, the following 2 UIButtons are quite suspicious, my intuition tells me “Send” is one of them. Meanwhile, the hidden property of the 2nd UIButton is set to YES, indicating it’s invisibility. Well, let’s test the 1st UIButton, “UIButton: 0x1605a8b0” with Cycript:

cy# [#0x1605a8b0 setHidden:YES]

The view changed to figure 10-56 after the above command:

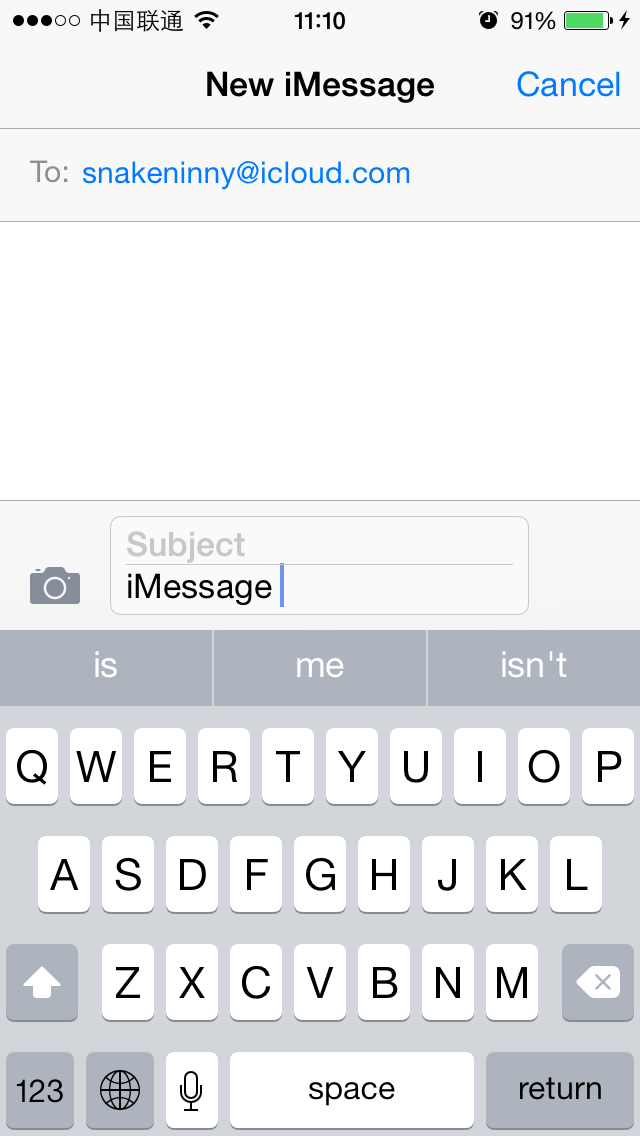


Figure 10- 56 Hide “Send”

Neat. After pressing this UIButton, an iMessage will be sent; a UIButton and its response action are always associated with [UIControl addTarget:action:forControlEvents:]. Since UIControl offers another method actionsForTarget:forControlEvent: to find its own response method, let’s see what method will get called after pressing “Send” with this method:

cy# [#0x1605a8b0 setHidden:NO]

cy# button = #0x1605a8b0

#"<UIButton: 0x1605a8b0; frame = (266 27; 53 33); hidden = YES; opaque = NO; layer = <CALayer: 0x16052a00>>"

cy# [button allTargets]

[NSSet setWithArray:@[#"<CKMessageEntryView: 0x160c6180; frame = (0 0; 320 65); opaque = NO; autoresize = W; layer = <CALayer: 0x16089920>>"]]]

cy# [button allControlEvents]

64

cy# [button actionsForTarget:#0x160c6180 forControlEvent:64]

@["touchUpInsideSendButton:"]

As we can see, the response method is [CKMessageEntryView touchUpInsideSendButton:]. Now let’s turn to IDA and LLDB for deeper analysis.

10.3.3 Find suspicious sending action in response method

[CKMessageEntryView touchUpInsideSendButton:] doesn’t do much, as shown in figure 10-57.



Figure 10- 57 [CKMessageEntryView touchUpInsideSendButton:button]

It first calls [[self delegate] messageEntryViewSendButtonHit:self] then calls [self updateEntryView]. As their names suggest, the latter method simply refreshes UI; so sending action should come from the former one. Use Cycript to find out what’s [self delegate]:

cy# [#0x160c6180 delegate]

#"<CKTranscriptController: 0x15537200>"

Go to [CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView] in IDA. This is a pretty simple method, as shown in figure 10-58.

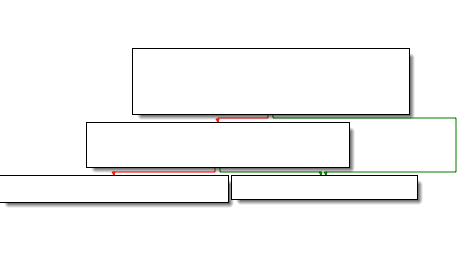


Figure 10- 58 [CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView]

By overviewing this method, I bet you can easily locate the actual sending action in [self sendComposition:[CKMessageEntryView compositionWithAcceptedAutocorrection]]. Let’s see what’s [self compositionWithAcceptedAutocorrection] in Cycript:

cy# [#0x160c6180 compositionWithAcceptedAutocorrection]

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

It’s an object of CKComposition, which clearly contains message text and subject. Keep digging into sendComposition:, as shown in figure 10-59.

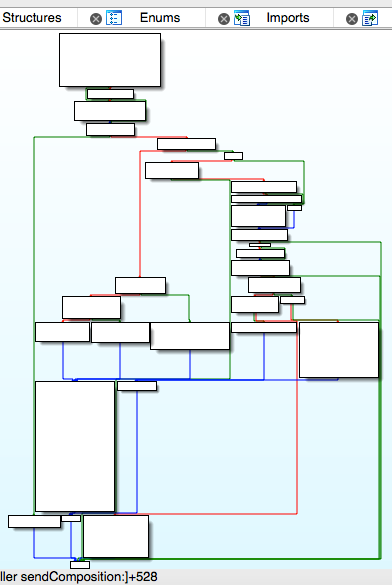


Figure 10- 59 [self sendComposition:]

The implementation is rather complicated. As we said earlier in this section, we’ll try to avoid massive use of LLDB, thus let’s first go over all branches in this method to glimpse the possible execution flows, then debug the uncertain ones with LLDB. We start from loc\_268D427C, as shown in figure 10-60.

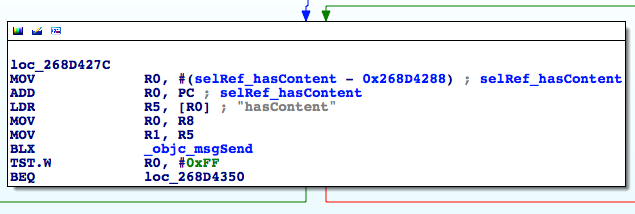


Figure 10- 60 loc\_268D427C

If the iMessage “hasContent”, branches right. According to figure 10-56, our content is “iMessage”, so branch right and arrive at figure 10-61.

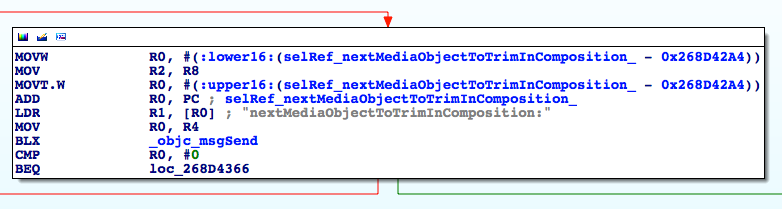


Figure 10- 61 branch

“nextMediaObjectToTrimInComposition:”? Is “media object” referring to image, audio or video kind of things? Since we’re sending plain text, there’s no media at all. Branch right and arrive at figure 10-62.

