# Chapter 6 ARM related iOS reverse engineering

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.

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 Introduction to ARM assembly

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.

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 Basic concepts

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.

### Register, memory, and stack

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.

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.

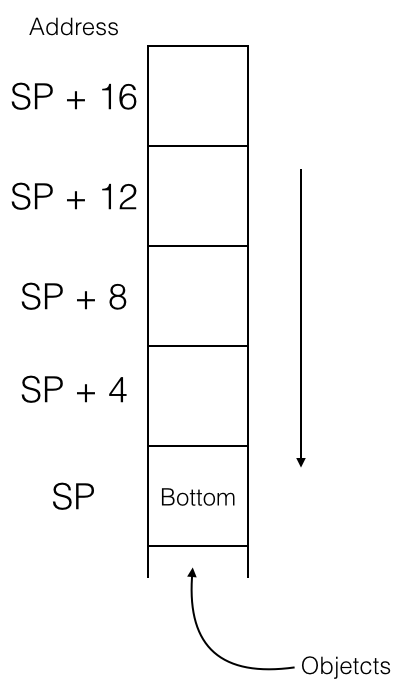


Figure 6-1 The stack of ARM

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;

}

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()

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

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.

### Preserved registers

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

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

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

### Branches

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.

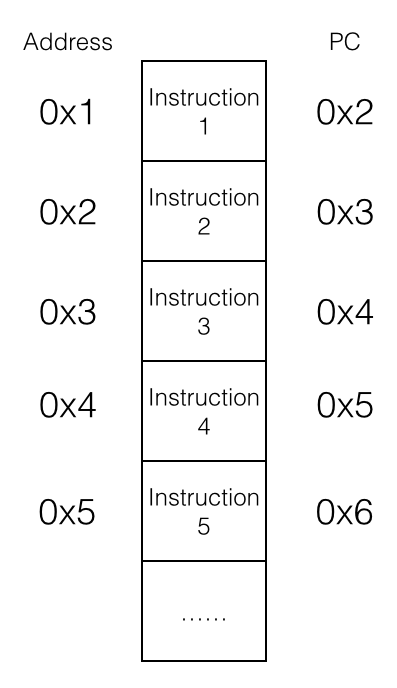


Figure 6-2 Execute instructions in order

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.

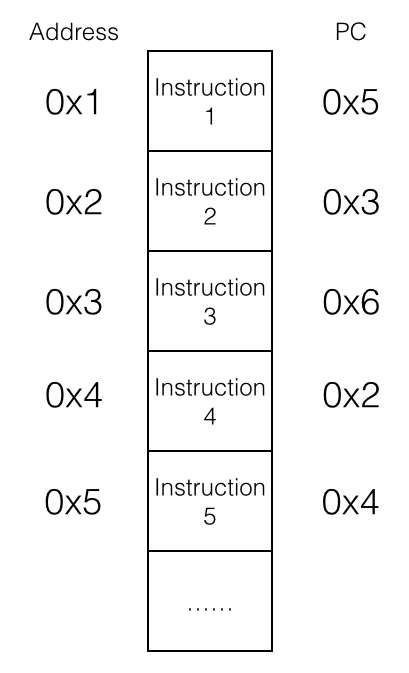


Figure 6-3 Execute instructions out of order

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()

endless:

operate op1, op2

branch endless

return // Dead loop, we cannot reach here!

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:

* The result of operation is zero (or non-zero).
* The result of operation is negative.
* The result of operation has carry.
* The operation overflows (for example, the sum of two positive numbers exceeds 32 bits).

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:

add A, #1

compare A, #16

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

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 Interpretation of ARM/THUMB instructions

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.

### Data processing instructions

There’re 2 rules in data processing instructions:

* 1. All operands are 32 bits.
  2. All results are 32 bits, and can only be stored in registers.

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

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

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

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” is easy to use, for example:

compare R0, R1

moveGE R2, R0

moveLT R2, R1

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

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

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.

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

Data processing instructions can be divided into 4 kinds:

* 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

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.

* 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

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 Logical Shift Left, as shown in figure 6-4

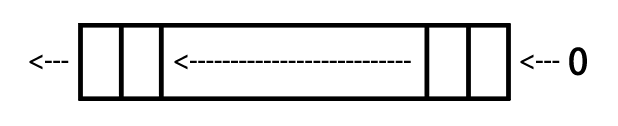


Figure 6-4 LSL

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

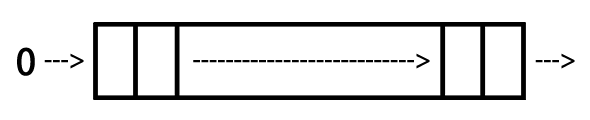


Figure 6-5 LSR

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

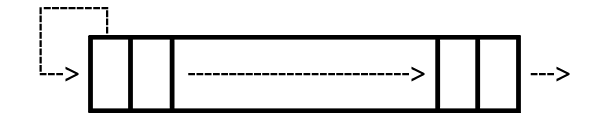


Figure 6-6 ASR

ROR ROtate Right, as shown in figure 6-7

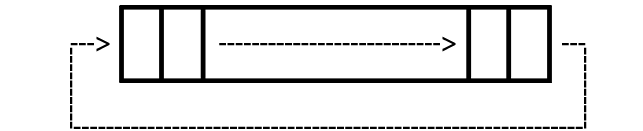


Figure 6-7 ROR

* Compare instructions

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

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

* Multiply instructions

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

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

The operands of multiply instructions must come from registers.

### Register processing instructions

The basic syntax of register processing instructions is:

op{cond}{type} Rd, [Rn, Op2]

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)

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.

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

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}), {} 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

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}]

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)

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

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.

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.

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

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”.

