2. 寻找“获取本机号码”的目标函数

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

接着上面的内容，根据找到的UI函数[PhoneSettingsController tableView:cellForRowAtIndexPath:]继续往下分析。因为UI函数的返回值存放在R0中，而从图6-17的“MOV R0, R4”可知，R0来自R4。在[PhoneSettingsController tableView:cellForRowAtIndexPath:]里，R4只在图6-28里的“MOV R4, R0”处被赋值了一次，这里的R0来自objc\_msgSendSuper2执行后的返回值。objc\_msgSendSuper2没有出现在文档中，由图6-29可知，它来自“/usr/lib/libobjc.A.dylib”。

Let’s continue our analysis from the UI function [PhoneSettingsController tableView:cellForRowAtIndexPath:]. Because the return value of UI function is stored in R0, and according to “MOV R0, R4” in figure 6-17, we know R0 comes from R4. As shown in figure 6-28, R4 is only assigned once at “MOV R4, R0” and R0 comes from the return value of objc\_msgSendSuper2. objc\_msgSendSuper2 is undocumented, as we can see in figure 6-29, it comes from “/usr/lib/libobjc.A.dylib”.

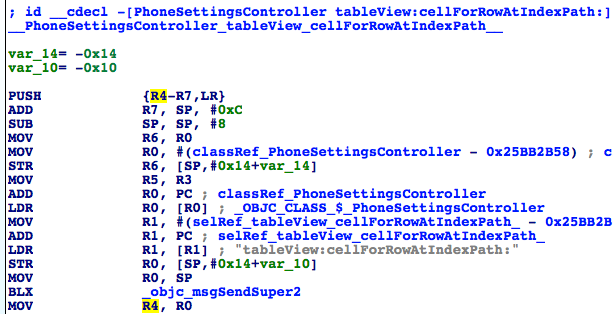


图6- 28 R4的来源

Figure 6-28 Source of R4

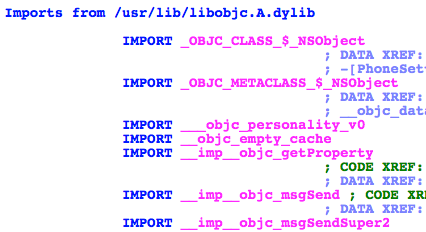


图6- 29 objc\_msgSendSuper2的来源

Figure 6-29 Source of objc\_msgSendSuper2

按字面意思理解，objc\_msgSendSuper2的作用应该跟objc\_msgSendSuper类似，即向调用者的父类发送消息。不用做过多猜测，在这个objc\_msgSendSuper2下个断点，看看它的参数和返回值就知道了。用debugserver附加Preferences，用LLDB连接，然后打印出MobilePhoneSettings的ASLR偏移：

According to the literal meaning, objc\_msgSendSuper2 and objc\_msgSendSuper are supposed to work similarly, namely send messages to callers’ superclasses. No more guesses, let’s set a breakpoint on objc\_msgSendSuper2 and check out its arguments as well return value. Attach debugserver to Preference, and connect with LLDB, then print out ASLR offset of MobilePhoneSettings:

(lldb) image list -o -f

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

[ 1] 0x00232000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000232000)

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

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

……

[330] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PreferenceBundles/MobilePhoneSettings.bundle/MobilePhoneSettings

……

MobilePhoneSettings的ASLR偏移是0x6db3000。然后看看objc\_msgSendSuper2的地址，如图6-30所示。

ASLR offset of MobilePhoneSettings is 0x6db3000. Then take a look at objc\_msgSendSuper2’s address, as shown in figure 6-30.

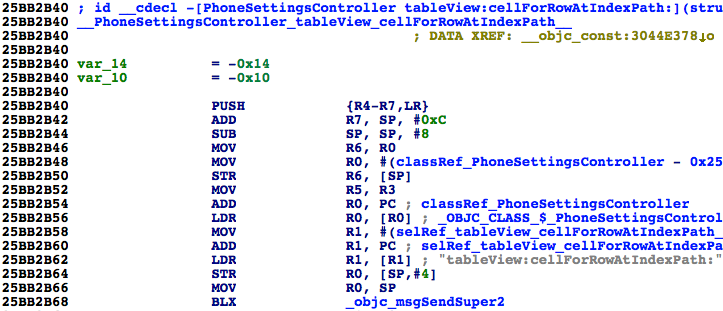


图6- 30 查看objc\_msgSendSuper2的地址

Figure 6-30 Check out address of objc\_msgSendSuper2

断点的地址应该是0x6db3000 + 0x25BB2B68 = 0x2C965B68。返回上一级目录，再进入MobilePhoneSettings触发断点：

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

(lldb) br s -a 0x2C965B68

Breakpoint 1: where = MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 40, address = 0x2c965b68

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x2c965b68 MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 40, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x2c965b68 MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 40

MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 40:

-> 0x2c965b68: blx 0x2c975fb8 ; symbol stub for: CTSettingRequest$shim

0x2c965b6c: mov r4, r0

0x2c965b6e: movw r0, #54708

0x2c965b72: movt r0, #2697

(lldb) p (char \*)$r1

(char \*) $0 = 0x2c3daf33 "tableView:cellForRowAtIndexPath:"

(lldb) po $r0

[no Objective-C description available]

(lldb) ni

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x2c965b6c MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 44, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x2c965b6c MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 44

MobilePhoneSettings`-[PhoneSettingsController tableView:cellForRowAtIndexPath:] + 44:

-> 0x2c965b6c: mov r4, r0

0x2c965b6e: movw r0, #54708

0x2c965b72: movt r0, #2697

0x2c965b76: mov r2, r5

(lldb) po $r0

<PSTableCell: 0x15fc6b00; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer = <CALayer: 0x15fbbe40>>

(lldb) po [$r0 detailTextLabel]

<UITableViewLabel: 0x15fb5590; frame = (0 0; 0 0); text = '+86PhoneNumber'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd87e0>>

值得一提的是，objc\_msgSendSuper2的第一个参数并不是一个Objective-C对象，我不清楚这到底是LLDB的bug，还是情况确实如此，但这不影响本节的分析，忽略这个细节就好。感兴趣的朋友可以继续研究，然后在<http://bbs.iosre.com>分享你的发现。

It’s worth mentioning that the 1st argument of objc\_msgSendSuper2 is not an Objective-C object. I’m not sure whether it is a bug of LLDB or it is the case. Anyway, it doesn’t influence our analysis, just ignore it for now. If you’re really interested in this detail, you are welcome to share your research on <http://bbs.iosre.com>.

话说回来，LLDB的输出结果预示着objc\_msgSendSuper2的返回结果就是初始化好的cell，里面已经含有了本机号码信息。跟上一节类似，到PhoneSettingsController的父类里看看tableView:cellForRowAtIndexPath:的实现。首先打开PhoneSettingsController.h，看看它的父类是谁：

Back on track, the output of LLDB indicates that the return value of objc\_msgSendSuper2 is an initialized cell, which contains my number already. Similar to what happened in the last section, let’s check out the implementation of tableView:cellForRowAtIndexPath: in PhoneSettingsController’s superclass. First of all let’s figure out who’s the superclass in PhoneSettingsController.h:

@interface PhoneSettingsController : PhoneSettingsListController <TPSetPINViewControllerDelegate>

……

@end

可以看到，PhoneSettingsController继承自PhoneSettingsListController，再打开PhoneSettingsListController.h，看看它有没有实现tableView:cellForRowAtIndexPath:方法：

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

@interface PhoneSettingsListController : PSListController

{

}

- (id)bundle;

- (void)dealloc;

- (id)init;

- (void)pushController:(Class)arg1 specifier:(id)arg2;

- (id)setCellEnabled:(BOOL)arg1 atIndex:(unsigned int)arg2;

- (id)setCellLoading:(BOOL)arg1 atIndex:(unsigned int)arg2;

- (id)setControlEnabled:(BOOL)arg1 atIndex:(unsigned int)arg2;

- (id)sheetSpecifierWithTitle:(id)arg1 controller:(Class)arg2 detail:(Class)arg3;

- (void)simRemoved:(id)arg1;

- (id)specifiers;

- (void)updateCellStates;

- (void)viewWillAppear:(BOOL)arg1;

@end

可见，PhoneSettingsListController没有实现tableView:cellForRowAtIndexPath:，继续去它的父类PSListController里看看。PSListController已经不在MobilePhoneSettings.bundle里了，用上一章介绍的搜索方法，很容易就可以在所有class-dump头文件里定位PSListController.h，如图6-31所示。

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

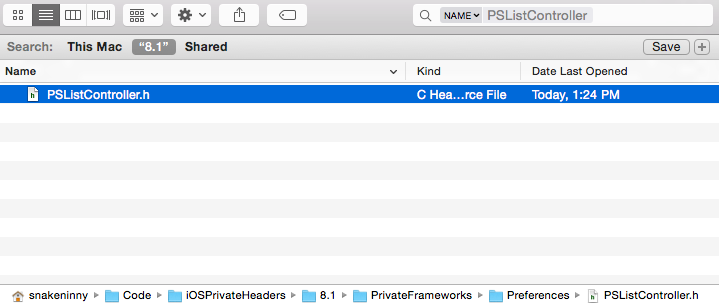


图6- 31 定位PSListController.h

Figure 6-31 Locate PSListController.h

注意，PSListController.h来自与Preferences.app同名的Preferences.framework，请大家注意分辨。打开它，看看有没有实现tableView:cellForRowAtIndexPath:方法：

