Lab Session 0x06

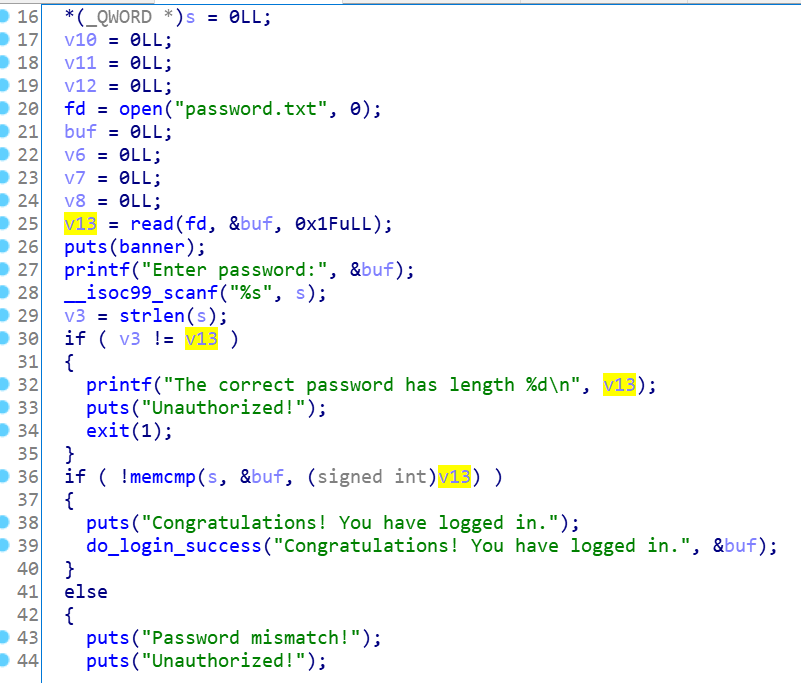
**3.1 Smashing the stack (7p)**

*For this section, use the binaries in the* ***05-lab-files.zip*** *file.*

**Task 3.1.1: stack-buffer overflow into data (2p)**

* *Do an initial analysis of the binary in IDA. Find the buffer overflow vulnerability and calculate the required input length to overwrite the* ***pass\_len*** *variable. (0.5p)*

This is the ***main()*** function:

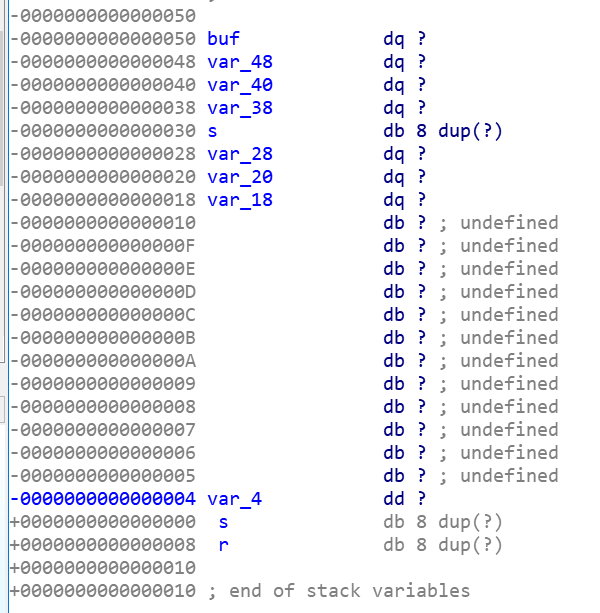


The buffer overflow takes place at line 28 – the ***scanf*** does not limit the number of characters that can be introduced by the user.

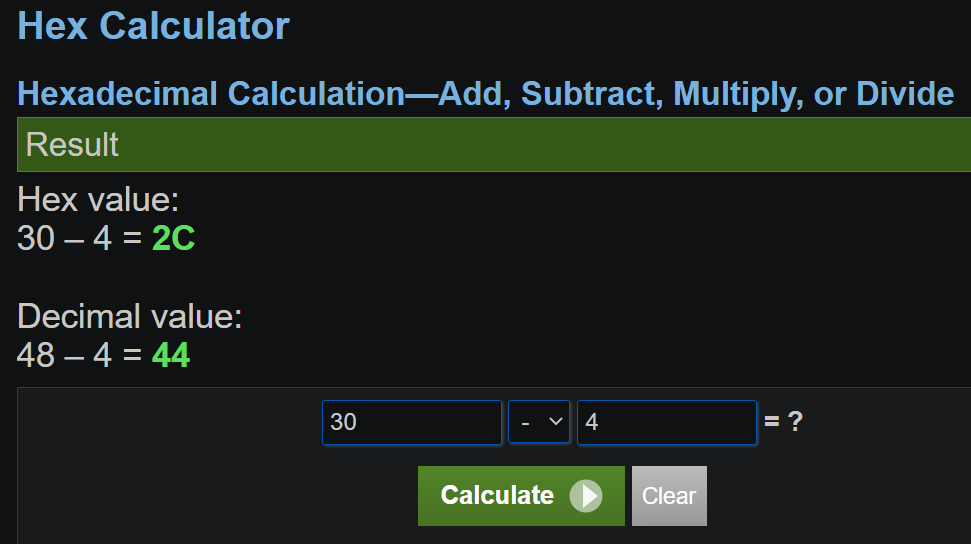
We can assume that ***v13*** is the ***pass\_len*** variable that we are looking for:



We double click it and it opens the stack frame (***var\_4*** is the corespondence of ***v13***):

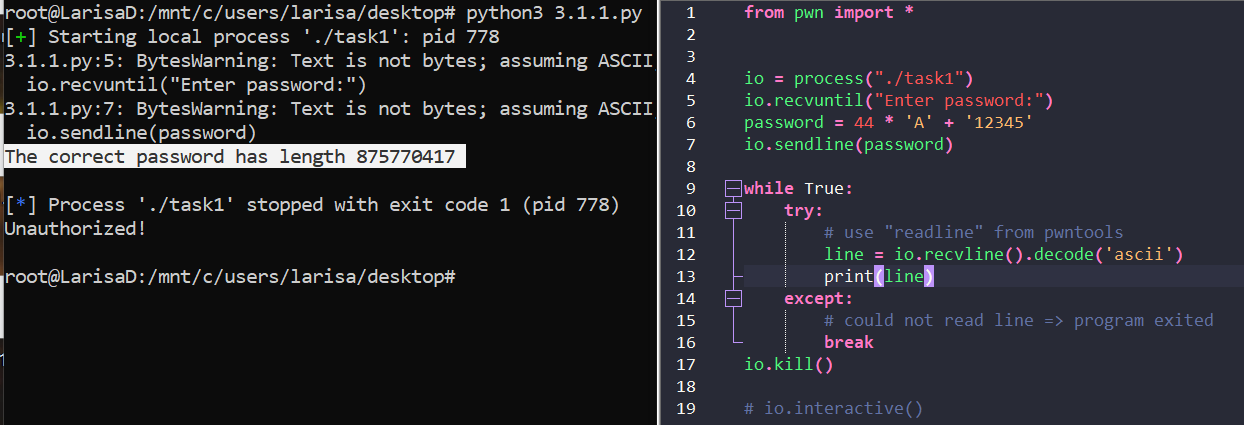


So, in order to overwrite it, we need: **44** bytes + ***new\_var\_4*** ([link](https://www.calculator.net/hex-calculator.html?number1=30&c2op=-&number2=4&calctype=op&x=53&y=32)).

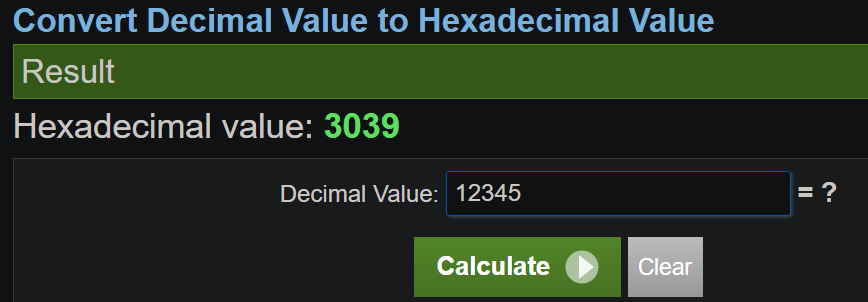


* *Starting from the template, construct an input that overflows into the* ***pass\_len*** *variable and make the program print: The correct password has length 12345 „\n” Unauthorized! (0.5p)*

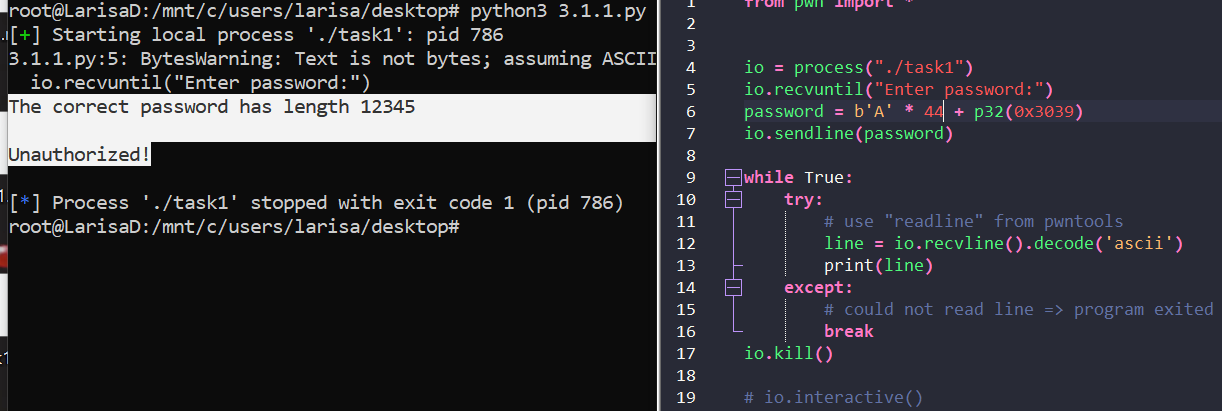
We need to put **44** random bytes and then the new value for ***var\_4***. So let’s try it:



It’s not really what we want. ([Link](https://www.calculator.net/hex-calculator.html?d2bnumber1=12345&calctype=d2b&x=85&y=13#decimal2hex))

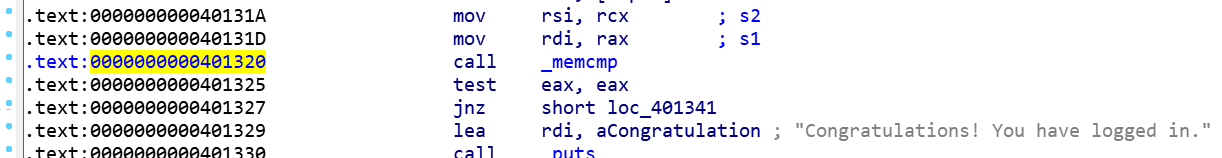


Let’s try using this value instead:

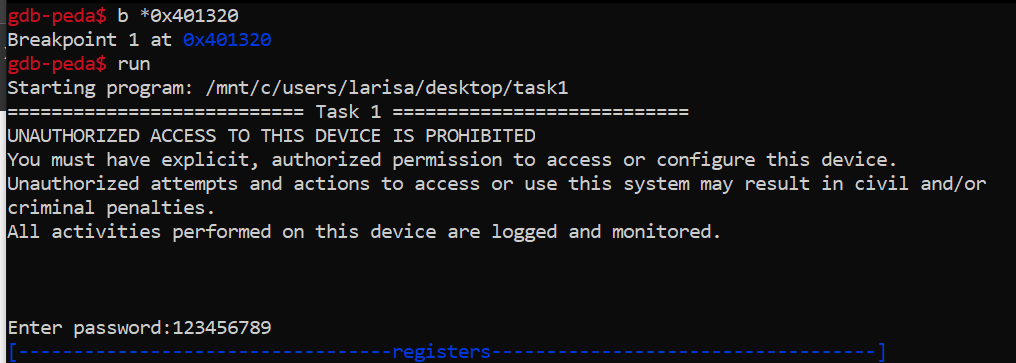


Our assumption was correct. The code is in ***3.1.1\_1.py***

* *Bypass the memcmp comparison by forcing its third parameter to be 0. (0.5p)*



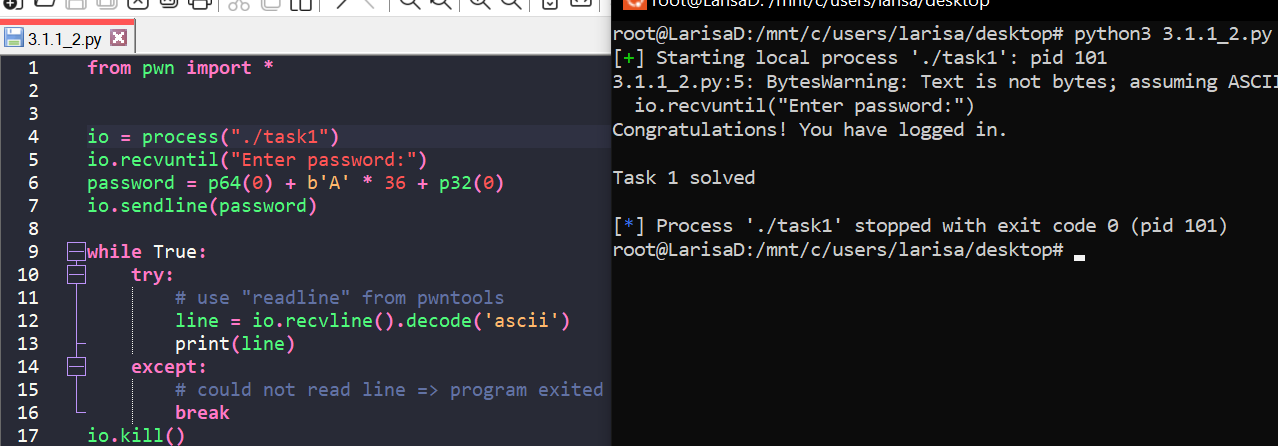
The address is: ***0x401320***. Using gdb-peda, we set a breakpoint at this address, then we input a password of 9 characters (in order to bypass the length check) and then set the 3rd parameter to 0:





Files: *peda-session-task1.txt*

We can obtain the same result if we send as password: ***p64(0) + b'A' \* 36 + p32(0)*** ([Source](https://ssst0n3.github.io/post/%E7%BD%91%E7%BB%9C%E5%AE%89%E5%85%A8/CTF/pwn/pwntools/p32-or-p64-or-struct.html))