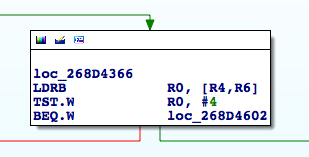


Figure 10- 62 Branch

What’s R0? Get back to the beginning of sendComposition:, as shown in figure 10-63.

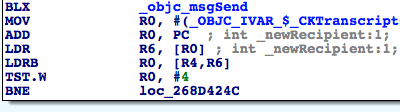


Figure 10- 63 Trace R0

R0 turns out to be self->\_newRecipient, let’s print its value in Cycript:

cy# #0x15537200->\_newRecipient

1

So the result of “TST.W R0, #4” is 0, branch right and arrive at loc\_268D4604, as shown in figure 10-64.

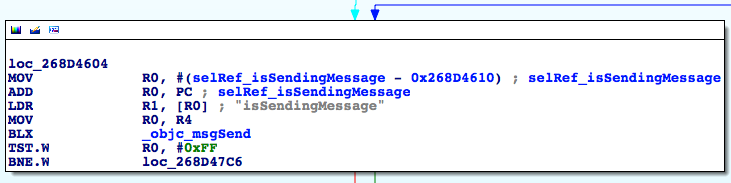


Figure 10- 64 loc\_268D4604

Whether iOS “isSendingMessage”? We don’t know if the timing is before or after pressing “Send” button, so let’s test them both. Before pressing “Send”:

cy# [#0x15537200 isSendingMessage]

0

And after pressing “Send”:

cy# [#0x15537200 isSendingMessage]

0

So, [self isSendingMessage] returns 0 anyway. Branch left and arrive at loc\_268D4636, as shown in figure 10-65.

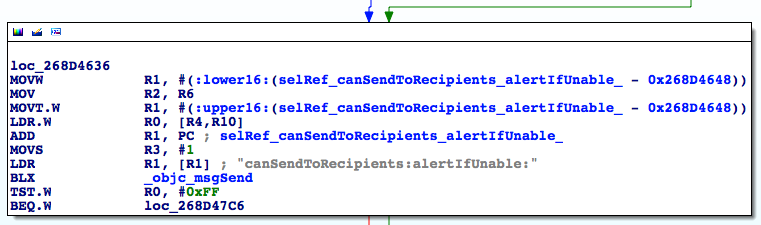


Figure 10- 65 loc\_268D4636

Can we send the iMessage to the recipient? Since the recipient is a valid iMessage account, of course we can! Branch left and arrive at figure 10-66.

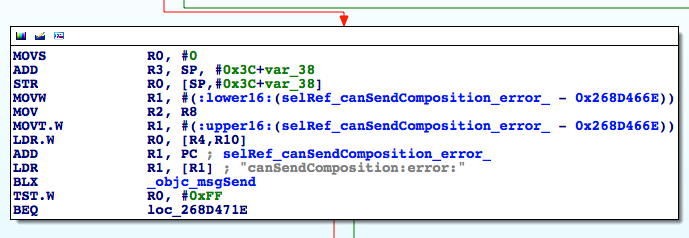


Figure 10- 66 Branch

Can we send the composition? Since we’ve already printed the CKComposition object, there doesn’t seem to be any problems. Branch left and arrive at figure 10-67.

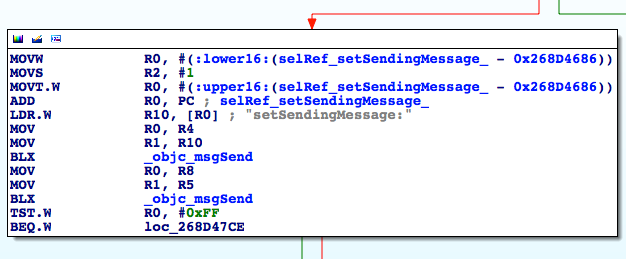


Figure 10- 67 Branch

The branch condition R0 comes from the return value of the 2nd objc\_msgSend. Search upwards, we can find R5 in figure 10-60; it’s determining if the iMessage “hasContent” again. Therefore, branch right and arrive at figure 10-68.



Figure 10- 68 Branch

This is an informative figure. If you look close, you’ll discover that most objc\_msgSends are just refreshing UI, making the last objc\_msgSend, i.e. [R4 sendMessage:R2] more eye-catching. What’s R4 and R2? Look upwards, you’ll see they’re CKTranscriptController and the argument of [self sendComposition:], respectively. Let’s continue analyzing from [CKTranscriptController sendMessage:], as shown in figure 10-69.

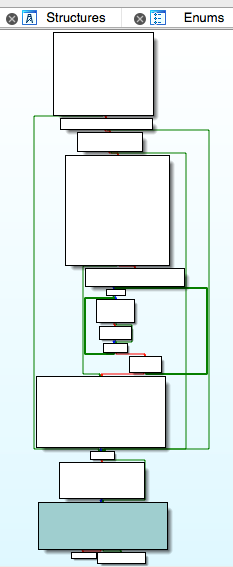


Figure 10- 69 [CKTranscriptController sendMessage:]

Another method full of branches. But after giving a glimpse to the possible execution flows just like what we did to sendComposition:, we can find that most branches are just making preparations, only “\_startCreatingNewMessageForSending:” looks promising. Let’s take a look at its implementation, as shown in figure 10-70.

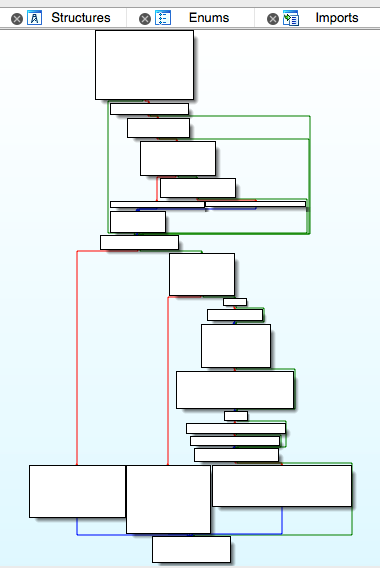


Figure 10- 70 [CKTranscriptController \_startCreatingNewMessageForSending:]

Again, it’s a method full of branches. Overview the implementation, I think you’ll notice the method “sendMessage:newComposition:” just like me. The method occurs twice in [CKTranscriptController \_startCreatingNewMessageForSending:], as shown in the 2 dark colored blocks in figure 10-71.

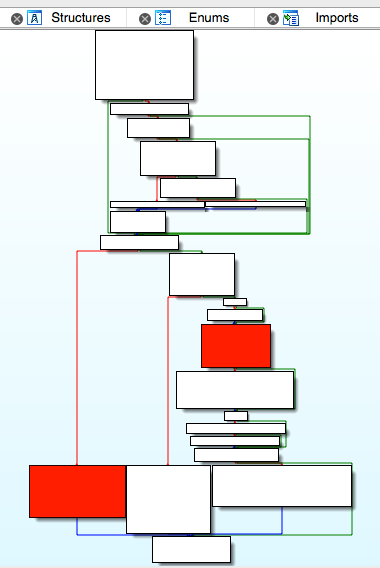


Figure 10- 71 [CKTranscriptController \_startCreatingNewMessageForSending:]

Take a look at the implementation of this method, as shown in figure 10-72.

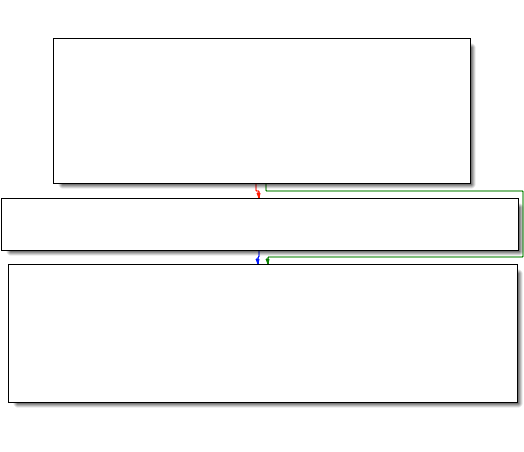


Figure 10- 72 [CKConversation sendMessage:newComposition:]

It further calls “sendMessage:onService:newComposition:”, so proceed to this method, as shown in figure 10-73.

