## 6.2 tweak的编写套路

## 6.2 Advanced methodology of writing a tweak

在第5章的“tweak的编写套路”一节里，归纳总结了5个步骤，分别是寻找灵感、定位目标文件、定位目标函数、测试函数功能，以及解析函数参数。这些步骤没问题，但“定位目标函数”这个关键环节的水分太大——在class-dump的头文件里搜索自己感兴趣的关键词，可以称之为“定位目标函数”吗？非也。

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

一般情况下，一个软件之所以能引起我们的兴趣，无非是2个元素：功能和数据。如果发现了自己感兴趣的功能，但class-dump的头文件里找不到可疑的关键词，怎么办？如果看到了自己感兴趣的数据，我们该怎么去寻找它的生成算法？对此，class-dump一点辙都没有。因此，通过class-dump及关键词搜索的方式只是“定位目标函数”中的一种情况，不能以偏概全。那么针对更普遍的情况，该怎么定位目标函数呢？

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

我们感兴趣的功能和数据，都是以软件中产生的某种现象为形式，直观地呈现在我们面前的，我们能看到，感受到。例如，图6-10所示的是邮件应用（以下简称Mail），右下角的那个书写图标代表了“编写邮件”功能；图6-11所示的是设置应用中的电话设置（以下简称MobilePhoneSettings），第一个cell中的内容代表了“本机号码”数据。功能是由函数提供的，数据是由函数生成的，也就是说，外在现象的内在本质，其实是函数。所以，“定位目标函数”实际上是如何从我们感兴趣的外在现象，定位到其内在函数的过程。

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

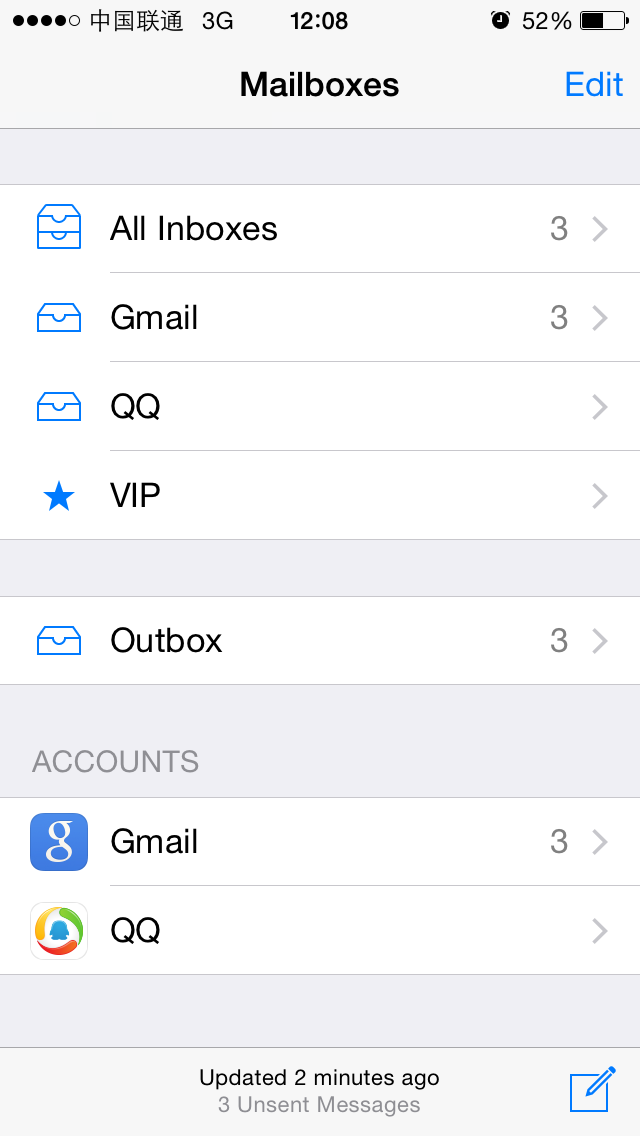


Figure 6- 10 Mail



Figure 6- 11 MobilePhoneSettings

面对这样的需求，class-dump明显已经不够用了。好在我们现在了解了Cycript、IDA、LLDB的基本用法，对ARM汇编也有了初步印象，有了它们的辅助，“定位目标函数”变得有规律可循了。iOS上最常见的是一个个App，我们对这种类型的文件也最熟悉，把它们作为初学阶段的练习对象再合适不过了。接下来，就以App为目标，用ARM汇编级别的逆向工程完善 “定位目标函数”环节，强化tweak的编写套路。

Facing such demands, class-dump is quite helpless. Luckily, we have already learned how to use Cycript, IDA and LLDB, and gained some basic knowledge about ARM assembly; with their help, there are patterns to follow for “locating target functions”. For most of us, among all iOS software, we know Apps the best, so if we’re choosing something to be our junior reverse targets, nothing is more appropriate than Apps. So in the following sections, we will take Apps as examples, and try to refine “locate target functions” with ARM level reverse engineering, as well enhance the methodology of writing a tweak.

### 6.2.1 从现象切入App，找出UI函数

## 6.2.1 Cut into the target App and find the UI function

对于App来说，我们感兴趣的现象往往体现在UI上，UI展示了函数的执行过程和结果。函数和UI之间的关联非常紧密，如果能拿到感兴趣的UI对象，就可以找到它所对应的函数，我们称该函数为UI函数。这个过程，一般是利用Cycript，结合UIView中的神奇私有函数recursiveDescription和UIResponder中的nextResponder来实现的。下面先以Mail为例讲解过程，然后把总结出来的方法用在MobilePhoneSettings上加深印象。这部分内容是在iPhone 5，iOS 8.1中完成的。

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

### 用Cycript注入Mail

#### 1. Inject Cycript into Mail

先用dumpdecrypted小节中提及的技巧，定位Mail的进程名并注入：

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

FunMaker-5:~ root# ps -e | grep /Applications

363 ?? 0:06.94 /Applications/MobileMail.app/MobileMail

596 ?? 0:01.50 /Applications/MessagesNotificationViewService.app/MessagesNotificationViewService

623 ?? 0:08.50 /Applications/InCallService.app/InCallService

713 ttys000 0:00.01 grep /Applications

FunMaker-5:~ root# cycript -p MobileMail

### 查看当前界面的UI层次结构，定位“编写邮件”按钮

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

UIView中的私有函数recursiveDescription可以返回这个view的UI层次结构。一般来说，当前界面是由至少一个UIWindow构成的，而UIWindow继承自UIView，因此我们可以利用这个私有函数来查看当前界面的UI层次结构。它的用法是这样的：

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

cy# ?expand

expand == true

首先执行Cycript的?expand命令来开启expand功能，Cycript会把格式符号翻译成相应的格式，如“\n”会被翻译成一个换行，让输出的可读性更高。

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

cy# [[UIApp keyWindow] recursiveDescription]

UIApp是[UIApplication sharedApplication]的简写，两者等价。调用上面的方法即可打印keyWindow的视图结构，输出类似下面的信息：

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

@"<UIWindow: 0x14587a70; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x147166b0>; layer = <UIWindowLayer: 0x14587e30>>

| <UIView: 0x146e6180; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x146e98d0>; layer = <CALayer: 0x146e61f0>>

| | <UIView: 0x146e5f60; frame = (0 0; 320 568); layer = <CALayer: 0x1460ec40>>

| | | <\_MFActorItemView: 0x14506a30; frame = (0 0; 320 568); layer = <CALayer: 0x14506c10>>

| | | | <UIView: 0x145074b0; frame = (-0.5 -0.5; 321 569); alpha = 0; layer = <CALayer: 0x14507520>>

| | | | <\_MFActorSnapshotView: 0x14506f70; baseClass = UISnapshotView; frame = (0 0; 320 568); clipsToBounds = YES; hidden = YES; layer = <CALayer: 0x145071c0>>

……

| | <MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568); userInteractionEnabled = NO; gestureRecognizers = <NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>

| | | <UIScrollView: 0x146bfa90; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x146e1e90>; layer = <CALayer: 0x146c8740>; contentOffset: {0, 0}; contentSize: {320, 77.5}>

| | | <\_TabGradientView: 0x146e7010; frame = (-320 -508; 960 568); alpha = 0; userInteractionEnabled = NO; layer = <CAGradientLayer: 0x146e7d80>>

| | | <UIView: 0x146e29c0; frame = (-10000 568; 10320 10000); layer = <CALayer: 0x146e2a30>>"

keyWindow的每个subview及二级subview的description会被完整展示在<……>里，包括每个view对象在内存中的地址，以及它的坐标、尺寸等基本信息。其中，缩进的多少体现了视图间的关系，同一缩进量的视图是平级的，如最下面的UIScrollView、\_TabGradientView及UIView；缩进少的视图是缩进多的视图的superview，如UIScrollView、\_TabGradientView和UIView都是MFTiltedTabView的subview。通过Cycript的“#”操作符，就可以拿到这个window上的任意view，如：