* *Exploit the service running at 45.76.91.112 port 10051. (0.5p)*

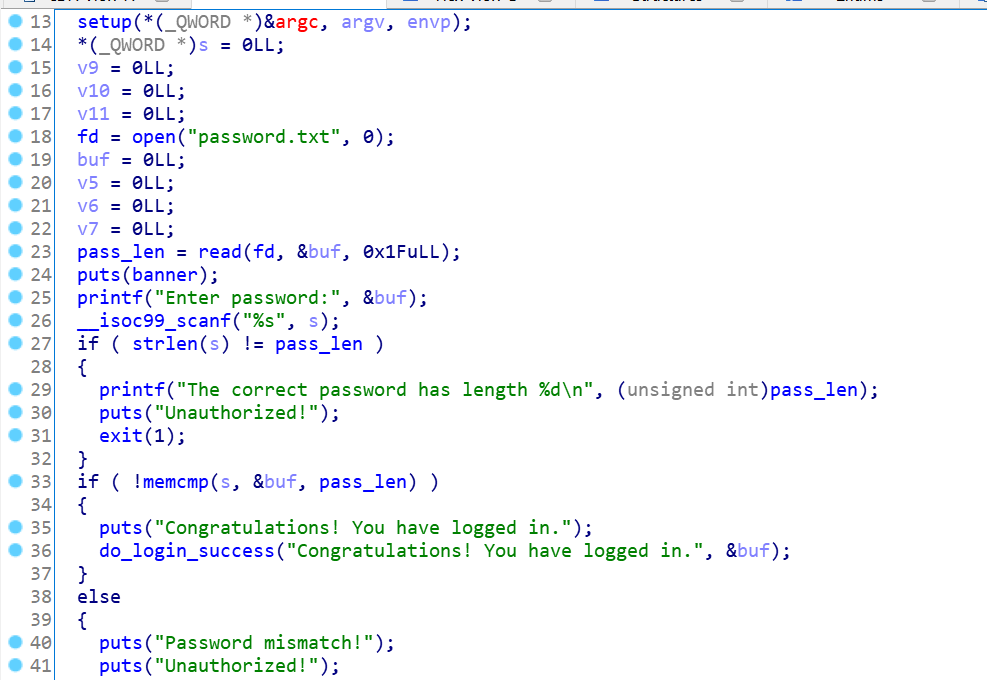
File is: ***3.1.1\_2.py*** ([Source](https://docs.pwntools.com/en/stable/intro.html))



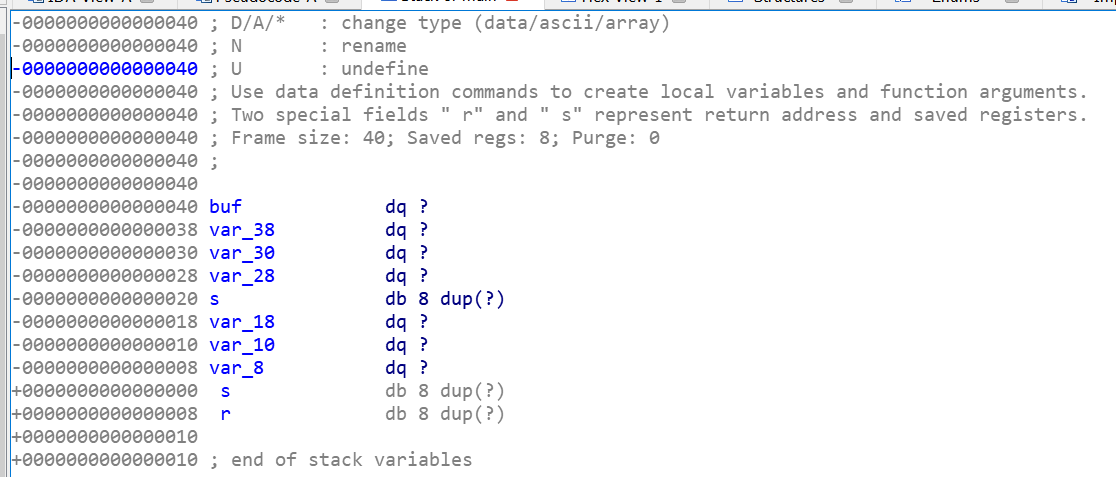
**Task 3.1.2: stack-buffer overflow into ret addr (3p)**

* *Do an initial analysis of the binary in IDA. Find the buffer overflow vulnerability, look at the stack frame and calculate the required input length to overwrite the return address. (1p)*

Just like above, the buffer overflow is caused by the *scanf* function:



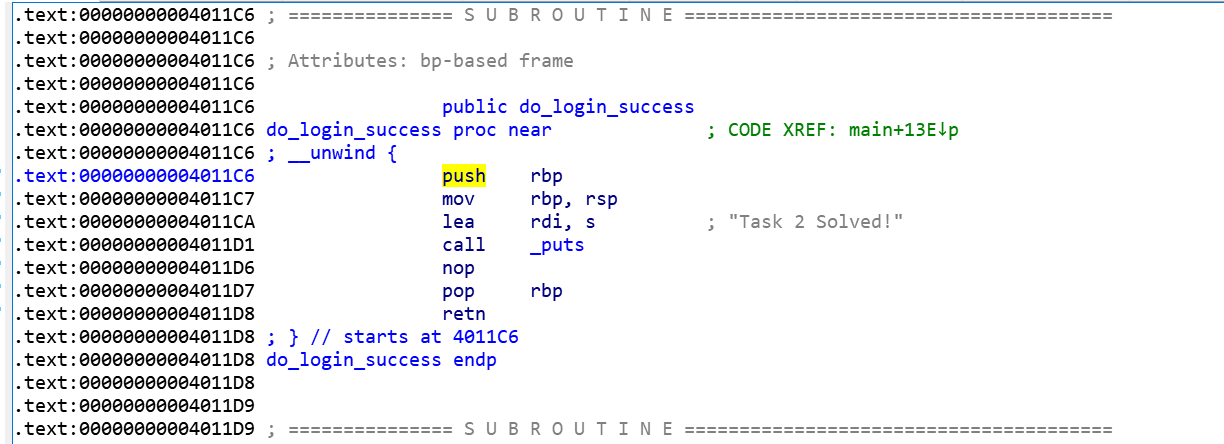
Double click on ***s*** and we see the stack:



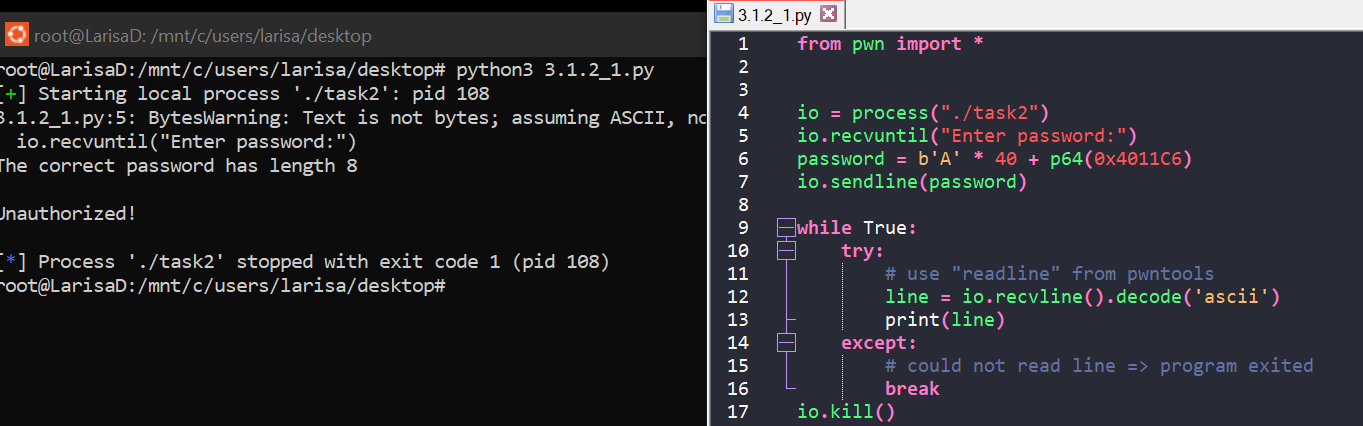
To overwrite the return address we need 8 bytes (for s) + 8 bytes (for var\_18) + 8 bytes (for var\_10) + 8 bytes (for var\_8) + 8 bytes (for s) + bytes\_for\_r = **40** bytes + ***bytes\_for\_r***

* *Starting from the template, construct an input that overflows into the return address and replaces it with the address of* ***do\_login\_success****. (1p)*

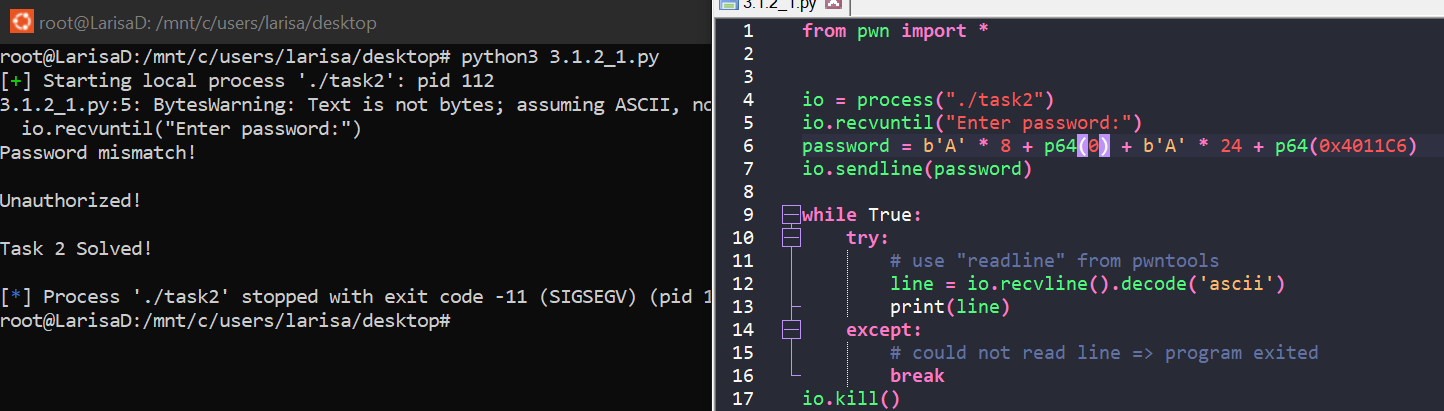
The address for ***do\_login\_success*** is ***0x4011C6***:



Let’s write the program:

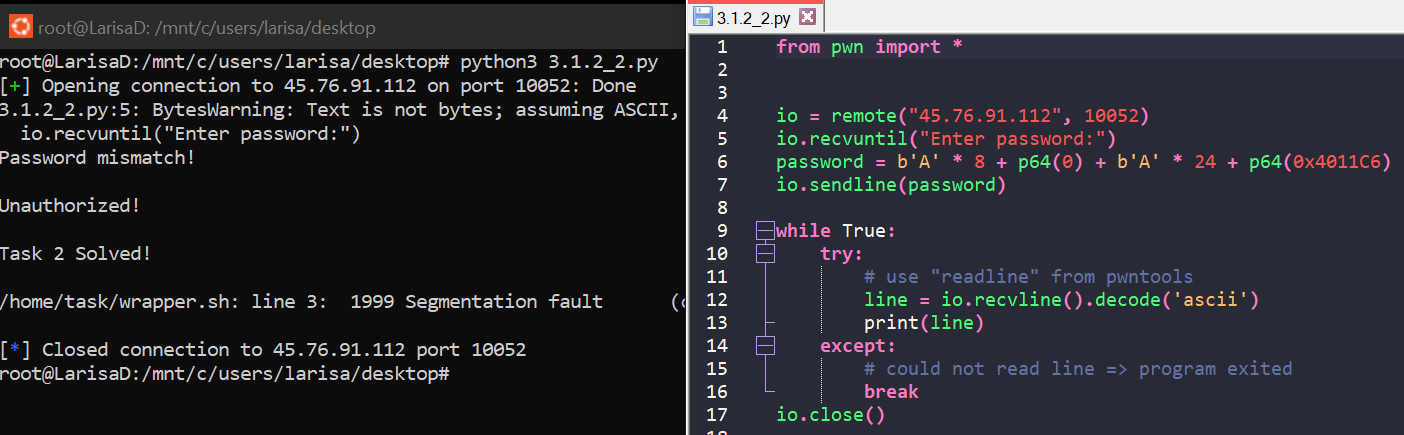


It seems it didn’t work. Overflowing it broke the length check for the password, so let’s fix the issue:



File: *3.1.2\_1.py*

* *Exploit the service running at 45.76.91.112 port 10052. (1p)*



File: *3.1.2\_2.py*

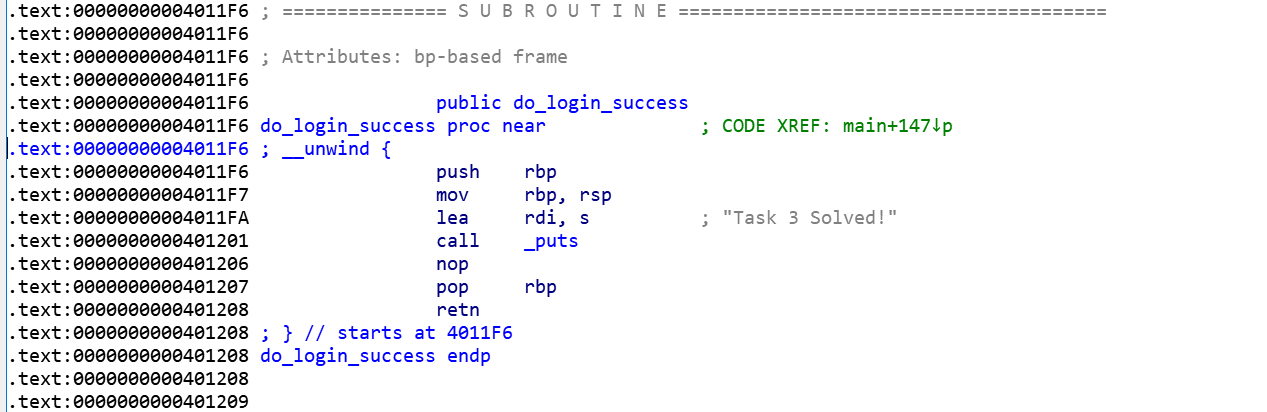
**Task 3.1.3: stack-buffer overflow protection (2p)**