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

@interface PSListController : PSViewController <UITableViewDelegate, UITableViewDataSource, UIActionSheetDelegate, UIAlertViewDelegate, UIPopoverControllerDelegate, PSSpecifierObserver, PSViewControllerOffsetProtocol>

……

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

……

@end

可以看到，它确实实现了这个方法，在IDA中打开Preferences.framework里的二进制文件，定位到tableView:cellForRowAtIndexPath:，如图6-32所示。

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

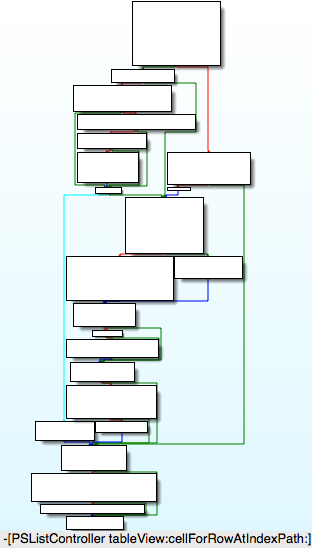


图6- 32 [PSListController tableView:cellForRowAtIndexPath:]

Figure 6-32 [PSListController tableView:cellForRowAtIndexPath:]

它的实现逻辑有些复杂，为了保险起见，先在它的尾部下一个断点，看看返回值里是否含有“本机号码”信息，确认objc\_msgSendSuper2是否调用了[PSListController tableView:cellForRowAtIndexPath:]。先看看Preferences.framework的ASLR偏移：

Its execution logic is complicated. To play it safe, let’s set a breakpoint at the end of this method to check if “my number” is contained in the return value, so that we can make sure objc\_msgSendSuper2 calls [PSListController tableView:cellForRowAtIndexPath:]. First, let’s check out ASLR offset of Preferences.framework:

(lldb) image list -o -f

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

[ 1] 0x00232000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x0000000000232000)

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

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

……

[ 42] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PrivateFrameworks/Preferences.framework/Preferences

……

它的ASLR偏移是0x6db3000。然后看看[PSListController tableView:cellForRowAtIndexPath:]尾部指令的地址，如图6-33所示。

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

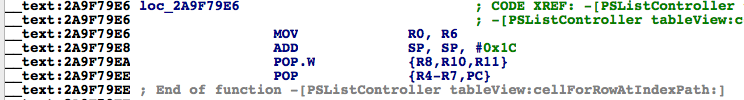


图6- 33 [PSListController tableView:cellForRowAtIndexPath:]

Figure 6-33 [PSListController tableView:cellForRowAtIndexPath:]

因为返回值存放在R0中，而R0来自“MOV R0, R6”，即R6，所以在这条指令上下一个断点，然后打印R6。这条指令的地址是0x2A9F79E6，因此断点的地址是0x6db3000 + 0x2A9F79E6 = 0x317AA9E6。返回上一页再重新进入MobilePhoneSettings，触发断点：

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

(lldb) br s -a 0x317AA9E6

Breakpoint 5: where = Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 1026, address = 0x317aa9e6

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9e6 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 1026, queue = 'com.apple.main-thread, stop reason = breakpoint 5.1

frame #0: 0x317aa9e6 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 1026

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 1026:

-> 0x317aa9e6: mov r0, r6

0x317aa9e8: add sp, #28

0x317aa9ea: pop.w {r8, r10, r11}

0x317aa9ee: pop {r4, r5, r6, r7, pc}

(lldb) po $r6

<PSTableCell: 0x15f8c6a0; baseClass = UITableViewCell; frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer = <CALayer: 0x15f7c0b0>>

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7b8d0; frame = (0 0; 0 0); text = '+86PhoneNumber'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15f7b990>>

从LLDB的输出可以确认objc\_msgSendSuper2调用了[PSListController tableView:cellForRowAtIndexPath:]，且它的返回值来自于R6。那R6来自于哪里呢？当我们往上回溯，查找R6来源的时候，可以看到R6作为objc\_msgSend的第一个参数，多次出现在了这个方法内部，如图6-34所示。

Now we can confirm that objc\_msgSendSuper2 calls [PSListController tableView:cellForRowAtIndexPath:], and its return value does come from R6. Well, where does R6 come from? When we track back to look for the source of R6, we can see multiple occurrences of R6 as the 1st argument of multiple objc\_msgSend, as shown in figure 6-34.

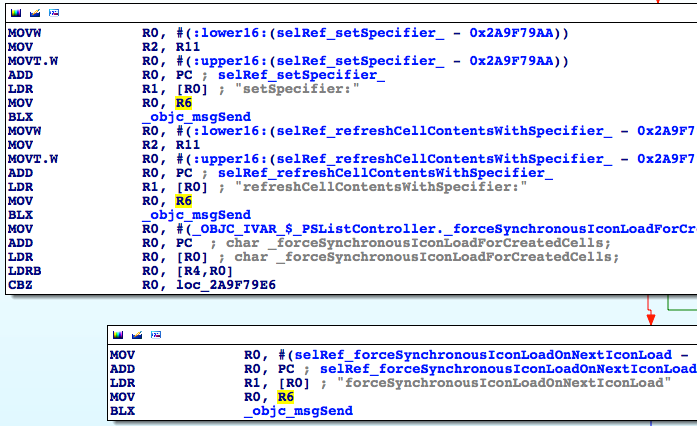


图6- 34 R6出现频率很高

Figure 6-34 Multiple occurrences of R6

再往上一点，会发现往R6里写入的，都是刚刚初始化的各种对象，如图6-35、图6-36、图6-37所示。

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

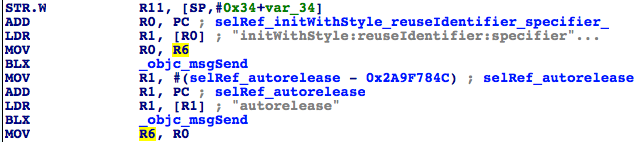


图6- 35 R6被赋值

Figure 6-35 The assignment of R6

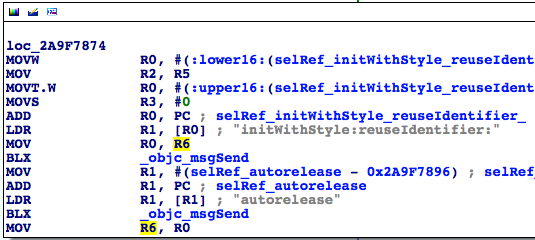


图6- 36 R6被赋值

Figure 6-36 The assignment of R6

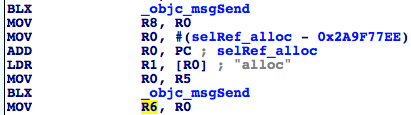


图6- 37 R6被赋值

Figure 6-37 The assignment of R6

这个现象很好理解，tableView:cellForRowAtIndexPath:的作用本来就是返回一个可用的cell。因此，常规的做法是在方法内部先创建一个空的cell，然后调用别的函数来配置它。那么，从一个空的PSTableCell到含有“本机号码”信息的这个配置过程发生在哪里呢？现在已知头部的PSTableCell不含有本机号码，尾部的PSTableCell含有本机号码，所以这个设置过程一定是发生在tableView:cellForRowAtIndexPath:内部的，且是通过一个objc\_msgSend函数完成的。因此，现在的问题变成了，在一堆objc\_msgSend函数中，怎么去定位那个设置“本机号码”的objc\_msgSend？

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

如果不考虑效率，可以从头开始一个个排查。tableView:cellForRowAtIndexPath:内部的objc\_msgSend个数毕竟有限，在执行objc\_msgSend之前和之后各打印一次[$r6 detailTextLabel]，对比两者的异同，就一定可以找到这个objc\_msgSend；数学比较好的朋友可能用二分法，从tableView:cellForRowAtIndexPath:中间部分的某个objc\_msgSend开始找，不断缩小排查范围。这就是见仁见智的问题了，大家选择一种自己喜欢的方式就好。在这里，笔者采取了折中的二分法，如图6-38所示。

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

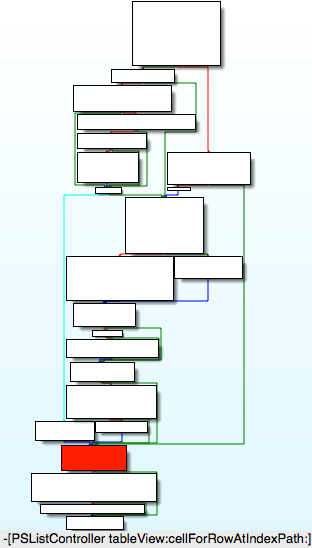


图6- 38 [PSListController tableView:cellForRowAtIndexPath:]

Figure 6-38 [PSListController tableView:cellForRowAtIndexPath:]