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

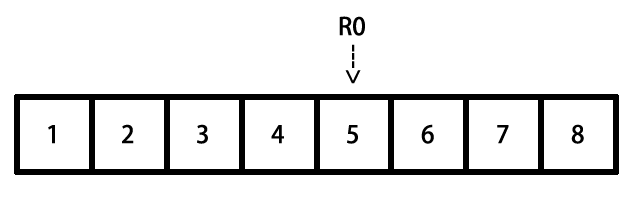


Figure 6-8 Simulation of LDM

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

### Branch instructions

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

* Unconditional branches

B Label ; PC = Label

BL Label ; LR = PC – 4; PC = Label

BX Rd ; PC = Rd ,and switch instruction set

Unconditional branches are easy to understand, for example:

foo():

B Label ; Jump to Label to keep executing

....... ; Can’t reach here

Label:

.......

* Conditional branches

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

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

BNE Label ; If Z == 0 then jump

### 4. THUMB instructions

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:

* There’re less THUMB instructions than ARM instructions

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.

* No conditional execution

Except branch instructions, other instructions cannot be executed conditionally.

* All THUMB instructions set flags by default
* Barrel shift cannot cooperate with other instructions

Shift instructions can only be executed alone, say:

LSL R0 #2

But cannot:

ADD R0, R1, LSL #2

* Limitation of registers

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.

* Limitation of immediate values and the second operand

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

* Doesn't support data write back

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

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 calling conventions

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.

### Prologs and epilogs

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:

* PUSH LR;
* PUSH R7;
* R7 = SP;
* PUSH registers that must be preserved;
* Allocates space in the stack frame for local storage.

And epilog does an opposite job to prolog:

* Deallocates space that the prolog allocates;
* POP preserved registers;
* POP R7;
* POP LR, and PC = LR.

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.

### Pass arguments and return values

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”:

“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.”

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;

}

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.

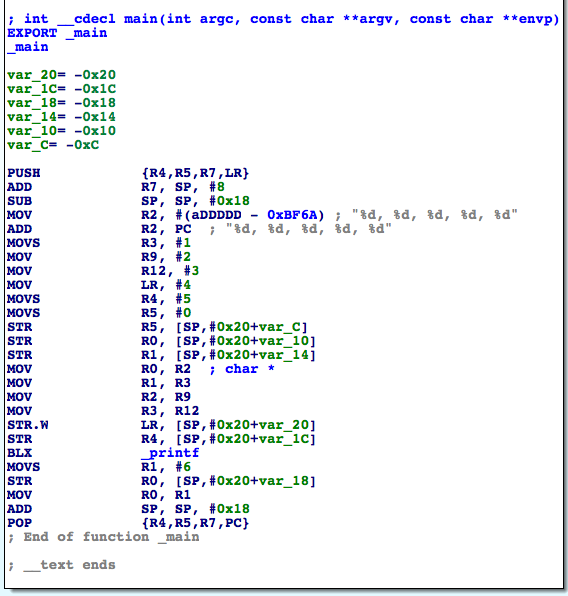


Figure 6-9 main in assembly

“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].

I don’t think we need further explanation.

“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.”

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

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 Advanced methodology of writing a tweak

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.

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?

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.

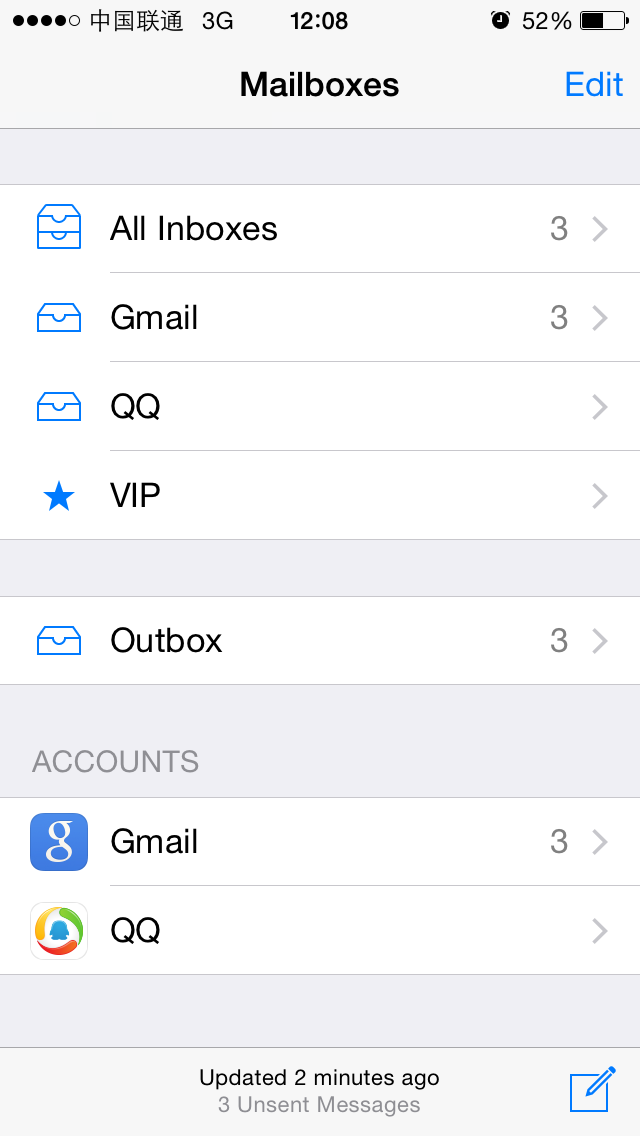


Figure 6- 10 Mail



Figure 6- 11 MobilePhoneSettings

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 Cut into the target App and find the UI function

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.

### Inject Cycript into Mail

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

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

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

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 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>>"

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>>"

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>>"]

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>>"

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.