* *Starting from the template, construct an input that bypasses the stack protection but still overflows into the return address and replaces it with the address of* ***do\_login\_success****. (1p)*

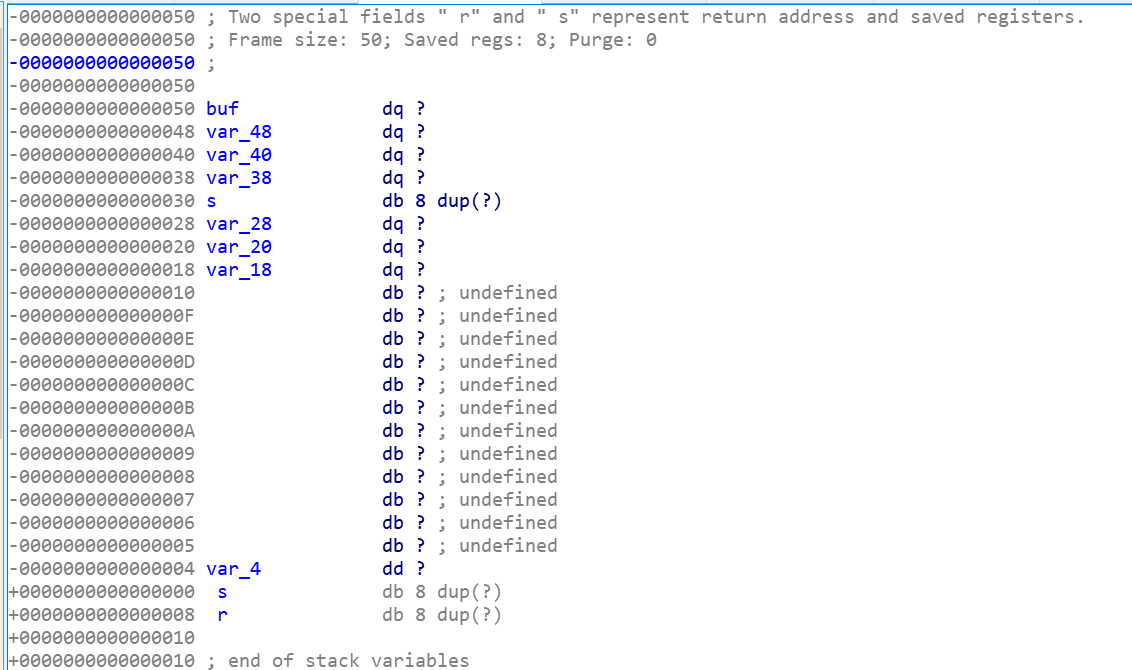
The stack protection occurs at line 45 (***global\_stack\_cookie***).



The address for ***do\_login\_success*** is ***0x4011F6***:



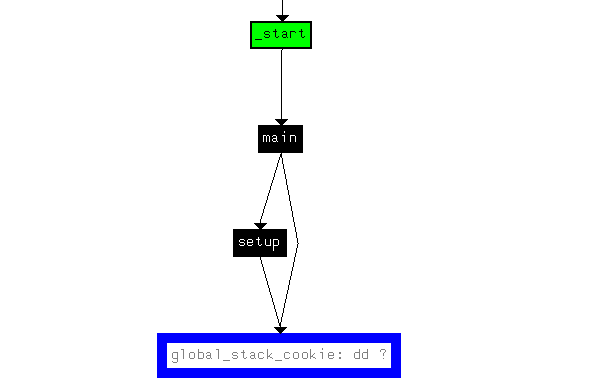
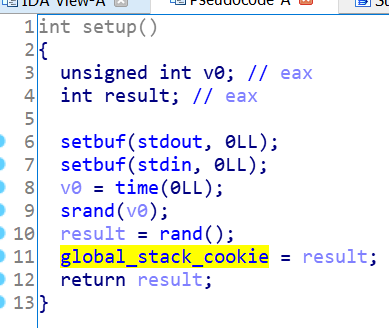
The stack:



We have to take care of ***var\_4*** (it is v12 from main; line 15):



Let’s see how ***global\_stack\_cookie*** is being generated. Using *xref* we can see:

So, ***global\_stack\_cookie*** is actually a random value generated using the seed of time. Let’s see how our password should look like, knowing this:

-0000000000000030 s db 8 dup(?) → b'A' \* 8 (8 bytes)

-0000000000000028 var\_28 dq ? → p64(0) (8 bytes)

-0000000000000020 var\_20 dq ? → b'A' \* 8 (8 bytes)

-0000000000000018 var\_18 dq ? → b'A' \* 20 (20 bytes)

-0000000000000010 db ? ; undefined

-000000000000000F db ? ; undefined

-000000000000000E db ? ; undefined

-000000000000000D db ? ; undefined

-000000000000000C db ? ; undefined

-000000000000000B db ? ; undefined

-000000000000000A db ? ; undefined

-0000000000000009 db ? ; undefined

-0000000000000008 db ? ; undefined

-0000000000000007 db ? ; undefined

-0000000000000006 db ? ; undefined

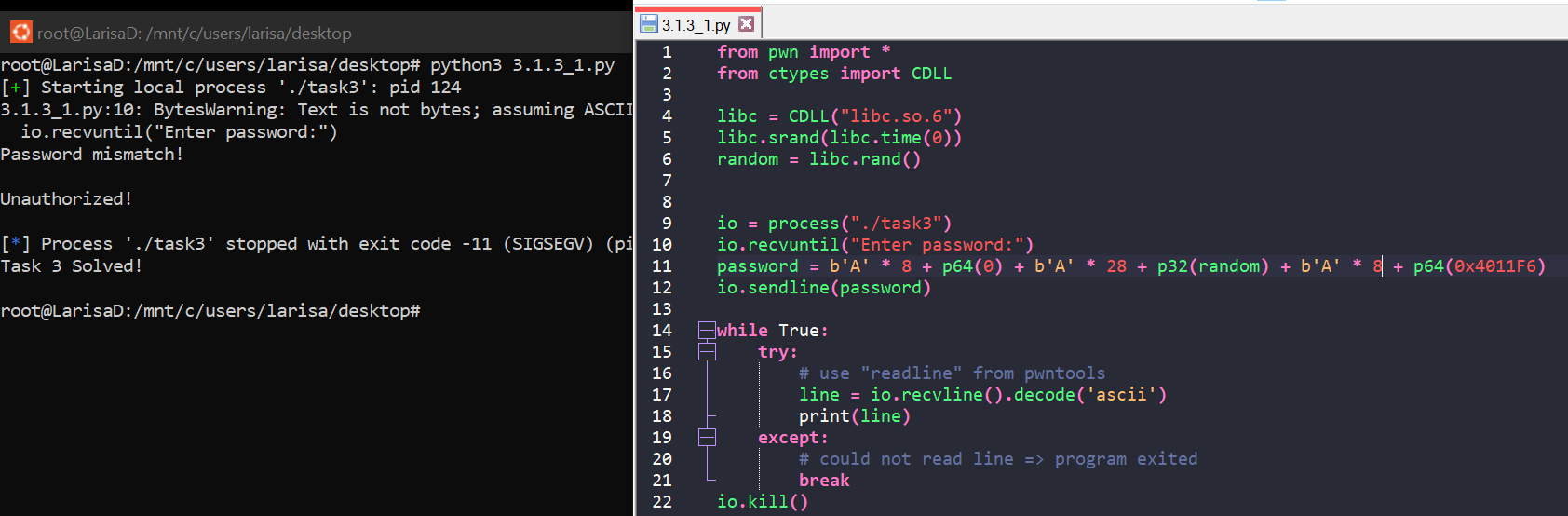
-0000000000000005 db ? ; undefined

-0000000000000004 var\_4 dd ? → p32(random\_value time(NULL)) (4 bytes)

+0000000000000000 s db 8 dup(?) → b'A' \* 8 (8 bytes)

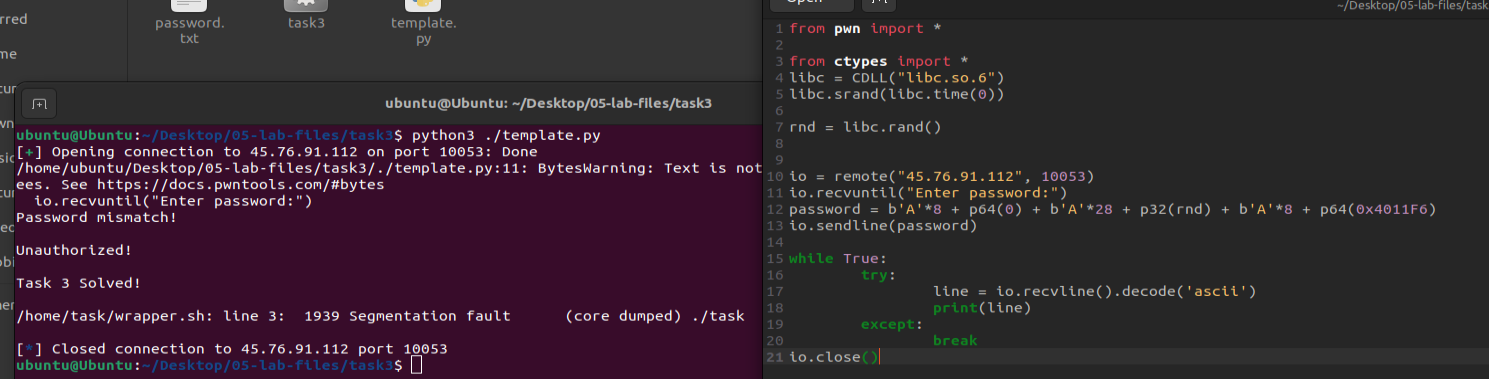
+0000000000000008 r db 8 dup(?) → p64(0x4011F6)

([Source](https://stackoverflow.com/questions/70300623/python-ctypes-time0-and-c-time0))



File: *3.1.3\_1.py*

* *Exploit the service running at 45.76.91.112 port 10053. (1p)*



(I don’t know why it doesn’t work with Linux subsystem for Windows)

File: *3.1.3\_2.py*

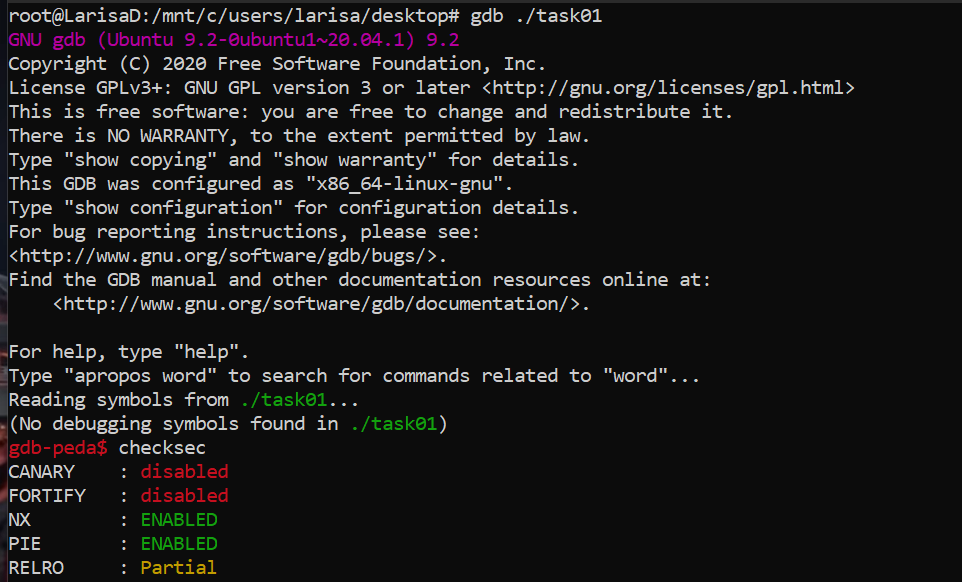
**3.2 PIE tasks (7p)**

*For this section, use the binaries in the* ***07-lab-files.zip*** *file.*

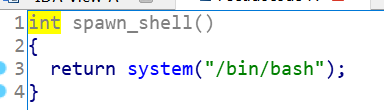
**Task 3.2.1: simple PIE (3p)**

* *Identify the binary protections and the helper function (which spawns a shell). (1p)*

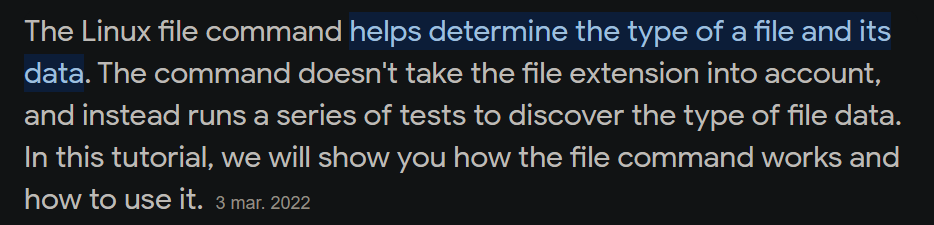
The binary protection:

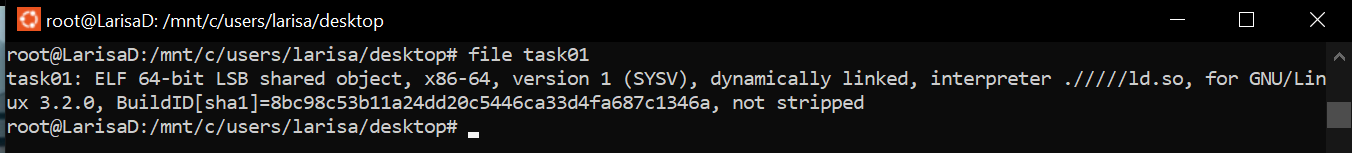


The helper function that spawns a shell:



* *Is the binary stripped? What approach is needed for breakpoints?*

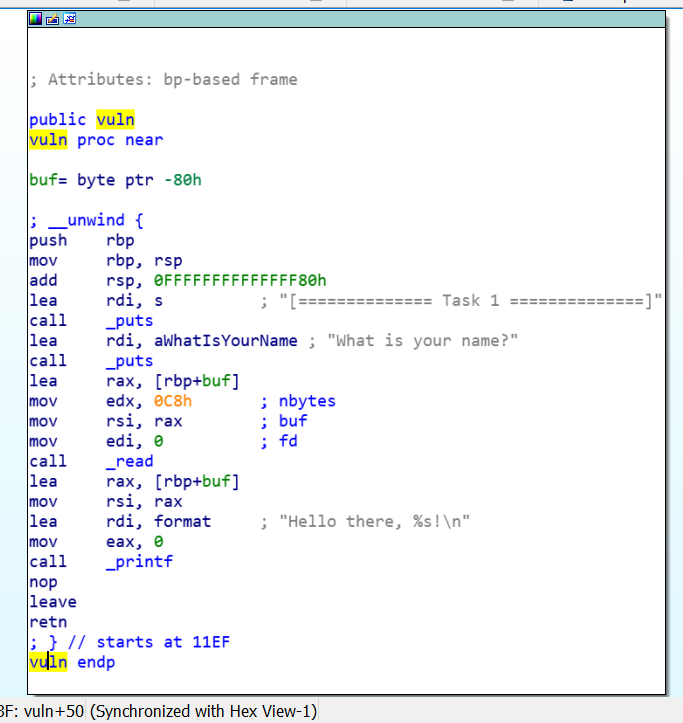




As we can see in the photo above, the binary is not stripped, but we saw at the previous point that PIE is enabled, so breakpoints for relative addresses don’t work. However, breakpoints for known symbols (e.g. *main*) work.