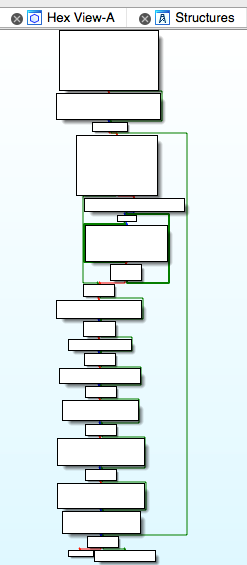


Figure 10- 73 [CKConversation sendMessage:onService:newComposition:]

The execution flow of this method is more straightforward than the previous ones. Skim it briefly, we can see phrases like “Sending message with guid: %@”, “ => Sending account: %@” and “=> Recipients: [%@]”, most of which are arguments of \_CKLogExternal. If MobileSMS has already started recording these into syslog, doesn’t it prove that “send iMessage” is happening? What’s more, we’ve seen the suspicious keyword “sendMessage:” again in figure 10-74:

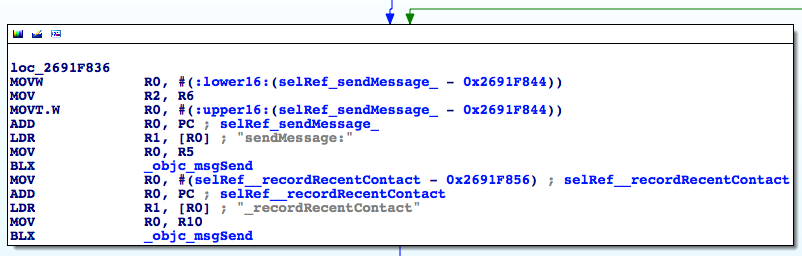


Figure 10- 74 loc\_2691f836

What’s the receiver and arguments of “sendMessage:”? Let’s find them in IDA; the receiver, R0, comes from R5. Where does R5 come from? Keep looking upwards until loc\_2691F726, as shown in figure 10-75.

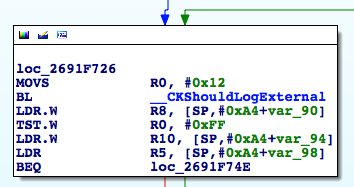


Figure 10- 75 loc\_2691f726

The instruction “LDR R5, [SP,#0xA4+var\_98]” decides R5. Well, what’s [SP,#0xA4+var\_98]? Do you remember how we’ve solved this kind of problems in section 10.2? Place the cursor on var\_98 and press “x” to view its cross references, as shown in figure 10-76.

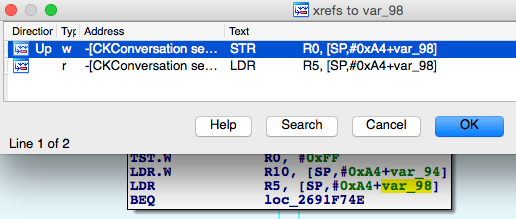


Figure 10- 76 Inspect cross references

Double click the first xref to jump to “STR R0, [SP,#0xA4+var\_98]”. Around here, R0 comes from [R6 chat]; R6 first appears in the beginning of [CKConversation sendMessage:onService:newComposition:], it’s “self”; so the receiver of “sendMessage:” is [self chat]. Back to figure 10-74, we can see the argument of “sendMessage:” is from R6. Go a little upwards to loc\_2691F6F4, R6 is set in “LDR R6, [SP,#0xA4+var\_80]”, as shown in figure 10-77.

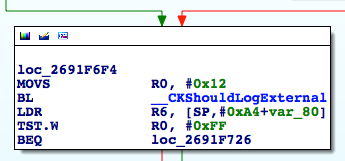


Figure 10- 77 loc\_2691f6f4

What’s next? We’ve performed the same operation just now, so I’ll leave some figures (from 10-78 to 10-80) rather than texts as references for you to follow:

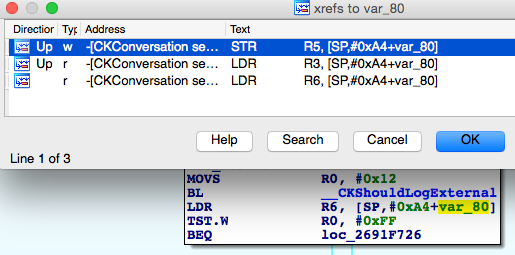


Figure 10- 78 Inspect cross references

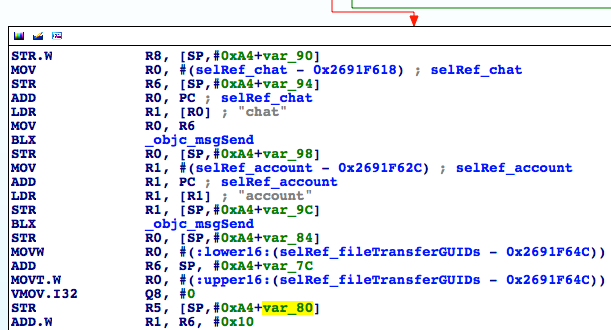


Figure 10- 79 [CKConversation setChat:]

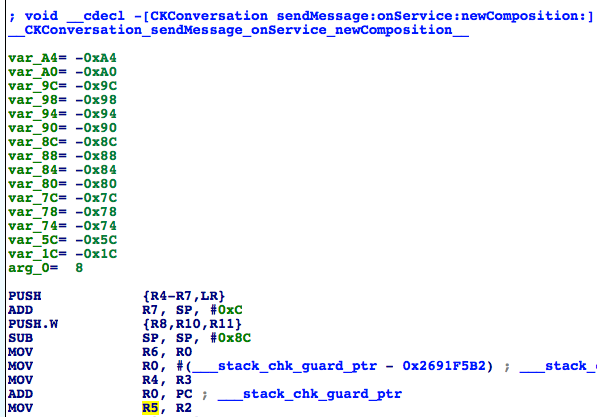


Figure 10- 80 [CKConversation sendMessage:onService:newComposition:]

So the argument of [[self chat] sendMessage:] is exactly the first argument of [self sendMessage:onService:newComposition:]. Well, what’re the types and values of [self chat] and the argument? We’ve gone out of clue in IDA, so it’s time to bring out LLDB.

First compose an iMessage, then set a breakpoint on the objc\_msgSend right under “sendMessage:” in figure 10-74, which is at the end of [CKConversation sendMessage:onService:newComposition:]. After that, press “Send” button to trigger the breakpoint:

Process 233590 stopped

\* thread #1: tid = 0x39076, 0x30ad1846 ChatKit`-[CKConversation sendMessage:onService:newComposition:] + 686, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x30ad1846 ChatKit`-[CKConversation sendMessage:onService:newComposition:] + 686

ChatKit`-[CKConversation sendMessage:onService:newComposition:] + 686:

-> 0x30ad1846: blx 0x30b3bf44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

0x30ad184a: movw r0, #49322

0x30ad184e: movt r0, #2541

0x30ad1852: add r0, pc

(lldb) p (char \*)$r1

(char \*) $0 = 0x32b26146 "sendMessage:"

(lldb) po $r0

<IMChat 0x5ef2ce0> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 3 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: F399B0B5-800F-47A4-A66C-72C43ACC0428 Unread Count: 0 Failure Count: 0]

(lldb) po $r2

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

The output contains exactly what we want: [IMChat sendMessage:IMMessage]. There’s one thing to mention: after printing out all necessary information, I’ve executed an extra “ni” command and heard a familiar “message sent” text tone. This phenomenon indicates that the operation of “send iMessage” is indeed performed inside [IMChat sendMessage:IMMessage]. Because the prefixes of IMChat and IMMessage are both IM, they come from a library other than ChatKit; the lowest level “send iMessage” function in ChatKit stops at [CKConversation sendMessage:onService:newComposition:]. We can confirm for now that if we're able to construct an IMChat object and an IMMessage object, we can successfully send an iMessage. Old problems solved, new problems occur: how do we compose these 2 objects? Let’s see if there’re any clues in class-dump headers.

