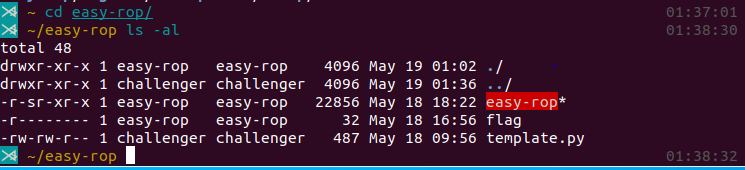
**Lab3 Report**

2018312567 조명하

**1. Easy-rop**

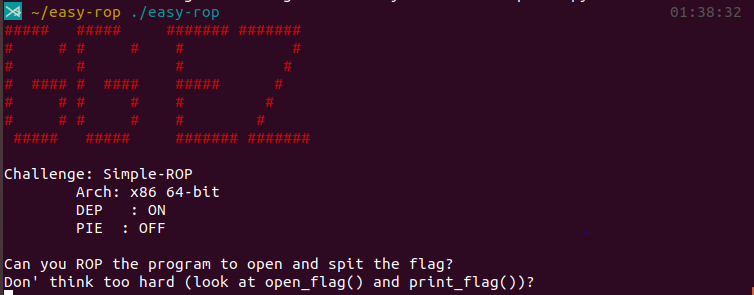
**1) Check folder structure and permission**



