# 第6章 ARM汇编相关的iOS逆向理论基础

# Chapter 6 ARM related iOS reverse engineering

前面的章节中介绍了iOS逆向工程的基础知识，包括一些常见工具的组合使用，在掌握了这些知识之后，简单地把玩一下Objective-C私有函数，满足一下自己的好奇心已经没问题了，可以针对App开发tweak了。但是，既然看到了这里，相信大家都具有比较强的钻研精神，如果想要真正提高自己的能力，就要尝试一些更有挑战性的内容。那么，从这一章开始，iOS逆向工程就将进入“极昼”，我们将零距离接触编程世界里最让人头大的知识。请先深呼吸一分钟，然后问问自己：“我是否真的适合深入学习iOS逆向工程？”在完成本章之后，相信你会得到答案。

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

下面即将面对iOS逆向工程中的第一个进阶难点：阅读ARM汇编语言。经过前几章的学习，相信大家已经知道，Objective-C代码在经过编译后形成机器码，它们由设备的CPU直接执行。别说编写，阅读机器码都已经是一个非常恼人的工作；好在Objective-C和机器码之间有汇编语言这座桥，它的可读性虽然远不如Objective-C，但比机器码要强多了——如果你能够啃下这块硬骨头，那么恭喜你，你有着成为逆向工程师的天份；如果你在啃骨头的时候牙被崩掉了，或许AppStore开发才是你更好的归宿……

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

## 6.1 ARM汇编基础

## 6.1 Introduction to ARM assembly

对于很多iOS开发者来说，ARM汇编是一门全新的语言；如果你是计算机专业科班出身，应该已经对汇编语言有了初步的印象，只是对于很多人来说，大学期间的汇编语言课简直跟天书一样深奥，它在我们心里埋下了恐惧的种子，仿佛一提到汇编语言，它就会像紧箍咒一样勒紧我们的头，让我们疼痛不已。汇编语言真的有这么难？是，因为汇编的语法晦涩难懂；但另一方面，毕竟它只是一门语言，跟英语一样，熟能生巧。

ARM assembly is a brand new language to most iOS developers. If your major in college is Computer related, you may already have some impression about assembly. Actually, assembly is too esoteric for most college students; we’re nervous and uncomfortable dealing with it. Is assembly really too hard to learn? Yes, it’s obscure and difficult to understand. On the other hand, however, as a human readable language, it is no much difference with other human languages, namely, if we use it more often, we will get familiar with it quicker.

我们一般的工作中与汇编打交道的机会并不多，如果不刻意练习，陡然面对时必然掌握不了，所以会觉得它很难。不过归根到底还是投入的时间和精力是否足够的问题——好了，iOS逆向工程给你学习ARM汇编提供了一个绝佳的条件——当我们在逆向一个功能时，往往需要分析大量ARM汇编代码，并把它们翻译成高级语言，试图重新实现这个功能；虽然暂时还不需要写汇编代码，但大量的阅读必然能加深我们对这门语言的理解。如果想在iOS逆向工程这条路上走下去，ARM汇编是必须掌握的语言，也是一定能够掌握的语言；跟英语类似，ARM汇编的基本概念相当于26个字母和音标；指令相当于单词，它们的变种相当于单词的各种形态；调用规则相当于语法，定义句子之间的联系。接下来，让我们一步步地深入。

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

### 6.1.1 基本概念

### 6.1.1 Basic concepts

如果要完整地介绍ARM汇编，ARM公司的用户手册已经做得足够好了。笔者对ARM汇编也只是略知一二，肯定没有用户手册那么全面，但对于iOS逆向工程初学者来说，这些知识足以应对，适度就好。随着iPhone 5s的推出，苹果引入了性能强大的64位处理器，但本书前半部分介绍的大多数工具对64位处理器的支持都不太好，因此后半部分的内容仍以32位处理器为准，但思路是通用的。

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

### 寄存器、内存和栈

### Register, memory, and stack

在高级语言，如Objective-C、C和C++里，操作对象是变量；在ARM汇编里，操作对象是寄存器（register）、内存和栈（stack）。其中，寄存器可以看成是CPU自带的变量，它们的数量一般是很有限的；当我们需要更多变量时，就可以把它们存放在内存中；不过，数量上去了，质量也下来了，对内存的操作比对寄存器的操作要慢得多。

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

栈其实也是一片内存区域，但它具有栈的特点：先进后出。ARM的栈是满递减（Full Descending）的，向下增长，也就是开口朝下，新的变量被存放到栈底的位置；越靠近栈底，内存地址越小，如图6-1所示。

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

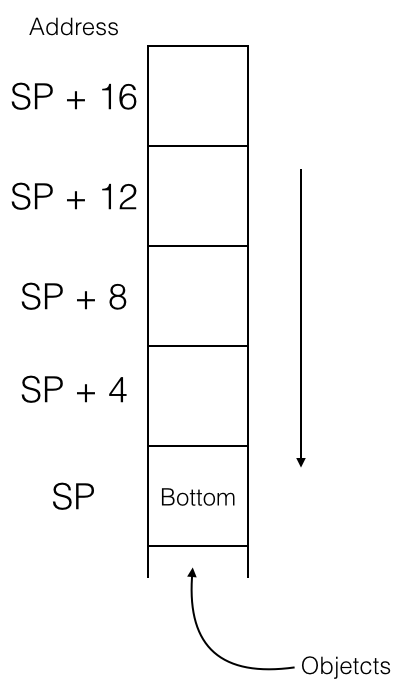


图6- 1 ARM的栈

Figure 6-1 The stack of ARM

一个名为“stack pointer”（简称SP）的寄存器保存栈的栈底地址，称为栈地址；我们可以把一个变量给入（push）栈以保存它的值，也可以让它出（pop）栈，恢复变量的原始值。在实际操作中，栈地址会不断变化；但是在执行一块代码的前后，栈地址应该是不变的，不然程序就要出问题了。为什么？举例说明：

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

static int global\_var0;

static int global\_var1;

…

void foo(void)

{

bar();

// other operations;

}

在上面4行代码中，假设函数foo()用到了A、B、C、D四个寄存器；foo()内部调用了bar()，假设bar()用到了A、B、C三个寄存器。因为2个不同的函数用到了3个相同的寄存器，所以bar()在开始执行前需要将3个寄存器中原来的值入栈以保存其原始值，在结束执行前将它们出栈以恢复其原始值，保证foo()能够正常执行。用伪汇编代码表示如下：

In the above code snippet, suppose that foo() uses registers A, B, C, and D; foo() calls bar(), and suppose that bar() uses registers A, B, and C. Because registers A, B and C are overlapped in foo() and bar(), bar() needs to save values of A, B, and C into stack before it starts execution. Also, it needs to restore these 3 registers from stack before it ends execution, to make sure foo() can work correctly. Let’s look at some pseudo code:

// foo() function函数

foo:

// 将A、B、C、D入栈，保存它们的原始值

//push A、B、C、D, store their values

入栈 {A, B, C, D}

// 使用A ~ D

移动 A, #1 // A = 1

移动 B, #2 // B = 2

移动 C, #3 // 你猜猜这行是什么意思？

调用 bar

移动 D, global\_var0

// global\_var1 = A + B + C + D

相加 A, B // A = A + B，注意此处A的值

相加 A, C // A = A + C，还要注意此处A的值

相加 A, D // 你再猜猜这行是什么意思？

移动 global\_var1, A

// 将A、B、C、D出栈，恢复它们的原始值

出栈 {A-D}

返回

// bar()函数

bar:

// 将A、B、C入栈，保存它们的原始值A == 1，B == 2，C == 3

入栈 {A-C}

// 使用A ~ C

移动 A, #2 // 还需要注释吗？

移动 B, #5

移动 C, A

相加 C, B // C = 7

// global\_var0 = A + B + C (== 2 \* C)

相加 C, C

移动 global\_var0, C // A = 2，B = 5，C = 14

// 现在你知道入栈和出栈的重要意义了吗？

出栈 {A-C}

返回

// foo()

foo:

// Push A, B, C, D into stack, save their values

push {A, B, C, D}

// Use A ~ D

move A, #1 // A = 1

move B, #2 // B = 2

move C, #3 // C = 3

call bar

move D, global\_var0

// global\_var1 = A + B + C + D

add A, B // A = A + B，notice A’s value

add A, C // A = A + C，notice A’s value

add A, D // A = A + D，notice A’s value

move global\_var1, A

// Pop A, B, C, D out of stack, restore their values

pop {A-D}

return

// bar()

bar:

// Push A、B、C into the stack, store their values

push {A-C}

// Use A ~ C

move A, #2 // Do you know what this instruction do?

move B, #5

move C, A

add C, B // C = 7

// global\_var0 = A + B + C (== 2 \* C)

add C, C

move global\_var0, C // A = 2，B = 5，C = 14

// Do you get the meaning of push and pop now?

pop {A-C}

return

简单解释一下这段伪代码：foo()先将A、B、C分别设置为1、2、3，然后调用bar()，bar()改变了A、B、C的值，并将全局变量global\_var0的值设置为ABC三者之和。如果把此时的A、B、C直接用于foo()，计算出的另一个全局变量global\_var1的值就是错的，因此在bar()执行前先要让A、B、C入栈，保存它们的值，执行完成后再出栈，使得foo()能够得到正确的global\_var1。注意一点，出于同样的目的，foo()在执行前后也对A、B、C、D执行了入栈和出栈操作，所以foo()的调用者也能够正常工作。

Let’s shortly explain this snippet of pseudo code: firstly, foo() sets registers A, B and C to 1, 2 and 3 respectively, then calls bar(), which changes values of A, B and C as well sets global\_var0, a global variable, to the sum of registers A, B and C. If we directly use the current values of A, B and C to calculate the value of global\_var1 for now, then the result would be wrong. So before executing bar(),values of A, B and C should be pushed into stack first, and pop them out after the execution of bar() for restoration, then we can get a correct global\_var1. Notice that, for the same reason, foo() has done the same operations on A, B, C and D, which saves its callers’ days.