采用二分查找法固然效率高，但[PSListController tableView:cellForRowAtIndexPath:]的分支很多，从哪个地方分，可以保证不遗漏每一个分支呢？因为[PSListController tableView:cellForRowAtIndexPath:]的执行一定会通过图6-38所示的深色方块，所以以这个地方为二分点肯定不会遗漏任何分支，然后从它的第一个objc\_msgSend开始排查，如果[$r6 detailTextLabel]含有本机号码信息，那么就往上找，否则往下找。我们去看看这个深色方块包含的汇编指令，如图6-39所示。

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

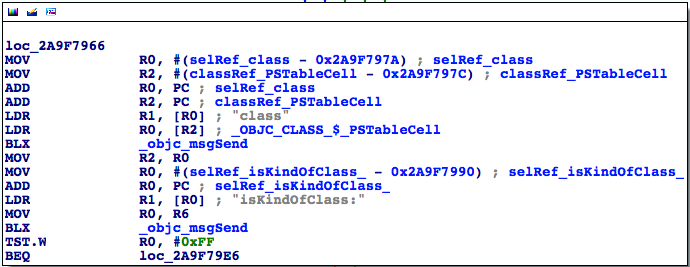


图6- 39 深色方块所在的loc\_2a9f7966

Figure 6-39 loc\_2a9f7966

这里有2个objc\_msgSend，就从最上面这一个开始吧，看看它的地址，如图6-40所示。

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

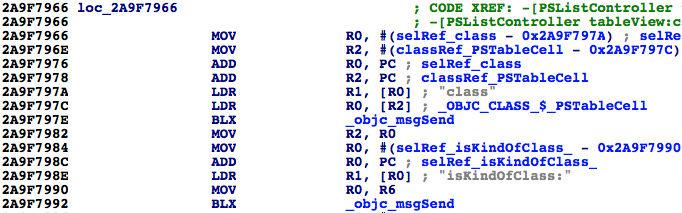


图6- 40 查看objc\_msgSend的地址

Figure 6-40 Check out address of objc\_msgSend

Preferences的ASLR偏移是0x6db3000，刚才已经用到了。所以断点的地址是0x6db3000 + 0x2A9F797E = 0x317AA97E。触发它，看看此时PSTableCell是否含有本机号码信息：

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

(lldb) br s -a 0x317AA97E

Breakpoint 10: where = Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 922, address = 0x317aa97e

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa97e Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 922, queue = 'com.apple.main-thread, stop reason = breakpoint 10.1

frame #0: 0x317aa97e Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 922

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 922:

-> 0x317aa97e: blx 0x31825f04 ; symbol stub for: \_\_\_\_NETRBClientResponseHandler\_block\_invoke

0x317aa982: mov r2, r0

0x317aa984: movw r0, #59804

0x317aa988: movt r0, #1736

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

它还不含有本机号码信息，说明本机号码信息一定是在图6-38深色方块下方的3个方块里生成的。接着执行“ni”命令，在每个objc\_msgSend的前后各“po [$r6 detailTextLabel]”一次：

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

(lldb) ni

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa982 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 926, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa982 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 926

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 926:

-> 0x317aa982: mov r2, r0

0x317aa984: movw r0, #59804

0x317aa988: movt r0, #1736

0x317aa98c: add r0, pc

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

……

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa992 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 942, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa992 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 942

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 942:

-> 0x317aa992: blx 0x31825f04 ; symbol stub for: \_\_\_\_NETRBClientResponseHandler\_block\_invoke

0x317aa996: tst.w r0, #255

0x317aa99a: beq 0x317aa9e6 ; -[PSListController tableView:cellForRowAtIndexPath:] + 1026

0x317aa99c: movw r0, #60302

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa996 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 946, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa996 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 946

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 946:

-> 0x317aa996: tst.w r0, #255

0x317aa99a: beq 0x317aa9e6 ; -[PSListController tableView:cellForRowAtIndexPath:] + 1026

0x317aa99c: movw r0, #60302

0x317aa9a0: mov r2, r11

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

……

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9ac Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 968, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa9ac Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 968

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 968:

-> 0x317aa9ac: blx 0x31825f04 ; symbol stub for: \_\_\_\_NETRBClientResponseHandler\_block\_invoke

0x317aa9b0: movw r0, #60822

0x317aa9b4: mov r2, r11

0x317aa9b6: movt r0, #1736

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9b0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 972, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa9b0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 972

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 972:

-> 0x317aa9b0: movw r0, #60822

0x317aa9b4: mov r2, r11

0x317aa9b6: movt r0, #1736

0x317aa9ba: add r0, pc

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

……

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa9c0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988:

-> 0x317aa9c0: blx 0x31825f04 ; symbol stub for: \_\_\_\_NETRBClientResponseHandler\_block\_invoke

0x317aa9c4: movw r0, #4312

0x317aa9c8: movt r0, #1737

0x317aa9cc: add r0, pc

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

(lldb) ni

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9c4 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 992, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x317aa9c4 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 992

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 992:

-> 0x317aa9c4: movw r0, #4312

0x317aa9c8: movt r0, #1737

0x317aa9cc: add r0, pc

0x317aa9ce: ldr r0, [r0]

(lldb) po [$r6 detailTextLabel]

<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); text = '+86PhoneNumber'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x15fd1c90>>

在0x317aa9c0处的objc\_msgSend前后PSTableCell的本机号码信息发生了变化，0x317aa9c0 - 0x6db3000 = 0x2A9F79C0，在IDA中定位到这个objc\_msgSend，如图6-41所示。

Obviously, my number appears after objc\_msgSend at 0x317aa9c0. Because 0x317aa9c0 - 0x6db3000 = 0x2A9F79C0, we can locate this address in IDA, as shown in figure 6-41.



图6- 41 设置本机号码的objc\_msgSend

Figure 6-41 The configuration objc\_msgSend

“用specifier刷新cell的内容”，这个方法的作用显而易见，我们看看这个specifier是什么。在这个objc\_msgSend上下个断点，触发后，打印它的参数：

As it name suggests, this method refreshes the cell contents with something specific. Let’s uncover this “something specific”: set a breakpoint at this objc\_msgSend, then trigger it and print its argument:

(lldb) br s -a 0x317AA9C0

Breakpoint 11: where = Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988, address = 0x317aa9c0

Process 268587 stopped

\* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988, queue = 'com.apple.main-thread, stop reason = breakpoint 11.1

frame #0: 0x317aa9c0 Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988

Preferences`-[PSListController tableView:cellForRowAtIndexPath:] + 988:

-> 0x317aa9c0: blx 0x31825f04 ; symbol stub for: \_\_\_\_NETRBClientResponseHandler\_block\_invoke

0x317aa9c4: movw r0, #4312

0x317aa9c8: movt r0, #1737

0x317aa9cc: add r0, pc

(lldb) p (char \*)$r1

(char \*) $97 = 0x318362d2 "refreshCellContentsWithSpecifier:"

(lldb) po $r2

My Number ID:myNumberCell 0x170ece60 target:<PhoneSettingsController 0x170ed760: navItem <UINavigationItem: 0x170d0b40>, view <UITableView: 0x16acb200; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x15d232d0>; layer = <CALayer: 0x15fc9110>; contentOffset: {0, -64}; contentSize: {320, 717.5}>>

(lldb) po [$r2 class]

PSSpecifier

可以看到，specifier是一个PSSpecifier对象，而且与本机号码相关。如果你在上一章的PreferenceBundle部分仔细阅读过preferences specifier plist标准，就知道PSTableCell的内容是由PSSpecifier指定的，因此可以通过[PSSpecifier propertyForKey:@”set”]和[PSSpecifier propertyForKey:@”get”]拿到PSSpecifier的setter和getter：

As the output shows, “something specific”, i.e. specifier, is a PSSpecifier object, and it’s tightly related to my number. If you have carefully read the preferences specifier plist standard in section PreferenceBundle of the last chapter, you would know that the contents of a PSTableCell is specified by PSSpecfier. Further more, we can acquire the setter and getter of PSSpecifier through [PSSpecifier propertyForKey:@“set”] and [PSSpecifier propertyForKey:@“get”] like this:

(lldb) po [$r2 propertyForKey:@"set"]

setMyNumber:specifier:

(lldb) po [$r2 propertyForKey:@"get"]

myNumber:

还可以通过[PSSpecifier target]拿到它们的target：

We also can get their target through [PSSpecifier target]:

(lldb) po [$r2 target]

<PhoneSettingsController 0x170ed760: navItem <UINavigationItem: 0x170d0b40>, view <UITableView: 0x16acb200; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x15d232d0>; layer = <CALayer: 0x15fc9110>; contentOffset: {0, -64}; contentSize: {320, 717.5}>>

非常好，现在我们知道PSTableCell的本机号码是通过[PhoneSettingsController setMyNumber:specifier:]方法设置的，通过[PhoneSettingsController myNumber:]读取的（你对它俩还有印象吗？），那么，在myNumber:内部，就一定有获取本机号码的方法，如图6-42所示。

Excellent! Now we know my number on PSTableCell is set through [PhoneSettingsController setMyNumber:specifier:], and is got through [PhoneSettingsController myNumber:] (Do you still remember these 2 methods?), so there must be a method inside myNumber: that returns my number, as shown in figure 6-42.