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

cy# tabView = #0x146e1af0

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

当然，也可以通过UIApplication和UIView的其他方法，获取我们感兴趣的其他view，如：

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

cy# [UIApp windows]

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

上面的代码可以拿到这个App的所有window；

The above code can get all windows of this App:

cy# [#0x146e1af0 subviews]

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

cy# [#0x146e29c0 superview]

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

上面的代码可以拿到subview和superview。总之，综合利用这几个函数，就可以拿到UI上的任意view，为下一步操作奠定基础。

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

要定位“编写邮件”按钮，就要寻找跟这个按钮相关的控件。对此，一般采用的方法是排查法，对于形如<UIView: viewAddress; …>的view来说，对其逐个调用[#viewAddress setHidden:YES]函数，UI上消失的那个控件就可以跟它对应起来。当然，一些小技巧可以加快排查的速度——因为这个按钮的左边是上下两排字，所以可以猜测，这个按钮跟两排字是共用一个superview的，如果找到这个superview，那么只排查这个superview的subview就好了，减少了我们的工作量。因为文字一般是会出现在description里的，所以可在recursiveDescription里搜索“3 Unsent Messages”：

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

| | | | | | | | <MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44); opaque = NO; autoresize = W+H; layer = <CALayer: 0x146c8840>>

| | | | | | | | | <UILabel: 0x14609610; frame = (40 21.5; 102 13.5); text = '3 Unsent Messages'; opaque = NO; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x146097f0>>

从而获取到它的superview，即MailStatusUpdateView。如果按钮是MailStatusUpdateView的一个subview，那么通过调用setHidden:函数隐藏MailStatusUpdateView，按钮也会被隐藏。下面试试看：

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

cy# [#0x146e6060 setHidden:YES]

执行之后，发现两排字被隐藏了，而按钮没有被隐藏，如图6-12所示。

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

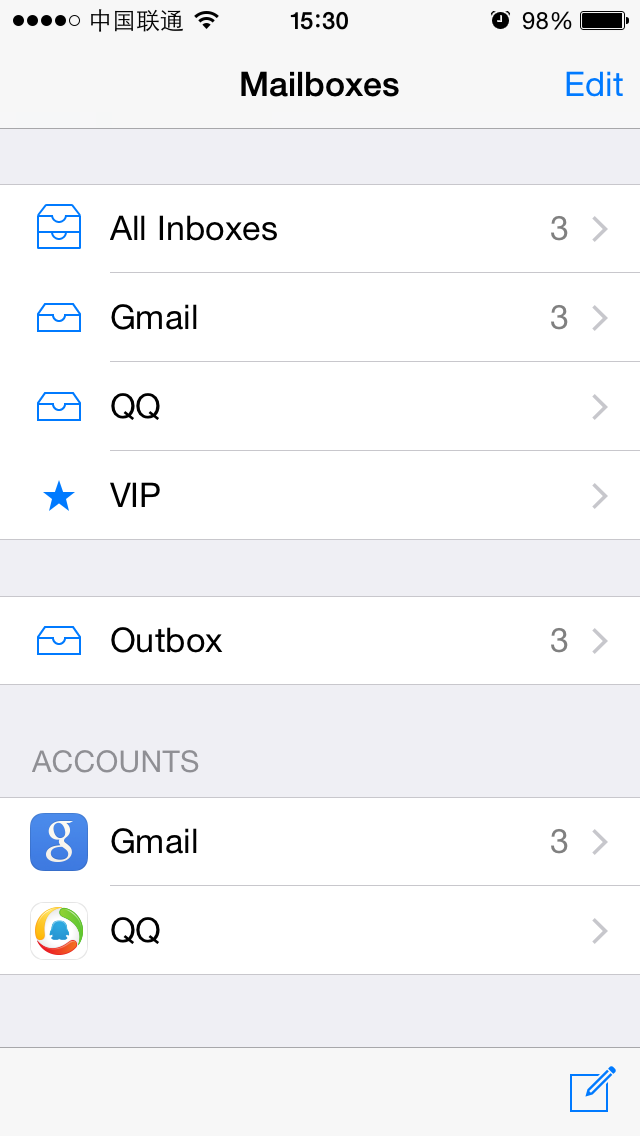


图6- 12 两排字被隐藏

Figure 6-12 Two lines of sentences are hidden

这说明MailStatusUpdateView的级别低于或等于按钮所在的view，对吧？因此，接下来要做的就是排查MailStatusUpdateView的superview。从recursiveDescription可知，它的superview是MailStatusBarView：

This indicates that the level of MailStatusUpdateView is lower than or equal to the button, right? So, next let’s check the superview of MailStatusUpdateView. From recursiveDescription, we realize that its superview is MailStatusBarView:

| | | | | | | <MailStatusBarView: 0x146c4110; frame = (69 0; 182 44); opaque = NO; autoresize = BM; layer = <CALayer: 0x146f9f90>>

| | | | | | | | <MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44); opaque = NO; autoresize = W+H; layer = <CALayer: 0x146c8840>>

试着隐藏它，看看按钮受不受影响：

Try to hide it and see if the button disappear:

cy# [#0x146e6060 setHidden:NO]

cy# [#0x146c4110 setHidden:YES]

效果跟刚才一样，两排字被隐藏，按钮还是没有被隐藏，说明MailStatusBarView的级别仍然不够高，继续找它的superview，即UIToolBar：

It’s disappointing; two sentences are hidden but not the button, which means that the level of MailStatusBarView is still not high enough, let’s keep looking for its superview, i.e. UIToolBar:

| | | | | | <UIToolbar: 0x146f62a0; frame = (0 524; 320 44); opaque = NO; autoresize = W+TM; layer = <CALayer: 0x146f6420>>

| | | | | | | <\_UIToolbarBackground: 0x14607ed0; frame = (0 0; 320 44); autoresize = W; userInteractionEnabled = NO; layer = <CALayer: 0x14607d40>>

| | | | | | | | <\_UIBackdropView: 0x15829590; frame = (0 0; 320 44); opaque = NO; autoresize = W+H; userInteractionEnabled = NO; layer = <\_UIBackdropViewLayer: 0x158297e0>>

| | | | | | | | | <\_UIBackdropEffectView: 0x14509020; frame = (0 0; 320 44); clipsToBounds = YES; opaque = NO; autoresize = W+H; userInteractionEnabled = NO; layer = <CABackdropLayer: 0x145a68d0>>

| | | | | | | | | <UIView: 0x147335c0; frame = (0 0; 320 44); hidden = YES; opaque = NO; autoresize = W+H; userInteractionEnabled = NO; layer = <CALayer: 0x145f3ab0>>

| | | | | | | <UIImageView: 0x14725730; frame = (0 -0.5; 320 0.5); autoresize = W+BM; userInteractionEnabled = NO; layer = <CALayer: 0x1472be40>>

| | | | | | | <MailStatusBarView: 0x146c4110; frame = (69 0; 182 44); opaque = NO; autoresize = BM; layer = <CALayer: 0x146f9f90>>

模仿之前的操作，隐藏UIToolBar：

Let’s repeat the operation to hide UIToolBar:

cy# [#0x146c4110 setHidden:NO]

cy# [#0x146f62a0 setHidden:YES]

效果如图6-13所示。

The effect is shown in figure 6-13:

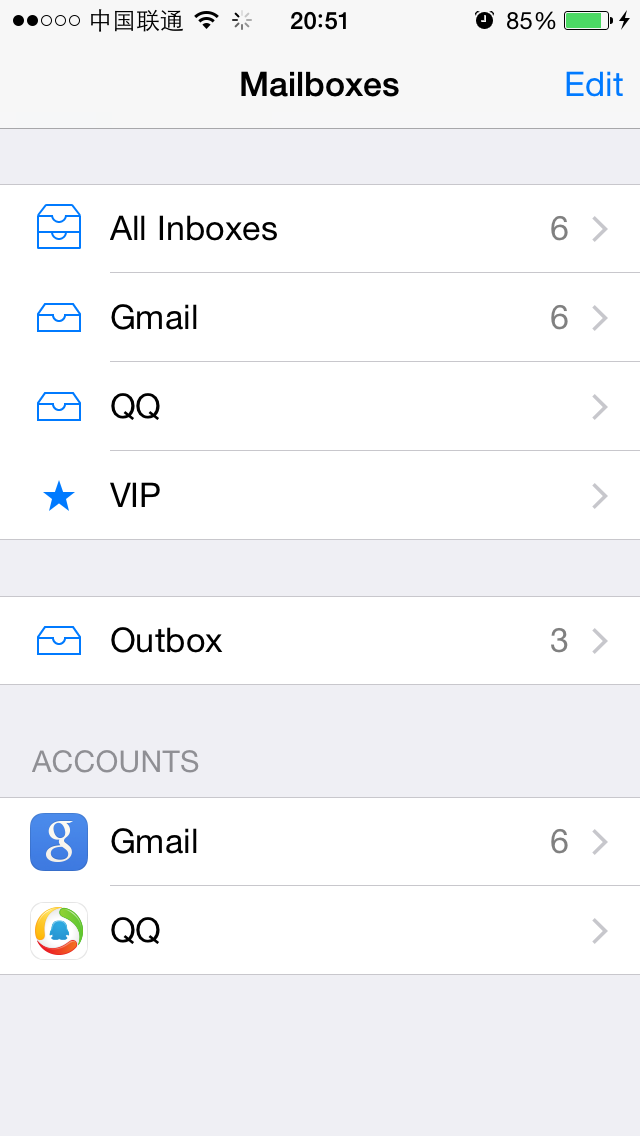


图6- 13 UIToolBar被隐藏

Figure 6-13 UIToolBar is hidden

此时，按钮被隐藏了，说明按钮是这个UIToolBar的一个subview。在这个UIToolBar的subview里面寻找带有“button”字样的view，很容易就定位到了UIToolbarButton：

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

| | | | | | | <MailStatusBarView: 0x146c4110; frame = (69 0; 182 44); opaque = NO; autoresize = BM; layer = <CALayer: 0x146f9f90>>

| | | | | | | | <MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44); opaque = NO; autoresize = W+H; layer = <CALayer: 0x146c8840>>

| | | | | | | | | <UILabel: 0x14609610; frame = (40 21.5; 102 13.5); text = '3 Unsent Messages'; opaque = NO; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x146097f0>>

| | | | | | | | | <UILabel: 0x145f3020; frame = (43 8; 96.5 13.5); text = 'Updated Just Now'; opaque = NO; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x145f2e50>>

| | | | | | | <UIToolbarButton: 0x14798410; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>

下面看看它是不是“编写邮件”按钮，命令如下：

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

cy# [#0x146f62a0 setHidden:NO]

cy# [#0x14798410 setHidden:YES]

按钮被成功隐藏，如图6-14所示。

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

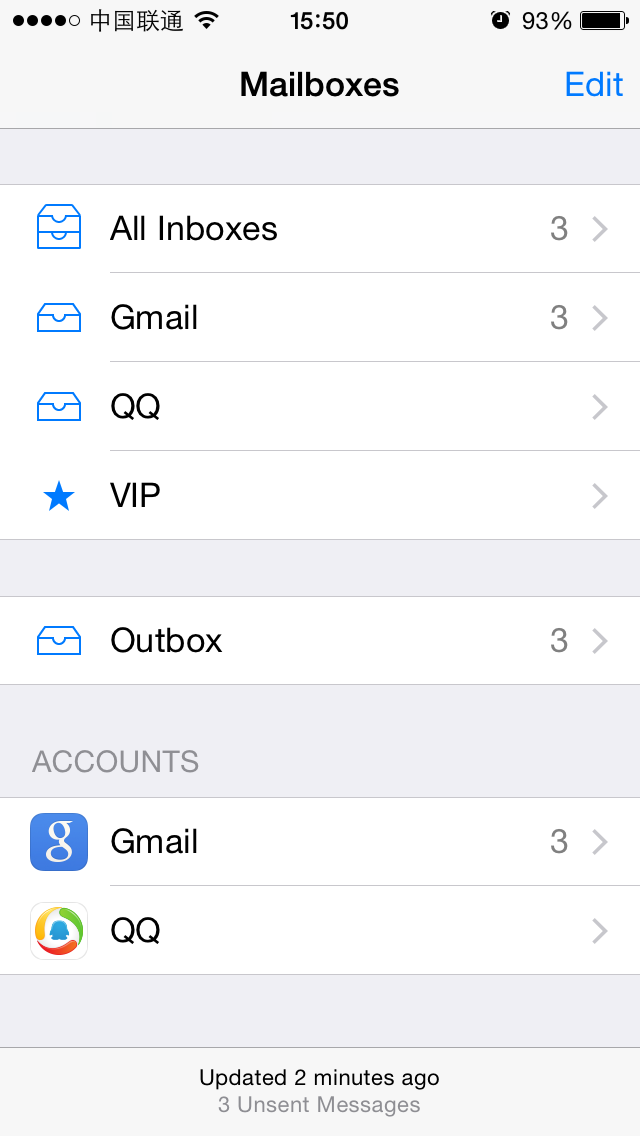


图6- 14 按钮被隐藏

Figure 6-14 Button is hidden

至此，我们成功定位到了“编写邮件”按钮，它的description是<UIToolbarButton: 0x14798410; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>。接下来要找出它的UI函数。

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

### 找出“编写邮件”按钮的UI函数

#### 3. Find out UI function of “compose” button

按钮的UI函数，就是点击它之后的响应函数。给UIView对象加上响应函数，一般是通过[UIControl addTarget:action:forControlEvents:]实现的（笔者还没有碰到过例外）；而UIControl提供了一个actionsForTarget:forControlEvent:方法，来获得这个UIControl的响应函数。基于这个条件，只要第2步里定位到的view是UIControl的子类（笔者也还没有碰到过例外），就可以通过这种方式找到它的响应函数。具体到书中的例子，是这样操作的：

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

cy# button = #0x14798410

#"<UIToolbarButton: 0x14798410; frame = (285 0; 23 44); hidden = YES; opaque = NO; gestureRecognizers = <NSArray: 0x14799510>; layer = <CALayer: 0x14798510>>"

cy# [button allTargets]

[NSSet setWithArray:@[#"<ComposeButtonItem: 0x14609d00>"]]]

cy# [button allControlEvents]

64

cy# [button actionsForTarget:#0x14609d00 forControlEvent:64]

@["\_sendAction:withEvent:"]

因此，按下“编写邮件”按钮，Mail会调用[ComposeButtonItem \_sendAction:withEvent:]，我们成功找到了它的响应函数。用Cycript注入，定位UI控件，找出UI函数，就这么简单。如果你还不理解，下面会用类似的套路分析MobilePhoneSettings，请注意总结。

Therefore, after tapping “compose” button, Mail calls [ComposeButtonItem \_sendAction:withEvent:], we have successfully found the response method. Inject with Cycript, locate UI control object, and then find out its UI function, it’s fairly easy as you see. If you still don’t get it, we will repeat these steps on MobilePhoneSettings, please pay attention.

### 用Cycript注入MobilePhoneSettings

### 4. Inject Cycript into MobilePhoneSettings

下面的操作大家应该都很熟悉了：

You should be very familiar with the following operation for now:

FunMaker-5:~ root# ps -e | grep /Applications

596 ?? 0:01.50 /Applications/MessagesNotificationViewService.app/MessagesNotificationViewService

623 ?? 0:08.55 /Applications/InCallService.app/InCallService

748 ?? 0:01.36 /Applications/MobileMail.app/MobileMail

750 ?? 0:01.82 /Applications/Preferences.app/Preferences

755 ttys000 0:00.01 grep /Applications

FunMaker-5:~ root# cycript -p Preferences

注意，桌面上Settings的应用名叫Preferences，下面会频繁出现，请大家留意。

Be careful, Settings App’s name is Preferences. It will show frequently in this chapter, please keep an eye.