### 特殊用途的寄存器

### Preserved registers

ARM处理器中的部分寄存器有特殊用途，如下所示：

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

R0-R3 传递参数与返回值

R7 帧指针，指向母函数与被调用子函数在栈中的交界

R9 在iOS 3.0以前被系统保留

R12 内部过程调用寄存器，dynamic linker会用到它

R13 SP寄存器

R14 LR寄存器，保存函数返回地址

R15 PC寄存器

R0-R3 Passes arguments and return values

R7 Frame pointer, which points to the previously saved stack frame and the saved link register

R9 Reserved by system before iOS 3.0

R12 IP register，used by dynamic linker

R13 Stack Pointer, i.e. SP

R14 Link Register, i.e. LR, saves function return address

R15 Program Counter, i.e. PC

因为现在还没有开始自己写汇编代码，所以对上述知识有简单了解就足够了。

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

### 分支跳转与条件判断

### Branches

处理器中名为“program counter”（简称PC）的寄存器用于存放下一条指令的地址。一般情况下，计算机一条接一条地顺序执行指令，处理器执行完一条指令后将PC加1，让它指向下一条指令，如图6-2所示。

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

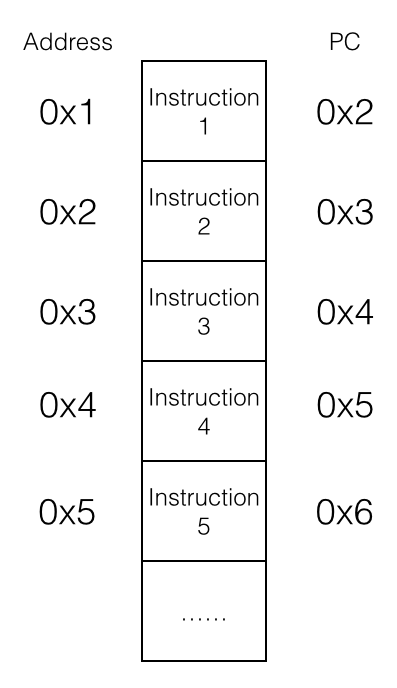


图6-2 顺序执行指令

Figure 6-2 Execute instructions in order

处理器顺序执行指令1到指令5，稀松平常、沉闷无聊。但是如果把PC的值变一变，指令的执行顺序就完全不同了，如图6-3所示。

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

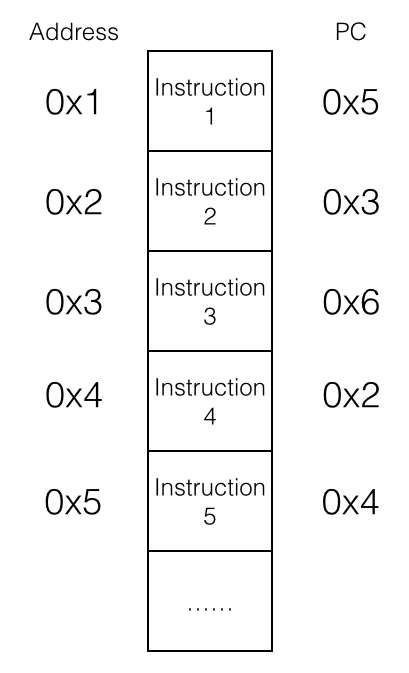


图6- 3 乱序执行指令

Figure 6-3 Execute instructions out of order

指令的执行顺序被打乱，变成指令1、指令5、指令4、指令2、指令3、指令6，光怪陆离、百花齐放。这种“乱序”的学名叫“分支”（branch），或“跳转”（jump），它使得循环和subroutine成为可能，例如：

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

// endless()函数

endless:

操作 操作数1, 操作数2

分支 endless

返回 // 死循环，执行不到这里啦！

// endless()

endless:

operate op1, op2

branch endless

return // Dead loop, we cannot reach here!

在实际情况中，满足一定条件才得以触发的分支是最实用的，这种分支称为条件分支。if else和while都是基于条件分支实现的。在ARM汇编中，分支的条件一般有4种：

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

* 操作结果为0（或不为0）；
* The result of operation is zero (or non-zero).
* 操作结果为负数；
* The result of operation is negative.
* 操作结果有进位；
* The result of operation has carry.
* 运算溢出（比如两个正数相加得到的数超过了寄存器位数）。
* The operation overflows (for example, the sum of two positive numbers exceeds 32 bits).

这些条件的判断准则（flag）存放在程序状态寄存器（Program Status Register，简称PSR）中，数据处理相关指令会改变这些flag，分支指令再根据这些flag决定是否跳转。下面的伪代码展示了一个for循环：

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

for:

相加 A, #1

比较 A, #16

不为0则跳转到for

for:

add A, #1

compare A, #16

bne for // If A - 16 != 0 then jump to for

此循环将A和#16作比较，如果两者不相等，则将A加1，继续比较。如果两者相等，则不再循环，继续往下执行。

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

### 6.1.2 ARM/THUMB指令解读

### 6.1.2 Interpretation of ARM/THUMB instructions

ARM处理器用到的指令集分为ARM和THUMB两种；ARM指令长度均为32bit，THUMB指令长度均为16bit。所有指令可大致分为3类，分别是数据操作指令、内存操作指令和分支指令。

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

### 数据操作指令

### Data processing instructions

数据操作指令有2条规则：

There’re 2 rules in data processing instructions:

* 1. 所有操作数均为32bit；
  2. All operands are 32 bits.
  3. 所有结果均为32bit，且只能存放在寄存器中。
  4. All results are 32 bits, and can only be stored in registers.

总的来说，数据操作指令的基本格式是：

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

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

其中，“cond”和“s”是两个可选后缀；“cond”的作用是指定指令“op”在什么条件下执行，共有17种条件：

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

EQ 结果为0（EQual to 0）

NE 结果不为0（Not Equal to 0）

CS 有进位或借位（Carry Set）

HS 同CS（unsigned Higher or Same）

CC 没有进位或借位（Carry clear）

LO 同CC（unsigned LOwer）

MI 结果小于0（MInus）

PL 结果大于等于0（PLus）

VS 溢出（oVerflow Set）

VC 无溢出（oVerflow Clear）

HI 无符号比较大于（unsigned HIgher）

LS 无符号比较小于等于（unsigned Lower or Same）

GE 有符号比较大于等于（signed Greater than or Equal）

LT 有符号比较小于（signed Less Than）

GT 有符号比较大于（signed Greater Than）

LE 无符号比较小于等于（signed Less than or Equal）

AL 无条件（ALways，默认）

EQ The result equals to 0 (EQual to 0)

NE The result doesn’t equal to 0 (Not Equal)

CS The operation has carry or borrow (Carry Set)

HS Same to CS (unsigned Higher or Same)

CC The operation has no carry or borrow (Carry Clear)

LO Same to CC (unsigned LOwer)

MI The result is negative (MInus)

PL The result is greater than or equal to 0 (PLus)

VS The operation overflows (oVerflow Set)

VC The operation doesn’t overflow (oVerflow Clear)

HI If operand1 is unsigned HIgher than operand2

LS If operand1 is unsigned Lower or Same than operand2

GE If operand1 is signed Greater than or Equal to operand2

LT If operand1 is signed Less Than operand2

GT If operand1 is signed Greater Than operand2

LE If operand1 is signed Less than or Equal operand2

AL ALways，this is the default

“cond”的用法很简单，例如：

“cond” is easy to use, for example:

比较 R0, R1

移动 GE R2, R0

移动 LT R2, R1

compare R0, R1

moveGE R2, R0

moveLT R2, R1

比较R0和R1的值，如果R0大于等于R1，则R2 = R0；否则R2 = R1。

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

“s”的作用是指定指令“op”是否设置flag，共有4种flag：

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

N（Negative）

如果结果小于0则置1，否则置0；

Z（Zero）

如果结果是0则置1，否则置0；

C（Carry）

对于加操作（包括CMN）来说，如果产生进位则置1，否则置0；对于减操作（包括CMP）来说，Carry相当于Not-Borrow，如果产生借位则置0，否则置1；对于有移位操作的非加/减操作来说，C置移出值的最后一位；对于其它的非加/减操作来说，C的值一般不变；

V（oVerflow）

如果操作导致溢出，则置1，否则置0。

N (Negative)

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

Z (Zero)

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

C (Carry)

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

V (oVerflow)

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

需要注意一点，C flag表示无符号数运算结果是否溢出；V flag表示有符号数运算结果是否溢出。

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

数据操作指令可以大致分为以下4类：

Data processing instructions can be divided into 4 kinds:

* 算术操作
* Arithmetic instructions

ADD R0, R1, R2 ; R0 = R1 + R2

ADC R0, R1, R2 ; R0 = R1 + R2 + C(arry)