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>>

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]

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

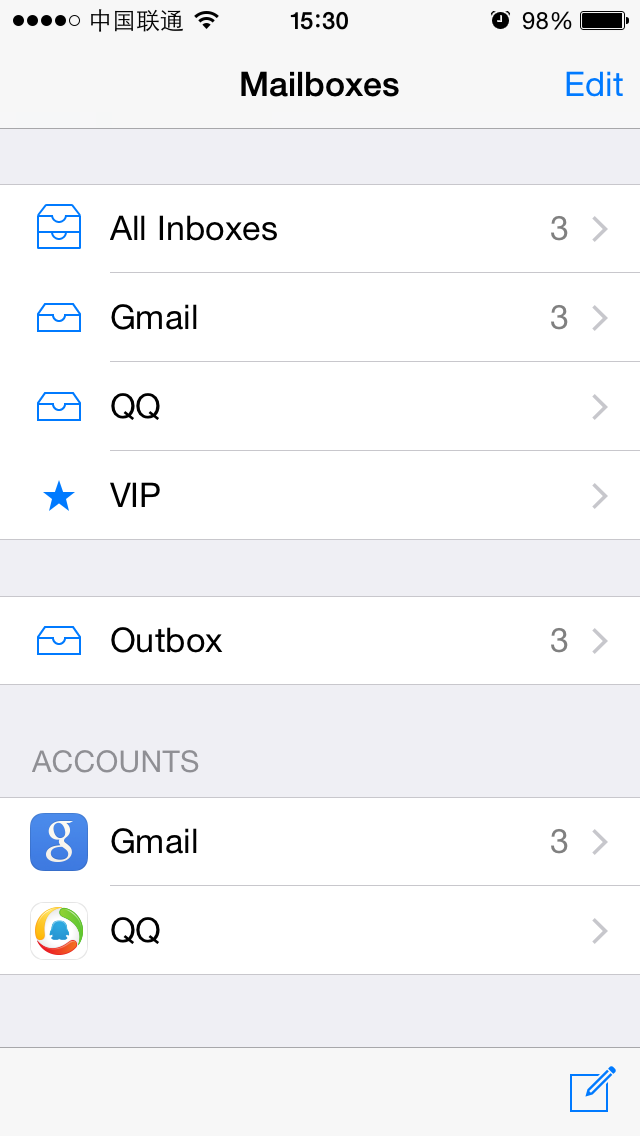


Figure 6-12 Two lines of sentences are hidden

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>>

Try to hide it and see if the button disappears:

cy# [#0x146e6060 setHidden:NO]

cy# [#0x146c4110 setHidden:YES]

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>>

Let’s repeat the operation to hide UIToolBar:

cy# [#0x146c4110 setHidden:NO]

cy# [#0x146f62a0 setHidden:YES]

The effect is shown in figure 6-13:

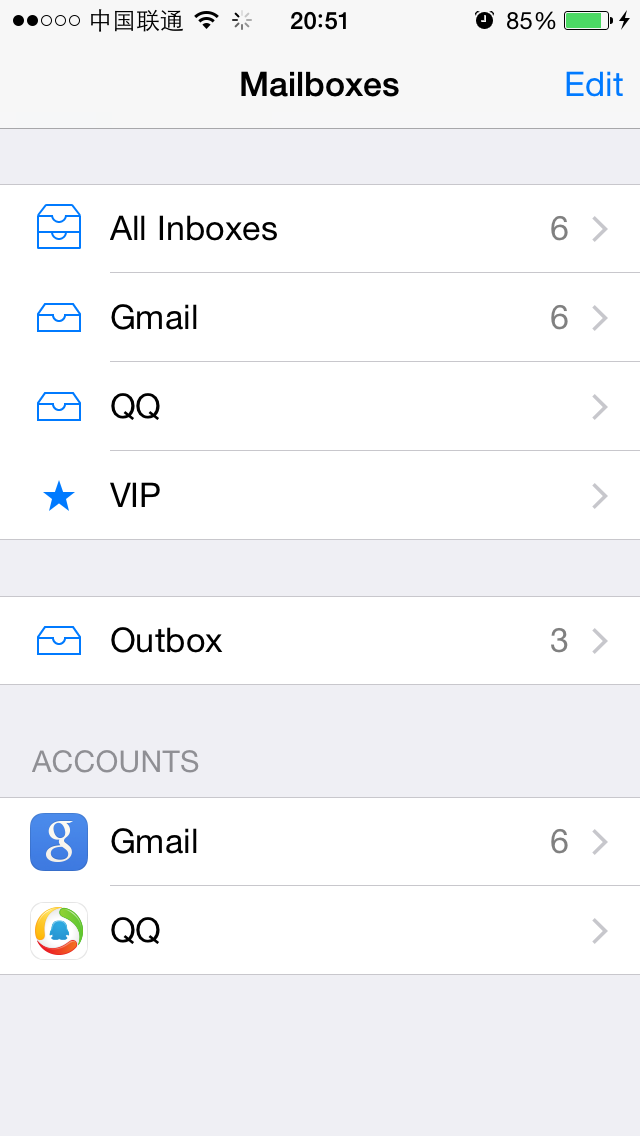


Figure 6-13 UIToolBar is hidden

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>>

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

cy# [#0x146f62a0 setHidden:NO]

cy# [#0x14798410 setHidden:YES]