### 查看当前界面的UI层次结构，定位第一个cell

#### 5. Examine the view hierarchy of “Phone” view, and locate the top cell

打印出当前界面的UI层次结构：

As usual, let’s take a look at the view hierarchy first:

cy# ?expand

expand == true

cy# [[UIApp keyWindow] recursiveDescription]

@"<UIWindow: 0x17d62e00; frame = (0 0; 320 568); autoresize = H; gestureRecognizers = <NSArray: 0x17d589b0>; layer = <UIWindowLayer: 0x17d21c60>>

| <UILayoutContainerView: 0x17d86620; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer: 0x17d863b0>>

| | <UIView: 0x17ef2430; frame = (0 0; 320 0); layer = <CALayer: 0x17ef24a0>>

| | <UILayoutContainerView: 0x17d7eb80; frame = (0 0; 320 568); clipsToBounds = YES; gestureRecognizers = <NSArray: 0x17eb6400>; layer = <CALayer: 0x17d7ed60>>

……

| | | | | | | | | | | <PSTableCell: 0x17f92890; baseClass = UITableViewCell; frame = (0 35; 320 44); text = 'My Number'; autoresize = W; tag = 2; layer = <CALayer: 0x17f92a60>>

| | | | | | | | | | | | <UITableViewCellContentView: 0x17f92ad0; frame = (0 0; 287 43.5); gestureRecognizers = <NSArray: 0x17f92ce0>; layer = <CALayer: 0x17f92b40>>

| | | | | | | | | | | | | <UITableViewLabel: 0x17f92d30; frame = (15 12; 90 20.5); text = 'My Number'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x17f92df0>>

| | | | | | | | | | | | | <UITableViewLabel: 0x17f93060; frame = (132.5 12; 152.5 20.5); text = '+86PhoneNumber'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x17f93120>>

很容易就可以定位到显示“+86PhoneNumber”的地方，而且几乎不需要测试，就可以知道它所在的cell是PSTableCell。尝试隐藏这个cell，验证一下猜测：

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

cy# [#0x17f92890 setHidden:YES]

此时，MobilePhoneSettings变成了如图6-15所示的这个样子。

Now, MobilePhoneSettings looks like figure 6-15:



图6- 15 隐藏第一个cell

Figure 6-15 Hide the top cell

所以第一个cell的description是<PSTableCell: 0x17f92890; baseClass = UITableViewCell; frame = (0 35; 320 44); text = 'My Number'; autoresize = W; tag = 2; layer = <CALayer: 0x17f92a60>>。与刚才“编写邮件”按钮不同的是，这次的目标不是这个cell的响应函数（功能），而是它上面显示的内容（数据）， actionsForTarget:forControlEvent:不再适用。面对这种情况，该怎么办呢？

So the description of the top cell is <PSTableCell: 0x17f92890; baseClass = UITableViewCell; frame = (0 35; 320 44); text = 'My Number'; autoresize = W; tag = 2; layer = <CALayer: 0x17f92a60>>. Unlike “compose” button, our current target is not the response method of this cell (i.e. function), but the content (i.e. data) it shows, hence actionsForTarget:forControlEvent: is not our choice. Facing this kind of situation, what shall we do?

在绝大多数情况下，我们感兴趣的数据不会是一个常量。如果这个数据永远显示1，笔者相信你看都不会多看它一眼。当我们的目标是一个变量时，则要思考一个问题：这个变量来自哪里？

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

任何变量都不是凭空出现的，它是由数据源，经过一定的算法生成的，而我们感兴趣的一般是这个算法，也就是数据源生成变量的这个过程，这个过程往往是由一个或多个函数串联而成的，它们形成了一个调用链，类似于下面的伪代码：

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

id dataSource = ?; // head

id a = function(dataSource);

id b = function(a);

id c = function(b);

…

id z = function(y);

NSString \*myPhoneNumber = function(z); // tail

变量是已知的，也就是说，我们位于链条的尾部。逆向工程，自然就能够让我们从尾部顺着链条回溯到头部，找出这个调用链上的一个个函数，从而还原一整套算法。总的来说，还原变量的生成算法，就要在回溯的过程中记录其数据源（的数据源的数据源……，以下简称N重数据源）和函数的调用轨迹，当它的N重数据源是一个你可以决定的数据时（比如本例的数据源是——SIM卡），从N重数据源到变量之间这段链条上的函数，就是变量的生成算法。有点不知所云？看完下面的内容，你就明白了。

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

### 找出第一个cell的UI函数

### 6. Find the UI function of the top cell

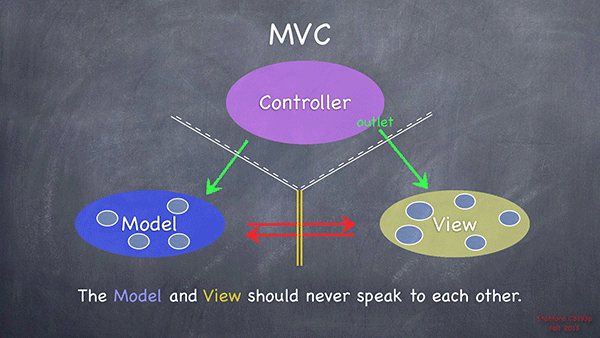


图6- 16 MVC设计标准（来自Stanford CS 193P）

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

按照MVC设计标准（如图6-16所示），M代表model，即数据源，是未知的；V代表view，即第一个cell，是已知的；C代表controller，是未知的。M和V之间没有直接联系，而C既可以访问M又可以访问V，是三者的交流中枢。如果我们能够利用已知的V，获得C，不就可以访问M，找到自己的数据源了吗？这种方式从逻辑上是说得通的，在实际操作中可行吗？

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

从笔者目前的职业经历来看，从V得到C，是100%可行的，用到的关键函数，就是在笔者心目中与recursiveDescription具有同等地位的公开函数[UIResponder nextResponder]，它的描述是这样的：

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

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

也就是说，对于一个V，调用nextResponder要么返回它对应的C，要么返回它的superview。因为MVC三者缺一不可，所以C是一定存在的，也就是说，一定有一个V的nextResponder是C；又因为通过recursiveDescription可以拿到所有的V，所以从已知的V获得C是可行的，进一步就可以访问M了。

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

因此，我们现在的目标是拿到cell的C，操作起来很简单——从cell处开始调用nextResponder，一直到返回一个C为止：

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

cy# [#0x17f92890 nextResponder]

#"<UITableViewWrapperView: 0x17eb4fc0; frame = (0 0; 320 504); gestureRecognizers = <NSArray: 0x17ee5230>; layer = <CALayer: 0x17ee5170>; contentOffset: {0, 0}; contentSize: {320, 504}>"

cy# [#0x17eb4fc0 nextResponder]

#"<UITableView: 0x16c69e00; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x17f4ace0>; layer = <CALayer: 0x17f4ac20>; contentOffset: {0, -64}; contentSize: {320, 717.5}>"

cy# [#0x16c69e00 nextResponder]

#"<UIView: 0x17ebf2b0; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer: 0x17ebf320>>"

cy# [#0x17ebf2b0 nextResponder]

#"<PhoneSettingsController 0x17f411e0: navItem <UINavigationItem: 0x17dae890>, view <UITableView: 0x16c69e00; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x17f4ace0>; layer = <CALayer: 0x17f4ac20>; contentOffset: {0, -64}; contentSize: {320, 717.5}>>"

拿到了C，就可以从C所在的头文件出发，踏上寻找M的旅途了。对于本例的情况，首先要定位PhoneSettingsController所在的目标文件，我们不确定它是来自Preferences.app本身，还是来自一个PreferenceBundle。对于这种情况，简单验证一下就好了：

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

FunMaker-5:~ root# grep -r PhoneSettingsController /Applications/Preferences.app/

FunMaker-5:~ root# grep -r PhoneSettingsController /System/Library/

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

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

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

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

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

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

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

看来这个类来自MobilePhoneSettings.bundle。下面class-dump它的二进制文件，然后打开PhoneSettingsController.h：

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

@interface PhoneSettingsController : PhoneSettingsListController <TPSetPINViewControllerDelegate>

……

- (id)myNumber:(id)arg1;

- (void)setMyNumber:(id)arg1 specifier:(id)arg2;

……

- (id)tableView:(id)arg1 cellForRowAtIndexPath:(id)arg2;

@end

从上面的代码可以看到，前两个方法明显跟本机号码相关，而第3个方法是用来初始化所有cell的数据源函数，每个cell显示的数据一般也都跟这个方法有着千丝万缕的联系。从这3个方法入手，一定可以找到第一个cell的数据源。我们用LLDB在[PhoneSettingsController tableView:cellForRowAtIndexPath:]的末尾下个断点，打印出返回值，也就是cell，看看有没有本机号码的踪迹。下面用debugserver附加Preferences，然后用LLDB连接，查看MobilePhoneSettings的ASLR偏移：

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

(lldb) image list -o -f

[ 0] 0x00078000 /private/var/db/stash/\_.29LMeZ/Applications/Preferences.app/Preferences(0x000000000007c000)

[ 1] 0x00231000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000231000)

[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard

[ 3] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation

……

[322] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PreferenceBundles/MobilePhoneSettings.bundle/MobilePhoneSettings