SUB R0, R1, R2 ; R0 = R1 - R2

SBC R0, R1, R2 ; R0 = R1 - R2 - !C

RSB R0, R1, R2 ; R0 = R2 - R1

RSC R0, R1, R2 ; R0 = R2 - R1 - !C

算术操作中，ADD和SUB为基础操作，其他均为两者的变种。RSB是“Reverse SuB”的缩写，仅仅是把SUB的两个操作数调换了位置而已；以“C”（即Carry）结尾的变种代表有进位和借位的加减法，当产生进位或没有借位时，将Carry flag置1。

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

* 逻辑操作
* Logical operation instructions

AND R0, R1, R2 ; R0 = R1 & R2

ORR R0, R1, R2 ; R0 = R1 | R2

EOR R0, R1, R2 ; R0 = R1 ^ R2

BIC R0, R1, R2 ; R0 = R1 &~ R2

MOV R0, R2 ; R0 = R2

MVN R0, R2 ; R0 = ~R2

逻辑操作指令没什么多说的，它们的作用都已经用C操作符表示出来了，大家应该很熟悉；但是C操作符里的移位操作并没有对应的逻辑操作指令，因为ARM采用了桶式移位，共有4种指令：

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

LSL 逻辑左移，见图6-4

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

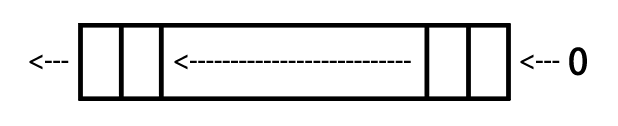


图6- 4 逻辑左移

Figure 6-4 LSL

LSR 逻辑右移，见图6-5

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

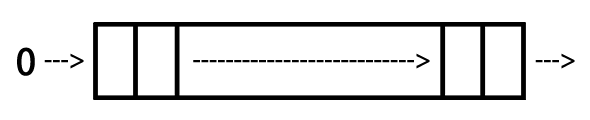


图6- 5 逻辑右移

Figure 6-5 LSR

ASR 算术右移，见图6-6

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

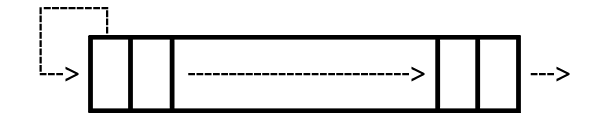


图6- 6 算术右移

Figure 6-6 ASR

ROR 循环右移，见图6-7

ROR ROtate Right, as shown in figure 6-7

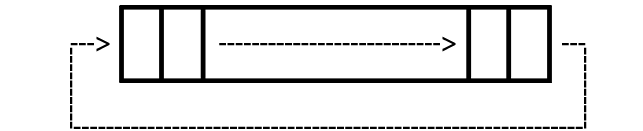


图6- 7 循环右移

Figure 6-7 ROR

* 比较操作
* Compare instructions

CMP R1, R2 ; 执行R1 - R2并依结果设置flag

CMN R1, R2 ; 执行R1 + R2并依结果设置flag

TST R1, R2 ; 执行R1 & R2并依结果设置flag

TEQ R1, R2 ; 执行R1 ^ R2并依结果设置flag

CMP R1, R2 ; Set flag according to the result of R1 - R2

CMN R1, R2 ; Set flag according to the result of R1 + R2

TST R1, R2 ; Set flag according to the result of R1 & R2

TEQ R1, R2 ; Set flag according to the result of R1 ^ R2

比较操作其实就是改变flag的算术操作或逻辑操作，只是操作结果不保留在寄存器里而已。

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

* 乘法操作
* Multiply instructions

MUL R4, R3, R2 ; R4 = R3 \* R2

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

乘法操作的操作数必须来自寄存器。

The operands of multiply instructions must come from registers.

### 内存操作指令

### Register processing instructions

内存操作指令的基本格式是：

The basic syntax of register processing instructions is:

op{cond}{type} Rd, [Rn, Op2]

其中Rn是基址寄存器，用于存放基地址；“cond”的作用与数据操作指令相同；“type”指定指令“op”操作的数据类型，共有4种：

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

B（unsigned Byte）

无符号byte（执行时扩展到32bit，以0填充）；

SB（Signed Byte）

有符号byte（仅用于LDR指令；执行时扩展到32bit，以符号位填充）；

H（unsigned Halfword）

无符号halfword（执行时扩展到32bit，以0填充）；

SH（Signed Halfword）

有符号halfword（仅用于LDR指令；执行时扩展到32bit，以符号位填充）。

B (unsigned Byte)

Extends to 32 bits when executing，filled with 0.

SB (Signed Byte)

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

H (unsigned Halfword)

Extends to 32 bits when executing，filled with 0.

SH (Signed Halfword)

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

如果不指定“type”，则默认数据类型是word。

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

ARM内存操作基础指令只有两个：LDR（LoaD Register）将数据从内存中读出来，存到寄存器中；STR（STore Register）将数据从寄存器中读出来，存到内存中。两个指令的使用情况如下：

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

* LDR

LDR Rt, [Rn {, #offset}] ; Rt = \*(Rn {+ offset})，{}代表可选

LDR Rt, [Rn, #offset]! ; Rt = \*(Rn + offset); Rn = Rn + offset

LDR Rt, [Rn], #offset ; Rt = \*Rn; Rn = Rn + offset

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

LDR Rt, [Rn, #offset]! ; Rt = \*(Rn + offset); Rn = Rn + offset

LDR Rt, [Rn], #offset ; Rt = \*Rn; Rn = Rn + offset

* STR

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

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

STR Rt, [Rn], #offset ; \*Rn = Rt; Rn = Rn + offset

此外，LDR和STR的变种LDRD和STRD还可以操作双字（Doubleword），即一次性操作2个寄存器，其基本格式是：

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

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

其用法与原型类似，如下：

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

* STRD

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

* LDRD

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

除了LDR和STR外，还可以通过LDM（LoaD Multiple）和STM（STore Multiple）进行块传输，一次性操作多个寄存器。块传输指令的基本格式是：

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

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

其中Rd是基址寄存器，可选的“!”指定Rd变化后的值是否写回Rd；reglist是一系列寄存器，用大括号括起来，它们之间可以用“,”分隔，也可以用“-”表示一个范围，比如，{R4 – R6, R8}表示寄存器R4、R5、R6、R8；这些寄存器的顺序是按照自身的编号由小到大排列的，与大括号内的排列顺序无关。

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

需要特别注意的是，LDM和STM的操作方向与LDR和STR完全相反：LDM是把从Rd开始，地址连续的内存数据存入reglist中，STM是把reglist中的值存入从Rd开始，地址连续的内存中。此处特别容易混淆，大家一定要注意！

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

“cond”的作用与数据操作指令相同。“mode”指定Rd值的4种变化规律，如下：

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

IA（Increment After）

每次传输后增加Rd的值；

IB（Increment Before）

每次传输前增加Rd的值；

DA（Decrement After）

每次传输后减少Rd的值；

DB（Decrement Before）

每次传输前减少Rd的值。

IA (Increment After)

Increment Rd after “op”.

IB (Increment Before)

Increment Rd before “op”.

DA (Decrement After)

Decrement Rd after “op”.

DB (Decrement Before)

Decrement Rd before “op”.

这是什么意思呢？下面以LDM为代表，举一个简单的例子，相信大家一看就明白了。在图6-8中，R0指向的值是5。

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

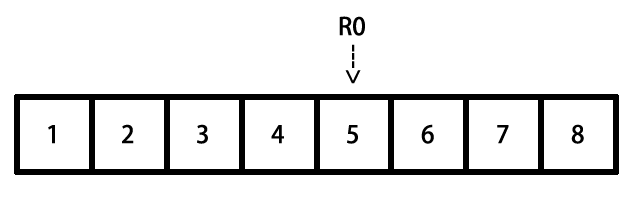


图6- 8 块传输指令模拟环境

Figure 6-8 Simulation of LDM

执行以下命令后，R4、R5、R6的值分别变成：

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

LDMIA R0, {R4 – R6} ; R4 = 5, R5 = 6, R6 = 7

LDMIB R0, {R4 – R6} ; R4 = 6, R5 = 7, R6 = 8

LDMDA R0, {R4 – R6} ; R4 = 5, R5 = 4, R6 = 3

LDMDB R0, {R4 – R6} ; R4 = 4, R5 = 3, R6 = 2

STM指令的作用方式与此类似，不再赘述。再次提醒，LDM和STM的操作方向与LDR和STR完全相反，切记切记！

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

### 分支指令

### Branch instructions

分支指令可以分为无条件分支和条件分支两种。

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

* 无条件分支
* Unconditional branches

B Label ; PC = Label

BL Label ; LR = PC – 4; PC = Label

BX Rd ; PC = Rd ,and switch instruction set

无条件分支很简单，举一个小例子就会了解：

Unconditional branches are easy to understand, for example:

foo():

B Label ; 跳转到Label处往下执行

....... ; 得不到执行

Label:

.......

foo():

B Label ; Jump to Label to keep executing

....... ; Can’t reach here

Label:

.......