To compose objects of IMChat and IMMessage from scratch, we need to see if there’re any constructors or initializers in their headers. Let’s start from IMChat.h and search for methods with the name “init”:

- (id)\_initWithDictionaryRepresentation:(id)arg1 items:(id)arg2 participantsHint:(id)arg3 accountHint:(id)arg4;

- (id)init;

- (id)\_initWithGUID:(id)arg1 account:(id)arg2 style:(unsigned char)arg3 roomName:(id)arg4 displayName:(id)arg5 items:(id)arg6 participants:(id)arg7;

Although they seem to be initializers, there’re various arguments, which we don’t know how to compose. The clues break, what’s next?

Do you still remember how we managed to find the receiver of “sendMessage:”? Yes, it’s [self chat]; self is a CKConversation object. Since [CKConversation chat] returns an IMChat object, let’s analyze this method in IDA to see if there’s any clue, as shown in figure 10-81.

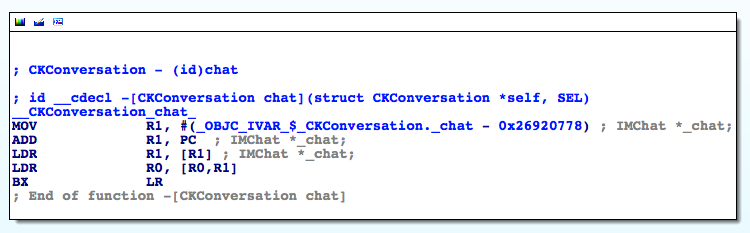


Figure 10- 81 [CKConversation chat]

[CKConversation chat] simply returns the an instance variable \_chat. This scenario is quite familiar, isn’t it? We’ve met a similar situation analyzing \_composeSendingService in figure 10-22. Once again, we have to let LLDB take the job for now. Delete this iMessage conversation (to delete this CKConversation obejct) and create a new iMessage (to create a new CKConversation object), then set a breakpoint on [CKConversation setChat:]. Press “Send” to trigger the breakpoint:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x30ad277c ChatKit`-[CKConversation setChat:], queue = 'com.apple.main-thread, stop reason = breakpoint 13.1

frame #0: 0x30ad277c ChatKit`-[CKConversation setChat:]

ChatKit`-[CKConversation setChat:]:

-> 0x30ad277c: movw r3, #55168

0x30ad2780: movt r3, #2541

0x30ad2784: add r3, pc

0x30ad2786: ldr r3, [r3]

(lldb) po $r2

<IMChat 0x1594f7e0> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]

(lldb) p/x $lr

(unsigned int) $20 = 0x30acf625

LR without offset is 0x30acf625 – 0xa1b2000 = 0x2691d625, it’s inside [CKConversation initWithChat:]. Since IMChat is the argument, to trace its source, we have to find out the method caller. Repeat the previous operations to recreate a new iMessage, then set a breakpoint at the beginning of [CKConversation initWithChat:] and trigger it:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x30acf5ec ChatKit`-[CKConversation initWithChat:], queue = 'com.apple.main-thread, stop reason = breakpoint 14.1

frame #0: 0x30acf5ec ChatKit`-[CKConversation initWithChat:]

ChatKit`-[CKConversation initWithChat:]:

-> 0x30acf5ec: push {r4, r5, r6, r7, lr}

0x30acf5ee: add r7, sp, #12

0x30acf5f0: push.w {r8, r10, r11}

0x30acf5f4: sub sp, #8

(lldb) po $r2

<IMChat 0x1470a520> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]

(lldb) p/x $lr

(unsigned int) $22 = 0x30a8d131

LR without offset is 0x30a8d131 – 0xa1b2000 = 0x268db131, which is inside [CKConversationList \_beginTrackingConversationWithChat:]. Again, it’s the argument, so let’s continue tracing the method caller:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x30a8d09c ChatKit`-[CKConversationList \_beginTrackingConversationWithChat:], queue = 'com.apple.main-thread, stop reason = breakpoint 15.1

frame #0: 0x30a8d09c ChatKit`-[CKConversationList \_beginTrackingConversationWithChat:]

ChatKit`-[CKConversationList \_beginTrackingConversationWithChat:]:

-> 0x30a8d09c: push {r4, r5, r6, r7, lr}

0x30a8d09e: mov r5, r0

0x30a8d0a0: movs r0, #25

0x30a8d0a2: add r7, sp, #12

(lldb) po $r2

<IMChat 0x15a326a0> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]

(lldb) p/x $lr

(unsigned int) $24 = 0x30a8d4f1

LR without offset is 0x30a8d4f1 – 0xa1b2000 = 0x268db4f1, which is inside [CKConversationList \_handleRegistryDidRegisterChatNotification:]; you’ll see in your IDA that this time IMChat is from [notification object] instead of the argument, which is a notification. Since this IMChat object is passed through a notification, to trace its source, we have to find the poster of this notification instead of the caller of [CKConversationList \_handleRegistryDidRegisterChatNotification:]. Let’s set a breakpoint on the base address of this method and take a look at the structure of notification:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x30a8d4ac ChatKit`-[CKConversationList \_handleRegistryDidRegisterChatNotification:], queue = 'com.apple.main-thread, stop reason = breakpoint 16.1

frame #0: 0x30a8d4ac ChatKit`-[CKConversationList \_handleRegistryDidRegisterChatNotification:]

ChatKit`-[CKConversationList \_handleRegistryDidRegisterChatNotification:]:

-> 0x30a8d4ac: push {r4, r5, r6, r7, lr}

0x30a8d4ae: add r7, sp, #12

0x30a8d4b0: push.w {r8, r10, r11}

0x30a8d4b4: sub.w r4, sp, #64

(lldb) po $r2

NSConcreteNotification 0x15934340 {name = \_\_kIMChatRegistryDidRegisterChatNotification; object = <IMChat 0x147c39f0> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]}

The name of the notification is “\_\_kIMChatRegistryDidRegisterChatNotification”. To find out its poster, a good solution is to grep the whole filesystem and see what binaries contain the notification name, as shown below:

FunMaker-5:~ root# grep -r \_handleRegistryDidRegisterChatNotification: /System/

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

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

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

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

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

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

The keyword appears in the cache. Naturally, let’s grep those decached files:

snakeninnys-MacBook:~ snakeninny$ grep -r \_\_kIMChatRegistryDidRegisterChatNotification /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5/

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

grep: /Users/snakeninny/Code/iOSSystemBinaries/8.1\_iPhone5//System/Library/Caches/com.apple.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

You may have already guessed from the results that both IMCore and ChatKit are in charge of iMessage related operations, but IMCore is lower level than ChatKit; ChatKit receives the commands from the user and hands them to IMCore for processing, then IMCore passes the result back to ChatKit for UI animation. By way of analogy, you can consider MobileSMS as a restaurant, ChatKit as the waiter and IMCore as the cook. Can you get it?

Naturally, drag and drop IMCore into IDA and search for “\_\_kIMChatRegistryDidRegisterChatNotification” globally, the results are shown in figure 10-82.

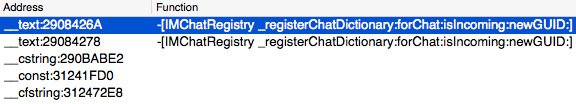


Figure 10- 82 Occurrences of “\_\_kIMChatRegistryDidRegisterChatNotification” in IDA

Good. Let’s double click the first row and take a look at its context, as shown in figure 10-83.