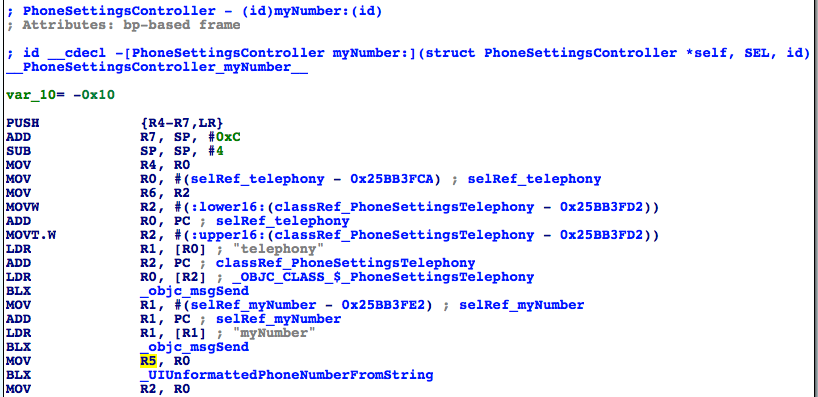


图6- 42 [PhoneSettingsController myNumber:]

Figure 6-42 [PhoneSettingsController myNumber:]

[PhoneSettingsController myNumber:]的逻辑比较简单，就是看[[PhoneSettingsTelephony telephony] myNumber]的长度是否为0，如果不为0，它就是本机号码，否则返回一个“未知号码”，告诉用户无法读取本机号码。用Cycript测试一下这个方法：

The implementation of [PhoneSettingsController myNumber:] is rather straightforward. This method simply checks whether the length of [[PhoneSettingsTelephony telephony] myNumber] is 0. If it is not 0, it is returned as my number, otherwise this method returns an “unknown number” as an error reminder. Let’s test [[PhoneSettingsTelephony telephony] myNumber] with Cycript:

FunMaker-5:~ root# cycript -p Preferences

cy# [[PhoneSettingsTelephony telephony] myNumber]

@"+86PhoneNumber"

现在，退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，再测试一次这个方法：

Now, press home button to quit Preferences, then terminate it completely and make sure it’s not running in the background. After that, launch it again and don’t enter MobilePhoneSettings for now, let’s test this method again:

FunMaker-5:~ root# cycript -p Preferences

cy# [[PhoneSettingsTelephony telephony] myNumber]

ReferenceError: Can't find variable: PhoneSettingsTelephony

出现了错误，这是怎么回事？那是因为PhoneSettingsTelephony是MobilePhoneSettings.bundle中的一个类，如果我们不进入MobilePhoneSettings界面，这个bundle是不会加载的，所以这个类也是不存在的。也就是说，要调用这个方法，需要先加载MobilePhoneSettings.bundle。Preference.app加载MobilePhoneSettings.bundle的方式被称为延迟加载（lazy load），在iOS逆向工程中出现类似状况的时候很多，当你碰到时，欢迎来<http://bbs.iosre.com>跟大家交流心得。

An error happens. What’s wrong? The reason is that PhoneSettingsTelephony is a class of MobilePhoneSettings.bundle. If we don’t enter MobilePhoneSettings, this bundle will not be loaded, so this class doesn’t exist. In other words, this method will only work after MobilePhoneSettings.bundle is loaded. The way Preference.app loads MobilePhoneSettings.bundle is called lazy load, which is common in iOS reverse engineering. When you come across it, welcome to share with us on <http://bbs.iosre.com>.

其实到此为止，可以认为我们已经找到了目标函数，因为我们拿到了这个方法的调用者和参数，而且这个方法不涉及UI操作，调用起来干净利落。但有一点让人不爽的是，调用这个方法前必须加载MobilePhoneSettings.bundle。有没有办法去掉这个硬指标，让我们不需要加载这个bundle就能拿到本机号码呢？应该存在这么一个方法。因为本机号码是存储在SIM卡上的，所以[PhoneSettingsTelephony myNumber]的原始数据源应该来自SIM卡。而能够访问SIM卡的显然不止MobilePhoneSetting.bundle，因此底层一定存在更通用的访问SIM卡的库，如果能定位到这个库，估计就可以直接读取本机号码了。既然是一个更底层的库，那么我们自然是要从[PhoneSettingsTelephony myNumber]入手，看看它的内部是如何读取本机号码的，如图6-43所示。

So far, we can say we have already found the target function, because we have got both the caller and arguments of this method, plus no UI operation is involved, we can call this method neatly. However, there is still a fly in the ointment: MobilePhoneSettings.bundle must be loaded, which weakens elegancy. Is there any way that enables us to get rid of this burden? I think so. Because ultimately, my number is stored on SIM card, the original data source of [PhoneSettingsTelephony myNumber] should come from SIM card. Whereas, SIM card accessibility is obviously not limited to MobilePhoneSetting.bundle, there must be a more common as well lower level library that can read SIM card. If we can locate this library, we can get my number without loading MobilePhoneSetting.bundle. Since it’s supposed to be a lower level library, naturally, we should dig into [PhoneSettingsTelephony myNumber] to find out how it reads my number, as shown in figure 6-43.

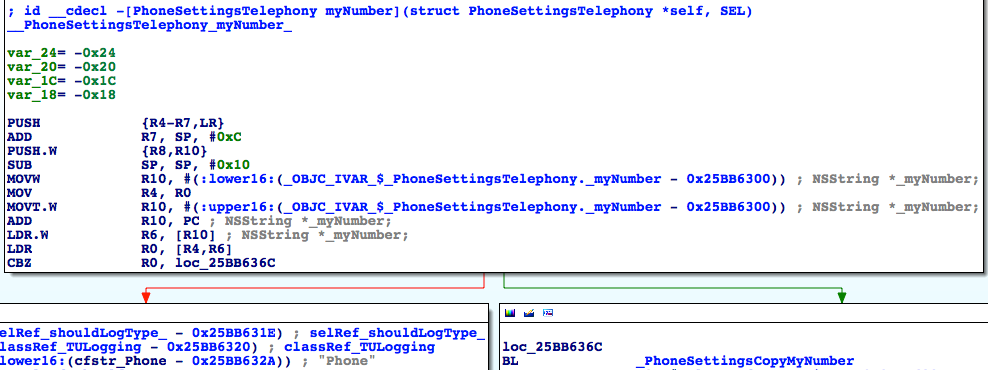


图6- 43 [PhoneSettingsTelephony myNumber]

Figure 6-43 [PhoneSettingsTelephony myNumber]

它的逻辑也比较简单，先取出实例变量\_myNumber，如果它不是nil，则走左边并记录“My Number requested, returning cached value: %@”，即返回一个缓存中的数据；否则走右边，先调用PhoneSettingsCopyMyNumber函数取得本机号码，再记录“My Number requested, no cached value, fetched: %@”，即没有在缓存中找到本机号码，返回一个现取的数据。因此，调用PhoneSettingsCopyMyNumber可以取得本机号码，但从名字来看，它仍然是MobilePhoneSettings.bundle里的一个函数，在这个bundle外不能调用，看来我们挖得还不够深。继续看看这个函数内部做了些什么，如图6-44所示。

This method is also very simple. It judges if the instance variable \_myNumber is nil; if not, branches left and records “My Number requested, returning cached value: %@”, namely, returns a data in cache; or else branches right, first get my number by calling PhoneSettingsCopyMyNumber, then records “My Number requested, no cached value, fetched: %@”, namely, my number is not in cache, so it returns a newly fetched data. In consequence, PhoneSettingsCopyMyNumber is able to get my number, but as its name suggests, it is still a function inside MobilePhoneSettings.bundle, we can’t call it from outside this bundle. We’re a step further, but not far enough. Let’s continue by digging into PhoneSettingsCopyMyNumber, as shown in figure 6-44.

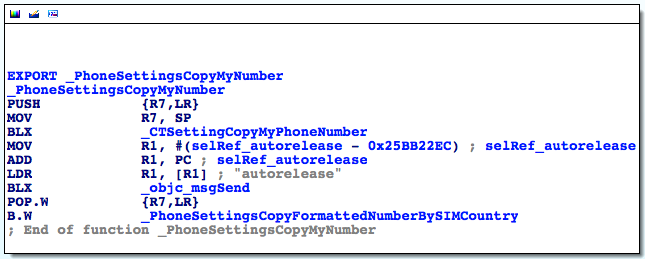


图6- 44 PhoneSettingsCopyMyNumber

Figure 6-44 PhoneSettingsCopyMyNumber

这段代码先调用CTSettingCopyMyPhoneNumber函数，把返回值给autorelease掉，然后再调用PhoneSettingsCopyFormattedNumberBySIMCountry，看其函数名好像是根据SIM卡所在的国家把号码给格式化了一下。那么CTSettingCopyMyPhoneNumber函数无论是从名字还是上下文来看，都非常疑似获取本机号码的函数，而且CT前缀说明它来自CoreTelephony，而不是MobilePhoneSettings。双击这个函数，看看它的内部实现，如图6-45所示。

This snippet first calls CTSettingCopyMyPhoneNumber and autorelease the return value, then calls PhoneSettingsCopyFormattedNumberBySIMCountry, which seems to format the phone number according to the country of the SIM card. Judging from the name and context, CTSettingCopyMyPhoneNumber looks like the target function we are looking for. And the prefix CT implies that it comes from CoreTelephony rather than MobilePhoneSettings. Double click this function to see its implementation, as shown in figure 6-45.