* 条件分支
* Conditional branches

条件分支的cond是依照6.2.1节提到的4种flag来判断的，它们的对应关系如下：

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

cond flag

EQ Z = 1

NE Z = 0

CS C = 1

HS C = 1

CC C = 0

LO C = 0

MI N = 1

PL N = 0

VS V = 1

VC V = 0

HI C = 1 & Z = 0

LS C = 0 | Z = 1

GE N = V

LT N != V

GT Z = 0 & N = V

LE Z = 1 | N != V

在条件分支指令前会有一条数据操作指令来设置flag，分支指令根据flag的值来决定代码走向，例如：

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

Label:

LDR R0, [R1], #4

CMP R0, 0 ; 如果R0 == 0，Z = 1；否则Z = 0

BNE Label ; Z == 0则跳转

Label:

LDR R0, [R1], #4

CMP R0, 0 ; If R0 == 0 then Z = 1; else Z = 0

BNE Label ; If Z == 0 then jump

### THUMB指令

### 4. THUMB instructions

THUMB指令集是ARM指令集的一个子集，每条THUMB指令均为16bit；因此THUMB指令比ARM指令更节省空间，且在16位数据总线上的传输效率更高。有得必有失，除了“b”之外，所有THUMB指令均无法条件执行；桶式移位无法结合其他指令执行；大多数THUMB指令只能使用R0~R7这8个寄存器等。相对于ARM指令，THUMB指令的特点是：

THUMB instruction set is a subset of ARM instruction set. Every THUMB instruction is 16 bits long, so THUMB instructions are more space saving than ARM instructions, and can be faster transferred on 16-bit data bus. However, you can’t make an omelet without breaking eggs. All THUMB instructions except “b” can’t be executed conditionally; barrel shift can’t cooperate with other instructions; most THUMB instructions can only make use of registers R0 to R7, etc. Compared with ARM instructions, the features of THUMB instructions are:

* 指令数量减少
* There’re less THUMB instructions than ARM instructions

既然THUMB只是一个子集，指令数量必然会减少。例如，乘法指令中只有MUL保留了下来，其他的都被精简了。

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

* 没有条件执行
* No conditional execution

除分支指令外，其他指令无法条件执行。

Except branch instructions, other instructions cannot be executed conditionally.

* 所有指令默认附带“s”
* All THUMB instructions set flags by default

即所有THUMB指令都会设置flag。

* 桶式移位无法结合其它指令执行
* Barrel shift cannot cooperate with other instructions

移位指令只能单独执行，无法与其他指令结合执行。即，可以：

Shift instructions can only be executed alone, say:

LSL R0 #2

而不可：

But cannot:

ADD R0, R1, LSL #2

* 寄存器使用受限
* Limitation of registers

除非显式声明，否则THUMB指令只能使用R0~R7寄存器；但也有例外：ADD、MOV和CMP指令可以将R8~R15作为操作数使用；LDR和STR可以使用PC或SP寄存器；PUSH可以使用LR，POP可以使用PC；BX可以使用所有寄存器。

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

* 立即数和第二操作数使用受限
* Limitation of immediate values and the second operand

大多数THUMB数据操作指令的形式是“op Rd, Rm”，只有移位指令、ADD、SUB、MOV和CMP是例外。

Most of THUMB instructions’ format is “op Rd, Rm”, excluding shift instructions, ADD, SUB, MOV and CMP.

* 不支持数据写回
* Doesn't support data write back

除了LDMIA和STMIA外，其他THUMB指令均不支持数据写回，即“!”不可用。

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

我们在iOS逆向工程初级阶段经常会碰到以上指令，如果你对前两节的内容还是感到一知半解，没关系，自己动手分析两个程序就熟悉了。这一节的内容只是一个引子，在实际操作中如果对指令作用不清楚，ARM的官方文档http://infocenter.arm.com永远是最好的教科书，<http://bbs.iosre.com>上的讨论也很有参考价值。

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

### 6.1.3 ARM调用规则

### 6.1.3 ARM calling conventions

了解了常用的ARM指令后，相信大家已经能够基本读懂一个函数的汇编代码了。当一个函数调用另一个函数时，常常需要传递参数和返回值；如何传递这些数据，称为ARM汇编的调用规则。

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

### 前言与后记

### 1. Prologs and epilogs

在6.1.1节提到，“在执行一块代码时，其前后栈地址应该是不变的”，这个操作是通过被执行代码块的前言（prologs）和后记（epilogs）完成的。前言所做的工作主要有：

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

* 将LR入栈；
* 将R7入栈；
* R7 = SP；
* 将需要保留的寄存器原始值入栈；
* 为本地变量开辟空间。
* PUSH LR;
* PUSH R7;
* R7 = SP;
* PUSH registers that must be preserved;
* Allocates space in the stack frame for local storage.

后记所做的主要工作跟前言正好相反：

And epilog does an opposite job to prolog:

* 释放本地变量占用的空间；
* 将需要保留的寄存器原始值出栈；
* 将R7出栈；
* 将LR出栈，PC = LR。
* Deallocates space that the prolog allocates;
* POP preserved registers;
* POP R7;
* POP LR, and PC = LR.

前言和后记中的这些工作并不是必须的，如果这块代码压根儿就没有用到栈，就不需要“保留寄存器原始值”这一步了。在逆向工程中，前言与后记的影响主要体现在SP的变化上，此处稍作了解即可，第10章的例子中会有详细的解答。

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

### 传递参数与返回值

### 2. Pass arguments and return values

如果想详细了解参数传递规则，可以通读http://infocenter.arm.com/help/topic/com.arm.doc.  
ihi0042e/IHI0042E\_aapcs.pdf。一般情况下，记住最重要的一个金句就好：

If you want to delve deeper into how arguments and return values are passed, you can read [http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E\_aapcs.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/ihi0042e_aapcs.pdf). However, in the majorty of cases, you just need to 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.”

这句话的意思很好理解，为了加深印象，我们看一个例子：

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

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

#include <stdio.h>

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

{

printf("%d, %d, %d, %d, %d", 1, 2, 3, 4, 5);

return 6;

}

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，生成的main汇编代码如图6-9所示。

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

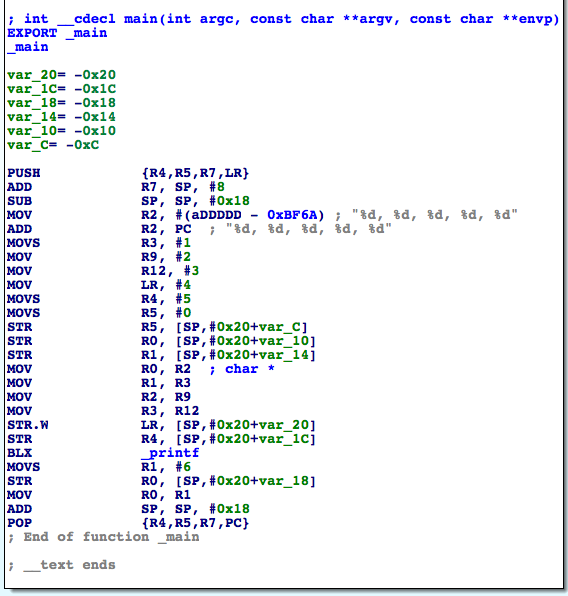


图6- 9 main的汇编代码

Figure 6-9 main in assembly

“BLX \_printf”执行printf函数，它的6个参数分别存放在R0、R1、R2、R3、[SP, #0x20 + var\_20]和[SP, #0x20 + var\_1C]中，返回值存放在R0里，其中var\_20 = -0x20，var\_1C = -0x1C，因此栈上的2个参数分别位于[SP]和[SP, #0x4]。

“BLX \_printf” calls printf, and its 6 arguments are stored in R0, R1, R2, R3, [SP, #0x20 + var\_20], and [SP, #0x20 + var\_1C] respectively; the return value is stored in R0. Because var\_20 = -0x20，var\_1C = -0x1C, 2 arguments in the stack are at [SP] and [SP, #0x4].

还需要更多解释吗？

I don’t think we need further explanation.

“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.”

一定要牢记上面这句话！

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

本节只是把iOS逆向工程用到的最基本的ARM汇编知识过了一遍，难免有遗漏，但说白了，只要记住刚才的“金句”，配合ARM官方网站，就已经可以开始分析程序了。接下来，就来实际动手，看看如何把刚刚学到的知识运用到iOS逆向工程中。

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

## 6.2 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 “look for interesting keywords in class-dump headers” as “locate 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 discover an interesting function, but fail to find the related keywords in class-dump headers? And how can we track an interesting data till we know how it’s generated? In these cases, class-dump is all thumbs. Thus, “look for interesting keywords in class-dump headers” is just one scenario in “locate target functions”, we’ve overgeneralized. Therefore, in more general cases, how should we locate target functions?

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 programmatic functions. So, “locate target functions” is actually the process of how we locate the source functions of our interested Apps’ visual expressions.