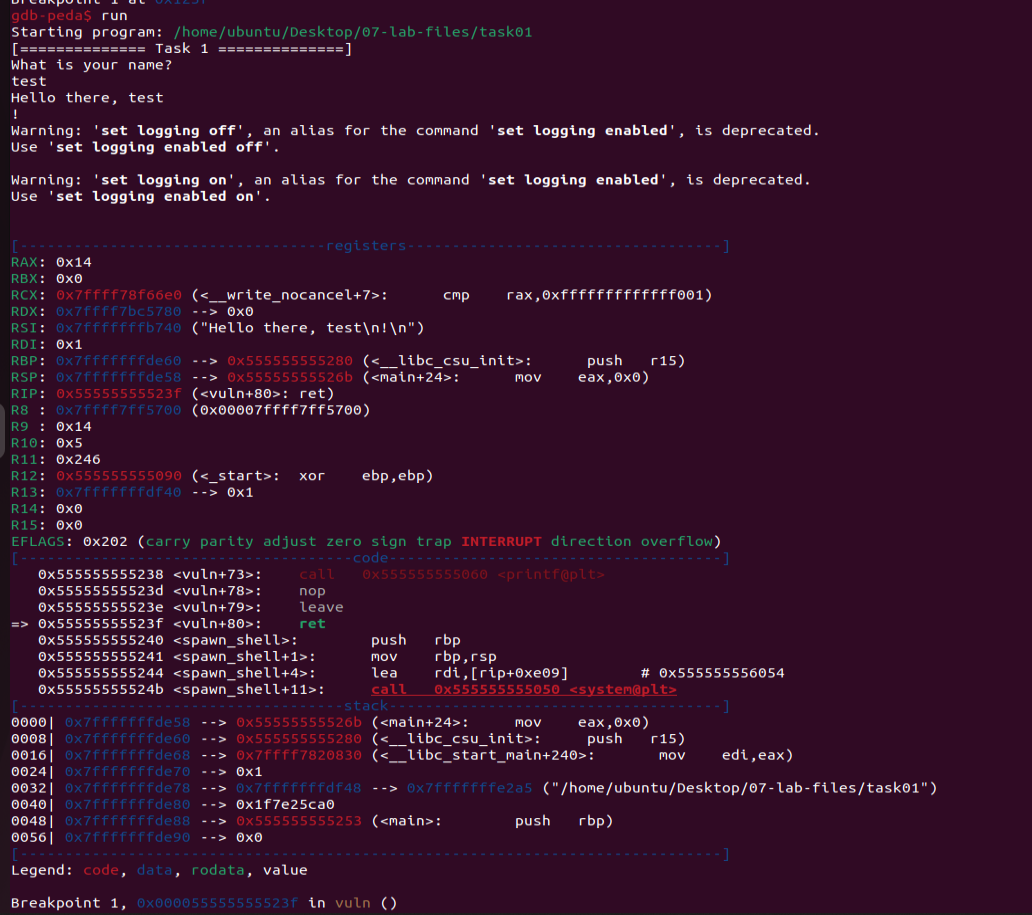
* *Set a breakpoint on the return address in the vulnerable function:*
  + *run a couple of times with ASLR.*
  + *for each run, observe the raw 8 bytes of the (overwriteable) return address.*
  + *for each run, also observe the raw 8 bytes of the target (helper function) address.*
  + *how many bytes differ between the (overwriteable) return address and the target (helper function) address. (1p)*
  + *calculate the probability that a partial overwrite of the return address succeeds.*

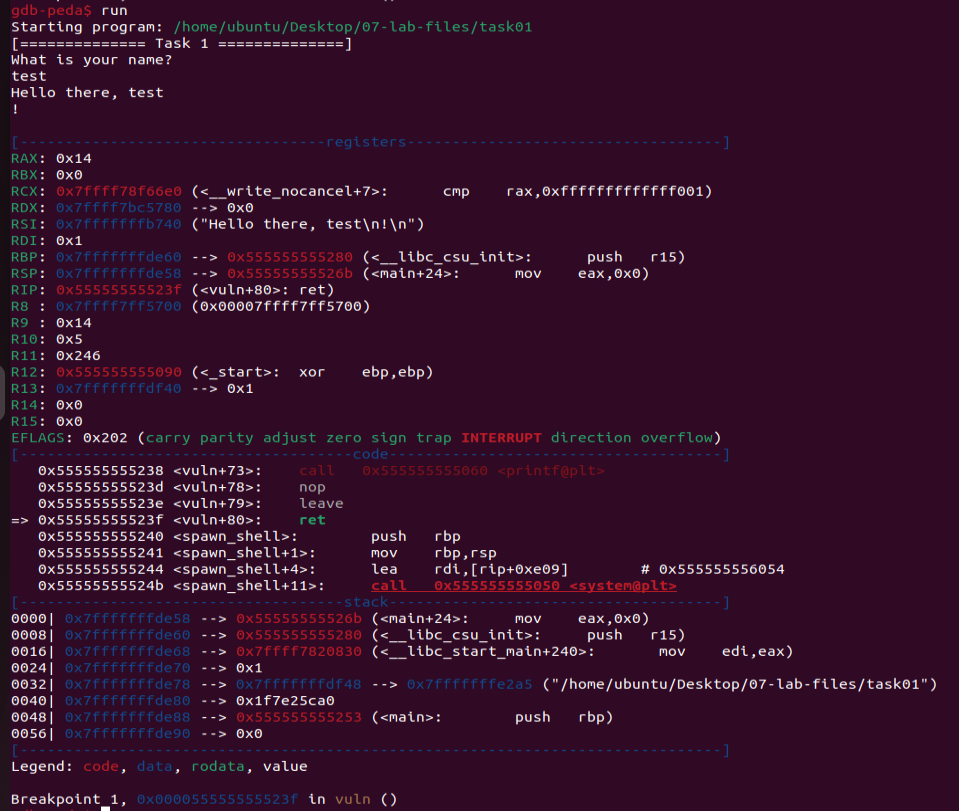
The vulnerable function in IDA:



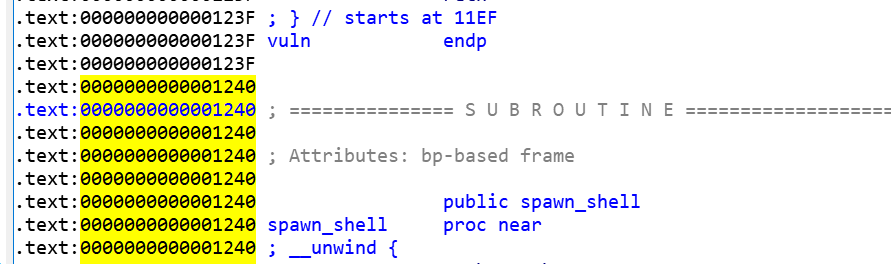
We can set a breakpoint at address (0x)vuln+50 → *vuln+80* (in decimal).



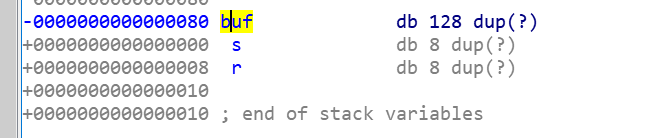




We can notice the 2 addresses of the functions and the difference between them is always 1 byte:

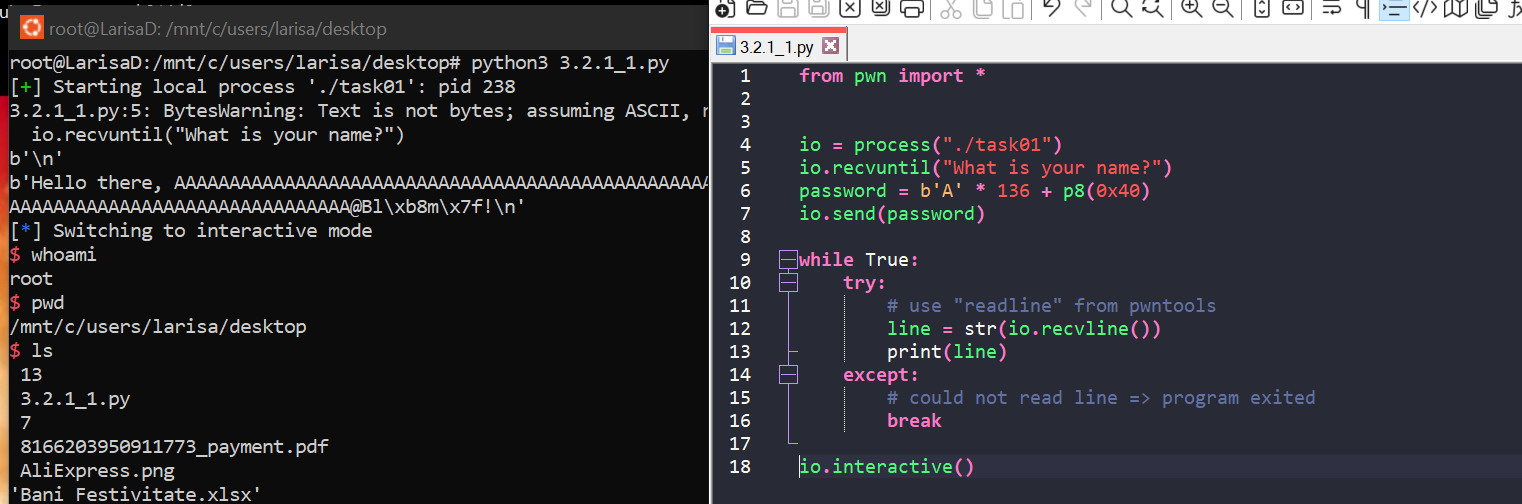


The helper function (aka *spawn\_shell*) is always loaded at an address ending with ***40***. That means, the most significant byte of the helper function is ***40*** and, by overwriting it, we should get the shell. By looking at the stack of *vuln* we see we need 128 bytes (for *buf*) + 8 bytes (for *s*) + „***40***” = 136 bytes + „***40***”.



(The photo is obtained after *right click* on ***buf*** and then select *Array*)

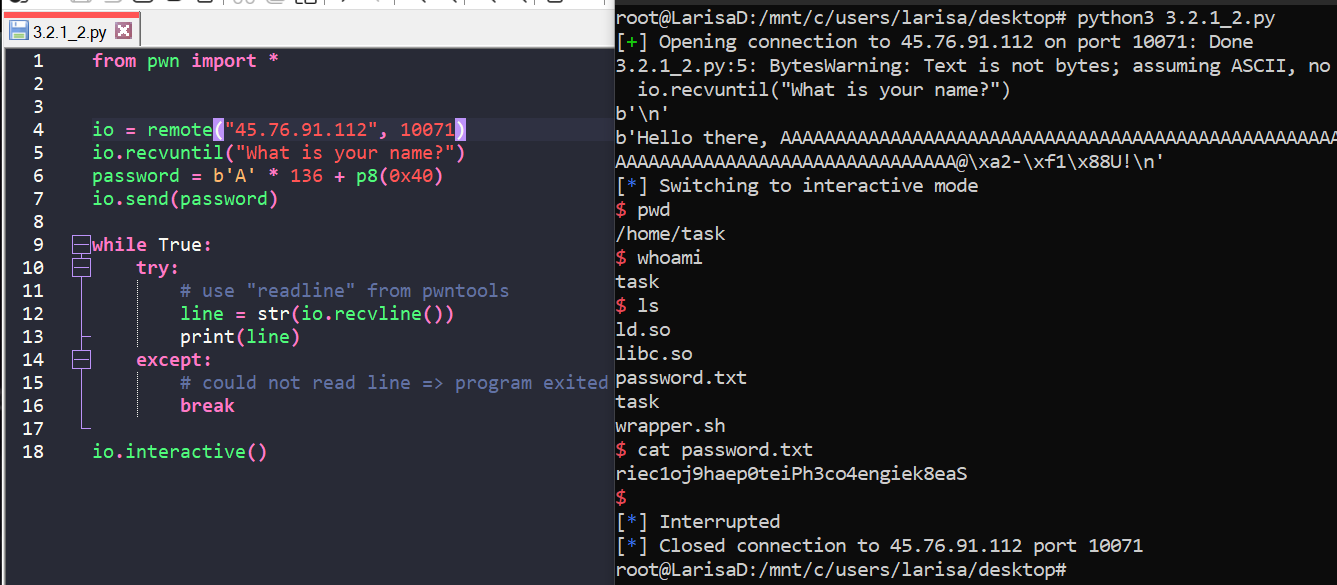
This is the result (when the script runs, in order to enter interactive mode → ***CTRL + C***):



File is *3.2.1\_1.py*

And, to answer the final question, the probability is 100%.