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

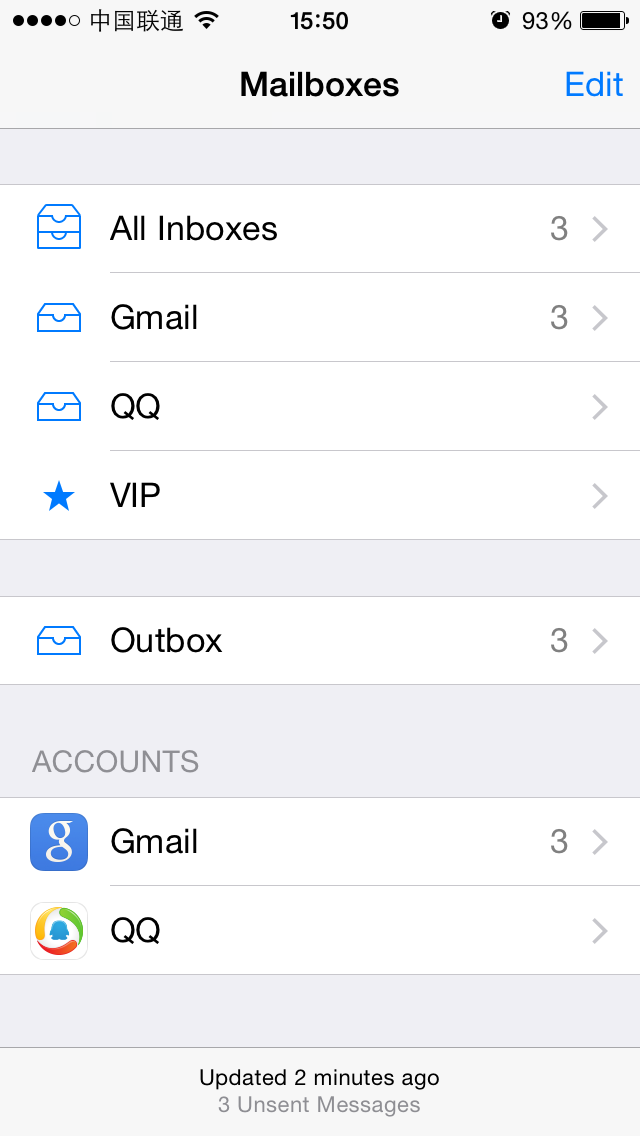


Figure 6-14 Button is hidden

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.

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

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:"]

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.

### Inject Cycript into MobilePhoneSettings

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

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

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

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>>

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]

Now, MobilePhoneSettings looks like figure 6-15:



Figure 6-15 Hide the top cell

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?

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?

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

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, and so on. 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.

### Find the UI function of the top cell

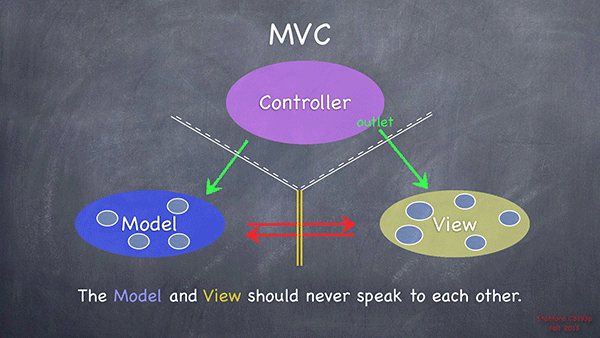


Figure 6-16 MVC design pattern, taken from Stanford CS 193P

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?

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.”*

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.

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}>>"

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

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

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

……

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:

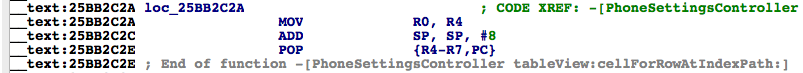


Figure 6-17 [PhoneSettingsController tableView:cellForRowAtIndexPath:]

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>>

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.

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.

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

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?

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)

}

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:].

### Look for the target function of “compose email”

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

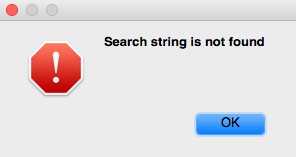


Figure 6-18 [ComposeButtonItem \_sendAction:withEvent:] is not found

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

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

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

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?

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

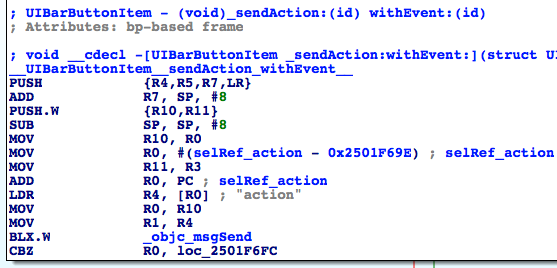


Figure 6-19 [UIBarButtonItem \_sendAction:withEvent:]

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.”*

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?

Still remember “sentence of the book”?

“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.”

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.

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”?

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.

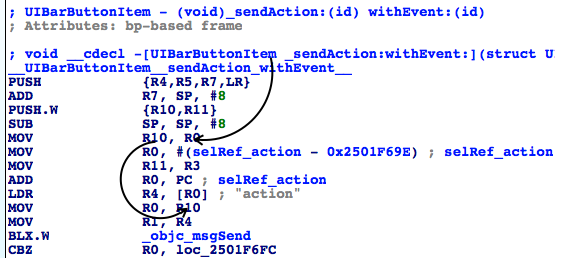


Figure 6-20 R0’s evolution

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:

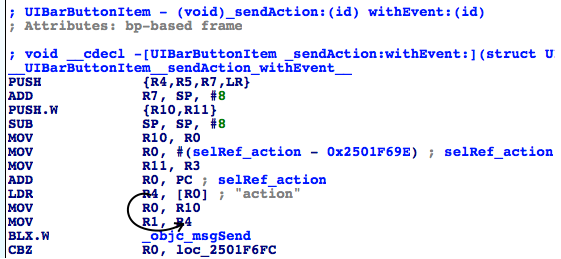


Figure 6-21 R1’s change process

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:

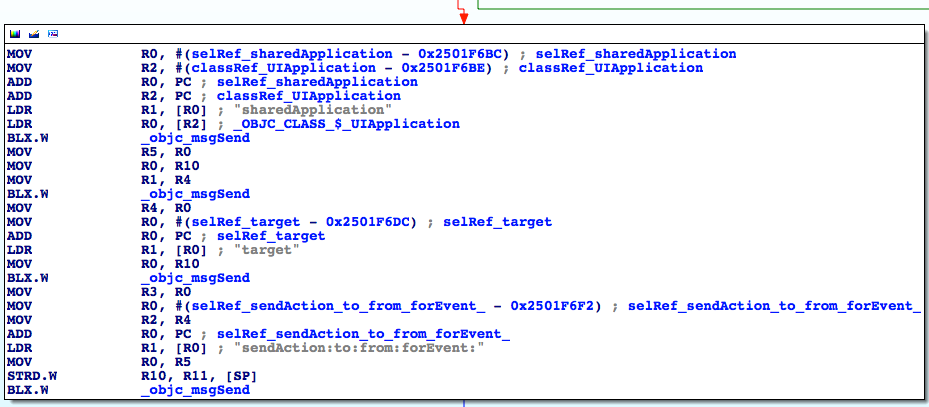


Figure 6-22 [UIBarButtonItem \_sendAction:withEvent:]

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

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.