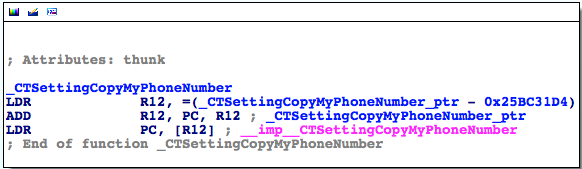


图6- 45 CTSettingCopyMyPhoneNumber

Figure 6-45 CTSettingCopyMyPhoneNumber

果然是一个外部函数，再次双击“\_\_imp\_\_CTSettingCopyMyPhoneNumber”，看看它来自哪个库——正是CoreTelephony。退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，然后用debugserver附加，用LLDB打印出image list，你会发现CoreTelephony赫然名列其中。这意味着，我们不需要加载MobilePhoneSettings.bundle就可以调用CTSettingCopyMyPhoneNumber获取未经格式化的本机号码，它就是我们要找的目标函数。那么还剩最后一个问题——它的参数和返回值是什么？

As expected, it’s an external function. Double click “\_\_imp\_\_CTSettingCopyMyPhoneNumber” to check out which library it originates from; it’s exactly CoreTelephony. Quit Preferences and terminate it completely in the background, then relaunch it and don’t enter MobilePhoneSettings. Now let’s attach debugserver to it and take a look at its image list with LLDB, we will see CoreTelephony is on the list. It means that we can call CTSettingCopyMyPhoneNumber to get my unformatted number without loading MobilePhoneSettings.bundle, which perfectly meets our requirements of a target function. Finally, the last question: what’s its arguments and return value?

从图6-44看来，CTSettingCopyMyPhoneNumber不像是有参数——它的前面甚至没有出现R0~R3寄存器。如果它有参数，那么R0~R3也是来自它的调用者，即PhoneSettingsCopyMyNumber。但从图6-43看来，PhoneSettingsCopyMyNumber之前也只出现了R0，且如果进程走右边，R0一定是0，PhoneSettingsCopyMyNumber看起来也没有参数。为了保险起见，还是去CoreTelephony里看看CTSettingCopyMyPhoneNumber的实现，如图6-46所示。

Judging from figure 6-44, CTSettingCopyMyPhoneNumber doesn’t seem to have any argument; before CTSettingCopyMyPhoneNumber, R0~R3 don’t even show at all. If it has any argument, then R0~R3 come from its caller, i.e. PhoneSettingsCopyMyNumber. However, as we can see in figure 6-43, before PhoneSettingsCopyMyNumber, only R0 occurs, and if it branches right, R0 is permanently 0, if R0 is an argument, it’s meaningless. Therefore, PhoneSettingsCopyMyNumber doesn’t seem to have any argument either. To play it safe, let’s reconfirm our guesses by checking the implementation of CTSettingCopyMyPhoneNumber in CoreTelephony, as shown in figure 6-46.

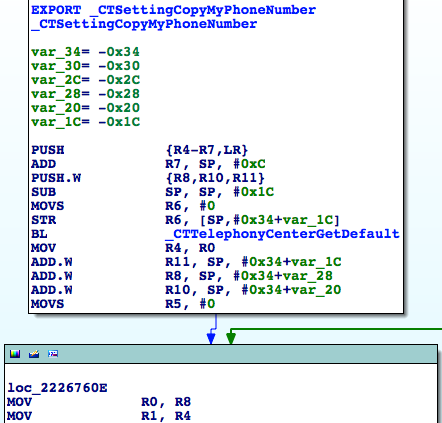


图6- 46 CTSettingCopyMyPhoneNumber

Figure 6-46 CTSettingCopyMyPhoneNumber

根据Objective-C函数的命名惯例，CTTelephonyCenterGetDefault是有返回值的；在“BL \_CTTelephonyCenterGetDefault”下面，R0被CTTelephonyCenterGetDefault的返回值覆盖掉了；而在图6-46的最下面，R1也被“MOV R1, R4”中的R4覆盖掉了。如果R0和R1是参数，那么这2个参数就没有起任何作用，不合常理，因此说明CTSettingCopyMyPhoneNumber没有参数。那么它的返回值呢？我们会很自然地猜测它的返回值是一个字符串，但为了保险起见，还是在CTSettingCopyMyPhoneNumber的尾部下个断点，把R0打印出来看看吧。先在IDA中看看它的地址，如图6-47所示。

According to the naming conventions of Objective-C functions, CTTelephonyCenterGetDefault is a getter and should return something; as a result, R0 under “BL \_CTTelephonyCenterGetDefault” is set to the return value of CTTelephonyCenterGetDefault. Meanwhile, at the bottom of figure 6-46, R1 is set to R4 in “MOV R1, R4”. If R0 and R1 are arguments, then they are useless, which doesn’t make sense. Now we can say for sure that CTSettingCopyMyPhoneNumber has no argument. What about its return value? We naturally guess it’s an NSString object. Let’s verify it by setting a breakpoint at the end of CTSettingCopyMyPhoneNumber, and print out R0. First locate to the end of CTSettingCopyMyPhoneNumber in IDA, as shown in figure 6-47.

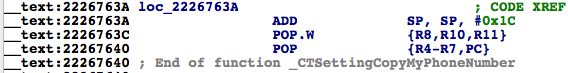


图6- 47 CTSettingCopyMyPhoneNumber

Figure 6-47 CTSettingCopyMyPhoneNumber

然后退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，然后用debugserver附加，用LLDB查看CoreTelephony的ASLR偏移：

Then quit Preferences and terminate it completely in the background, then relaunch it and don’t enter MobilePhoneSettings. Next attach debugserver to it and take a look at CoreTelephony’s ASLR offset with LLDB:

(lldb) image list -o -f

[ 0] 0x000b3000 /private/var/db/stash/\_.29LMeZ/Applications/Preferences.app/Preferences(0x00000000000b7000)

[ 1] 0x0026c000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x000000000026c000)

[ 2] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard [ 51] 0x06db3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/System/Library/Frameworks/CoreTelephony.framework/CoreTelephony

……

我们就把断点下在0x6db3000 + 0x2226763A = 0x2901A63A上吧。然后进入MobilePhoneSettings界面，触发断点：

The breakpoint should be set at 0x6db3000 + 0x2226763A = 0x2901A63A, right? Then enter MobilePhoneSettings to trigger the breakpoint:

(lldb) br s -a 0x2901A63A

Breakpoint 1: where = CoreTelephony`CTSettingCopyMyPhoneNumber + 78, address = 0x2901a63a

Process 330210 stopped

\* thread #1: tid = 0x509e2, 0x2901a63a CoreTelephony`CTSettingCopyMyPhoneNumber + 78, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x2901a63a CoreTelephony`CTSettingCopyMyPhoneNumber + 78

CoreTelephony`CTSettingCopyMyPhoneNumber + 78:

-> 0x2901a63a: add sp, #28

0x2901a63c: pop.w {r8, r10, r11}

0x2901a640: pop {r4, r5, r6, r7, pc}

0x2901a642: nop

(lldb) po $r0

+86PhoneNumber

(lldb) po [$r0 class]

\_\_NSCFString

它就是一个NSString，这样就可以还原这个函数的原型啦——

It is indeed an NSString, so the prototype of this function can be reconstructed:

NSString \*CTSettingCopyMyPhoneNumber(void);

它就是我们的目标函数，也就是PSTableCell的数据源，我们通过分析[PhoneSettingsController tableView:cellForRowAtIndexPath:]所在的函数调用链找到了它。在调用它的时候，注意参数传0，然后释放返回值就好了。写一个小tweak测测这个函数，确保它是正确的。

This is our target function, as well the data source of PSTableCell. We’ve finally found it through analyzing the call chain of [PhoneSettingsController tableView:cellForRowAtIndexPath:], hurray! Just remember to release the return value when we make use of this function. At last, let’s write a tweak to test this function.

1. 用Theos新建tweak工程“iOSREGetMyNumber”，命令如下：

Create tweak project “ iOSREGetMyNumber” using Theos:

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): iOSREGetMyNumber

Package Name [com.yourcompany.iosregetmynumber]: com.iosre.iosregetmynumber

Author/Maintainer Name [snakeninny]: snakeninny

[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]: com.apple.Preferences

[iphone/tweak] List of applications to terminate upon installation (space-separated, '-' for none) [SpringBoard]: Preferences

Instantiating iphone/tweak in iosregetmynumber/...

Done.