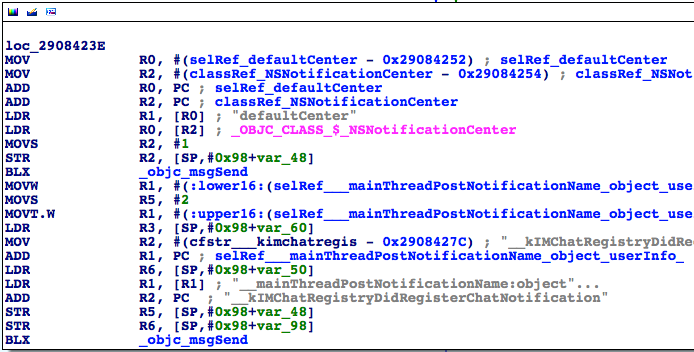


Figure 10- 83 loc\_2908423E

After seeing the keyword “PostNotification”, we know the notification that ChatKit received is right from here. Since IMChat is the 2nd argument, i.e. R3, and R3 comes from [SP, #0x98+var\_60]. You know what to do by referring to figure 10-84 and figure 10-85.

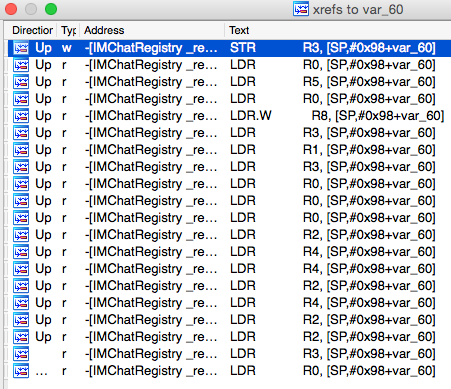


Figure 10- 84 Inspect cross references



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

According to the above figures, IMChat comes from the 2nd argument of [IMChatRegistry \_registerChatDictionary:forChat:isIncoming:newGUID:], whose caller is:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x33235944 IMCore`\_\_\_lldb\_unnamed\_function2048$$IMCore, queue = 'com.apple.main-thread, stop reason = breakpoint 17.1

frame #0: 0x33235944 IMCore`\_\_\_lldb\_unnamed\_function2048$$IMCore

IMCore`\_\_\_lldb\_unnamed\_function2048$$IMCore:

-> 0x33235944: push {r4, r5, r6, r7, lr}

0x33235946: add r7, sp, #12

0x33235948: push.w {r8, r10, r11}

0x3323594c: sub.w r4, sp, #64

(lldb) po $r3

<IMChat 0x147c7f30> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]

(lldb) p/x $lr

(unsigned int) $27 = 0x3323646f

LR without offset is 0x3323646f – 0xa1b2000 = 0x2908446F, which is located inside [IMChatRegistry \_registerChat:isIncoming:guid:]. Keep tracing the caller:

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x3323644c IMCore`\_\_\_lldb\_unnamed\_function2049$$IMCore, queue = 'com.apple.main-thread, stop reason = breakpoint 20.1

frame #0: 0x3323644c IMCore`\_\_\_lldb\_unnamed\_function2049$$IMCore

IMCore`\_\_\_lldb\_unnamed\_function2049$$IMCore:

-> 0x3323644c: push {r4, r5, r7, lr}

0x3323644e: add r7, sp, #8

0x33236450: sub sp, #8

0x33236452: movw r1, #9840

(lldb) po $r2

<IMChat 0x15972f20> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 0 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: (null) Unread Count: 0 Failure Count: 0]

(lldb) p/x $lr

(unsigned int) $30 = 0x33237173

LR without offset is 0x33237173 – 0xa1b2000 = 0x29085173, which is located inside [IMChatRegistry chatForIMHandle:]. Meanwhile, the 1st argument of [IMChatRegistry \_registerChat:isIncoming:guid:], i.e. IMChat, is from R5; at the end of [IMChatRegistry chatForIMHandle:], R5 appears as the return value. In other words, [IMChatRegistry chatForIMHandle:] returns an IMChat object! Further more, as the name suggests, IMChatRegistry is a class for registering chats, so getting an IMChat object from this class is quite reasonable. Old questions go, new questions come: How do we get an IMChatRegistry object and the argument of chatForIMHandle:? Let’s get to them one by one, starting from IMChatRegistry.

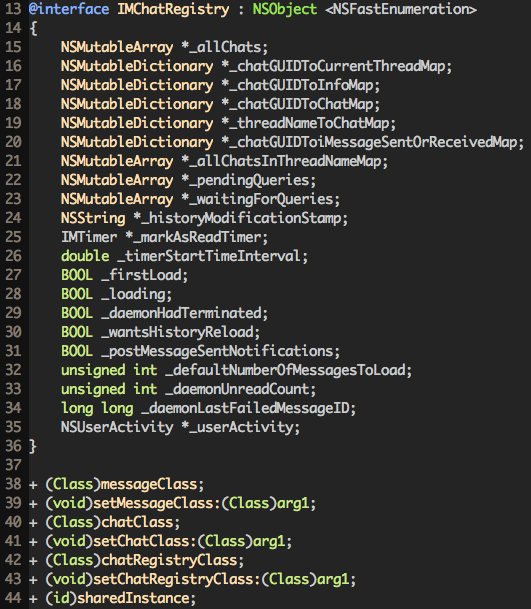


Figure 10- 86 IMChatRegistry.h

According to line 44, we know that IMChatRegistry is a singleton, we can get the registry by calling [IMChatRegistry sharedInstance]. So easy!

Next question, where does the argument of chatForIMHandle: come from? It definitely comes from its caller. It’s LLDB’s show time again.

Process 248623 stopped

\* thread #1: tid = 0x3cb2f, 0x33236d8c IMCore`\_\_\_lldb\_unnamed\_function2054$$IMCore, queue = 'com.apple.main-thread, stop reason = breakpoint 21.1

frame #0: 0x33236d8c IMCore`\_\_\_lldb\_unnamed\_function2054$$IMCore

IMCore`\_\_\_lldb\_unnamed\_function2054$$IMCore:

-> 0x33236d8c: push {r4, r5, r6, r7, lr}

0x33236d8e: add r7, sp, #12

0x33236d90: str r11, [sp, #-4]!

0x33236d94: sub sp, #20

(lldb) po $r2

[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account: P:+86PhoneNumber]

(lldb) p/x $lr

(unsigned int) $32 = 0x30a8dca5

LR without offset is 0x30a8dca5 – 0xa1b2000 = 0x268dbca5, which is not located inside IMCore anymore. Like we’ve just said, we’re jumping between IMCore and ChatKit, and ChatKit’s ASLR offset happens to be 0xa1b2000 too, so let’s head to ChatKit to see if 0x268dbca5 is there:

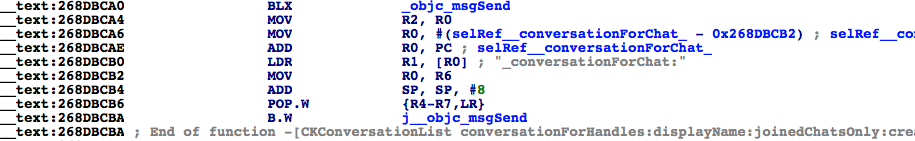


Figure 10- 87 [CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]

0x268dbca5 is inside [CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:], whose 1st argument is the source of the argument of chatForIMHandle:. Keep tracing the caller:

Process 292950 stopped

\* thread #1: tid = 0x47856, 0x30a8dc60 ChatKit`-[CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:], queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x30a8dc60 ChatKit`-[CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]

ChatKit`-[CKConversationList conversationForHandles:displayName:joinedChatsOnly:create:]:

-> 0x30a8dc60: push {r4, r5, r6, r7, lr}

0x30a8dc62: add r7, sp, #12

0x30a8dc64: sub sp, #8

0x30a8dc66: mov r6, r0

(lldb) po $r2

<\_\_NSArrayM 0x178d2290>(

[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account: P:+86PhoneNumber]

)

(lldb) p/x $lr

(unsigned int) $1 = 0x30a84efd

LR without offset is 0x30a84efd – 0xa1b2000 = 0x268d2efd, which is located inside [CKTranscriptController sendMessage:]. Can you believe it? We’ve walked through a big circle and returned to our starting point, which brings us a mixed feeling. Keep calm and carry on, let’s see how this NSArray is composed, as shown in figure 10-88.