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.

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.

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.

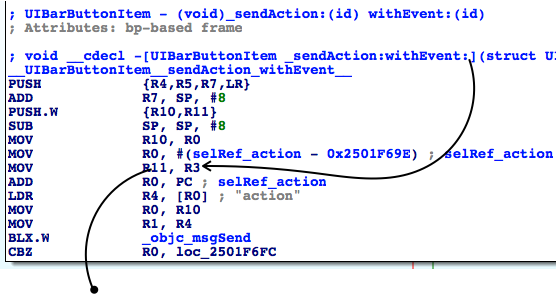


Figure 6-23 The relationship of objc\_msgSend’s arguments

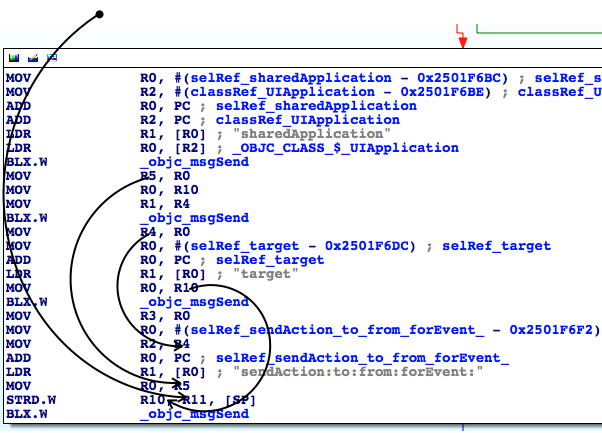


Figure 6-24 The relationship of objc\_msgSend’s arguments

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.

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

……

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

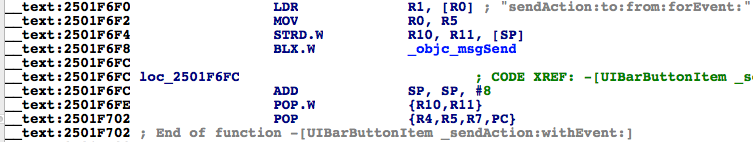


Figure 6-25 Check out address of objc\_msgSend

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}

)}

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.

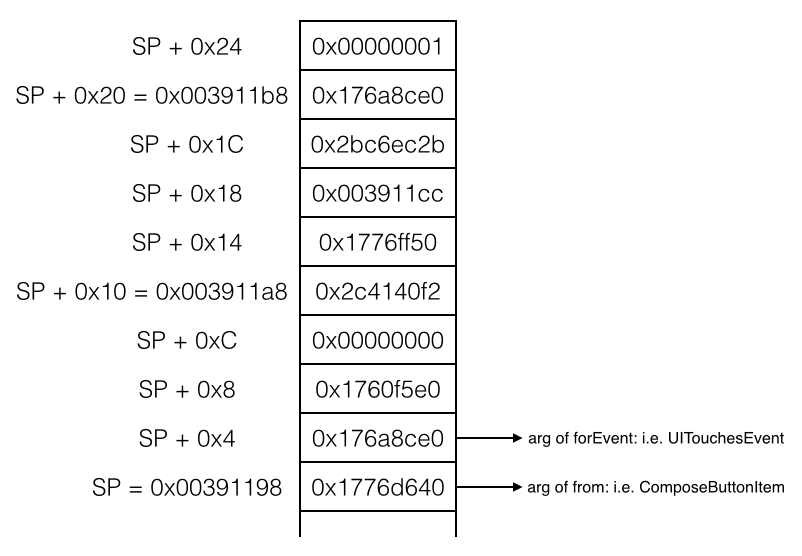


Figure 6-26 The relationship of SP, value in the stack and arguments

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.

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:

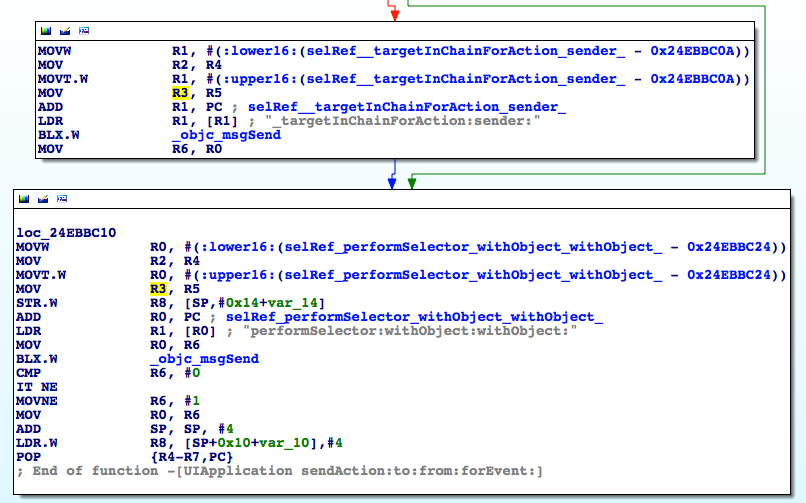


Figure 6- 27 [UIApplication sendAction:to:from:forEvent:]

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}

)}

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}

)}

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”.