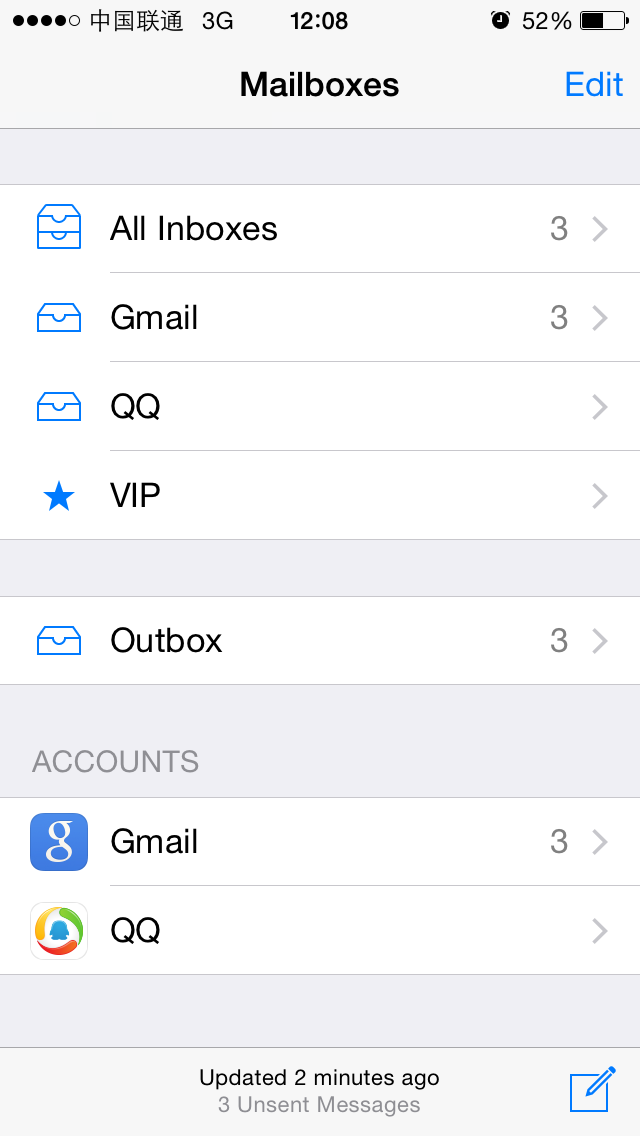


Figure 6- 10 Mail



Figure 6- 11 MobilePhoneSettings

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 “locate target functions”. For most of us, among all iOS software, we know Apps the best, so if we’re to choose something as our junior reverse targets, nothing is more appropriate than Apps. As a result, in the following sections, we will take Apps as examples, and try to refine “locate target functions” with ARM level reverse engineering, as well enhance the methodology of writing a tweak.

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

### 用Cycript注入Mail

### 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

### 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 the superview, the rest of work is only examining subviews of this superview, hence reduce our work burden. Commonly, texts will be printed in description, so we can directly search “3 Unsent Messages” in recursiveDescription:

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

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

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

cy# [#0x146e6060 setHidden:YES]

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:

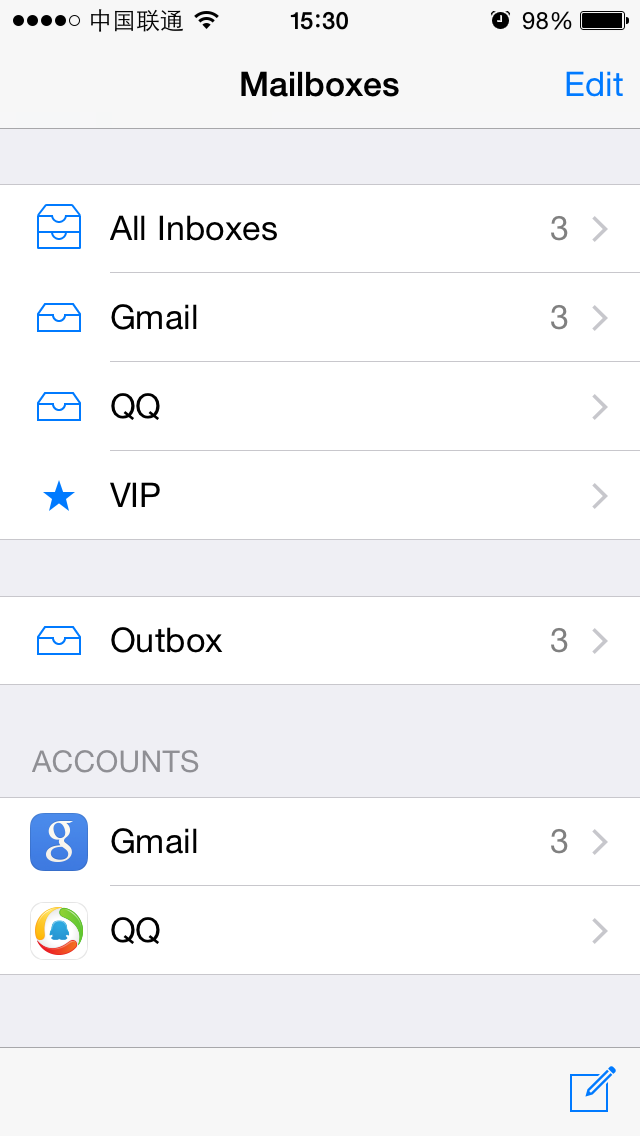


图6- 12 两排字被隐藏

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 disappears:

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:

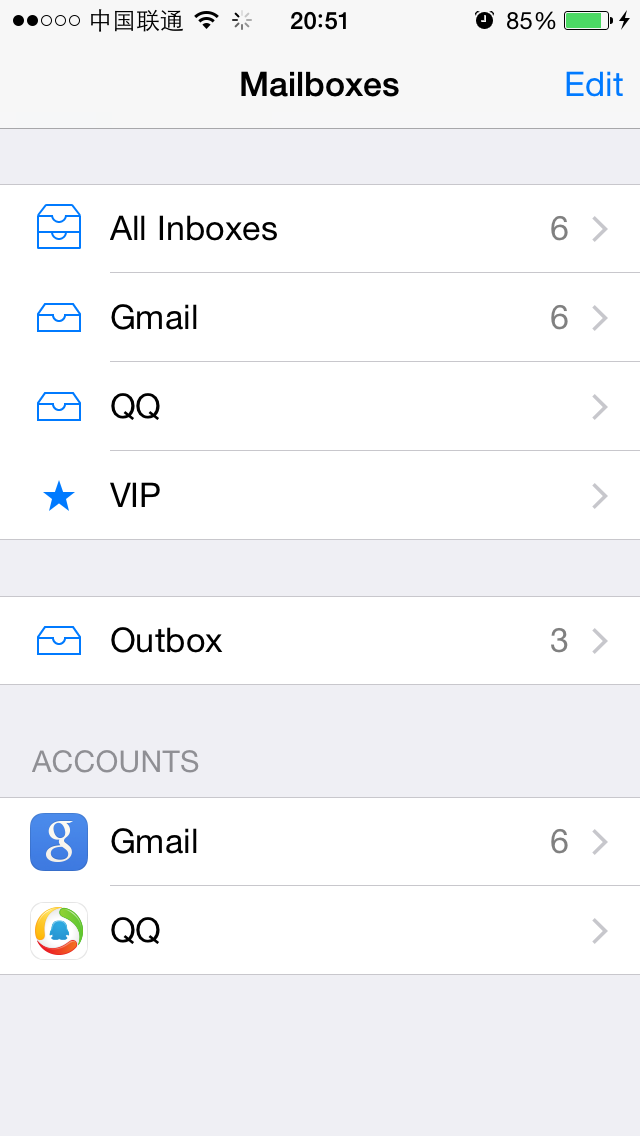


图6- 13 UIToolBar被隐藏

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 expected, as shown in figure 6-14:

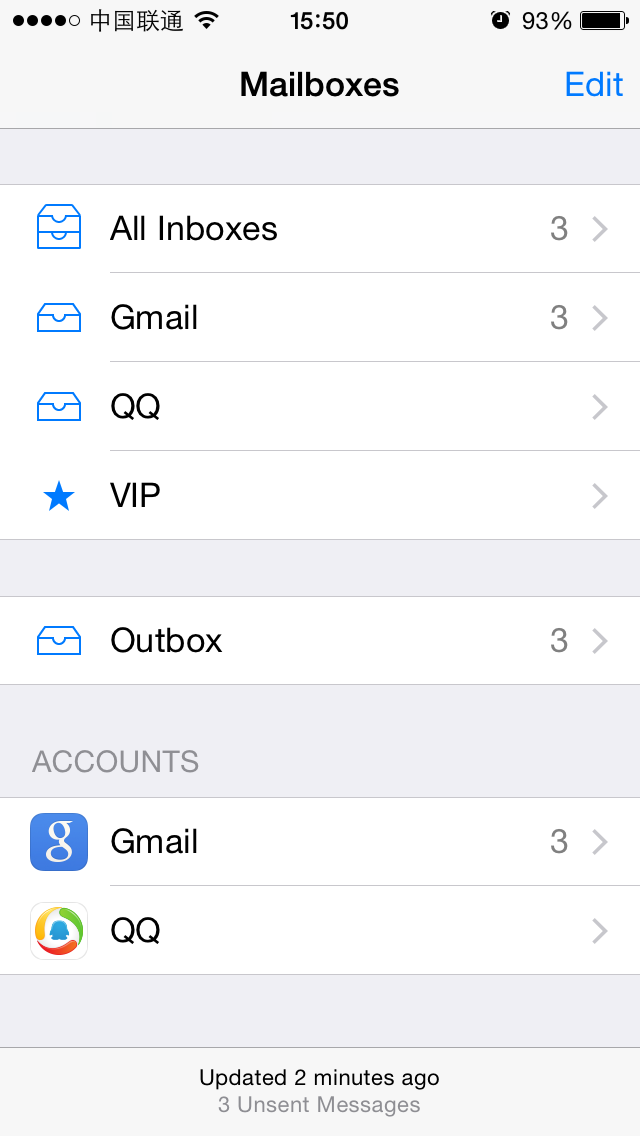


图6- 14 按钮被隐藏

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

### 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

### 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

### 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 no longer 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