1. 编辑Tweak.xm，代码如下：

Edit Tweak.xm as follows:

extern "C" NSString \*CTSettingCopyMyPhoneNumber(int); // From CoreTelephony

%hook PreferencesAppController

- (BOOL)application:(id)arg1 didFinishLaunchingWithOptions:(id)arg2

{

BOOL result = %orig;

NSLog(@"iOSRE: my number = %@", [CTSettingCopyMyPhoneNumber() autorelease]);

return result;

}

%end

1. 编辑Makefile以及control

Edit Makefile and control

编辑后的Makefile内容如下：

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 = iOSREGetMyNumber

iOSREGetMyNumber\_FILES = Tweak.xm

iOSREGetMyNumber\_FRAMEWORKS = CoreTelephony # CTSettingCopyMyPhoneNumber来自这里

include $(THEOS\_MAKE\_PATH)/tweak.mk

after-install::

install.exec "killall -9 Preferences"

编辑后的control内容如下：

The finalized control looks like this:

Package: com.iosre.iosregetmynumber

Name: iOSREGetMyNumber

Depends: mobilesubstrate, firmware (>= 8.0)

Version: 1.0

Architecture: iphoneos-arm

Description: Get my number just like MobilePhoneSettings!

Maintainer: snakeninny

Author: snakeninny

Section: Tweaks

Homepage: http://bbs.iosre.com

1. 测试

Test

将写好的tweak编译打包安装到iOS后，打开Preferences，不要进入MobilePhoneSettings界面。然后ssh到iOS上看看syslog：

Compile and install the tweak on iOS, then launch Preferences without entering MobilePhoneSettings. After that, ssh into iOS and take a look at the syslog:

FunMaker-5:~ root# grep iOSRE: /var/log/syslog

Nov 29 23:23:01 FunMaker-5 Preferences[2078]: iOSRE: my number = +86PhoneNumber

1. 补充

P.S.

因为笔者的iPhone 5将地区设置为了美国，所以格式化之前的本机号码是“+86PhoneNumber”，被PhoneSettingsCopyFormattedNumberBySIMCountry格式化之后变成了“+86 Pho-neNu-mber”，即美国电话号码格式。

I have set the region of my iPhone 5 to US, so PhoneSettingsCopyFormattedNumberBySIMCountry has formatted my number from “+86PhoneNumber” to “+86 Pho-neNu-mber”, which the American phone number format.

在逆向其他目标碰到CTSettingCopyMyPhoneNumber时，随着iOS逆向工程熟练度的增加，你就会慢慢发现，它的正确原型其实是：

You’ll run into CTSettingCopyMyPhoneNumber more frequently as your hands get dirtier. Actually, the prototype of CTSettingCopyMyPhoneNumber is:

CFStringRef CTSettingCopyMyPhoneNumber(void);

因为NSString \*和CFStringRef是等价的，而kCFAllocatorDefault与NULL是等价的，所以我们的写法也没问题。

Since NSString \* and CFStringRef are toll-free bridged, our prototype is OK.

因为CTSettingCopyMyPhoneNumber的函数名中含有“copy”字样，且它返回了一个CoreData对象，所以根据苹果的“Ownership Policy”（Google搜索“apple ownership policy”），我们要负责释放这个函数的返回值。

Because there is a keyword “copy” in the name of CTSettingCopyMyPhoneNumber and it returns a CoreData object, we are responsible to release the return value according to Apple’s “Ownership Policy”.

本节用大量篇幅，用ARM汇编完善了“定位目标函数”环节，并将其细分为“从现象切入App，找出UI函数”和“以UI函数为起点，寻找目标函数”两步，结合Cycript、IDA和LLDB，既定位了目标函数，又解析了一些不够直观的函数参数。两个例子中演示的套路基本可以应付现在95%的App，如果你有幸碰到了那5%搞不定的，欢迎来<http://bbs.iosre.com>提供案例，我们一起来寻求解决方案。

In this section, we have shed considerable light to refine “locate target functions” with ARM level reverse engineering and enhanced the methodology of writing a tweak. Specifically, we’ve divided “locate target functions” into 2 steps, i.e. “cut into the target App and find the UI function” and “locate the target function from the UI function”. By combining Cycript, IDA and LLDB, we have not only located the target functions, but also analyzed their arguments to reconstruct their prototypes. The methodology we used in the examples can work on at least 95% of all Apps; however, if you unfortunately encounter those 5%, please share and discuss with us on <http://bbs.iosre.com>.

6.3 LLDB的使用技巧

6.3 Advanced LLDB usage

上一节是不是为你开启了iOS逆向工程的另一扇门？IDA和LLDB的配合简直是无坚不摧，再配合ARM指令集文档，似乎已经达成了“它俩在手，天下我有”的境界。你是不是已经迫不及待，想要废寝忘食地开始实践刚学到的新知识了呢？

I bet the last section has opened a new chapter of iOS reverse engineering for you. The combination of IDA and LLDB can easily beat them all, and with the help of ARM architecture reference manual, you can conquer almost all Apps. I know you’re already desperate to practice what you have just learned.

先别急。6.2节的2个例子虽然已经综合运用了IDA和LLDB，但仍没有涵盖LLDB的常用场景。因此下面以几个短例示范一下LLDB的使用技巧，它们在实战中的合理运用能够大大减少我们的工作量。

Hold your horses for now. Although the examples in section 6.2 have synthetically made use of IDA and LLDB, they haven’t covered LLDB’s common usage yet. In the next section, we’ll go over some short LLDB examples for a better comprehension, which can greatly reduce our workload in practice.

6.3.1 寻找函数调用者

6.3.1 Look for a function’s caller

在上一节的2个例子里，我们在还原函数调用链时，主要分析的是一个函数调用了哪些函数，也就是还原了函数调用链的下游。当我们需要追溯函数调用链上游的时候，那就需要分析一个函数的调用者是谁了。看这样一段代码：

In the examples of the previous section, when we were restoring call chains, we primarily focused on the callees of a function, i.e. we’ve restored the bottom half of a call chain. When we’re to restore the top half, we need to find out the caller of a function. Look at this snippet:

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

#include <stdio.h>

#include <dlfcn.h>

#import <Foundation/Foundation.h>

extern void TestFunction0(void)

{

NSLog(@"iOSRE: %u", arc4random\_uniform(0));

}

extern void TestFunction1(void)

{

NSLog(@"iOSRE: %u", arc4random\_uniform(1));

}

extern void TestFunction2(void)

{

NSLog(@"iOSRE: %u", arc4random\_uniform(2));

}

extern void TestFunction3(void)

{

NSLog(@"iOSRE: %u", arc4random\_uniform(3));

}

int main(int argc, char \*\*argv)

{

TestFunction3();

return 0;

}

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拖进IDA，并查看NSLog的交叉引用，如图6-48所示。

Save this snippet as a file named main.m, and compile it with the sentence in the comments. Drag and drop MainBinary into IDA, and then check the cross references of NSLog, as shown in figure 6-48.

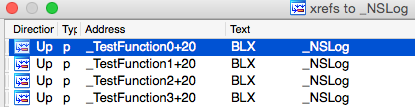


图6- 48 查看NSLog的交叉引用

Figure 6-48 Check the cross references of NSLog

可以看到，在这段代码中，NSLog出现在了4个函数里，如果我们在逆向的时候发现syslog中出现了“iOSRE: 0”，那么这个输出到底是来自哪个NSLog呢？当代码的逻辑比较简单时，靠人工就可以指出只有TestFunction3得到了调用，它进而又调用了NSLog。可如果这里有20个TestFunction，分别被8个不同的函数调用呢？逻辑变得复杂，人工分析就很吃力了。在这种情况下要寻找NSLog的调用者，LLDB就能起到很大的作用；用LLDB寻找函数调用者，主要有2种方法。

As we can see, NSLog appears in 4 functions. If we see “iOSRE: 0” in syslog when we are reversing, how can we know which NSLog it’s from? When there’re only tens lines of code, we can figure out by hand that only TestFunction3 is called, and it further calls NSLog. What if there are 20 TestFunctions which are called by 8 separate functions? When the amount of code increases, it’ll be too complicate to analyze manually. If we want to find the caller of NSLog under such circumstances, LLBD will be very helpful. Generally, there are 2 main methods.

1. 查看LR

1. Inspect LR

还记得6.1.3节介绍的LR寄存器吗？它的作用是保存返回地址。什么是返回地址？举个例子：

Still remember LR register introduced in section 6.1? Its function is to save the return address of a function. So what’s a return address? Take an example:

void FunctionA()

{

……

FunctionB();

……

}

在上面的伪代码中，FunctionA调用FunctionB，而A和B一般位于内存中的2块不同区域，它们的地址没有直接关联。B执行结束后，需要回到A里继续执行接下来的指令，如图6-49所示。

In the above pseudo code, FunctionA calls FunctionB, while A and B are located in 2 different memory areas, and their addresses have no direct connection. After the execution of B, the process needs to go back to A to continue execution, as shown in figure 6-49.