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

FunMaker-5:~ root# cycript -p MobileMail

cy# [UIApp composeButtonClicked:[ComposeButtonItem composeButtonItem]]

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. Look for the target function of “my number”

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”.

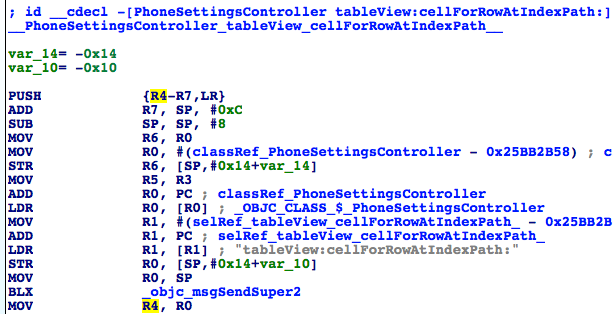


Figure 6-28 Source of R4

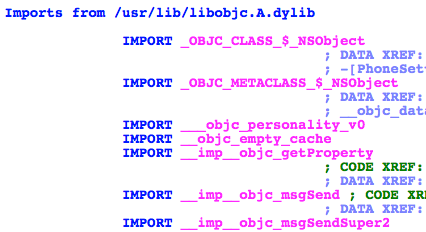


Figure 6-29 Source of objc\_msgSendSuper2

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

……

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

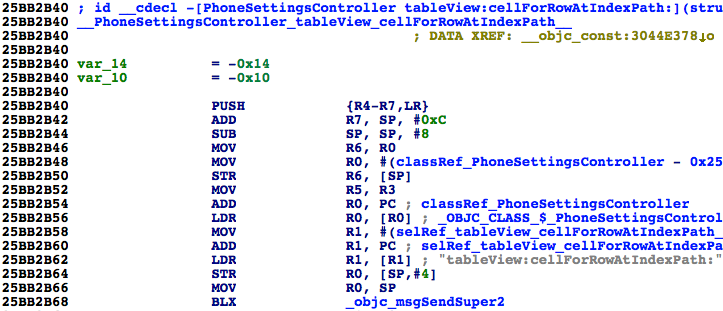


Figure 6-30 Check out address of objc\_msgSendSuper2

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>>

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>.

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

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

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.

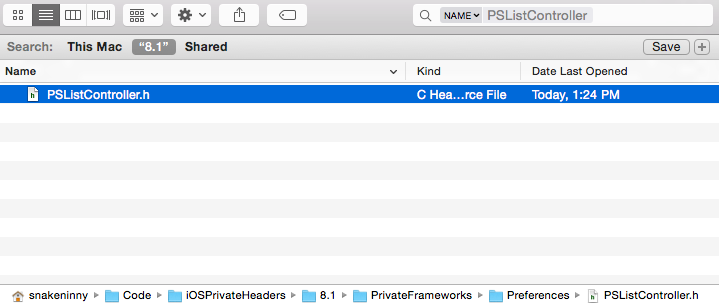


Figure 6-31 Locate PSListController.h

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

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.

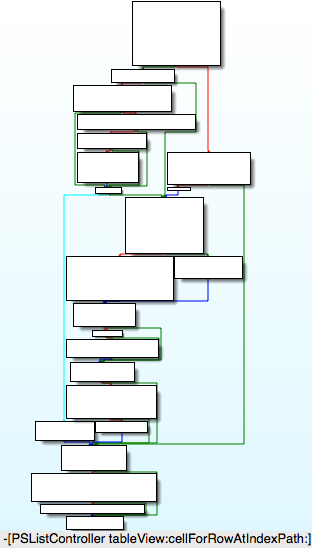


Figure 6-32 [PSListController tableView:cellForRowAtIndexPath:]

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

……

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

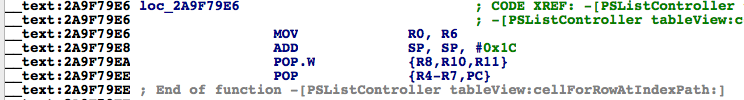


Figure 6-33 [PSListController tableView:cellForRowAtIndexPath:]

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>>

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.

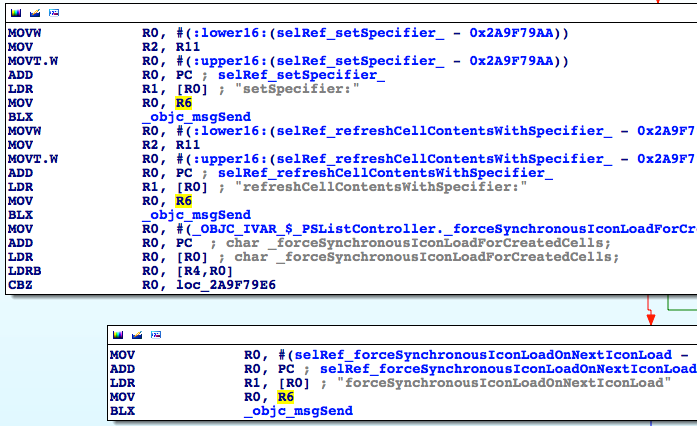


Figure 6-34 Multiple occurrences of R6

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.

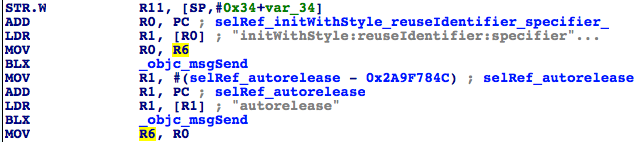


Figure 6-35 The assignment of R6

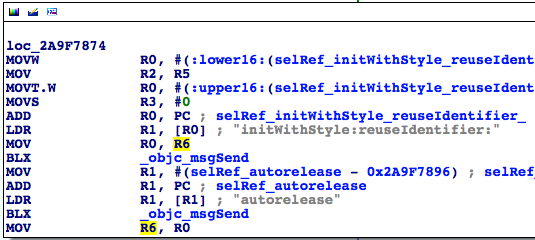


Figure 6-36 The assignment of R6

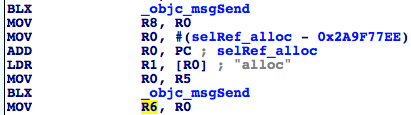


Figure 6-37 The assignment of R6

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?