……

可以看到，MobilePhoneSettings的ASLR偏移是0x6db3000。然后在IDA中看看[PhoneSettingsController tableView:cellForRowAtIndexPath:]末尾指令的地址，如图6-17所示。

As we can see, the ASLR offset of MobilePhoneSettings is 0x6db3000. Then check the address of the last instruction in [PhoneSettingsController tableView:cellForRowAtIndexPath:], as shown in figure 6-17:

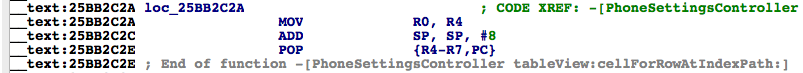


图6- 17 [PhoneSettingsController tableView:cellForRowAtIndexPath:]

Figure 6-17 [PhoneSettingsController tableView:cellForRowAtIndexPath:]

因为返回值存放在R0中，所以我们把断点下在“ADD SP, SP, #8”上，然后返回上一级目录，再重新进入MobilePhoneSettings，待断点触发后打印R0，其中应该存放了已经初始化的cell：

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

(lldb) br s -a 0x2c965c2c

Breakpoint 2: where = MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 236, address = 0x2c965c2c

Process 115525 stopped

\* thread #1: tid = 0x1c345, 0x2c965c2c MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 236, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x2c965c2c MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 236

MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 236:

-> 0x2c965c2c: add sp, #8

0x2c965c2e: pop {r4, r5, r6, r7, pc}

MobilePhoneSettings`-[PhoneSettingsController applicationWillSuspend]:

0x2c965c30: push {r7, lr}

0x2c965c32: mov r7, sp

(lldb) po $r0

<PSTableCell: 0x15f41440; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer = <CALayer: 0x15f4c930>>

(lldb) po [$r0 subviews]

<\_\_NSArrayM 0x17060e50>(

<UITableViewCellContentView: 0x15ed0660; frame = (0 0; 320 44); gestureRecognizers = <NSArray: 0x15f491e0>; layer = <CALayer: 0x15ed06d0>>,

<UIButton: 0x15f26f50; frame = (302 16; 8 13); opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x15f27050>>

)

(lldb) po [$r0 detailTextLabel]

<UITableViewLabel: 0x15eb3480; frame = (0 0; 0 0); text = '+86PhoneNumber'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15eb3540>>

可以看到，第一个cell的UI函数确实是[PhoneSettingsController tableView:cellForRowAtIndexPath:]，我们成功完成了本节的任务。我们有信心，通过PhoneSettingsController类一定可以拿到访问M的方法，在tableView:cellForRowAtIndexPath:内部也一定有M的线索，在下一小节中就会见证。

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

注意，游戏一般不是采用UIKit来构建UI的， recursiveDescription和nextResponder不适用于游戏。在逆向工程初期，不建议把游戏作为练习目标。如果你在熟悉了本书的内容后想要逆向游戏，可以来<http://bbs.iosre.com>参与讨论。

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

### 以UI函数为起点，寻找目标函数

### 6.2.2 Locate the target function from the UI function

拿到UI函数，预示着首战告捷。但是，UI函数是跟UI密切相关的，也就是说，要想调用[ComposeButtonItem \_sendAction:withEvent:]来编写邮件，或者调用[PhoneSettingsController tableView:cellForRowAtIndexPath:]来获取本机号码，会关联很多UI操作，比如刷新界面、尺寸布局等，有一种牵一发而动全身的感觉。在绝大多数情况下，我们不想搞得这么大张旗鼓，希望只是安静地牵一发，而不会动全身。面对这种挑战，我们该何去何从呢？

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

作为工程师，一定要具备基本的代码常识：最底层的函数通常是直接用汇编代码编写的，我们还接触不到；而这层以上的函数全都是嵌套调用的。UI函数也不例外——它嵌套调用了我们的目标函数。用伪代码表示如下：

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

drink GetRegular(water arg)

{

Functions();

return MakeRegular(arg);

}

drink GetDiet(void)

{

Functions();

return MakeDiet();

}

drink GetZero(void)

{

Functions();

return MakeZero();

}

drink GetCoke(sugar arg1, water arg2, color arg3)

{

if (arg1 > 0 && arg1 < 3) return GetDiet();

else if (arg1 == 0) return GetZero();

return GetRegular(arg2);

}

drink Get7Up(void)

{

Functions();

return Make7Up();

}

drink GetMirinda(void)

{

Functions();

return MakeMirinda();

}

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

{

if (arg3 == clear) Get7Up();

else if (arg3 == orange) GetMirinda();

return GetRegular(arg2);

}

array GetDrinks(sugar arg1, color arg2) // UIFunction

{

drink coke = GetCoke(arg1, 100, arg3);

drink pepsi = GetPepsi(arg1, 105, arg3);

return ArrayWithComponents(coke, pepsi)

}

我们不想每次都喝两种饮料（UI函数），如果只想喝七喜（数据），就要找到Get7Up（生成数据的目标函数）；如果想知道零度是怎么制作的（功能），就要找到MakeZero（提供功能的目标函数）。嵌套调用的函数之间其实也是一个链条，只要已知链条上的一个环节，就可用通过逆向工程还原整个链条。这个过程主要用到的工具是IDA和LLDB，我们接着上面2个App例子，看看怎么以[ComposeButtonItem \_sendAction:withEvent:]和[PhoneSettingsController tableView:cellForRowAtIndexPath:]这2个UI函数为线索，寻找“编写邮件”和“获取本机号码”的目标函数。

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

### 寻找“编写邮件”的目标函数

#### 1. Looking for the target function of “compose email”

把MobileMail丢进IDA开始分析，然后在Functions window里搜索[ComposeButtonItem \_sendAction:withEvent:]，如图6-18所示。

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

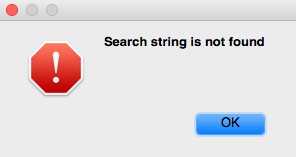


图6- 18 找不到[ComposeButtonItem \_sendAction:withEvent:]

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

说好的[ComposeButtonItem \_sendAction:withEvent:]呢？既然ComposeButtonItem没有实现这个方法，那么我们去它的父类里看看。打开ComposeButtonItem.h，看看它继承自哪个类：

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

@interface ComposeButtonItem : LongPressableButtonItem

+(id)composeButtonItem;

@end

然后打开LongPressableButtonItem.h，看看它有没有实现\_sendAction:withEvent:方法：

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

@interface LongPressableButtonItem : UIBarButtonItem

{

id \_longPressTarget;

SEL \_longPressAction;

}

- (void)\_attachGestureRecognizerToView:(id)arg1;

- (id)createViewForNavigationItem:(id)arg1;

- (id)createViewForToolbar:(id)arg1;

- (void)longPressGestureRecognized:(id)arg1;

- (void)setLongPressTarget:(id)arg1 action:(SEL)arg2;

@end

它也没有实现这个方法，那就再到它的父类里去看看。打开UIBarButtonItem.h：

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

@interface UIBarButtonItem : UIBarItem <NSCoding>

……

- (void)\_sendAction:(id)arg1 withEvent:(id)arg2;

……

@end

原来这个函数是在UIBarButtonItem类中实现的，那么我们把UIKit的二进制文件拖到IDA里开始分析。UIKit二进制文件较大，IDA分析耗时较长，在等待的间隙，来<http://bbs.iosre.com>跟大家聊聊吧！

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

UIKit初始分析结束后，定位到[UIBarButtonItem \_sendAction:withEvent:]，如图6-19所示。

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

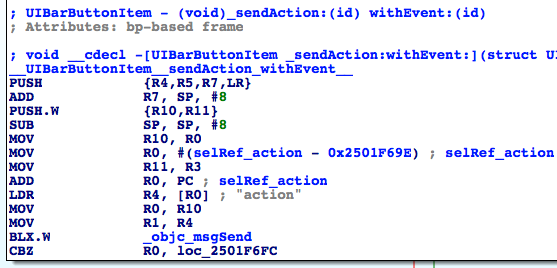


图6- 19 [UIBarButtonItem \_sendAction:withEvent:]

Figure 6-19 [UIBarButtonItem \_sendAction:withEvent:]

第一个调用的函数是objc\_msgSend。官方文档的注释是这样的：

The first function to be called is objc\_msgSend. Its official documentation is:

*“When it encounters a method call, the compiler generates a call to one of the functions objc\_msgSend, objc\_msgSend\_stret, objc\_msgSendSuper, or objc\_msgSendSuper\_stret. Messages sent to an object’s superclass (using the super keyword) are sent using objc\_msgSendSuper; other messages are sent using objc\_msgSend. Methods that have data structures as return values are sent using objc\_msgSendSuper\_stret and objc\_msgSend\_stret.”*

依据第5章中“对象”、“方法”和“实现”的关系来进一步探索，[receiver message]在编译后变成了objc\_msgSend(receiver, @selector(message))；当方法有参数时，则由[receiver message:arg1 foo:arg2 bar:arg3]变成objc\_msgSend(receiver, @selector(message), arg1, arg2, arg3)，依此类推。因此，第一个objc\_msgSend其实是执行了一个Objective-C方法。那么它具体是执行的什么方法呢？调用者是谁，参数又是什么呢？

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

还记得我们的金句吗？

Still remember “sentence of the book”?