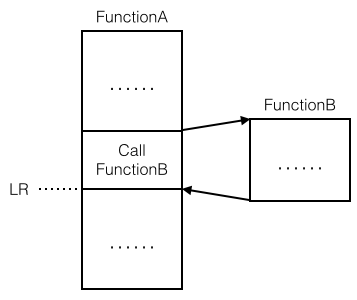


图6- 49 返回地址示意图

Figure 6-49 An illustration of return address

B执行结束后返回的那个地方，就是返回地址。因为它位于调用者的内部，所以如果能知道LR的值，就可以知道调用者是谁；概念不好懂，操作一遍你就全明白了。先把Foundation.framework的二进制文件拖进IDA，初始分析结束后定位到NSLog，查看其基地址，如图6-50所示。

The address that the process returns to after the execution of B, is the return address, i.e. LR. Because it’s inside B’s caller, if we know the value of LR we can track to the caller. Let’s explain this theory with an example. Drag and drop Foundation.framework’s binary into IDA; locate to NSLog after the initial analysis, and check out its base address, as shown in figure 6-50.

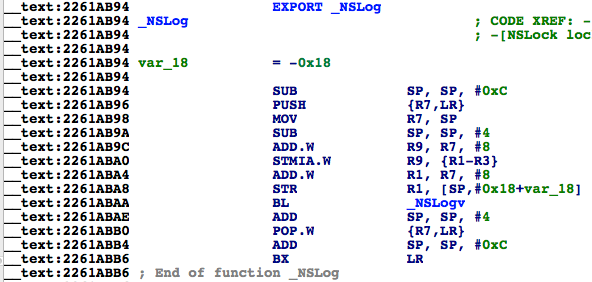


图6- 50 查看NSLog基地址

Figure 6-50 Check out NSLog’s base address

它的基地址是0x2261ab94，等会我们要在它上下断点，打印LR的值。接着用debugserver启动MainBinary：

Its base address is 0x2261ab94, we will set a breakpoint on it shortly and print out the value of LR. Next, launch MainBinary with debugserver:

FunMaker-5:~ root# debugserver -x backboard \*:1234 /var/tmp/MainBinary

debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89

for armv7.

Listening to port 1234 for a connection from \*...

再用LLDB连过去：

Then connect with LLDB:

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

Process 450336 stopped

\* thread #1: tid = 0x6df20, 0x1fec7000 dyld`\_dyld\_start, stop reason = signal SIGSTOP

frame #0: 0x1fec7000 dyld`\_dyld\_start

dyld`\_dyld\_start:

-> 0x1fec7000: mov r8, sp

0x1fec7004: sub sp, sp, #16

0x1fec7008: bic sp, sp, #7

0x1fec700c: ldr r3, [pc, #112] ; \_dyld\_start + 132

(lldb) image list -f

[ 0] /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/usr/lib/dyld

此时MainBinary还未启动，我们位于dyld内部。接下来，一直执行“ni”命令，直到出现“error: invalid thread”的提示：

Right at this moment, MainBinary is not run yet, and we are inside dyld. Next, keep entering “ni” until LLDB outputs “error: invalid thread”:

(lldb) ni

Process 450336 stopped

\* thread #1: tid = 0x6df20, 0x1fec7004 dyld`\_dyld\_start + 4, stop reason = instruction step over

frame #0: 0x1fec7004 dyld`\_dyld\_start + 4

dyld`\_dyld\_start + 4:

-> 0x1fec7004: sub sp, sp, #16

0x1fec7008: bic sp, sp, #7

0x1fec700c: ldr r3, [pc, #112] ; \_dyld\_start + 132

0x1fec7010: sub r0, pc, #8

(lldb)

Process 450336 stopped

\* thread #1: tid = 0x6df20, 0x1fec7008 dyld`\_dyld\_start + 8, stop reason = instruction step over

frame #0: 0x1fec7008 dyld`\_dyld\_start + 8

dyld`\_dyld\_start + 8:

-> 0x1fec7008: bic sp, sp, #7

0x1fec700c: ldr r3, [pc, #112] ; \_dyld\_start + 132

0x1fec7010: sub r0, pc, #8

0x1fec7014: ldr r3, [r0, r3]

……

(lldb)

error: invalid thread

到这里，不要再执行“ni”命令了，此时dyld开始加载MainBinary，等待一会，进程又会停下来，这时我们已经在MainBinary内部，可以开始调试了：

No more “ni” when the error occurs; now dyld begins to load MainBinary. Wait a moment, the process will stop again, and we are inside MainBinary, it’s okay to debug:

Process 450336 stopped

\* thread #1: tid = 0x6df20, 0x1fec7040 dyld`\_dyld\_start + 64, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x1fec7040 dyld`\_dyld\_start + 64

dyld`\_dyld\_start + 64:

-> 0x1fec7040: ldr r5, [sp, #12]

0x1fec7044: cmp r5, #0

0x1fec7048: bne 0x1fec7054 ; \_dyld\_start + 84

0x1fec704c: add sp, r8, #4

下面看看Foundation.framework的ASLR偏移：

Check out ASLR offset of Foundation.framework:

(lldb) image list -o -f

[ 0] 0x000fc000 /private/var/tmp/MainBinary(0x0000000000100000)

[ 1] 0x000c6000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/usr/lib/dyld

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

……

断点下在0x6db3000 + 0x2261ab94 = 0x293CDB94。然后执行“c”命令，触发断点：

As usual, we should set the breakpoint at 0x6db3000 + 0x2261ab94 = 0x293CDB94. Execute “c” to trigger the breakpoint:

(lldb) br s -a 0x293CDB94

Breakpoint 1: where = Foundation`NSLog, address = 0x293cdb94

(lldb) c

Process 450336 resuming

Process 450336 stopped

\* thread #1: tid = 0x6df20, 0x293cdb94 Foundation`NSLog, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x293cdb94 Foundation`NSLog

Foundation`NSLog:

-> 0x293cdb94: sub sp, #12

0x293cdb96: push {r7, lr}

0x293cdb98: mov r7, sp

0x293cdb9a: sub sp, #4

最后打印LR的值：

Print out LR:

(lldb) p/x $lr

(unsigned int) $0 = 0x00107f8d

因为MainBinary的基地址是0x000fc000，所以在IDA里打开MainBinary，然后跳转到0x107f8d - 0xfc000 = 0xBF8D，如图6-51所示。

Because the base address of MainBinary is 0x000fc000, open MainBinary in IDA and jump to 0x107f8d - 0xfc000 = 0xBF8D, as shown in figure 6-51.

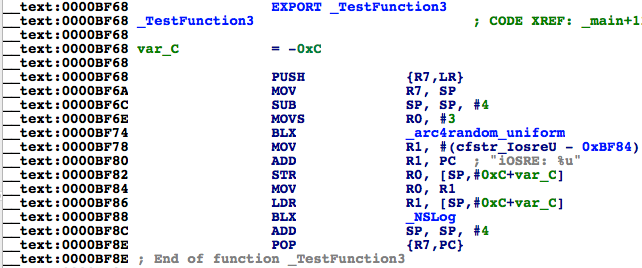


图6- 51 TestFunction3

Figure 6-51 TestFunction3

它位于TestFunction3中“BLX \_NSLog”的正下方，我们找到了NSLog的调用者。有一点需要强调的是，因为LR在被调用者内部可能会产生变化，所以断点一定要下在基地址上。很简单吧？

0xBF8D is right below “BLX \_NSLog”, so we have found the caller of NSLog. One thing should be noted is that because LR may change in the caller, the breakpoint should be set at the base address. Pretty easy, huh?

1. 执行“ni”命令到调用者内部

2. Execute “ni” to get inside caller

虽然“查看LR”的方法很简单，但在上面的例子里，我们耍了个小花样：因为事先知道MainBinary调用了NSLog，所以才用LR减去MainBinary的ASLR偏移得到地址，然后在IDA中跳过去。而一般情况下，我们不知道哪个函数调用了NSLog，更不知道哪个模块调用了NSLog，因此也就不知道该用LR减去谁的ASLR偏移了。要解决这个问题，我们的理论依据仍是“B执行结束后，需要回到A里，继续执行接下来的指令”——只要在被调用者的末尾下个断点，然后一直执行“ni”命令，就会回到调用者内部，从而发现调用者。还是来操作一遍：重复上面的步骤，用debugserver重新启动MainBinary，用LLDB挂接过去，直到进入MainBinary内部，然后查看Foundation.framework的ASLR偏移：

Although “Inspect LR” is straightforward enough, but we’ve played a trick: because we’ve already known NSLog is called inside MainBinary, we’ve subtracted MainBinary’s ASLR offset from LR to get the final result. But in more general cases, we don’t know which function calls NSLog, not to mention which image calls NSLog, so we don’t know whose ASLR offset should be subtracted from LR. To solve this problem, our theoretical base is still “After the execution of B, the process needs to go back to A to continue execution”; if we set a breakpoint at the end of the callee and keep executing “ni”, we will come back to the caller. Let’s take another example: repeat the steps in last section to check out ASLR offset of Foundation.framework in MainBinary:

(lldb) image list -o -f

[ 0] 0x0000c000 /private/var/tmp/MainBinary(0x0000000000010000)

[ 1] 0x000c5000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/usr/lib/dyld

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

……

它的ASLR偏移是0x6db3000。依图6-50，NSLog最后一条指令的地址是0x2261ABB6，因此，在0x6db3000 + 0x2261ABB6 = 0x293CDBB6上下一个断点，然后“c”一下，触发断点：