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.

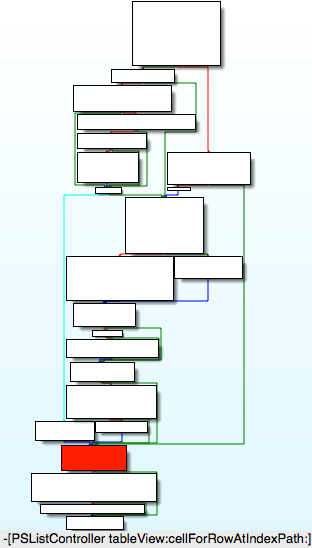


Figure 6-38 [PSListController tableView:cellForRowAtIndexPath:]

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.

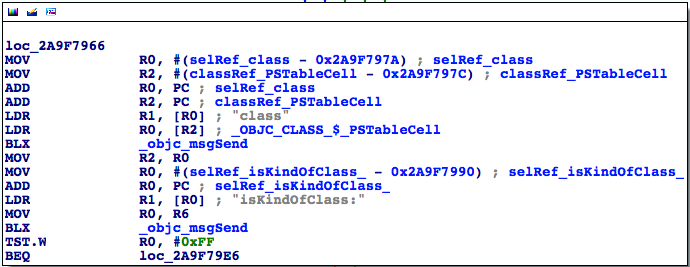


Figure 6-39 loc\_2a9f7966

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

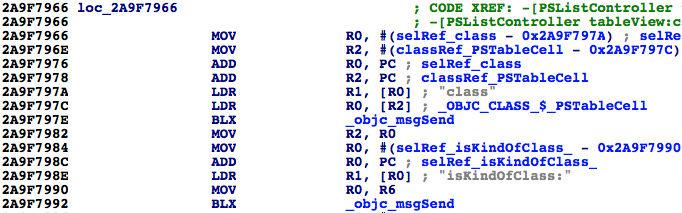


Figure 6-40 Check out address of objc\_msgSend

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>>

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>>

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.



Figure 6-41 The configuration objc\_msgSend

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

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:

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}>>

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.

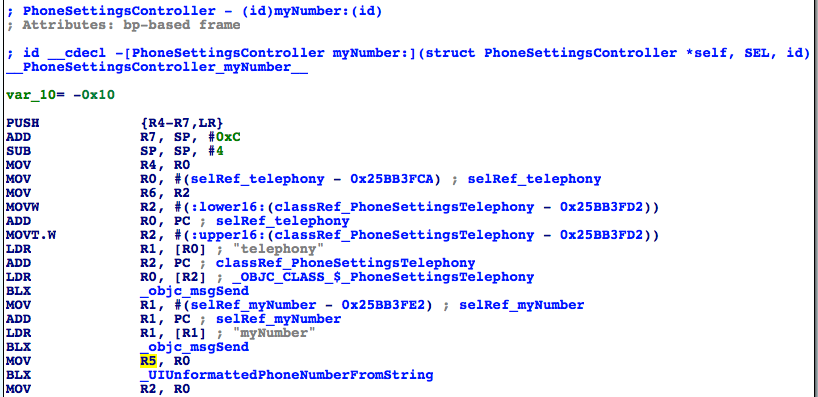


Figure 6-42 [PhoneSettingsController myNumber:]

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"

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

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>.

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.

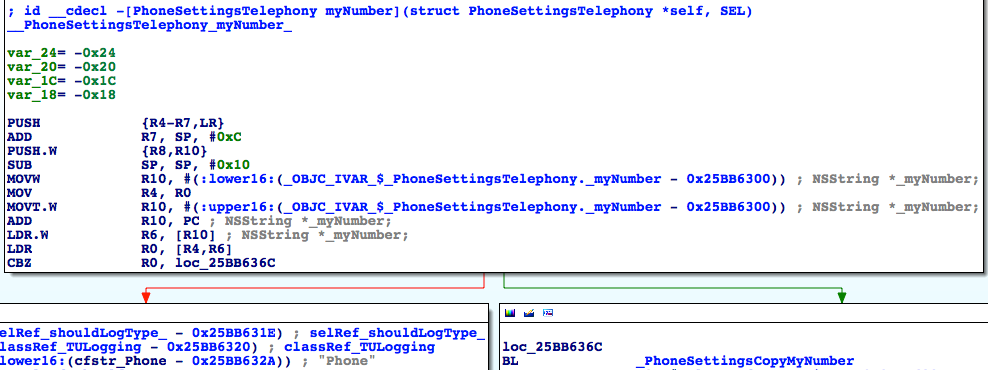


Figure 6-43 [PhoneSettingsTelephony myNumber]

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 one step further, but not far enough. Let’s continue by digging into PhoneSettingsCopyMyNumber, as shown in figure 6-44.

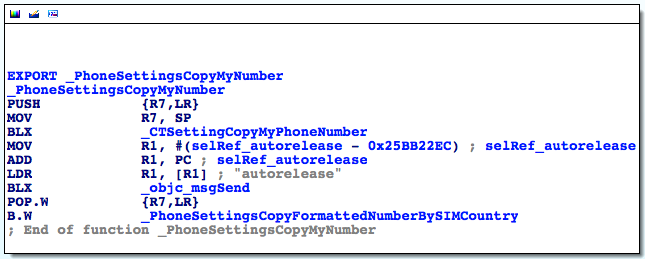


Figure 6-44 PhoneSettingsCopyMyNumber

This snippet first calls CTSettingCopyMyPhoneNumber and autoreleases 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.