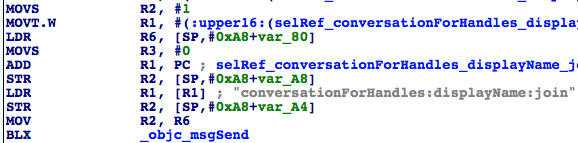


Figure 10- 88 Tracing the NSArray

R2 comes from R6, and R6 comes from [SP, #0xA8+var\_80]. The same pattern has reappeared, so as usual, I’ll replace text illustration with figure references, as shown in figure 10-89 and 10-90.

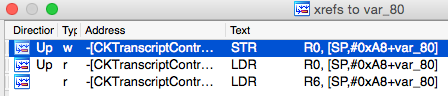


Figure 10- 89 Inspect cross references

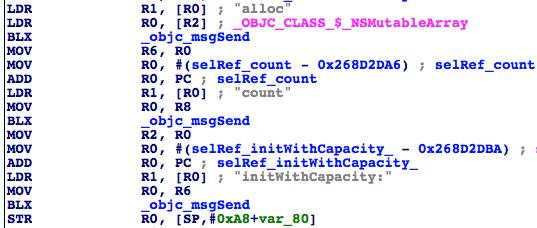


Figure 10- 90 [CKTranscriptController sendMessage:]

You may have already found that things are getting a little bit different. “STR R0, [SP,#0xA8+var\_80]” is just storing an initialized NSMutableArray into [SP, #0xA8+var\_80], it doesn’t contain any IMHandle yet. Hehe, since it’s an NSMutableArray, it can be extended by addObject:, which could happen in the 2nd “LDR R0, [SP,#0xA8+var\_80]” of figure 10-89. Let’s jump there for a look, as shown in figure 10-91.

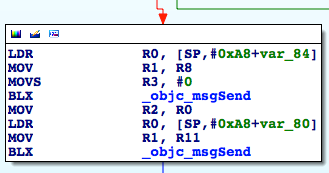


Figure 10- 91 Trace IMHandle

You’ll find it is indeed an addObject:, and by its context, you’ll see its argument comes from imHandleWithID:alreadyCanonical:. As the name suggests, it returns an IMHandle object. It’s getting closer, let’s set a breakpoint on the first objc\_msgSend in figure 10-91 to reconstruct the prototype of imHandleWithID:alreadyCanonical:.

Process 343388 stopped

\* thread #1: tid = 0x53d5c, 0x30a84e98 ChatKit`-[CKTranscriptController sendMessage:] + 516, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x30a84e98 ChatKit`-[CKTranscriptController sendMessage:] + 516

ChatKit`-[CKTranscriptController sendMessage:] + 516:

-> 0x30a84e98: blx 0x30b3bf44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

0x30a84e9c: mov r2, r0

0x30a84e9e: ldr r0, [sp, #40]

0x30a84ea0: mov r1, r11

(lldb) p (char \*)$r1

(char \*) $0 = 0x30b55fb4 "imHandleWithID:alreadyCanonical:"

(lldb) po $r0

IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Service: IMService[iMessage] Login: P:+86PhoneNumber Active: YES LoginStatus: Connected]

(lldb) po $r2

snakeninny@icloud.com

(lldb) p $r3

(unsigned int) $3 = 0

Both arguments are revealed; the 1st is my iMessage address, the 2nd is 0, i.e. NO in BOOL. The receiver is an IMAccount object, where is it from? As shown in figure 10-91, R0 comes from [SP, #0xA8+var\_84], so according to figure 10-92 and 10-93, IMAccount comes from [[IMAccountController sharedInstance] \_\_ck\_defaultAccountForService:[CKConversation sendingService]].

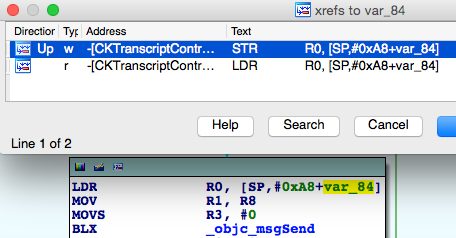


Figure 10- 92 Inspect cross references

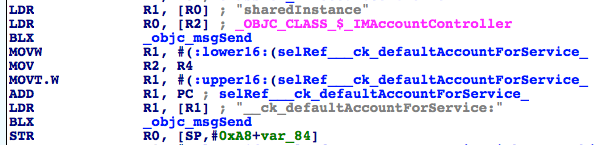


Figure 10- 93 [CKTranscriptController sendMessage:]

OK, let’s figure out what’s [CKConversation sendingService]. Set a breakpoint on the 2nd objc\_msgSend of figure 10-93 and trigger it:

Process 343388 stopped

\* thread #1: tid = 0x53d5c, 0x30a84e08 ChatKit`-[CKTranscriptController sendMessage:] + 372, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x30a84e08 ChatKit`-[CKTranscriptController sendMessage:] + 372

ChatKit`-[CKTranscriptController sendMessage:] + 372:

-> 0x30a84e08: blx 0x30b3bf44 ; symbol stub for: MarcoShouldLogMadridLevel$shim

0x30a84e0c: str r0, [sp, #36]

0x30a84e0e: movw r0, #23756

0x30a84e12: add r2, sp, #44

(lldb) p (char \*)$r1

(char \*) $4 = 0x30b55f95 "\_\_ck\_defaultAccountForService:"

(lldb) po $r2

IMService[iMessage]

(lldb) po [$r2 class]

IMServiceImpl

So it’s an IMServiceImpl object. How do we get such an object? In fact, we’ve already done this in section 10.2. Open IMServiceImpl.h, as shown in figuree 10-94.

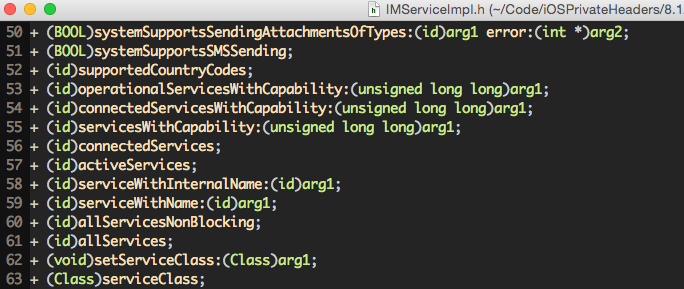


Figure 10- 94 IMServiceImpl.h

[IMServiceImpl iMessageService] and that's it. Reconfirm with Cycript:

cy# [IMServiceImpl iMessageService]

#"IMService[iMessage]"

By far, we’ve completely reversed the generation of an IMChat object. Let’s try it out in Cycript:

FunMaker-5:~ root# cycript -p MobileSMS

cy# service = [IMServiceImpl iMessageService]

#"IMService[iMessage]"

cy# account = [[IMAccountController sharedInstance] \_\_ck\_defaultAccountForService:service]

#"IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Service: IMService[iMessage] Login: P:+86PhoneNumber Active: YES LoginStatus: Connected]"

cy# handle = [account imHandleWithID:@"snakeninny@icloud.com" alreadyCanonical:NO]

#"[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No AB Match>) (Account: P:+86 MyPhoneNumber]"

cy# chat = [[IMChatRegistry sharedInstance] chatForIMHandle:handle]

#"<IMChat 0x15809000> [Identifier: snakeninny@icloud.com GUID: iMessage;-;snakeninny@icloud.com Persistent ID: snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-F58FBBB22363 Style: - State: 3 Participants: 1 Room Name: (null) Display Name: (null) Last Addressed: (null) Group ID: 6592DD84-4B34-4D54-BB40-E2AB17B2FC67 Unread Count: 0 Failure Count: 0]"

Gorgeous! To finally make it, we need to construct an IMMessage object for sending. Let’s move it now.