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

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

依照金句来看，objc\_msgSend调用时的参数应该是objc\_msgSend(R0, R1, R2, R3, \*SP, \*(SP + sizeOfLastArg), ...)的形式，还原成等价的Objective-C方法，就是[R0 R1:R2 foo:R3 bar:\*SP baz:\*(SP + sizeOfLastArg) qux:...]。把这个套路运用在第一个objc\_msgSend上，想要知道它的等价Objective-C方法，就要看在“BLX.W \_objc\_msgSend”之前，R0~R3及SP都是什么。这是个从下往上倒推的分析过程，是名符其实的逆向工程。一起来看一下。

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

在“BLX.W \_objc\_msgSend”之前，R0最近的一次赋值来自“MOV R0, R10”，即R0来自R10；R10的最近一次赋值来自“MOV R10, R0”，即R10来自R0。在“MOV R10, R0”之前，R0没有被赋值就直接取值了；这显然是不合逻辑的，汇编语言不可能出现这么严重的设计漏洞。那么R0肯定还是在某个地方被赋值了——问题来了，“某个地方”是哪个地方呢？

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

既然在[UIBarButtonItem \_sendAction:withEvent:]的内部R0没有被赋值，那么唯一的可能就是它在[UIBarButtonItem \_sendAction:withEvent:]的调用者中被赋值。[UIBarButtonItem \_sendAction:withEvent:]在编译后变成了objc\_msgSend(UIBarButtonItem, @selector(\_sendAction:withEvent:), action, event)，四个参数分别放在了R0~R3中。因此，[UIBarButtonItem \_sendAction:withEvent:]得到调用时，R0的值就是UIBarButtonItem，进而调用“MOV R10, R0”时的R0也是UIBarButtonItem，即调用“BLX.W \_objc\_msgSend”时的R0是UIBarButtonItem。有点迷糊？对照着图6-20再想一想就明白了。

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

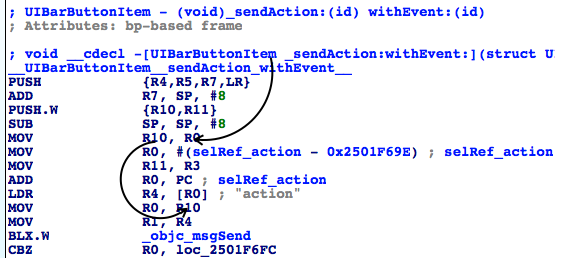


图6- 20 R0的演变过程

Figure 6-20 R0’s evolution

同理，在“BLX.W \_objc\_msgSend”之前，R1最近的一次赋值来自“MOV R1, R4”，即R1来自R4；R4的最近一次赋值来自“LDR R4, [R0]”，R4来自\*R0，即IDA已经标出的“action”。R1的演变过程如图6-21所示。

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

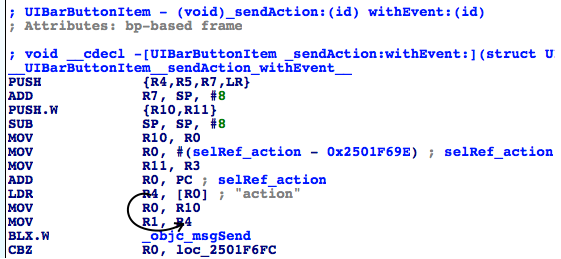


图6- 21 R1的演变过程

Figure 6-21 R1’s change process

因此，第一个objc\_msgSend还原成Objective-C方法后，是[self action]，返回值存放在接下来的R0中。没问题吧？接着进程判断[self action]是否为0，如果是0，则不执行任何操作；否则到达图6-22。

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

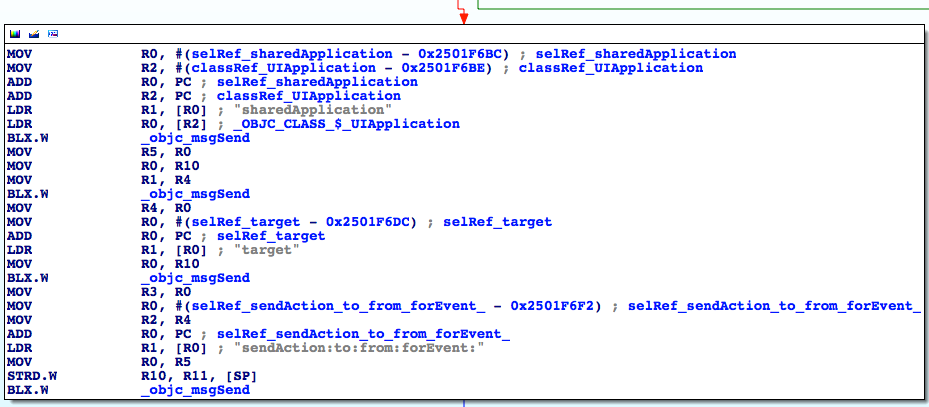


图6- 22 [UIBarButtonItem \_sendAction:withEvent:]

Figure 6-22 [UIBarButtonItem \_sendAction:withEvent:]

又是4个objc\_msgSend，从上到下逐个分析：

There’re 4 objc\_msgSend, let’s analyze them with the same thought one by one:

第一个objc\_msgSend的R0来自“LDR R0, [R2]”，IDA已经分析出[R2]是UIApplication类；R1来自“LDR R1, [R0]”，即“sharedApplication”，因此第一个objc\_msgSend还原成Objective-C方法就是[UIApplication sharedApplication]，且返回值放入R0。

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

第二个objc\_msgSend的R0来自“MOV R0, R10”，即R10；在图6-20中，我们知道R10的值是UIBarButtonItem；R1来自“MOV R1, R4”，即R4；在图6-21中，R4的值是“action”。因此第二个objc\_msgSend还原成Objective-C方法就是[UIBarButtonItem action]，并将返回值存放在R0中。

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

第三个objc\_msgSend的R0仍来自“MOV R0, R10”，即UIBarButtonItem；R1来自“LDR R1, [R0]”，即“target”。因此第三个objc\_msgSend还原成Objective-C方法就是[UIBarButtonItem target]，并将返回值保存在R0中。

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

第四个objc\_msgSend的R0来自“MOV R0, R5”，即R5；R5来自第一个objc\_msgSend下方的“MOV R5, R0”，即R0；R0是什么呢？因为第一个objc\_msgSend执行之后，把返回值存放在了R0里，所以这个R0就是[UIApplication sharedApplication]的返回值，它是objc\_msgSend的第一个参数。R1来自“LDR R1, [R0]”，即“sendAction:to:from:forEvent:”，这是一个有4个参数的方法，加上objc\_msgSend的前2个参数，一共6个参数，因此R0~R3寄存器不够用了，有2个参数要放在栈上。R2来自“MOV R2, R4”，即R4；R4来自第二个objc\_msgSend下方的“MOV R4, R0”，即R0；R0来自第二个objc\_msgSend执行之后的返回值，即[UIBarButtonItem action]，这是第3个参数。R3来自第三个objc\_msgSend下方的“MOV R3, R0”，即R0；R0来自第三个objc\_msgSend执行之后的返回值，[UIBarButtonItem target]，这是第4个参数。接下来的2个参数来自栈，而在第四个objc\_msgSend以前，栈的最近一次改动来自“STRD.W R10, R11, [SP]”，即先后把R10和R11入栈，因此接下来的2个参数就是R10和R11。R10是刚才已经分析了好几遍的UIBarButtonItem，而R11来自图6-21的“MOV R11, R3”，即R3；R3又是一个没有被赋值就直接取值的寄存器，因此它也是来自[UIBarButtonItem \_sendAction:withEvent:]的调用者。根据我们之前的分析，R11就是\_sendAction:withEvent:的第二个参数，即event。这4个objc\_msgSend的参数关系可以用图6-23和图6-24表示。