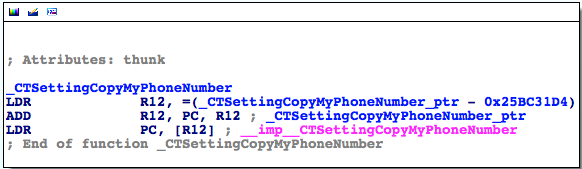


Figure 6-45 CTSettingCopyMyPhoneNumber

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?

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.

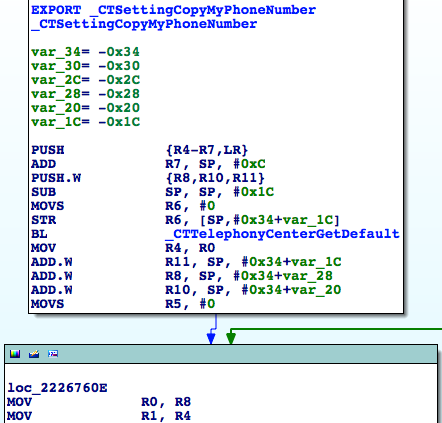


Figure 6-46 CTSettingCopyMyPhoneNumber

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.

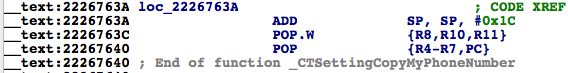


Figure 6-47 CTSettingCopyMyPhoneNumber

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

……

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

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

NSString \*CTSettingCopyMyPhoneNumber(void);

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 you make use of this function. At last, let’s write a tweak to test this function.

1. 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. 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. Edit Makefile and control

The finalized Makefile looks like this:

export THEOS\_DEVICE\_IP = iOSIP

export ARCHS = armv7 arm64

export TARGET = iphone:clang: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"

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. Test

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. P.S.

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 is the American phone number format.

You’ll run into CTSettingCopyMyPhoneNumber more frequently as you reverse more. Actually, the prototype of CTSettingCopyMyPhoneNumber should be:

CFStringRef CTSettingCopyMyPhoneNumber(void);

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

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”.

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 and return values 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 Advanced LLDB usage

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.

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 Look for a function’s caller

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;

}

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.

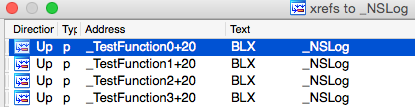


Figure 6-48 Check the cross references of NSLog

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 that 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.

#### Inspect LR

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();

……

}

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.

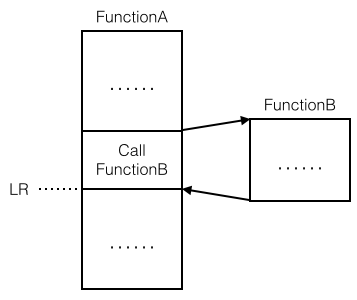


Figure 6-49 An illustration of return address

The address that the process returns to after the execution of FunctionB, is the return address, i.e. LR. Because it’s inside FunctionB’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.

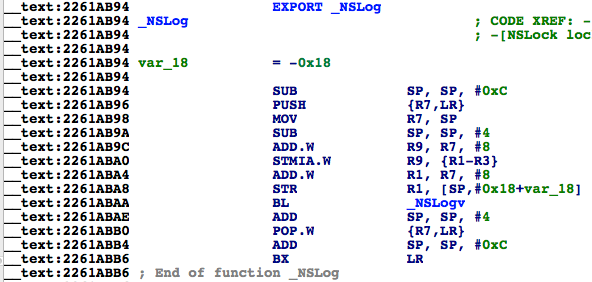


Figure 6-50 Check out NSLog’s base address

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 \*...

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

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

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

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

……

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

Print out LR:

(lldb) p/x $lr

(unsigned int) $0 = 0x00107f8d

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

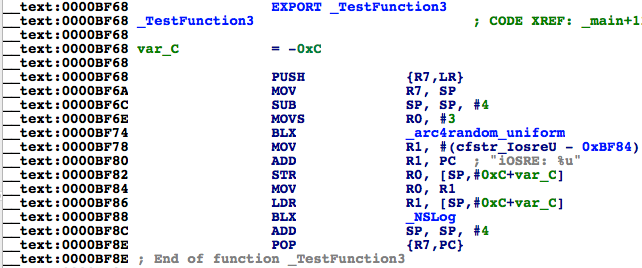


Figure 6-51 TestFunction3

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?

#### Execute “ni” to get inside caller

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

……

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

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}

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.

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

#### 6.3.2 Change process execution flow

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;

}

Save this snippet as 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

Run it:

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

FunMaker-5:~ root#

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 get called. I’ll show you how, please keep focused. Drag and drop MainBinary into IDA, then locate to the branch before ImportantAndComplicatedFunction, as shown in figure 6-52.

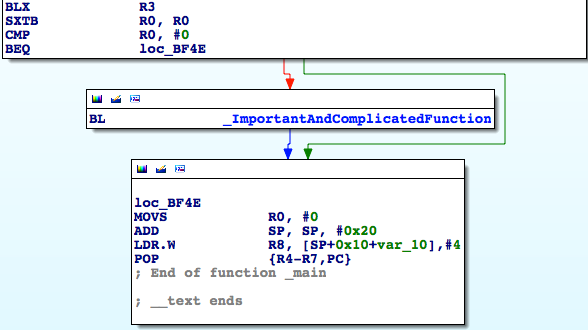


Figure 6-52 Before ImportantAndComplicatedFunction

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

(lldb) image list -o -f

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

……

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 is 0, so ImportantAndComplicatedFunction will not 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)

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

#### 6.4 Conclusion

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 book. 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.

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 after all, it’s up to you. Good luck!