Open IMMessage.h, as shown in figure 10-95.

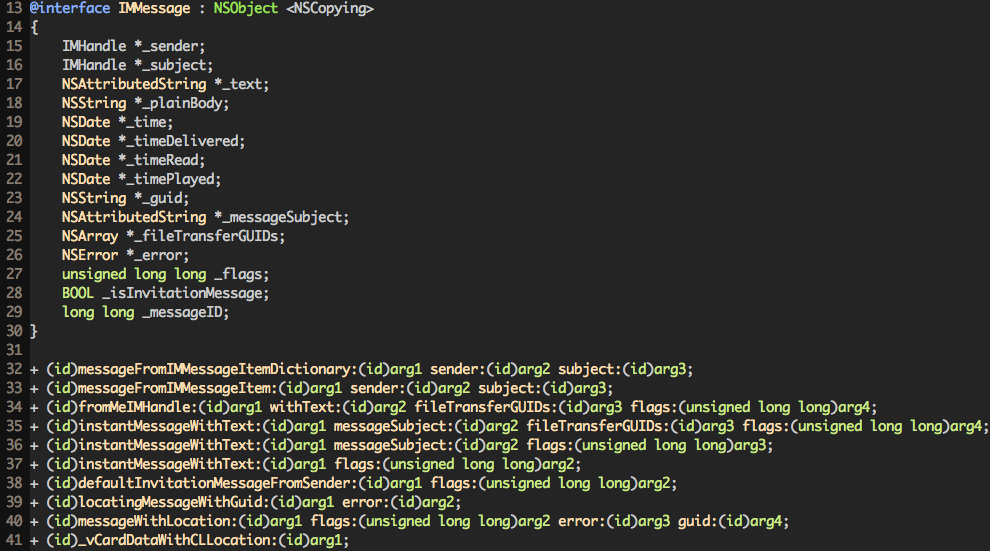


Figure 10- 95 IMMessage.h

There’re lots of class methods, among which “instantMessageWithText:flags:” gets our attention. Seems it returns an IMMessage object, but what’re the 2 arguments? The 1st may be an NSString object, what about “flag”? I don’t know if you still remember that earlier in this section, when we were locating [IMChat sendMessage:IMMessage], we’ve “po”ed an IMMessage object in LLDB:

(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)]

Although the “text” is “not loggable”, the “flag” is 100005. Let’s try it out in Cycript:

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

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

Cycript reminds us that NSString failed to respond to @selector(string). In other words, the 1st argument is not an NSString object, but instead some class that can respond to @selector(string). Let’s try to get some clues in figure 10-95, do you see “NSAttributedString \*\_text” in line 17? According to the official documents, NSAttributedString does have a property named “string”, whose getter is “- (NSString \*)string”, as shown in figure 10-96.

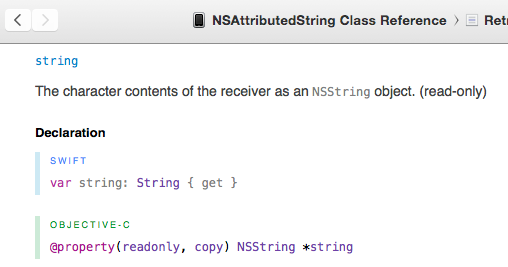


Figure 10- 96 [NSAttributedString string]

Let’s test “instantMessageWithText:flags:” with an NSAttributedString object:

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]

An IMMessage object appears, but as you can see, the value of flags is presented in hexadecimal instead of decimal. Only a tiny fix is needed to make it correct:

cy# message = [IMMessage instantMessageWithText:attributedString flags:1048581]

#"IMMessage[from=(null); msg-subject=(null); account:(null); flags=100005; subject='<< Message Not Loggable >>' text='<< Message Not Loggable >>' messageID: 0 GUID:'61012DF3-1C0F-4DED-9451-975E5771D493' date:'447412682.028256' date-delivered:'0.000000' date-read:'0.000000' date-played:'0.000000' empty: NO finished: YES sent: NO read: NO delivered: NO audio: NO played: NO from-me: YES emote: NO dd-results: NO dd-scanned: YES error: (null)]”

Everything should be ready by now. Last but not least:

cy# [chat sendMessage:message]

The effect is shown in figure 10-97. Let’s call it a day.

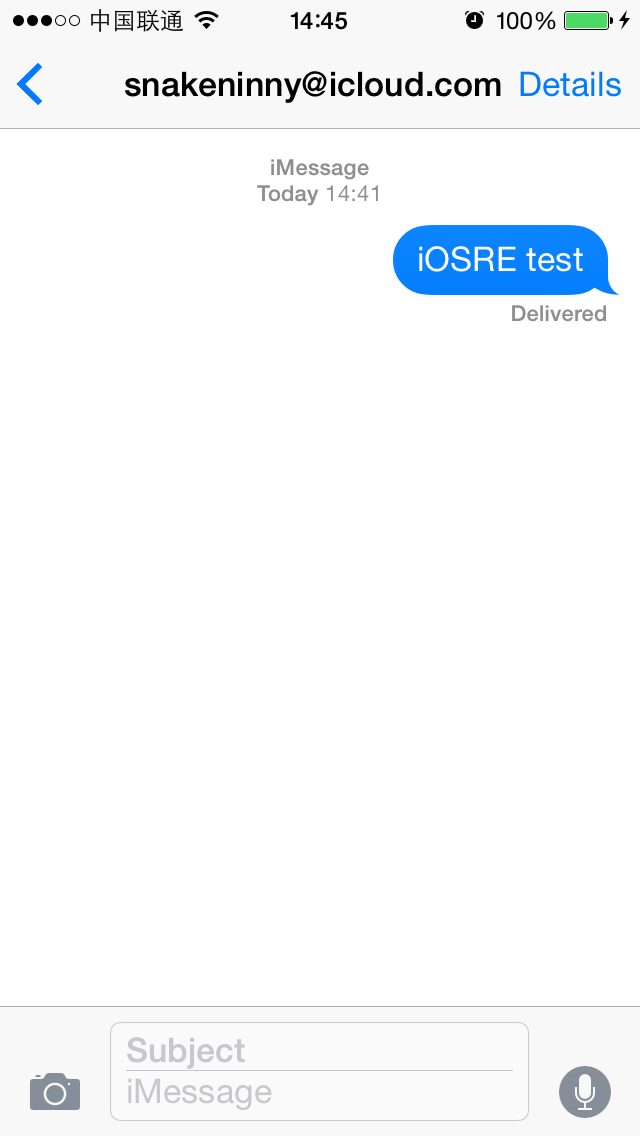


Figure 10- 97 iMessage delivered

10.4 Result Interpretation

Compared to previous practices, the reverse engineering methodology used in this chapter doesn’t change much, but the overall workload has increased tremendously; As for difficulty, this chapter is way harder than chapter 7 and 8, though they're all targeting system Apps. To reverse the functions of detecting and sending iMessages, our general thoughts are as follows.

1. Cut into the code via UI

The changing from “Text Message” to “iMessage”, green color to blue color, and “Send” button itself are all UI visualizations produced by programs. As long as we can describe what we see on UI, we can cut into the App from there. In this chapter, our cut-in points are message placeholder and “Send” button. Their UI functions can be easily located with Cycript, and are helpful in further analysis.