R0 of the 4th objc\_msgSend comes from “MOV R0, R5”, i.e. R5; R5 comes from “MOV R5, R0” under the 1st objc\_msgSend, i.e. R0. What’s R0? Because the 1st objc\_msgSend stores its return value in R0, R0 is the return value of [UIApplication sharedApplication] as well the 1st argument of the 4th objc\_msgSend. R1 comes from “LDR R1, [R0]”, i.e. “sendAction:to:from:forEvent:”, which has 4 arguments. Since objc\_msgSend already has 2 arguments, there’re 6 arguments in total, R0~R3 are not enough to hold all arguments, the last 2 arguments have to be stored in the stack. R2 comes from “MOV R2, R4”, i.e. R4; R4 comes from “MOV R4, R0” under the 2nd objc\_msgSend, i.e. R0; R0 comes from the return value of the 2nd objc\_msgSend, i.e. [UIBarButtonItem action], which is the 3rd argument. R3 comes from “MOV R3, R0” under the 3rd objc\_msgSend, i.e. R0; R0 comes from the return value of the 3rd objc\_msgSend, i.e. [UIBarButtonItem target], which is the 4th argument. The rest 2 arguments come from the stack, and before the 4th objc\_msgSend, the latest change of stack comes from “STRD.W R10, R11, [SP]”, i.e. R10 and R11 are saved onto the stack; therefore, the rest 2 arguments are R10 and R11. R10 is UIBarButtonItem, which is discussed several times; whereas R11 comes from “MOV R11, R3” in figure 6-21, i.e. R3, which is another unassigned register, so it must come from the caller of [UIBarButtonItem \_sendAction:withEvent:]. Based on our previous analysis, R11 is the 2nd argument of \_sendAction:withEvent:, i.e. event. The relationship of these 4 arguments is a little complicated, hope figure 6-23 and 6-24 can give you a better illustration.

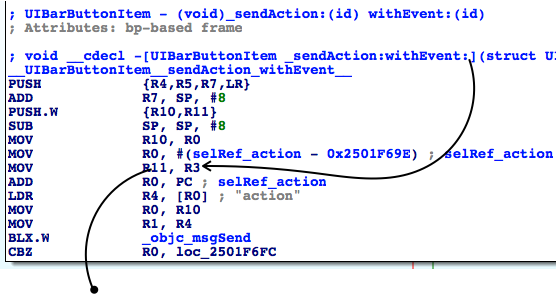


图6- 23 objc\_msgSend的参数关系

Figure 6-23 The relationship of objc\_msgSend’s arguments

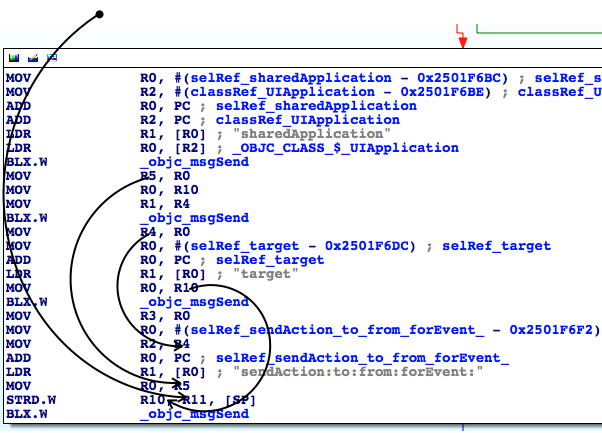


图6- 24 objc\_msgSend的参数关系

Figure 6-24 The relationship of objc\_msgSend’s arguments

这样看来，[UIBarButtonItem \_sendAction:withEvent:]内最关键的就是[[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:event]这个方法了。因为我们已经知道[UIBarButtonItem \_sendAction:withEvent:]会执行“编写邮件”操作，所以[[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:event]肯定会得到调用。虽然上面用IDA厘清了每个参数的来源，但是这些参数在运行时的值是什么，用IDA仍看不出来；是时候借助LLDB的威力了，一起来看看在运行时这段代码都做了些什么。

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

用debugserver附加MobileMail，然后用LLDB连过去，打印出UIKit的ASLR偏移：

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

(lldb) image list -o -f

[ 0] 0x0008e000 /private/var/db/stash/\_.29LMeZ/Applications/MobileMail.app/MobileMail(0x0000000000092000)

[ 1] 0x00393000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000393000)

[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/usr/lib/libarchive.2.dylib

……

[ 45] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/Frameworks/UIKit.framework/UIKit

……

UIKit的ASLR偏移是0x6db3000。再看看第四个objc\_msgSend地址是多少，如图6-25所示。

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

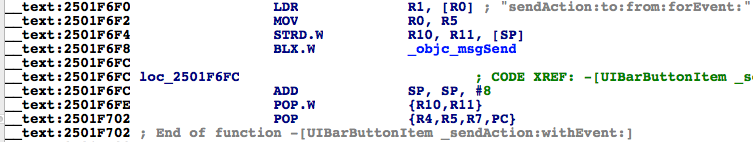


图6- 25 查看objc\_msgSend的地址

Figure 6-25 Check out address of objc\_msgSend

在0x6db3000 + 0x2501F6F8 = 0x2BDD26F8上下个断点，然后按下“编写邮件”按钮触发断点，我们看看[[UIApplication sharedApplication] sendAction:[self action] to:[self target] from:self forEvent:eventFromArg2]的几个参数都是什么：

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

(lldb) br s -a 0x2BDD26F8

Breakpoint 4: where = UIKit`-[UIBarButtonItem(UIInternal) \_sendAction:withEvent:] + 116, address = 0x2bdd26f8

Process 44785 stopped

\* thread #1: tid = 0xaef1, 0x2bdd26f8 UIKit`-[UIBarButtonItem(UIInternal) \_sendAction:withEvent:] + 116, queue = 'com.apple.main-thread, stop reason = breakpoint 4.1

frame #0: 0x2bdd26f8 UIKit`-[UIBarButtonItem(UIInternal) \_sendAction:withEvent:] + 116

UIKit`-[UIBarButtonItem(UIInternal) \_sendAction:withEvent:] + 116:

-> 0x2bdd26f8: blx 0x2c3539f8 ; symbol stub for: roundf$shim

0x2bdd26fc: add sp, #8

0x2bdd26fe: pop.w {r10, r11}

0x2bdd2702: pop {r4, r5, r7, pc}

(lldb) p (char \*)$r1

(char \*) $48 = 0x2c3de501 "sendAction:to:from:forEvent:"

(lldb) po $r0

<MailAppController: 0x176a8820>

(lldb) po $r2

[no Objective-C description available]

(lldb) p (char \*)$r2

(char \*) $51 = 0x2d763308 "composeButtonClicked:"

(lldb) po $r3

<nil>

(lldb) x/10 $sp

0x00391198: 0x1776d640 0x176a8ce0 0x1760f5e0 0x00000000

0x003911a8: 0x2c4140f2 0x1776ff50 0x003911cc 0x2bc6ec2b

0x003911b8: 0x176a8ce0 0x00000001

(lldb) po 0x1776d640

<ComposeButtonItem: 0x1776d640>

(lldb) po 0x176a8ce0

<UITouchesEvent: 0x176a8ce0> timestamp: 58147.4 touches: {(

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

)}

其中，objc\_msgSend的参数R0~R3很容易理解，分别是self、@selector(sendAction:to:from:forEvent:)、sendAction:的参数和to:的参数，直接打印寄存器就可以了。注意，在执行“po $r2”的时候，LLDB提示“no Objective-C description available”，即R2不是一个Objective-C对象，因此结合“action”的含义，笔者猜测它是一个SEL，就用“p (char \*)$r2”打印了它。如何解析栈中的参数呢？因为SP是指向栈底的指针，而我们知道余下的2个参数都在栈中，且大小均为1个字，所以，可用“x/10 $sp”打印从栈底开始的连续10个字，前2个字就是from:和forEvent:的参数。Objective-C方法的大多数参数都是1个字长度的指针，指向一个Objective-C对象，因此我们“po”了前2个字，把参数打印了出来。为了更便于理解，这里SP、栈上存储的值和参数的关系，可以参考图6-26。