### Find the UI function of the top cell

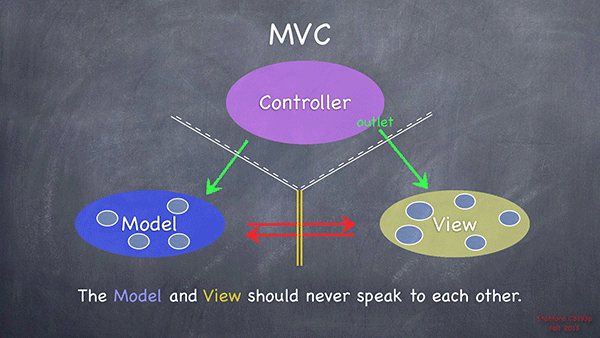


图6- 16 MVC设计标准（来自Stanford CS 193P）

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:

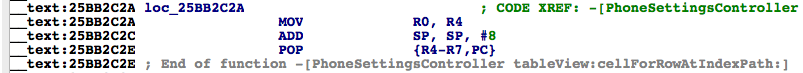


图6- 17 [PhoneSettingsController tableView:cellForRowAtIndexPath:]

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 to trigger the breakpoint. Print R0 out when the process stops, an initialized cell should be ready by then:

(lldb) br s -a 0x2c965c2c

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

Process 115525 stopped

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

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

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

-> 0x2c965c2c: add sp, #8

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

MobilePhoneSettings`-[PhoneSettingsController applicationWillSuspend]:

0x2c965c30: push {r7, lr}

0x2c965c32: mov r7, sp

(lldb) po $r0

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

(lldb) po [$r0 subviews]

<\_\_NSArrayM 0x17060e50>(

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

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

)

(lldb) po [$r0 detailTextLabel]

<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 games, 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 functions). If we only want to drink 7Up (data), we need to find Get7Up (target function which generates the data); if we want to know how Zero is made (function), we need to find MakeZero (target function which provides function). Actually, the “nest” of nested called functions are also consists of chains, so if we can get to know any link of the chain, we can regenerate the whole chain by reverse engineering, and the tools we mainly use are IDA and LLDB. Let’s continue with the previous 2 examples to learn how to find target functions of “compose email” and “get my number” by referring to [ComposeButtonItem \_sendAction:withEvent:] and [PhoneSettingsController tableView:cellForRowAtIndexPath:].

### 寻找“编写邮件”的目标函数

### Look 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.

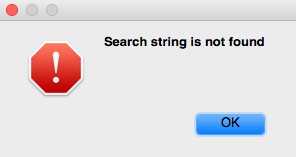


图6- 18 找不到[ComposeButtonItem \_sendAction:withEvent:]

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.

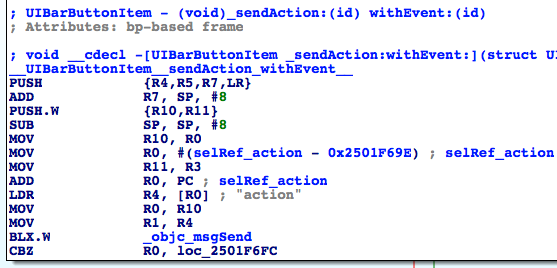


图6- 19 [UIBarButtonItem \_sendAction:withEvent:]

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. Based on this, the first objc\_msgSend actually executes an Objective-C method. So what exactly is the method? Who’s the receiver, and what are the arguments?

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 that it’s 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 confused? Refer to figure 6-20, I bet you can understand.

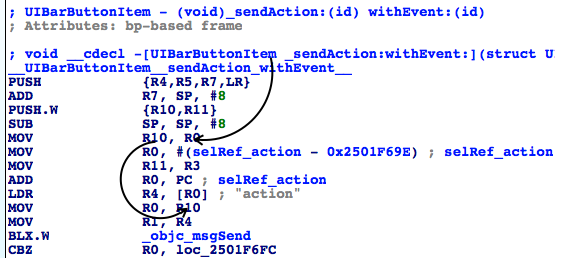


图6- 20 R0的演变过程

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:

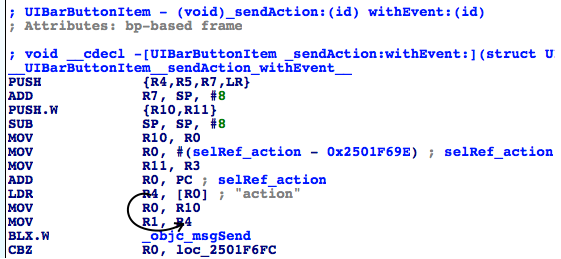


图6- 21 R1的演变过程

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:

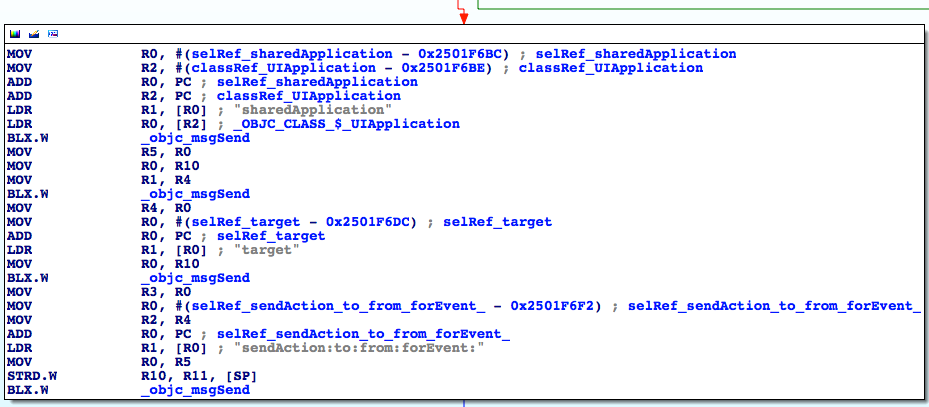


图6- 22 [UIBarButtonItem \_sendAction:withEvent:]

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

又是4个objc\_msgSend，从上到下逐个分析：

There’re 4 objc\_msgSends, 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 on the stack. R2 comes from “MOV R2, R4”, i.e. R4; R4 comes from “MOV R4, R0” under the 2nd objc\_msgSend, i.e. R0; R0 comes from the return value of the 2nd objc\_msgSend, i.e. [UIBarButtonItem action], which is the 3rd argument. R3 comes from “MOV R3, R0” under the 3rd objc\_msgSend, i.e. R0; R0 comes from the return value of the 3rd objc\_msgSend, i.e. [UIBarButtonItem target], which is the 4th argument. The rest 2 arguments come from the stack, and before the 4th objc\_msgSend, the latest change of stack comes from “STRD.W R10, R11, [SP]”, i.e. R10 and R11 are saved onto the stack; therefore, the rest 2 arguments are R10 and R11. R10 is UIBarButtonItem, which is discussed several times; whereas R11 comes from “MOV R11, R3” in figure 6-21, i.e. R3, which is another unassigned register, so it must come from the caller of [UIBarButtonItem \_sendAction:withEvent:]. Based on our previous analysis, R11 is the 2nd argument of \_sendAction:withEvent:, i.e. event. The relationship of these 4 arguments is a little complicated, hope figure 6-23 and 6-24 can give you a better illustration.

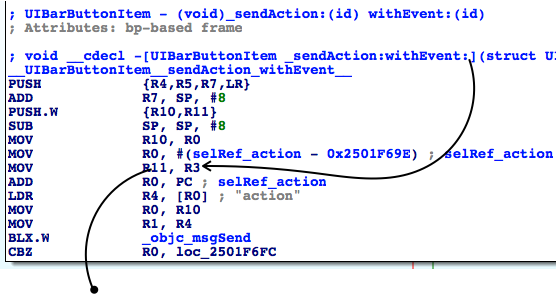


图6- 23 objc\_msgSend的参数关系

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

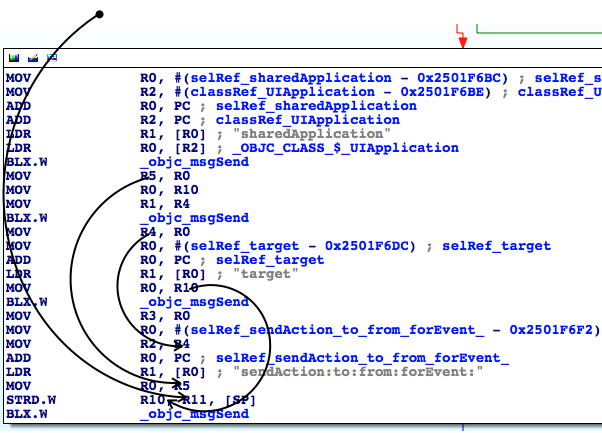


图6- 24 objc\_msgSend的参数关系

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 their values are 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.

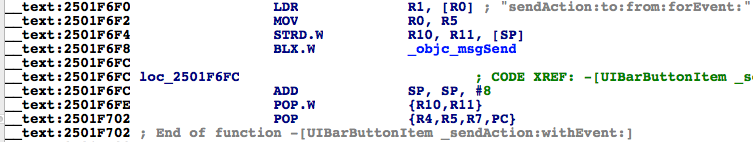


图6- 25 查看objc\_msgSend的地址

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 on the stack, and they are both one word long, I’ve printed out the continuous 10 words from the bottom of the stack using “x/10 $sp”, and the first 2 were the arguments on stack. Most Objective-C arguments are one word long pointers, which point at Objective-C objects, so I’ve “po”ed the first 2 words, they were the arguments. For ease of understanding, the relationship of SP, values on the stack and arguments are shown in figure 6-26.