Its ASLR offset is 0x6db3000. According to figure 6-50, the address of the last instruction of NSLog is 0x2261ABB6, so set a breakpoint at 0x6db3000 + 0x2261ABB6 = 0x293CDBB6, then enter “c” to trigger the breakpoint:

(lldb) br s -a 0x293CDBB6

Breakpoint 1: where = Foundation`NSLog + 34, address = 0x293cdbb6

(lldb) c

Process 452269 resuming

(lldb) 2014-11-30 23:45:37.070 MainBinary[3454:452269] iOSRE: 1

Process 452269 stopped

\* thread #1: tid = 0x6e6ad, 0x293cdbb6 Foundation`NSLog + 34, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x293cdbb6 Foundation`NSLog + 34

Foundation`NSLog + 34:

-> 0x293cdbb6: bx lr

Foundation`NSLogv:

0x293cdbb8: push {r4, r5, r6, r7, lr}

0x293cdbba: add r7, sp, #12

0x293cdbbc: sub sp, #12

注意“->”上方的文字，它指示了当前的模块。接着执行“ni”命令：

Notice the texts above “->”, it implies the present image. Keep executing “ni”:

(lldb) ni

Process 452269 stopped

\* thread #1: tid = 0x6e6ad, 0x00017fa6 MainBinary`main + 22, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x00017fa6 MainBinary`main + 22

MainBinary`main + 22:

-> 0x17fa6: movs r0, #0

0x17fa8: movt r0, #0

0x17fac: add sp, #12

0x17fae: pop {r7, pc}

进入了MainBinary，停在了0x17fa6。0x17fa6 – 0xc000 = 0xbfa6，对照图6-51，我们找到了NSLog的调用者TestFunction3。

Here comes MainBinary and the process stops at 0x17fa6. 0x17fa6 – 0xc000 = 0xbfa6, so again, we have found NSLog’s caller TestFunction3 according to figure 6-51.

两种寻找调用者的方法都很简单粗暴，大家根据自己的喜好随便选一种就可以了。

Both methods are simple and direct, choose whatever you like.

6.3.2 更改进程执行逻辑

6.3.2 Change process execution flow

为什么要更改进程执行逻辑？最常见的原因之一是因为有些时候，你想要调试的代码需要满足一定的条件才能触发执行，而这种条件不借助外界力量很难重现，所以可以更改进程执行逻辑，把进程引导向你的目标代码，从而调试它们。这句话听起来很拗口，举一个例子你就清楚了。看这样一段代码：

Why do we need to change process execution flow? Commonly it’s because the code we want to debug could only be executed in specific conditions, which are hard to meet in the original execution flow. Under such circumstances, we have to change the flow to redirect the process to execute the target code for debugging. Reads awkward? Let’s see an example.

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

#include <stdio.h>

#include <dlfcn.h>

#import <Foundation/Foundation.h>

#import <UIKit/UIKit.h>

extern void ImportantAndComplicatedFunction(void)

{

NSLog(@"iOSRE: Suppose I'm a very important and complicated function");

}

int main(int argc, char \*\*argv)

{

if ([[[UIDevice currentDevice] systemVersion] isEqualToString:@"8.1.1"]) ImportantAndComplicatedFunction();

return 0;

}

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拷到iOS的“/var/tmp/”下：

Save this snippet as a file named main.m, and compile it with the sentence in the comments, then copy MainBinary to “/var/tmp/” on iOS:

snakeninnys-MacBook:6 snakeninny$ scp MainBinary root@iOSIP:/var/tmp/

MainBinary 100% 49KB 48.6KB/s 00:00

运行它，看看效果：

Run it:

FunMaker-5:~ root# /var/tmp/MainBinary

FunMaker-5:~ root#

因为笔者的iOS系统是8.1，所以自然没有任何输出。笔者对ImportantAndComplicatedFunction很感兴趣，想动态调试它，但手头没有8.1.1的系统，怎么办呢？那就动态更改代码，让这个函数得到执行。下面来操作一遍，请读者注意观察。先把MainBinary拖进IDA，定位到ImportantAndComplicatedFunction被调用之前的指令，如图6-52所示。

Because I’m using iOS 8.1, there is no output for sure. What if I am interested in ImportantAndComplicatedFunction but don’t have iOS 8.1.1 in hand? Then I have to dynamically change the execution flow to make this function gets called. I’ll show you how, please keep focuses. Drag and drop MainBinary into IDA, then locate to the branch before ImportantAndComplicatedFunction, as shown in figure 6-52.

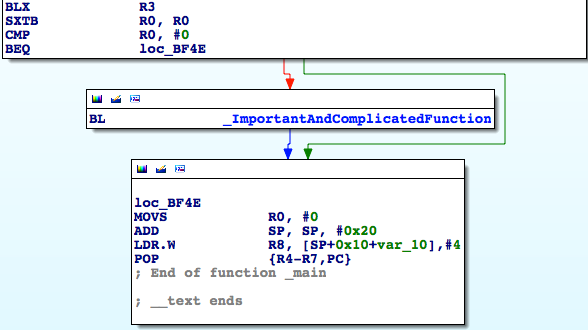


图6- 52 ImportantAndComplicatedFunction得到调用之前

Figure 6-52 Before ImportantAndComplicatedFunction

然后用debugserver启动MainBinary，用LLDB挂接过去，直到进入MainBinary内部，再查看MainBinary的ASLR偏移：

Repeat the previous steps to check out MainBinary’s ASLR offset:

(lldb) image list -o -f

[ 0] 0x0000e000 /private/var/tmp/MainBinary(0x0000000000012000)

……

因为图6-52最上面的那个“CMP R0, #0”地址是0xBF46，所以把断点下在0xbf46 + 0xe000 = 0x19F46，然后“c”一下触发它，然后看看R0的值：

Because the address of “CMP R0, #0” in figure 6-52 is 0xBF46, the breakpoint should be set at 0xbf46 + 0xe000 = 0x19F46. Trigger it with “c”, and print R0:

(lldb) br s -a 0x19F46

Breakpoint 1: where = MainBinary`main + 134, address = 0x00019f46

(lldb) c

Process 456316 resuming

Process 456316 stopped

\* thread #1: tid = 0x6f67c, 0x00019f46 MainBinary`main + 134, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x00019f46 MainBinary`main + 134

MainBinary`main + 134:

-> 0x19f46: cmp r0, #0

0x19f48: beq 0x19f4e ; main + 142

0x19f4a: bl 0x19ea4 ; ImportantAndComplicatedFunction

0x19f4e: movs r0, #0

(lldb) p $r0

(unsigned int) $0 = 0

R0是0，因此ImportantAndComplicatedFunction得不到执行。如果把R0改成1，情况就不同了：

R0 is 0, so ImportantAndComplicatedFunction will be executed. If we change R0 to 1, the situation changes all together:

(lldb) register write r0 1

(lldb) p $r0

(unsigned int) $1 = 1

(lldb) c

Process 456316 resuming

(lldb) 2014-12-01 00:41:47.779 MainBinary[3482:457105] iOSRE: Suppose I'm a very important and complicated function

Process 456316 exited with status = 0 (0x00000000)

我们通过动态更改寄存器的值来改变进程执行逻辑，达到了我们的目的。

As we can see, we’ve changed the process execution flow by modifying the value of a register, thus achieved our goal.

6.4 小结

6.4 Conclusion

IDA和LLDB两大神器的作用当然不止于本章所介绍的这些，它们的有效范围很广，小到分析App，大到动手越狱，是两款“老少咸宜”的工具。不过，相信在iOS逆向工程初级阶段，大家应用它们的场合不会超出本章的内容范围。当然，熟练掌握它们以后，对iOS的理解一定会上升到一个新的层次；届时，大家就能举一反三，根据自己的需求摸索它们的新用法了。在ARM汇编级别的iOS逆向工程里，值得悉心研究的课题还有很多，我们会在<http://bbs.iosre.com>上展开旷日持久的讨论。

The combination of IDA and LLDB is far more powerful than what we have introduced in this chapter, their usage ranges from App analysis to jailbreak, showing their omnipotence. Nonetheless, in the beginning stage of iOS reverse engineering, their usage is not likely to exceed the scope of this chapter. As soon as you can use them proficiently, your understanding of iOS would rise to a new level and you'll be able to summarize your own methodologies. There’re still lots and lots of topics in ARM related iOS reverse engineering to further explore, and we’re unable to cover them all in one book. Therefore, we will leave them to <http://bbs.iosre.com>, please stay focused.

本章的内容虽然有些艰深，但却是入门iOS逆向工程的基础。接下来的4章会将本章内容运用到实战中去，在阅读完那些内容之后，大家就能根据自己的掌握情况判断是知难而退，还是迎难而上了。无论如何，这是一个很有意思的方向，能走多远则完全看个人的功底和兴趣了。

To be honest, this chapter is rather difficult to understand, but this is the only path to be a real iOS reverse engineer. In part 4 of this book, we will turn methodologies in part 3 into practices and write 4 tweaks. I hope you know from the bottom of your heart whether you are talented enough to be an iOS reverse engineer after finishing all 4 practices. As Steve Jobs said, “It's more fun to be a pirate than to join the Navy”. IMHO, being an iOS reverse engineer is way more fun than being just an App developer, but it’s all up to you. Good luck!