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

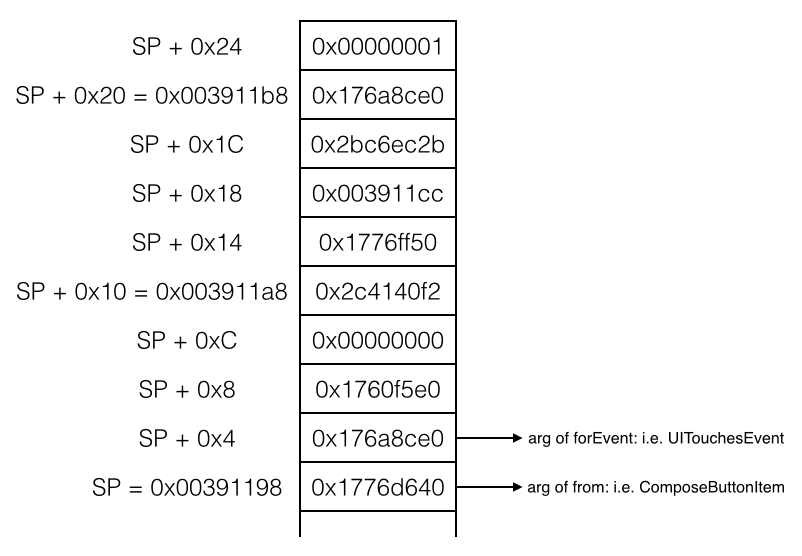


图6- 26 SP、栈值和参数的关系

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

一般情况下，Objective-C方法在栈中的参数不会超过10个，“x/10 $sp”就足够了，挨个打印，就能找到栈上的所有参数。

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

结合IDA和LLDB，我们知道[UIBarButtonItem \_sendAction:withEvent:]的核心在于[MailAppController sendAction:@selector(composeButtonClicked:) to:nil from:ComposeButtonItem forEvent:UITouchesEvent]，离“编写邮件”的目标函数又近了一层。下面在IDA里看看[UIApplication sendAction:to:from:forEvent:]的内部做了些什么，如图6-27所示。

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

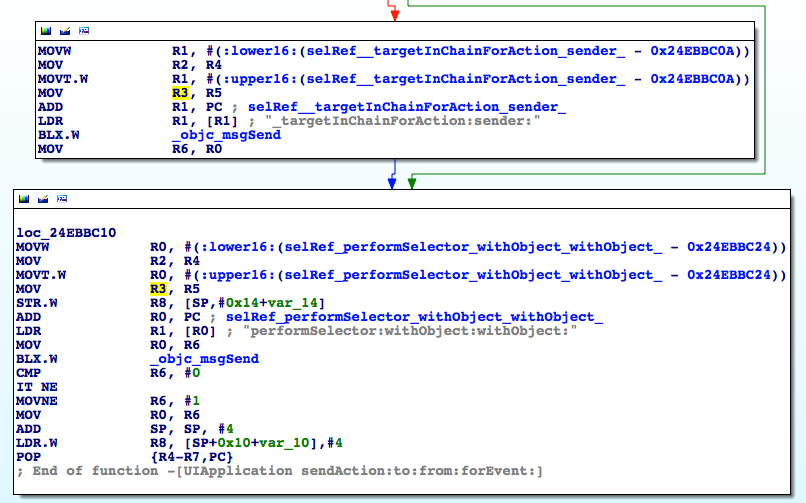


图6- 27 [UIApplication sendAction:to:from:forEvent:]

无论如何，loc\_24ebbc10中的“performSelector:withObject:withObject:”都会得到执行，我们自然猜测它就是做出实际操作的地方。跟刚才一样，用LLDB看看这个方法到底执行了什么操作。UIKit的ASLR偏移是0x6db3000，最下面的那个objc\_msgSend地址是0x24EBBC26，故而在0x6db3000 + 0x24EBBC26 = 0x2BC6EC26上下断点，然后按下“编写邮件”按钮触发断点，再看看这个方法的参数：

Whatever, “performSelector:withObject:withObject:” in loc\_24ebbc10 will get executed, so naturally we can guess it is where actual operations are performed. Just like before, let’s figure out what does this method do with LLDB. The ASLR offset of UIKit is 0x6db3000, and the address of the last objc\_msgSend is 0x24EBBC26, so we set a breakpoint at 0x6db3000 + 0x24EBBC26 = 0x2BC6EC26, then tap “compose” button to trigger the breakpoint to inspect the arguments:

(lldb) br s -a 0x2BC6EC26

Breakpoint 1: where = UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66, address = 0x2bc6ec26

Process 226191 stopped

\* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66

UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:

-> 0x2bc6ec26: blx 0x2c3539f8 ; symbol stub for: roundf$shim

0x2bc6ec2a: cmp r6, #0

0x2bc6ec2c: it ne

0x2bc6ec2e: movne r6, #1

(lldb) p (char \*)$r1

(char \*) $0 = 0x2c3dac95 "performSelector:withObject:withObject:"

(lldb) po $r0

<ComposeButtonItem: 0x14ddf5f0>

(lldb) p (char \*)$r2

(char \*) $2 = 0x2c4140f2 "\_sendAction:withEvent:"

(lldb) po $r3

<UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>>

(lldb) x/10 $sp

0x003735a8: 0x160a6120 0x00000001 0x14d73c90 0x160a6120

0x003735b8: 0x2c3d9be5 0x003735d4 0x2bc6ebd1 0x14d73c90

0x003735c8: 0x160a6120 0x00000040

(lldb) po 0x160a6120

<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches: {(

<UITouch: 0x14ff2f20> phase: Ended tap count: 1 window: <UIWindow: 0x14d878b0; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x14dba890>; layer = <UIWindowLayer: 0x14d87a30>> view: <UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>> location in window: {308, 545} previous location in window: {308, 545} location in view: {23, 21} previous location in view: {23, 21}

)}

这是怎么回事？performSelector:withObject:withObject:调用了[ComposeButtonItem \_sendAction:withEvent:]，而[ComposeButtonItem \_sendAction:withEvent:]又会调用performSelector:withObject:withObject:，如果它再次调用[ComposeButtonItem \_sendAction:withEvent:]，那这段代码就出现循环调用了，跟我们观察到的现象不符，也是不合常理的。那我们执行一下“c”命令，断点一定会被再次触发，看看performSelector:withObject:withObject:有没有发生变化：

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

(lldb) c

Process 226191 resuming

Process 226191 stopped

\* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x2bc6ec26 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66

UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:

-> 0x2bc6ec26: blx 0x2c3539f8 ; symbol stub for: roundf$shim

0x2bc6ec2a: cmp r6, #0

0x2bc6ec2c: it ne

0x2bc6ec2e: movne r6, #1

(lldb) p (char \*)$r1

(char \*) $6 = 0x2c3dac95 "performSelector:withObject:withObject:"

(lldb) po $r0

<MailAppController: 0x14e7a7a0>

(lldb) p (char \*)$r2

(char \*) $7 = 0x2d763308 "composeButtonClicked:"

(lldb) po $r3

<ComposeButtonItem: 0x14ddf5f0>

(lldb) x/10 $sp

0x0037356c: 0x160a6120 0x160a6120 0x2d763308 0x14e7a7a0

0x0037357c: 0x14ddf5f0 0x003735a0 0x2bdd26fd 0x14ddf5f0

0x0037358c: 0x160a6120 0x160fbdf0

(lldb) po 0x160a6120

<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches: {(

<UITouch: 0x14ff2f20> phase: Ended tap count: 1 window: <UIWindow: 0x14d878b0; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x14dba890>; layer = <UIWindowLayer: 0x14d87a30>> view: <UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44); opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>; layer = <CALayer: 0x14d73ea0>> location in window: {308, 545} previous location in window: {308, 545} location in view: {23, 21} previous location in view: {23, 21}

)}

可以看到，performSelector:withObject:withObject:的参数发生了变化，[MailAppController composeButtonClicked:ComposeButtonItem]得到了调用，如果再“c”一下，发现断点不再触发，所以可以确定执行实际操作的是composeButtonClicked:。因为在MobileMail内部，调用[UIApplication sharedApplication]可以拿到MailAppController对象；而在本小节开始的时候，我们在ComposeButtonItem.h里看到了可以通过一个类方法+composeButtonItem来拿到ComposeButtonItem对象；所以我们可以拿到调用[MailAppController composeButtonClicked:ComposeButtonItem]所需的全部对象，且在MobileMail的内部任何地方都可以调用这个方法，它可以算作是“编写邮件”的目标函数了。

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

在Cycript里做最后测试，看看这个目标函数好用不好用：

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

FunMaker-5:~ root# cycript -p MobileMail

cy# [UIApp composeButtonClicked:[ComposeButtonItem composeButtonItem]]

执行后成功调出“编写邮件”界面。在本例中，我们用IDA追踪函数的调用链，找到目标函数，然后用LLDB解析出了它的参数，虽然有点复杂，但其实不难，不是吗？接下来，将用类似的套路来找出“获取本机号码”的目标函数，请大家注意总结。

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