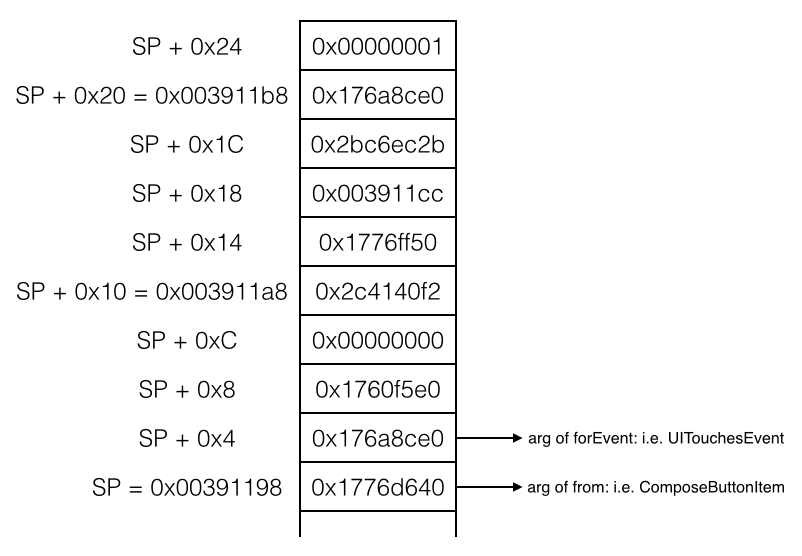


图6- 26 SP、栈值和参数的关系

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:

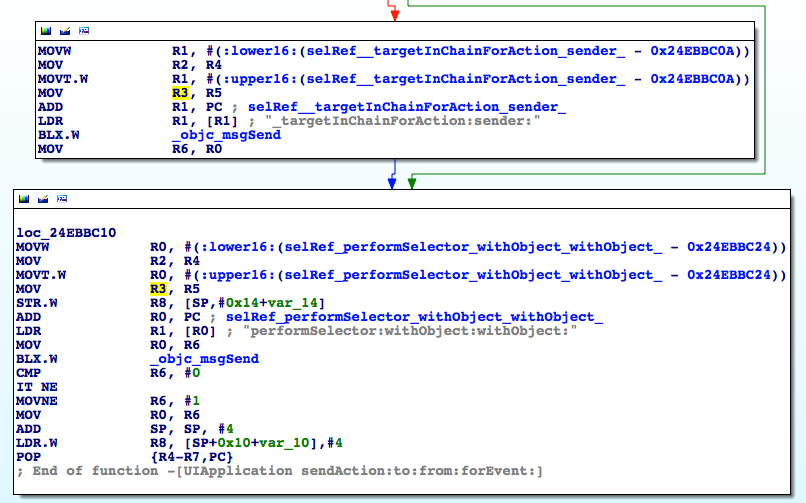


图6- 27 [UIApplication sendAction:to:from:forEvent:]

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.

### 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”.

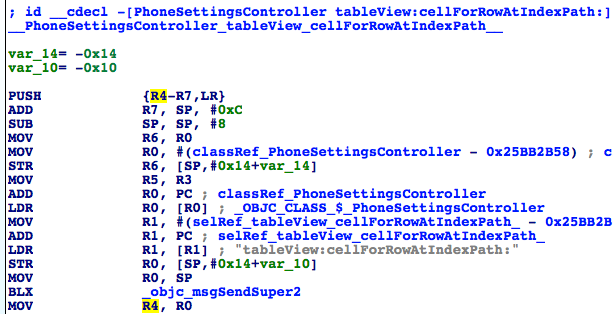


图6- 28 R4的来源

Figure 6-28 Source of R4

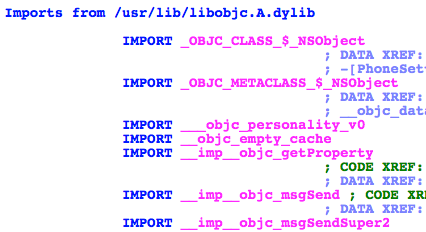


图6- 29 objc\_msgSendSuper2的来源

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.

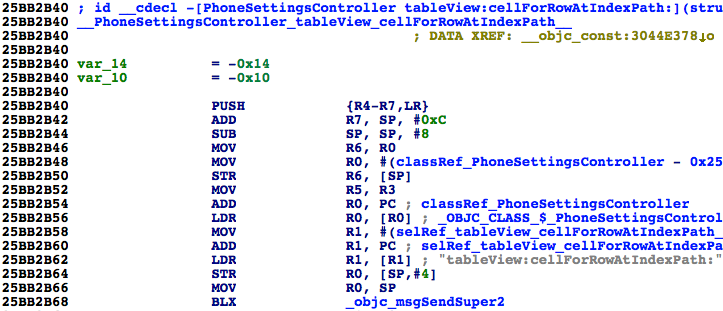


图6- 30 查看objc\_msgSendSuper2的地址

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.

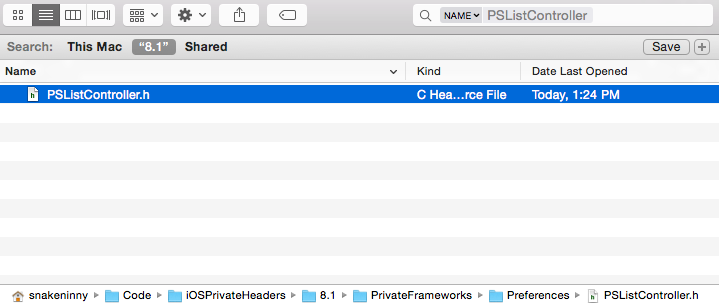


图6- 31 定位PSListController.h

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 Preferences.framework into IDA and jump to tableView:cellForRowAtIndexPath:, as shown in figure 6-32.

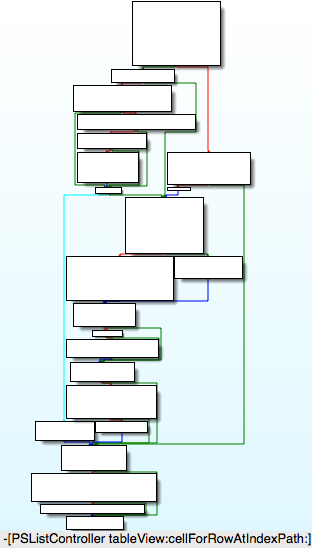


图6- 32 [PSListController tableView:cellForRowAtIndexPath:]

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.

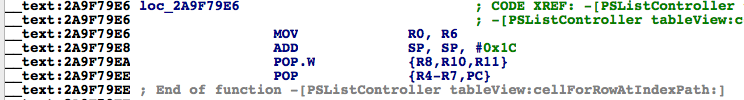


图6- 33 [PSListController tableView:cellForRowAtIndexPath:]

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.

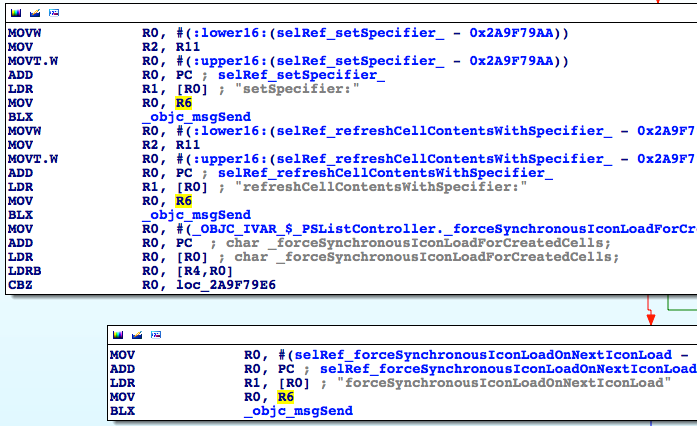


图6- 34 R6出现频率很高

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.

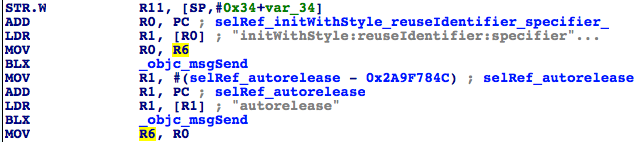


图6- 35 R6被赋值

Figure 6-35 The assignment of R6

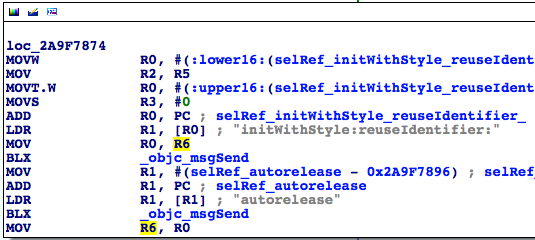


图6- 36 R6被赋值

Figure 6-36 The assignment of R6

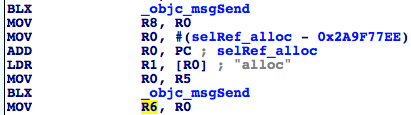


图6- 37 R6被赋值

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 differences, we can definitely locate this configuration objc\_msgSend; if you’re good at math, dichotomy can be used in this scenario, you can inspect from the middle. Anyway, it’s just a matter of personal preferences. In this example, I use a compromised dichotomy, as shown in figure 6-38.