2. Browse and test class-dump headers to find interesting dots

Objective-C headers are clearly organized, methods are explicitly named. Their high readability is the perfect place for us to look for reverse engineering clues. Testing private methods, properties and instance variables with Cycript can be really helpful when we want to know a certain private class better. In this chapter, when we came across some suspicious variables, we didn’t strictly analyze them with IDA and LLDB, but by only browsing corresponding headers, guessing their prototypes and usages, then testing with Cycript to achieve our goals. The famous leader in my country Deng Xiaoping once said:"It doesn't matter whether a cat is white or black, as long as it catches mice", which applies to iOS reverse engineering too.

3. Analyze functions in IDA to connect the dots and form a plane

As to inspect the implementation of a function, IDA is one of the most handy tools. Cross references, addresses jumping, global search and whatever, they help us quickly locate what we’re interested in, as well browse the context to form an overall understanding. In detecting iMessages, we’ve straightened out the relationships of [CKMessageEntryView updateEntryView], [CKPendingConversation sendingService], [CKPendingConversation composeSendingService], IMChatCalculateServiceForSendingNewCompose and so on; among them IMChatCalculateServiceForSendingNewCompose is a C function, hence is immune to class-dump. In sending iMessages, we’ve traced from the high level method [CKTranscriptController sendComposition:CKComposition], through [CKTranscriptController \_startCreatingNewMessageForSending:], [CKConversation sendMessage:newComposition:] and [CKConversation sendMessage:onService:newComposition:], to the low level method [IMChat sendMessage:IMMessage]. All these operations are picking call chains from a plane according to keywords and clues provided by IDA. That’s a lot of handwork, but thanks to the assistance of IDA, the workload is totally acceptable.

4. Pick out the exact line, i.e. call chain from the plane with LLDB

LLDB plays a significant role throughout the whole chapter. Although we’ve tried to limit its usage in section 10.3, we have to bring it out when tracing function callers and dynamically analyzing arguments. Compared with GDB, LLDB is more iOS supportive, there’re rare crashes and bugs; it works great on Objective-C objects, making our debugging much smoother. When we were working on the detecting and sending of iMessages, LLDB helped us clarify great amounts of details; based on the careful analysis of tightly related data sources, we’ve abstracted a short piece of the working principles and designing ideas of iMessage: MobileSMS plays the role of a post office; its building materials, office equipments and clerks are all from ChatKit, while IMCore is the postman. When I have a letter to send, I’ll go to the post office and put the letter in the mailbox. Then a clerk will sort the letters out and hand them to the postman; later the postman will give feedback of the delivery progress and result to the clerk, who are in charge of informing me what’s happening to my letter. This kind of closed-loop service is very Apple-ish; MobileSMS, ChatKit and IMCore play different roles, bringing Apple fans terrific user experiences. If we can learn how Apple design and implement all kinds of services via iOS reverse engineering, put them together and make them our own, it’ll bring dramatically improvement to the elegance, design and robustness of our products, which is unattainable by only reading the official documents.

10.5 Tweak writing

After prototyping the tweak with Cycript, coding with Theos is just physical labor without much thinking. We’ll add 2 methods to SMSApplication in MobileSMS, namely “- (int)madridStatusForAddress:(NSString \*)address” and “- (void)sendMadridMessageToAddress:(NSString \*)address withText:(NSString \*)text” then test them with Cycript. Let’s move it!

10.5.1 Create tweak project “iOSREMadridMessenger” using Theos

The Theos commands are as follows:

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

NIC 2.0 - New Instance Creator

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

[1.] iphone/application

[2.] iphone/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 Compose iOSREMadridMessenger.h

We’ve made use of multiple private classes and methods in previous sections, so we need to provide their definitions to avoid any compiler warning or error. Of course, the contents of iOSREMadridMessenger.h don’t come from nowhere, it’s composed by picking snippets from other class-dump headers, forming a “select header”. The finalized iOSREMadridMessenger.h looks like this:

@interface IDSIDQueryController

+ (instancetype)sharedInstance;

- (NSDictionary \*)\_currentIDStatusForDestinations:(NSArray \*)arg1 service:(NSString \*)arg2 listenerID:(NSString \*)arg3;

@end

@interface IMServiceImpl : NSObject

+ (instancetype)iMessageService;

@end

@class IMHandle;

@interface IMAccount : NSObject

- (IMHandle \*)imHandleWithID:(NSString \*)arg1 alreadyCanonical:(BOOL)arg2;

@end

@interface IMAccountController : NSObject

+ (instancetype)sharedInstance;

- (IMAccount \*)\_\_ck\_defaultAccountForService:(IMServiceImpl \*)arg1;

@end

@interface 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 Edit Tweak.xm

The finalized Tweak.xm looks like this:

#import "iOSREMadridMessenger.h"

%hook SMSApplication

%new

- (int)madridStatusForAddress:(NSString \*)address

{

NSString \*formattedAddress = nil;

if ([address rangeOfString:@"@"].location != NSNotFound) formattedAddress = [@"mailto:" stringByAppendingString:address];

else formattedAddress = [@"tel:" stringByAppendingString:address];

NSDictionary \*status = [[IDSIDQueryController sharedInstance] \_currentIDStatusForDestinations:@[formattedAddress] service:@"com.apple.madrid" listenerID:@"\_\_kIMChatServiceForSendingIDSQueryControllerListenerID"];

return [status[formattedAddress] intValue];

}

%new

- (void)sendMadridMessageToAddress:(NSString \*)address withText:(NSString \*)text

{

IMServiceImpl \*service = [IMServiceImpl iMessageService];

IMAccount \*account = [[IMAccountController sharedInstance] \_\_ck\_defaultAccountForService:service];

IMHandle \*handle = [account imHandleWithID:address alreadyCanonical:NO];

IMChat \*chat = [[IMChatRegistry sharedInstance] chatForIMHandle:handle];

NSAttributedString \*attributedString = [[NSAttributedString alloc] initWithString:text];

IMMessage \*message = [IMMessage instantMessageWithText:attributedString flags:1048581];

[chat sendMessage:message];

[attributedString release];

}

%end

10.5.4 Edit Makefile and control files

The finalized Makefile looks like this:

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"

The finalized control looks like this:

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 Test with Cycript

Compile and install the tweak, then ssh into iOS and execute the following commands:

FunMaker-5:~ root# cycript -p MobileSMS

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

1

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

The madrid status of “snakeninny@icloud.com” is 1, indicating it supports iMessage; plus the iMessage is sent and delivered like silk, as shown in figure 10-98.



Figure 10- 98 iMessage sent

If you’ve been so far, feel free to iMessage me at “snakeninny@gmail.com” with iOSREMadridMessenger and share your joy :)

10.6 Conclusion

Being one of the key services since iOS 5, iMessage is greatly enhanced in iOS 8. Whether it’s plain text, image, audio, or even video, iMessage can handle them all. Although detecting and sending iMessages is only a tip of iceberg in all iMessage operations, we’ve switched among IDS, ChatKit and IMCore, as well felt the high complexity of the entire iMessage service. According to our analysis, the class in charge of iMessage accounts is IMAccountController; the class of iMessage accounts is IMAccount; recipient class is IMHandle; a message conversation is an IMChat or CKConversation object; IMChatRegistry is responsible for managing all IMChats; an iMessage is an IMMessage or CKComposition object. For those IM developers, the design of iMessage can be a precious reference.

If you’re still unsatisfied with this chapter and want to dig deeper into iMessage, try the following bonuses:

* Send SMS programmatically (Tip: just replace IMServiceImpl would be alright).
* Send iMessage with ChatKit (Tip: you can get a CKConversationfrom object from an IMChat object).
* Send iMessage inside SpringBoard (Tip: calling [IMChat sendMessage:IMMessage] inside SpringBoard fails to send messages, because SpringBoard lacks certain “capabilities”).

If you can digest the contents of this chapter inside-out, and prototype the tweak without referring to the book, congratulations, you’re a fairly outstanding iOS reverse engineer now, you are qualified and encouraged to step toward a higher goal, say, jailbreak. Before you begin the new journey, share your knowledge and experiences with us on <http://bbs.iosre.com> to help build the jailbreak community. Thank you!