* *Exploit the vulnerability by doing a partial overwrite of the return address. Remote end: 45.76.91.112 10071. (1p)*

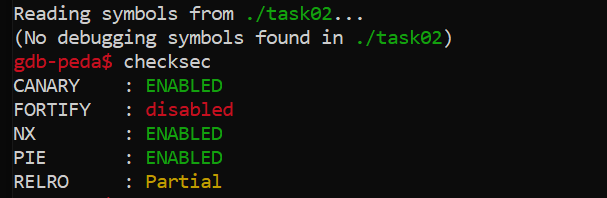


File is *3.2.1\_2.py*

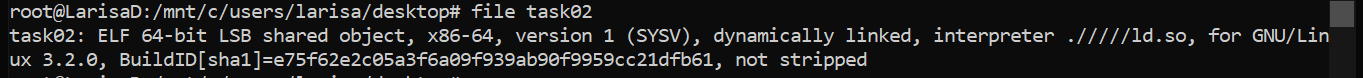
**Task 3.2.2: complex PIE (4p)**

* *Identify the binary protections. (1p)*

The binary protection:



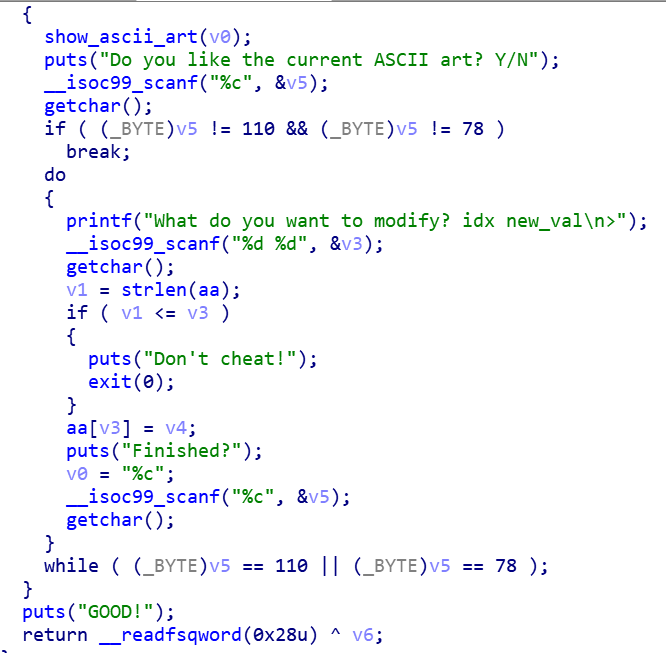
* *Is the binary stripped? What approach is needed for breakpoints?*



As we can see in the photo above, the binary is not stripped, but we saw at the previous point that PIE is enabled, so breakpoints for relative addresses don’t work. However, breakpoints for known symbols (e.g. *main*) work.

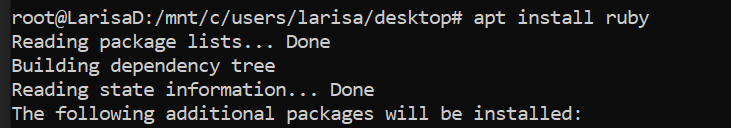
* *Analyze the binary in IDA. What is the vulnerability present? What can it be used for? (1p)*

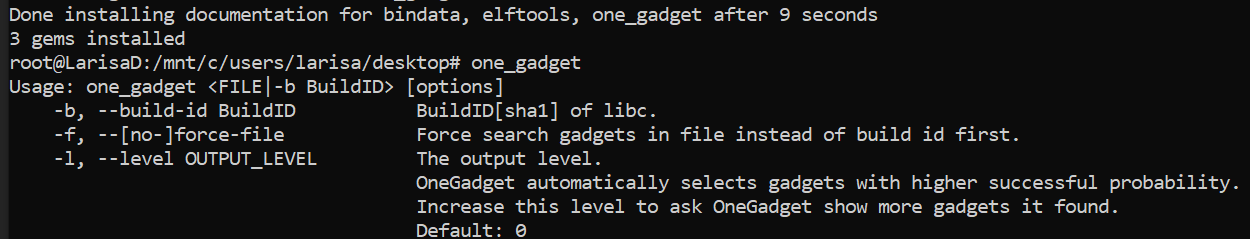
We have buffer overflow, caused by unbounded scanf. We can use this for a ROP attack.



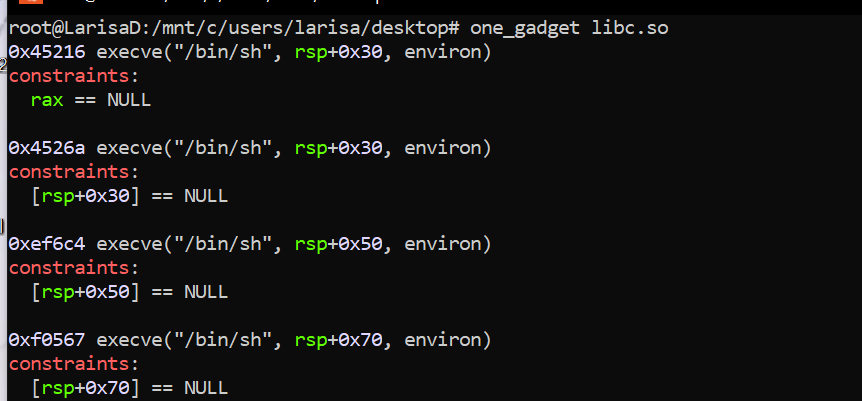
* *Use the one gadget\_tool to find a couple of offsets into libc for shell spawn.*

First, we need to install one\_gadget from [here](https://github.com/david942j/one_gadget). Commands: ***apt install ruby***; (***ruby --version***; just to see that the instalation was successful); ***gem install one\_gadget***





This is what we get after running the command: ***one\_gadget libc.so***

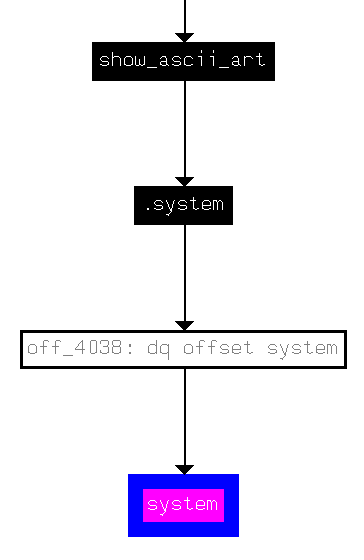
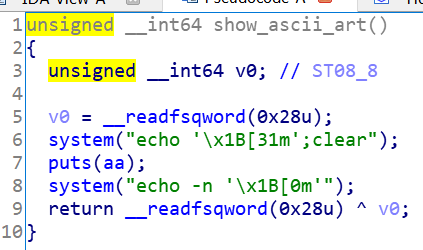


* *Scan the GOT table to see which of those addresses differs the least (less bytes to overwrite => less failed tries) and calculate the probability here as well (use the same approach as in task 3.2.1). (1p)*

By looking in IDA:



As we can see, *system* is at address ***0x4038***. And is being used only in function *show\_ascii\_art*:



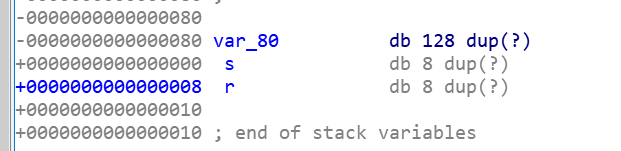
**3.3 ROP tricks (8p)**

*For this section, use the binaries in the* ***06-lab-files.zip*** *file.*

*The binaries have a trivial vulnerability as in the previous section. However, this time, the end game is not to just print “Task X solved” but to obtain code execution. We achieve this by calling system(”/bin/sh”). To this end, you will need to construct increasingly difficult ROP chains.*

**Task 3.3.1: first ROP (3p)**

* *Find the offset until the return address.*

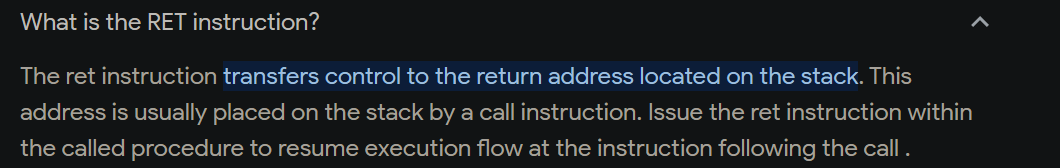


(In order to get this picture, right click on var\_80 and then click Array)

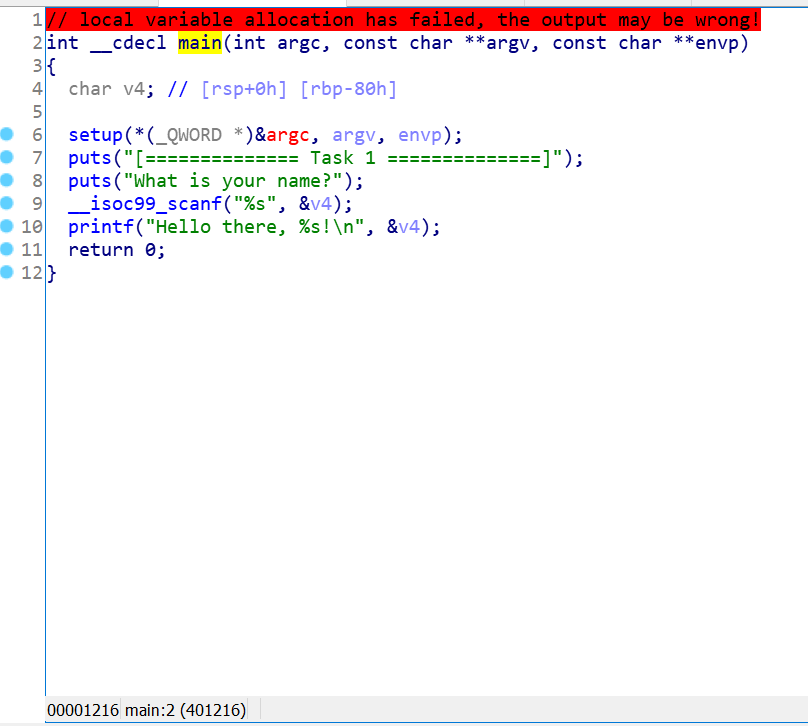
The offset of the return address = 128 bytes (for *var\_80*) + 8 bytes (for *s*) = ***136 bytes***

* *Find any ret instruction and construct a return sled. Step through it using gdb.*

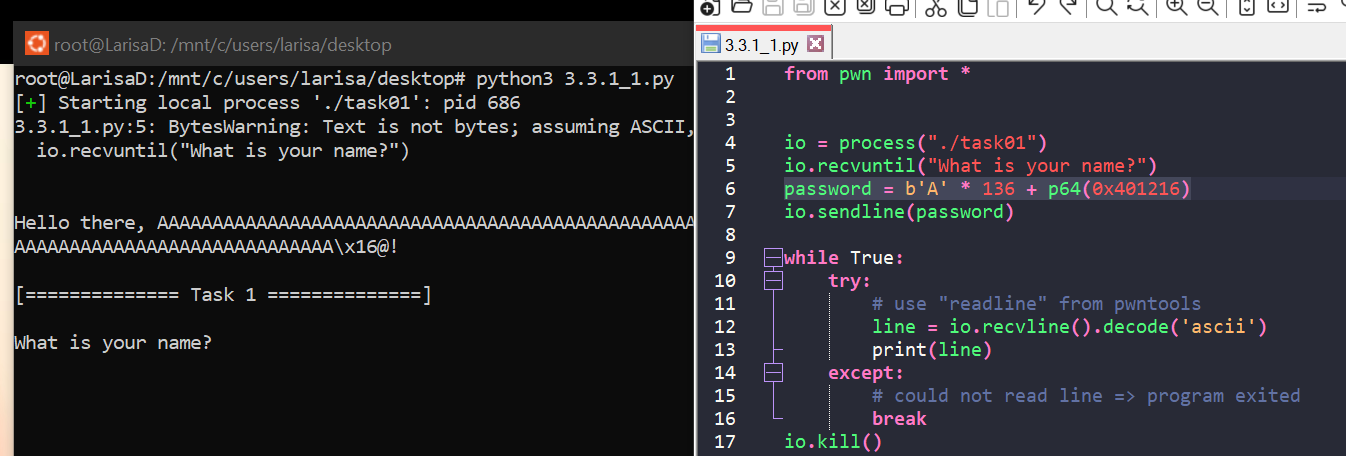
Just to remember:



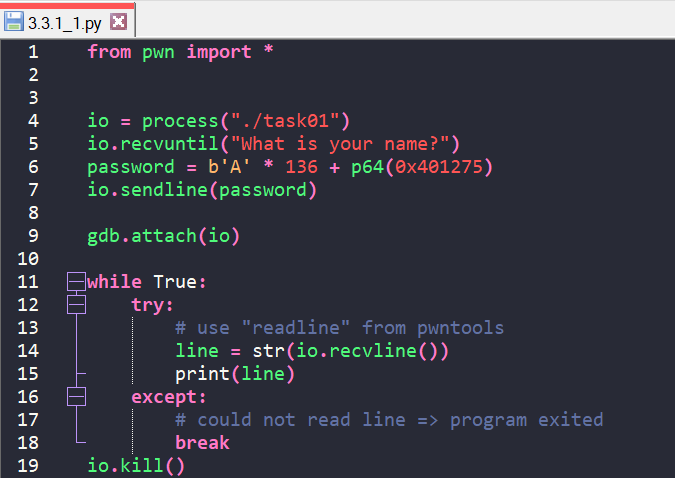
We will be using the ***main()*** function:



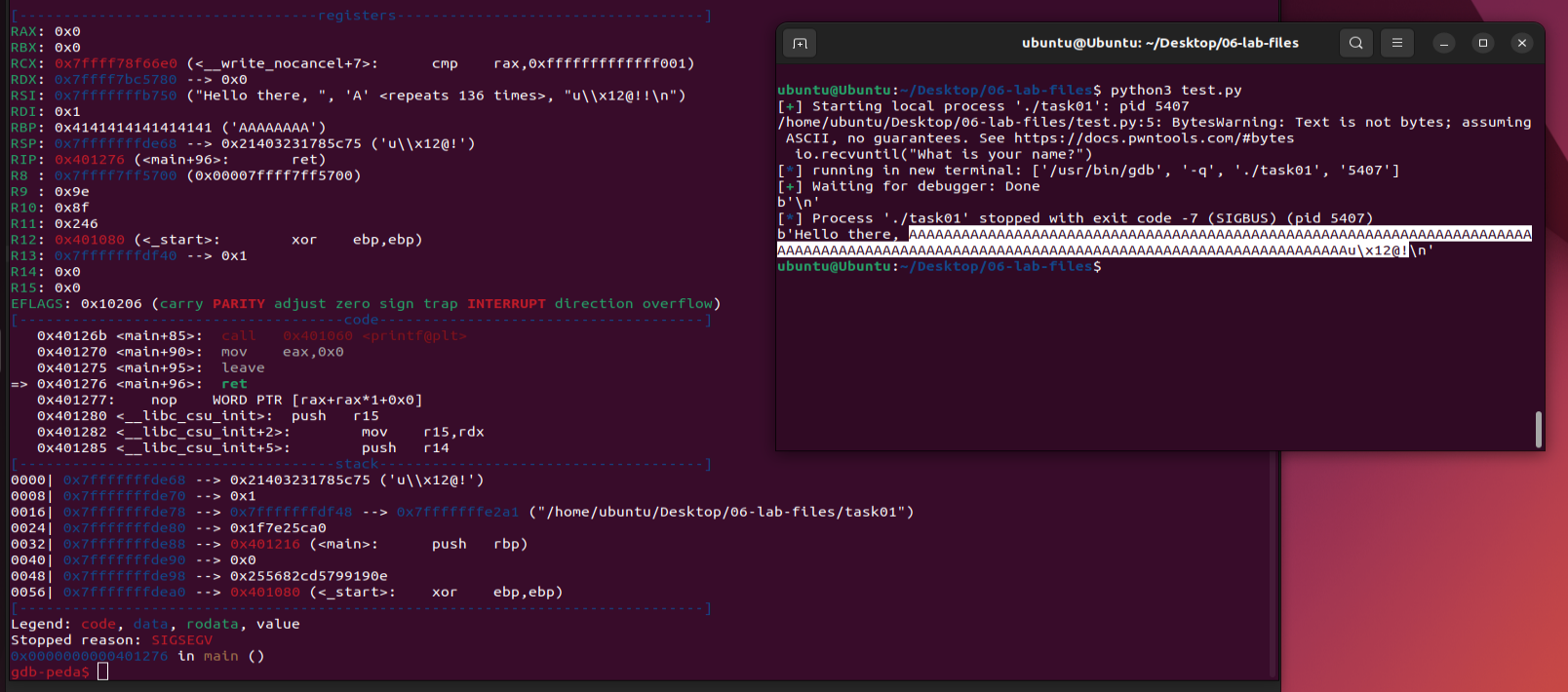
Knowing the informations from above, we need:



It’s working. So let’s access the gdb now. Here is the code that will help us (*3.3.1\_1.py*):

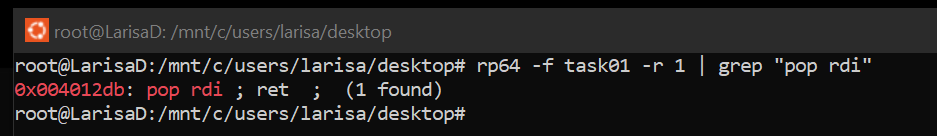


And, after typing ***run*** and the input (copy-paste) in *gdb-peda*:



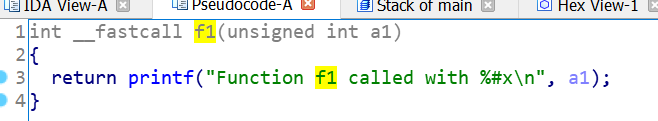
* *Using rp++ find a pop rdi; ret gadget.*

First, we need to install rp++ from [here](https://github.com/0vercl0k/rp). ([source](https://blog.int80.kr/94); don’t delete the files)



* *Call function f1 with the parameter 0xdeadbeef. (1p)*

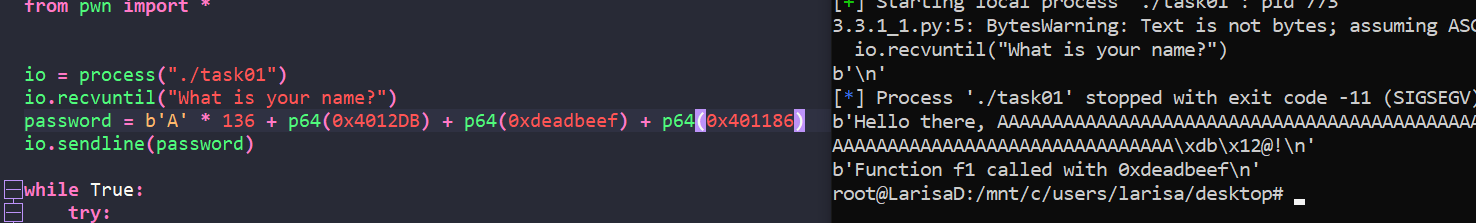
The *f1()* function:



The address:

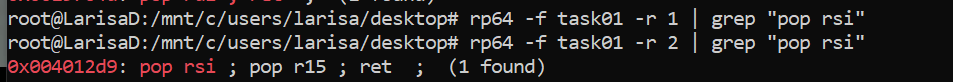


We need the following payload = offset + address of „*pop rdi; ret*” + parameter + address of f1() = b'A' \* 136 + p64(0x4012DB) + p64(0xdeadbeef) + p64(0x401186)



Code in: *3.3.1\_2.py*

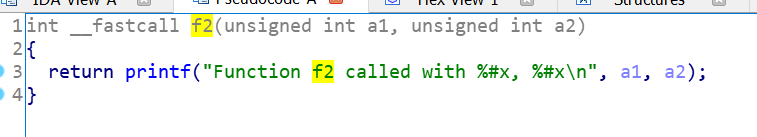
* *Using rp++ find a pop rsi; ret gadget.*



* *Call function f1 with the parameter 0xdeadbeef and f2 with the parameters 0x1234, 0xabcd. (1p)*

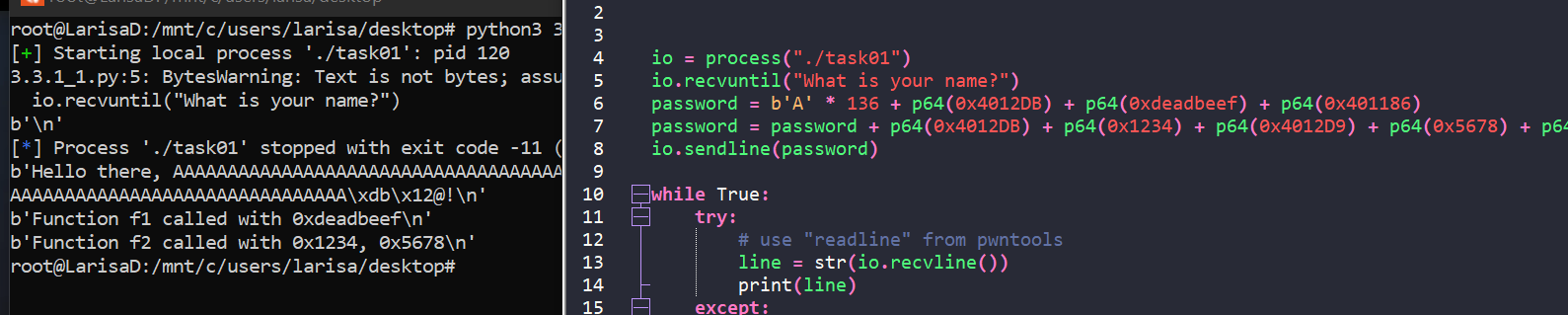
We’ve already discussed previously the payload for f1(): ***b'A' \* 136 + p64(0x4012DB) + p64(0xdeadbeef) + p64(0x401186)***

For function *f2()*:





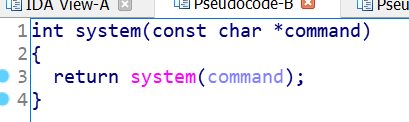
The payload for f2 = offset + address of „*pop rdi; ret*” + first parameter + address of „*pop rsi; pop r15; ret*” + second parameter + garbage value + address of f2() = ***offset + p64(0x4012DB) + p64(0x1234) + p64(0x4012D9) + p64(0x5678) + p64(0xffffff) + p64(0x4011AA)***



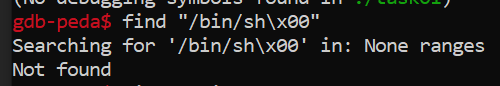
Code file: *3.3.1\_2.py*

* *Using IDA find the address of* ***system*** *in the binary. Using gdb find the address of the string “****/bin/sh\x00****” in the binary. Note that not all payloads work. If you have a whitespace character such as „****\n****” or „”, the scanf function terminates. Choose addresses according to these constraints.*

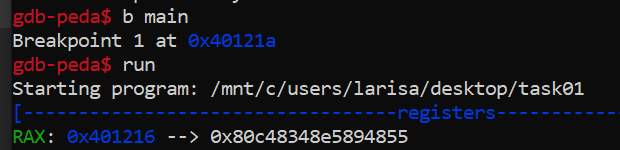
Function ***system*** with the address:

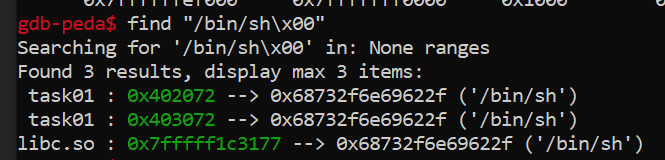
 

We use gdb in order to find the address of the string „/*bin/sh\x00*” ([source](https://stackoverflow.com/questions/6637448/how-to-find-the-address-of-a-string-in-memory-using-gdb)):



Because of this, we try an alternative way: first, we put a breakpoint in order to run the process, otherwise we get the error „*No current process: you must name one.*”:



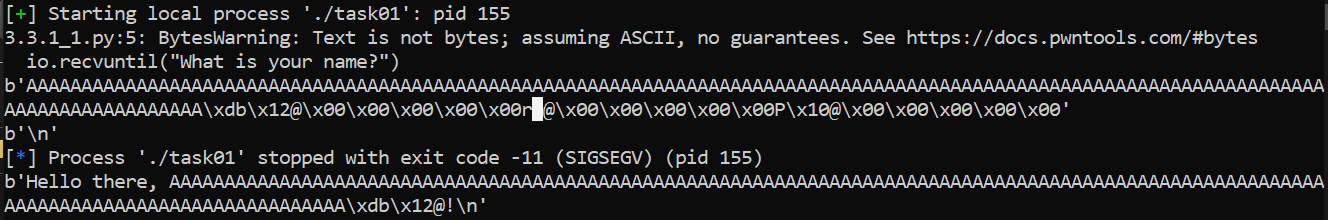


As we can notice, more than one address was found for the searched string.

* *Construct a ROP chain that loads the address of “****/bin/sh****” as the first argument and calls* ***system****.*

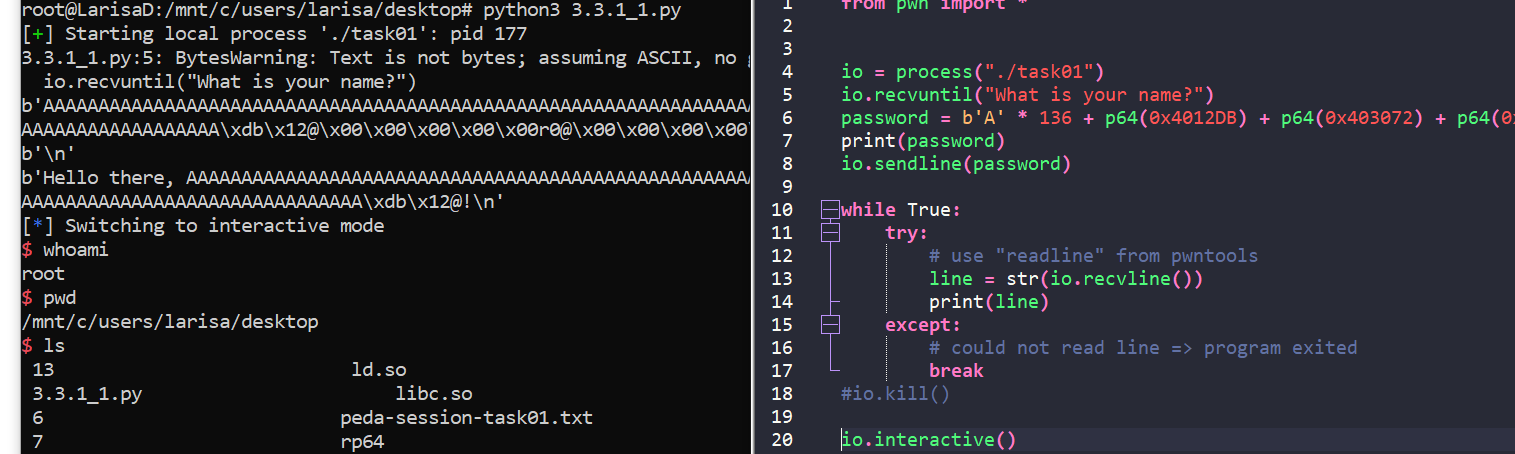
We will attempt the addresses, in order, until we find one that works. Our payload = offset + address of „*pop rdi; ret*” + address of „*/bin/sh*” + address of *system* = ***b'A' \* 136 + p64(0x4012DB) + address of „/bin/sh” + p64(0x401050)***

Let’s try with address *0x402072*:



It’s not working because we have a whitespace (the observation from above).

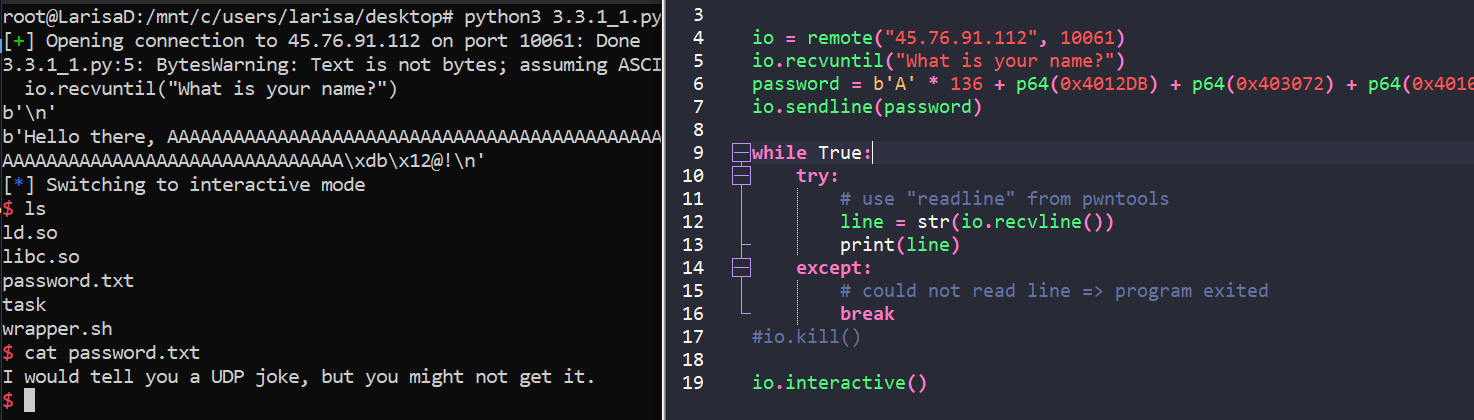
Let’s try with address *0x403072*:



We can notice, it is working with this address.