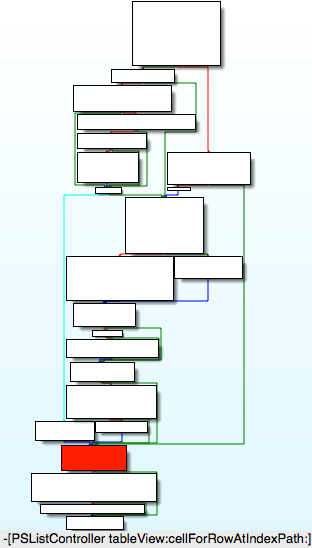


图6- 38 [PSListController tableView:cellForRowAtIndexPath:]

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.

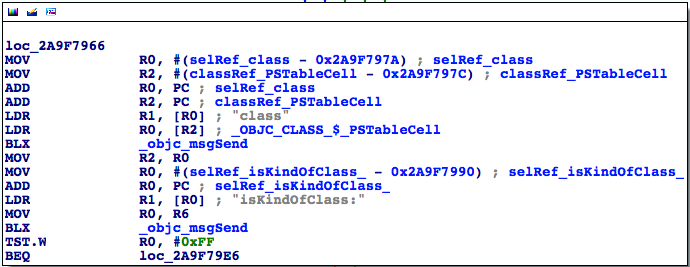


图6- 39 深色方块所在的loc\_2a9f7966

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.

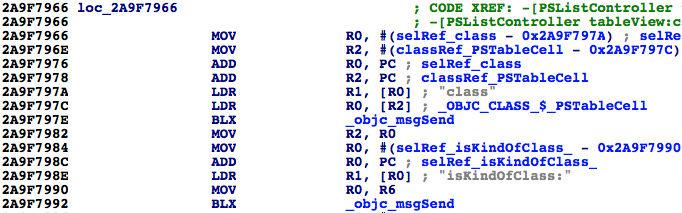


图6- 40 查看objc\_msgSend的地址

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 seen it. So the breakpoint should be set at 0x6db3000 + 0x2A9F797E = 0x317AA97E. Trigger it and see if PSTableCell contains my number already:

(lldb) br s -a 0x317AA97E

Breakpoint 10: where = Preferences`-[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 are specified by a 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 can also 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.

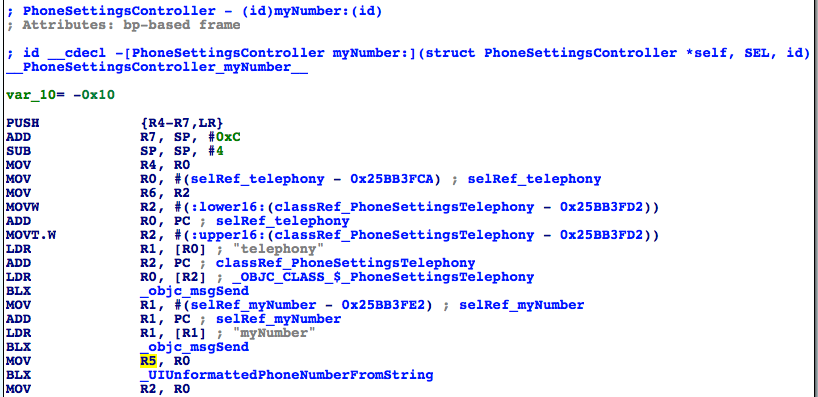


图6- 42 [PhoneSettingsController myNumber:]

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 discuss 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MobilePhoneSettings.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 MobilePhoneSettings.bundle, there must be a more common as well lower level library that can read SIM card. If we can locate this library, we can get my number without loading MobilePhoneSettings.bundle. Since it’s supposed to be a lower level library, naturally, we should dig into [PhoneSettingsTelephony myNumber] to find out how it reads my number, as shown in figure 6-43.

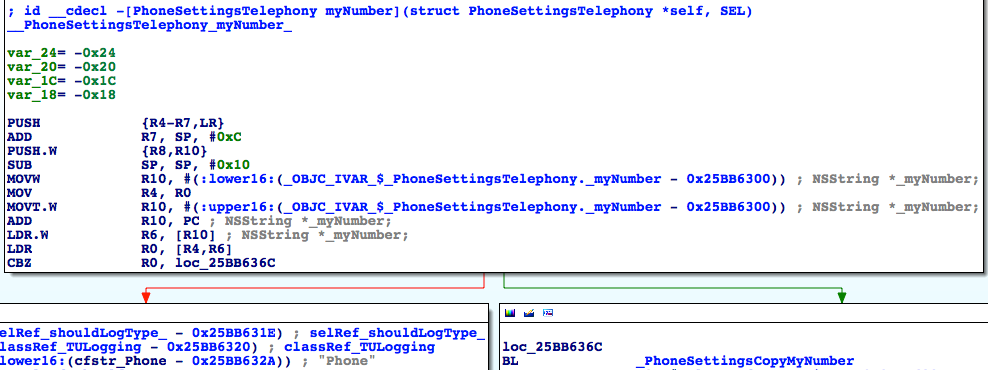


图6- 43 [PhoneSettingsTelephony myNumber]

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.

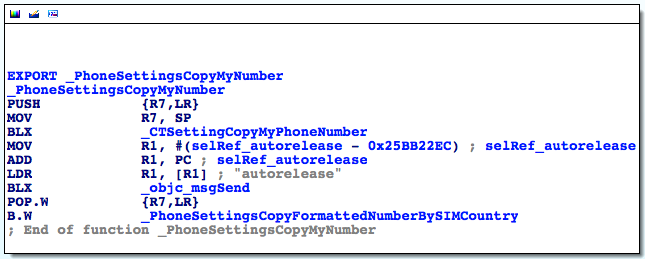


图6- 44 PhoneSettingsCopyMyNumber

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.

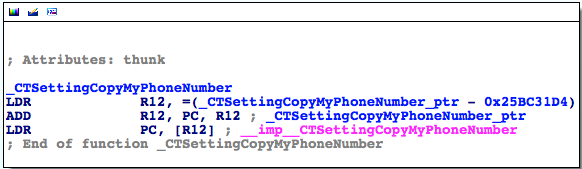


图6- 45 CTSettingCopyMyPhoneNumber

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.

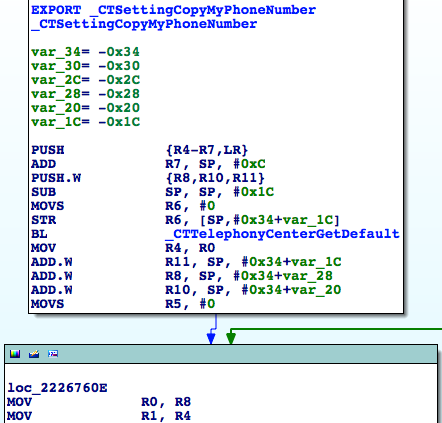


图6- 46 CTSettingCopyMyPhoneNumber

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.

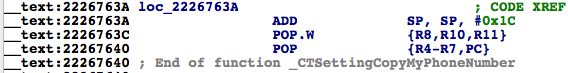


图6- 47 CTSettingCopyMyPhoneNumber

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.

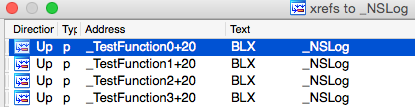


图6- 48 查看NSLog的交叉引用

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.

#### 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.

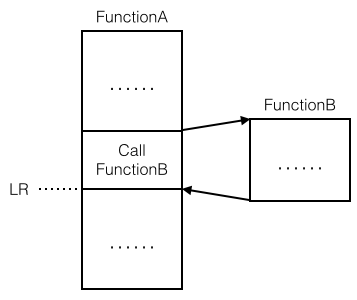


图6- 49 返回地址示意图

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.

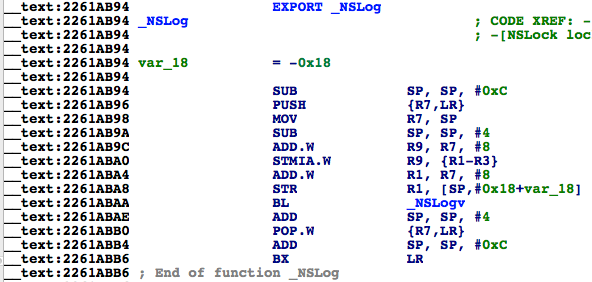


图6- 50 查看NSLog基地址

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.

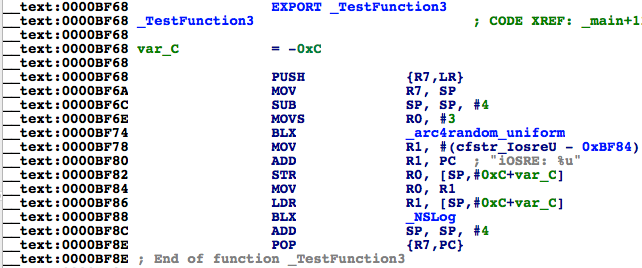


图6- 51 TestFunction3

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?

#### 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.

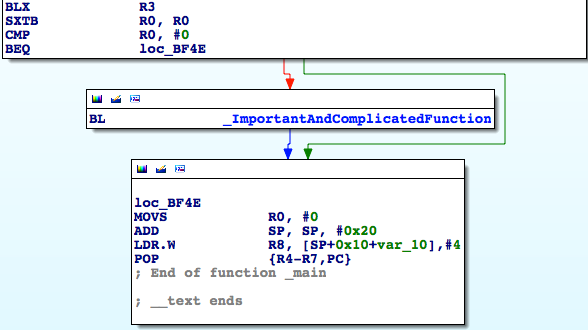


图6- 52 ImportantAndComplicatedFunction得到调用之前

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!