Code file: *3.3.1\_4.py*

* *Exploit the service running at 45.76.91.112 10061 and read password.txt. (1p)*

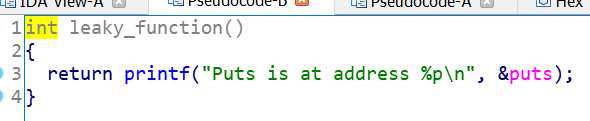


Code file: *3.3.1\_5.py*

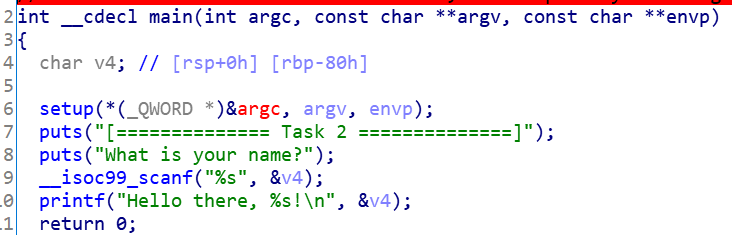
**Task 3.3.2: multi-step ROP (3p)**

* *In this task,* ***system*** *is no longer called. However, it is possible to recover its address using a helper function.*
* *Call the* ***leaky\_function*** *and then* ***main*** *again. Using the address leak, calculate the base of libc. (1p)*

The function *leaky\_function()*:

The function *main()*:

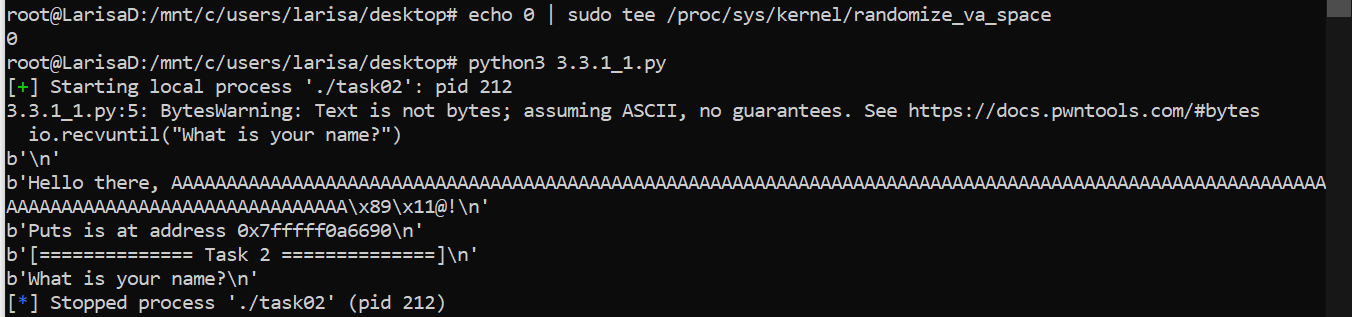
 

The offset will be ***136 bytes***:

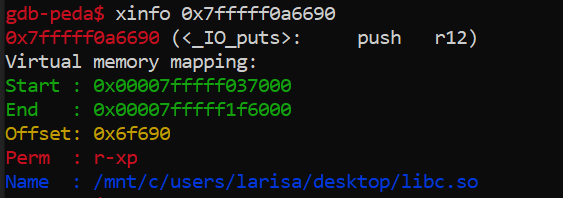


Knowing this we can calculate the payload = offset + address of *leaky\_function()* + address of *main()* = ***b'A' \* 136 + p64(0x401189) + p64(0x4011DA)***

We run: ***echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space*** Code file: *3.3.2\_1.py*



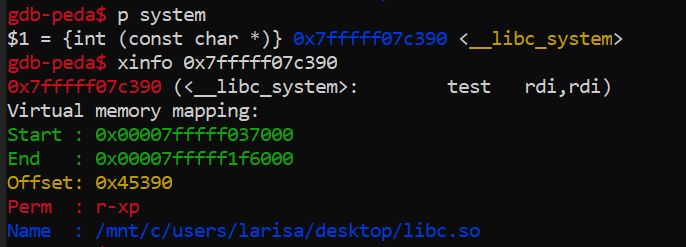
We can see from this photo that *puts* is at address ***0x7fffff0a6690***. With this information, we open gdb, set a breakpoint in main (*b main*), hit *run*, and then request information about this address (*xinfo 0x7fffff0a6690*).



So the offset of the puts is ***0x6f690***. And we can calculate the address of libc = address of puts - 0x6f690 = 0x7FFFFF037000 (address of ***libc***)

* *Turn the exploit into a full Remote Code Execution exploit. Use the service running at 45.76.91.112 10062 and read password.txt. (2p)*

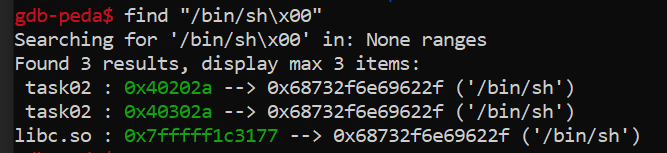
We need to locate *system*.



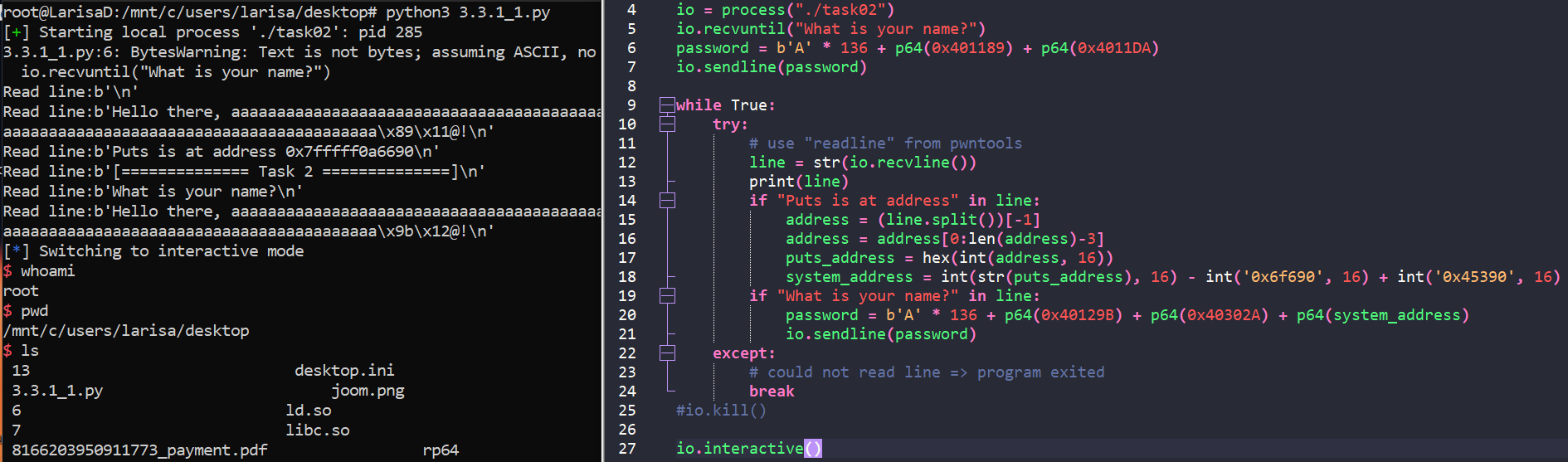
So we can conclude that the address of *system* = address of libc + 0x45390 (offset) = address of puts - 0x6f690 + 0x45390

We need, for our payload, the address of „*pop rdi; ret*” and the address of „*/bin/sh*”:





So, our payload = offset + address of „*pop rdi; ret*” + address of „*/bin/sh*” (0x40302A) + address of *system*



Code file: *3.3.2\_2.py*

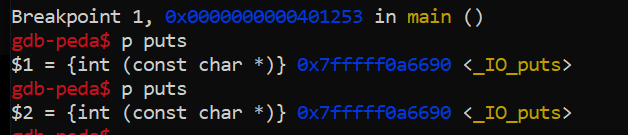
And for the remote service – *3.3.2\_3.py*:



**Task 3.3.3: format a string info leak (2p)**

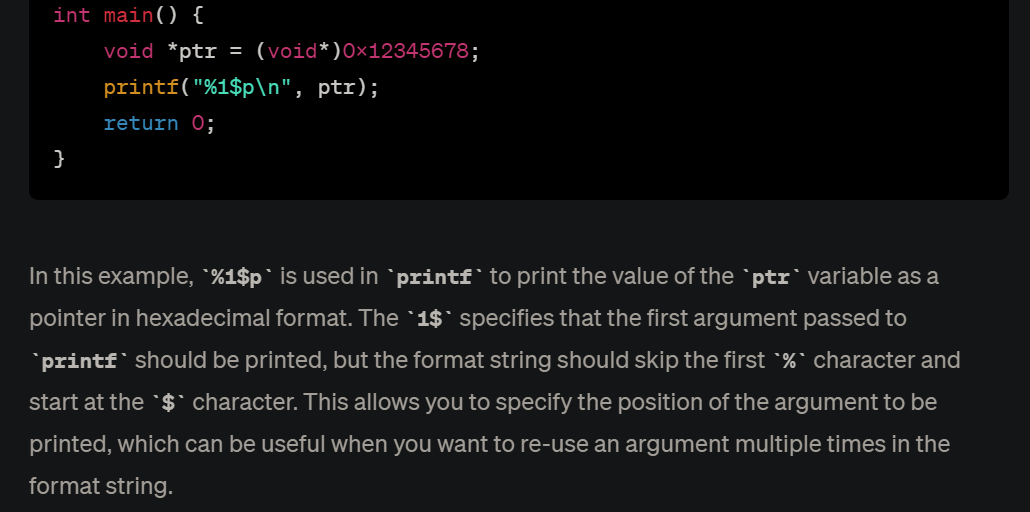
* *Use the input to leak values from the stack (find the* ***puts*** *pointer stored on the stack in* ***main****) and obtain the address of* ***libc****. (1p)*

Let’s start with finding the address of *puts*. We use gdb for this, place a breakpoint in main, hit *run* and then run the command *p puts*:



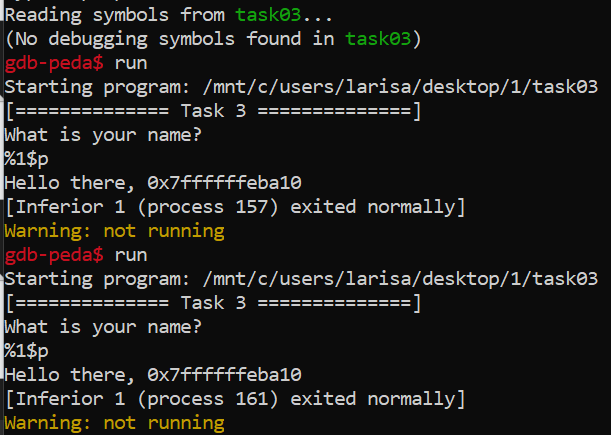
The address is ***0x7fffff0a6690***.

We can exploit the stack using inputs formated as „*%\_no\_$p*” ([source1](https://hackingiscool.pl/stack-canary-rop-format-string-leak-also-how-i-learned-nullbyte-is-not-a-badchar-to-scanf-string/), [source2](https://hackingiscool.pl/heap-overflow-with-stack-pivoting-format-string-leaking-first-stage-rop-ing-to-shellcode-after-making-it-executable-on-the-heap-on-a-statically-linked-binary-mbe-lab7a/)):

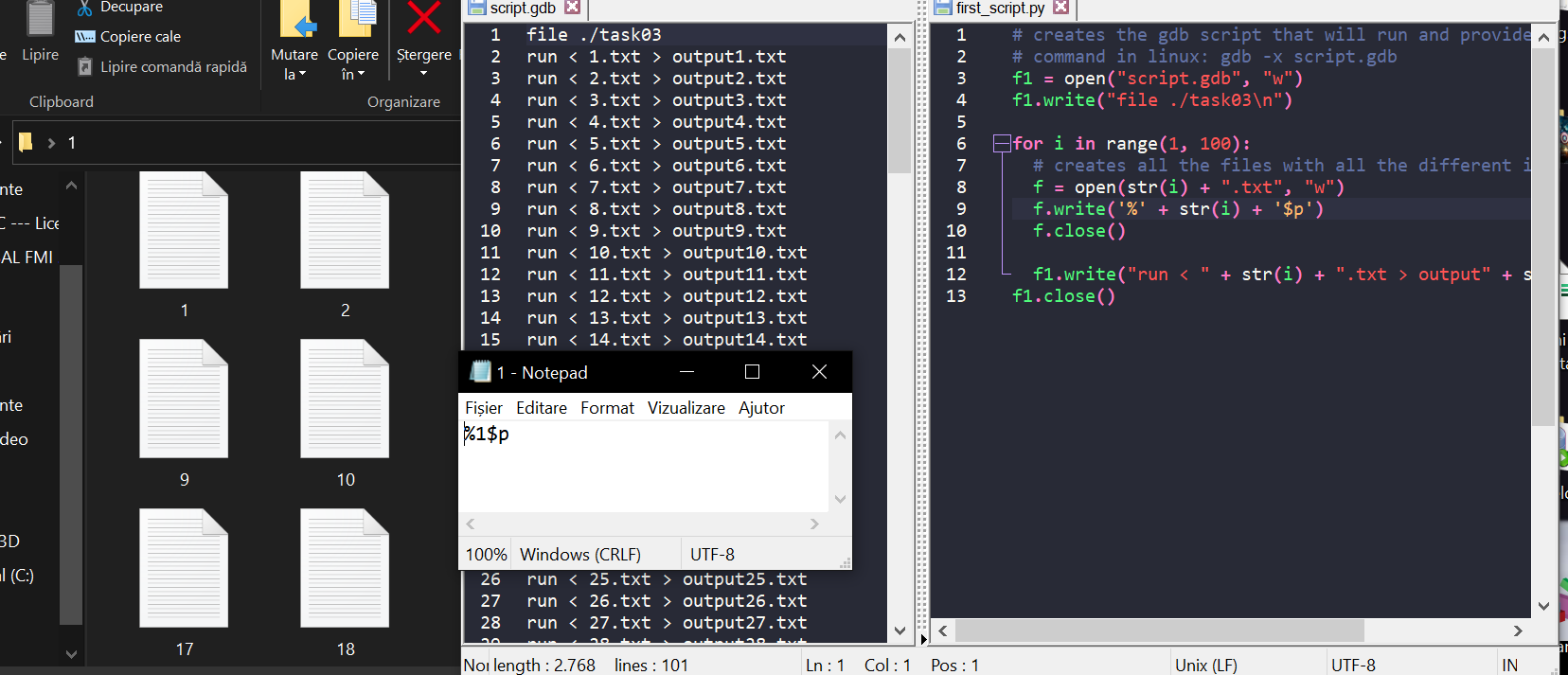




As we can notice, the address changes. Let’s try inside gdb and see the behaviour:

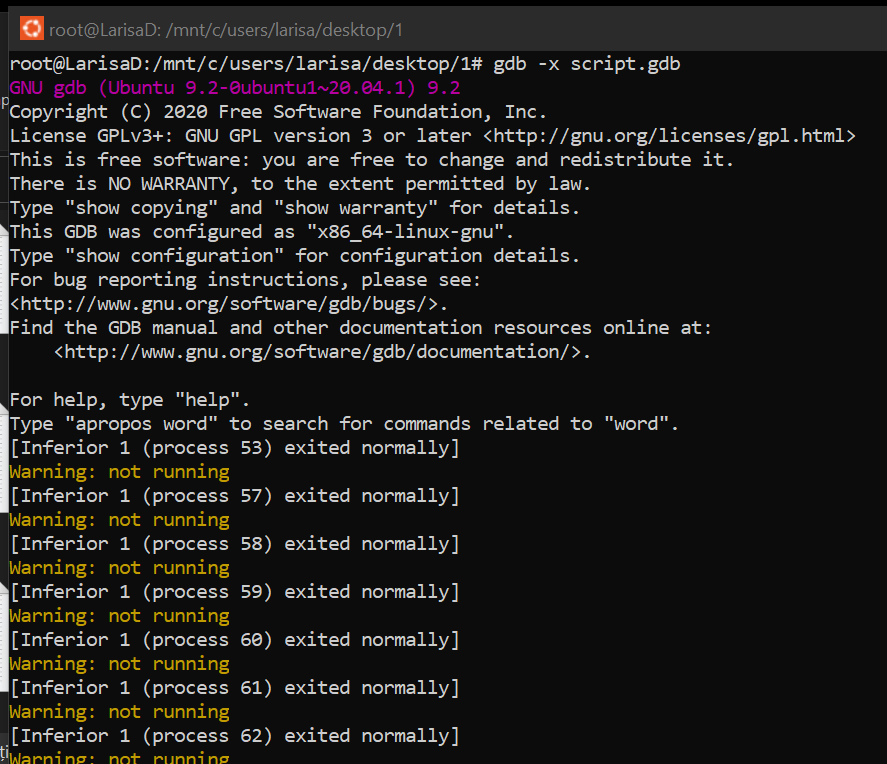
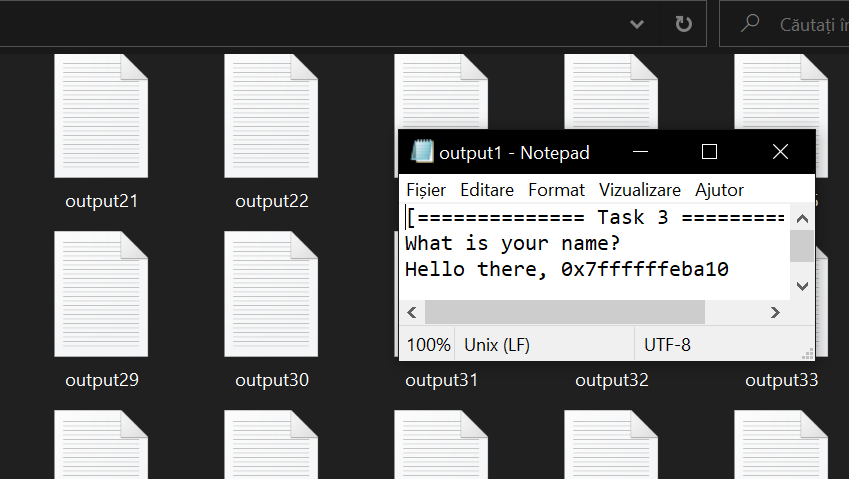


We can exploit this. We know the address of *puts* inside the *main* stack is ***0x7fffff0a6690***. So we need to provide input to gdb until the output returned is the address that we desire. First, we create a python program that creates the script and the files of input (each input file will be named *%no%.txt* and will contain one line „*%\_no\_$p*”):

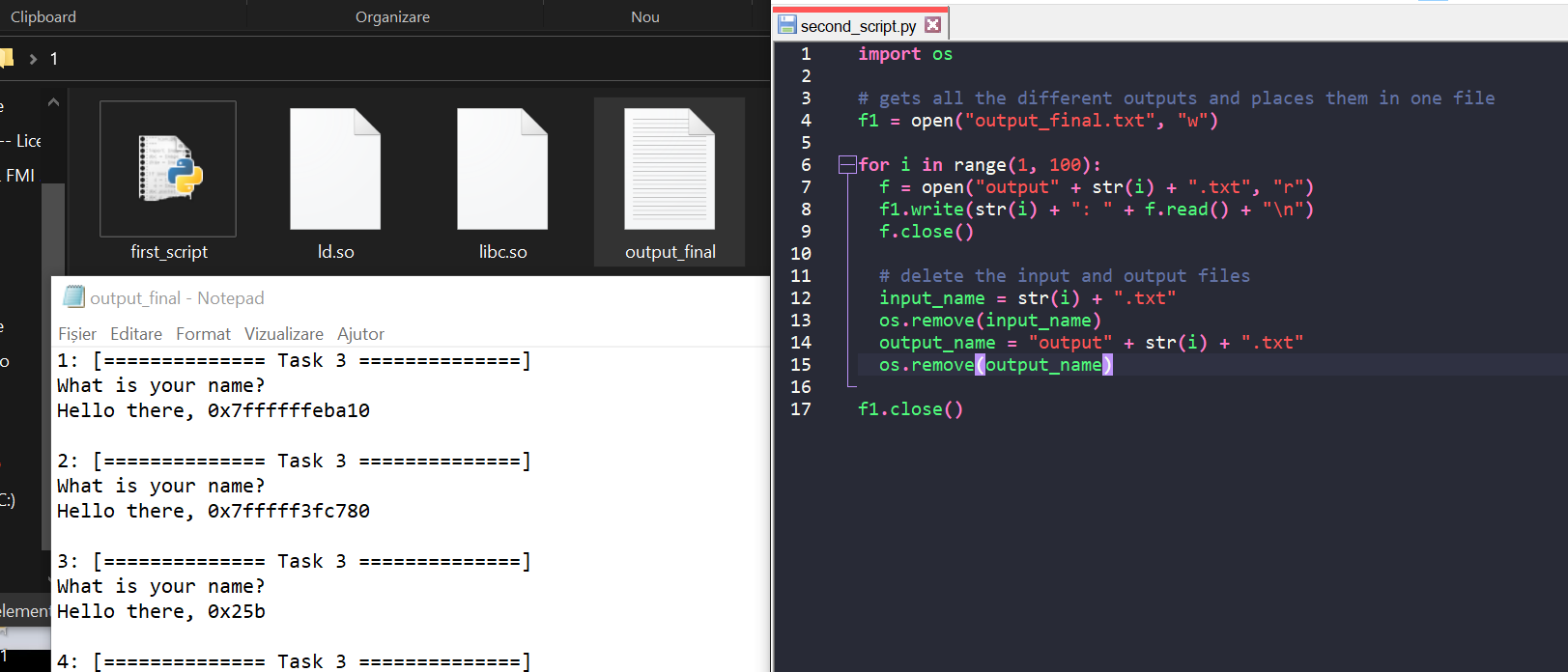


Code is in *first\_script.py* and *script.gdb* (we run for the first 100 numbers; the 100 input files were not provided).

In the terminal we run: *gdb -x script.gdb*

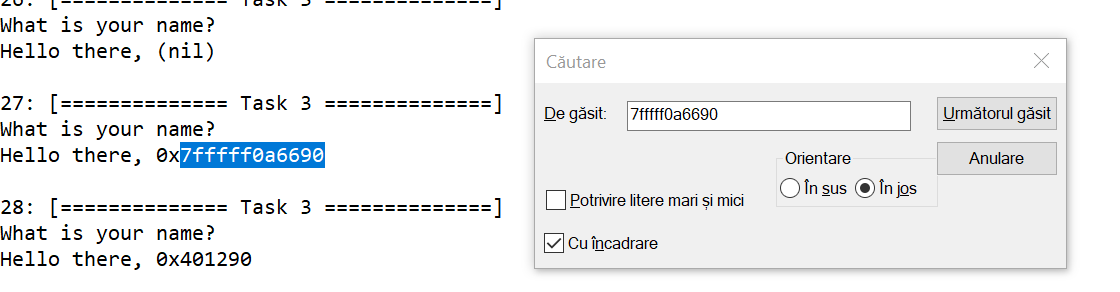
 

Now, gdb runs the commands and creates 100 files titled *output%no%.txt* that contain the addresses of every input. As we can see from the picture above, *output1.txt* contains exactly the output from above, when we manually entered the value „*%1$p*”. We run a second script that will take all the outputs, will place them inside one file and then we will search for the address of *puts* inside it, referencing the index (we will also delete the input and output files).



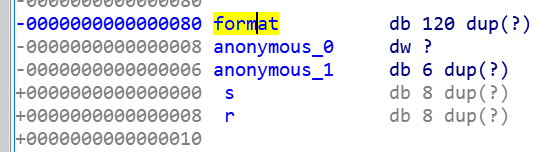
Files: *second\_script.py*, *output\_final.txt*

We find the index of *puts* – 27:



We know from the previous exercise, that the **address of *libc* = address of *puts* - 0x6f690 = 0x7FFFFF037000** (address of ***libc***) and that the **address of *system* = address of *libc* + 0x45390 (offset) = address of *puts* - 0x6f690 + 0x45390** (same method)

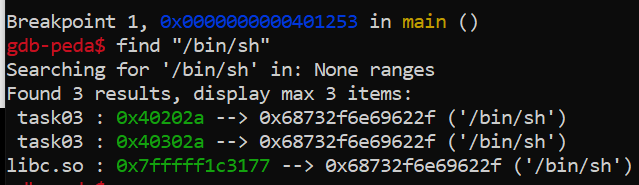
Similar to our previous payloads, we will be using payload = malicious input + p64(0) + rest of offset + address of main. The purpose of payload will be to place the index found above at the puts address and call main again. The address of main is ***0x40124F***.

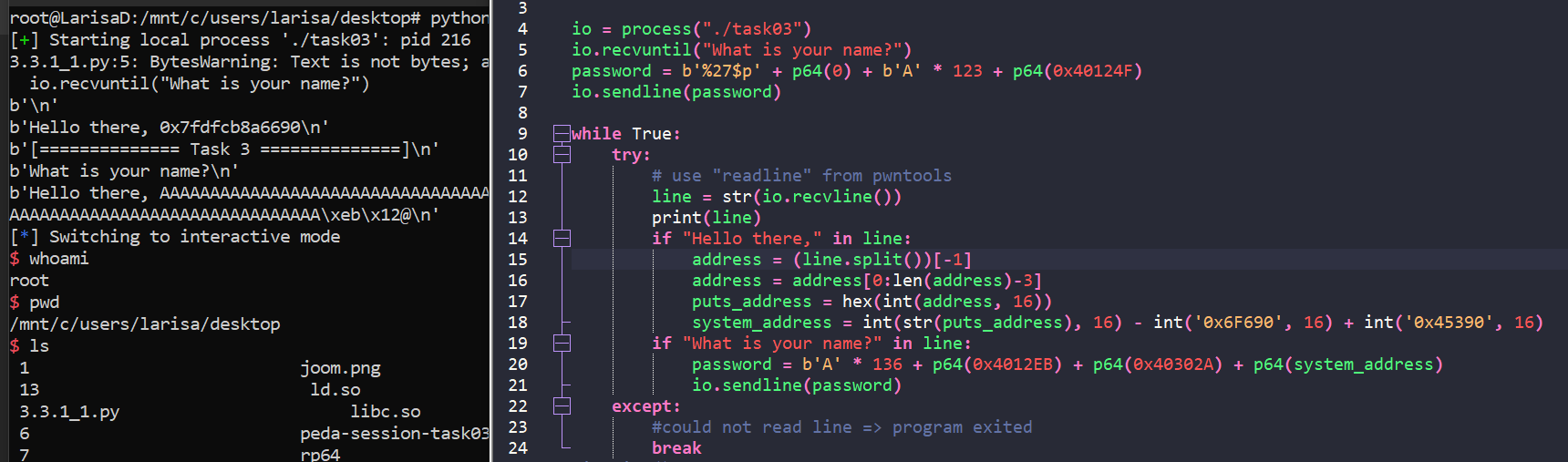
In conclusion, first ***payload = b'%27$p' + p64(0) + b'A' \* 123 + p64(0x40124F)***

For our second payload, we need payload = offset + address of „*pop rdi; ret*” + address of „*/bin/sh*” + address of *system*





So, second payload = ***b'A' \* 136 + p64(0x4012EB) + p64(0x40302A) + 0x7FFFFF07C390 (***address of *system****)***



Code: *3.3.3\_1.py*

* *Turn the exploit into a full Remote Code Execution exploit. Use the service running at 45.76.91.112 10063 and read password.txt. (1p)*

Code: *3.3.3\_2.py*