Flag is only read by easy-rop

I can execute easy-rop and read/write template.py

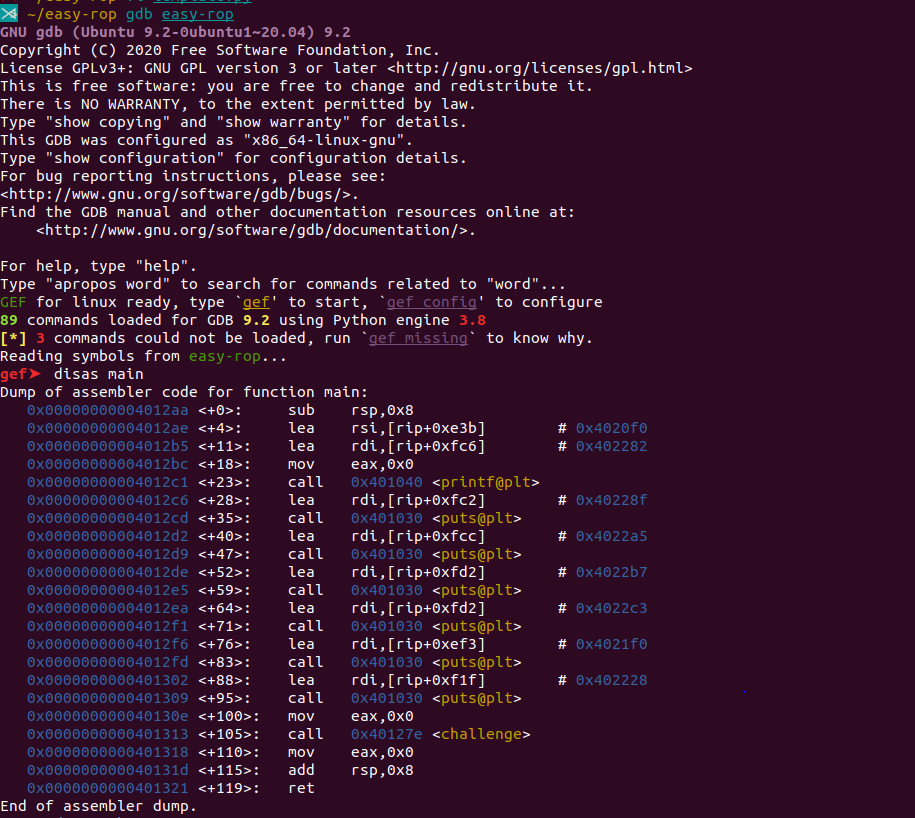
**2) Execute easy-rop**



It seems that I should go to open\_flag and then go to print\_flag

**3) Debug easy-rop**

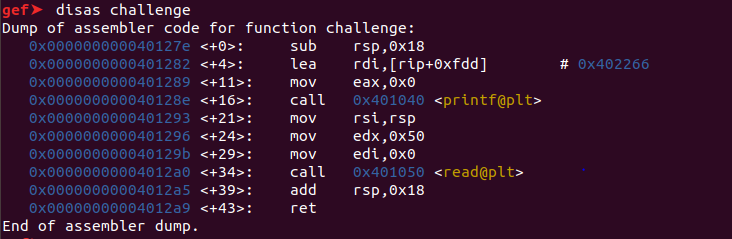
*- disassemble main*



Disassemble main

Return address of challenge() is 0x401318

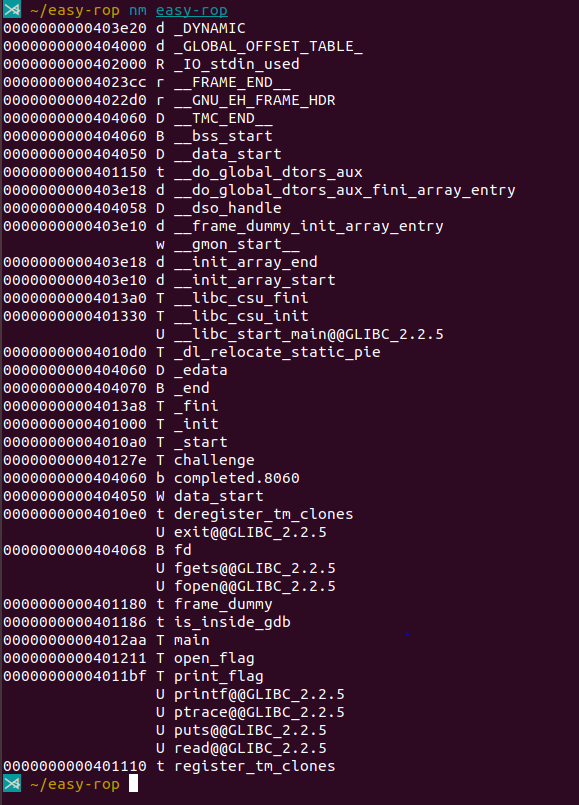
*- disassemble challenge*



Read() reads data from where rsp points to rsp+0x50.

However, because challenge() has 0x18 bytes, BOF occurs.

*- Find print\_flag, and open\_flag*



Open\_flag is in 0x401211

Print\_flag is in 0x4011bf

*- disassemble open\_flag*



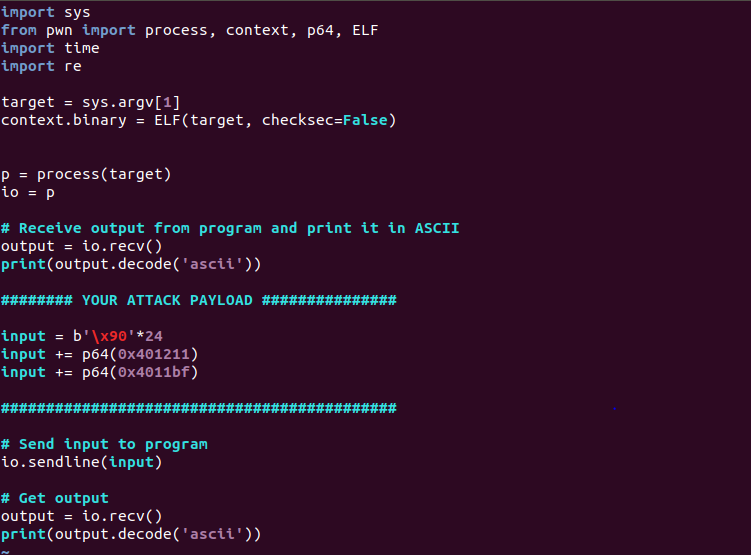
It seems that it takes no arguments

*-disassemble print\_flag*



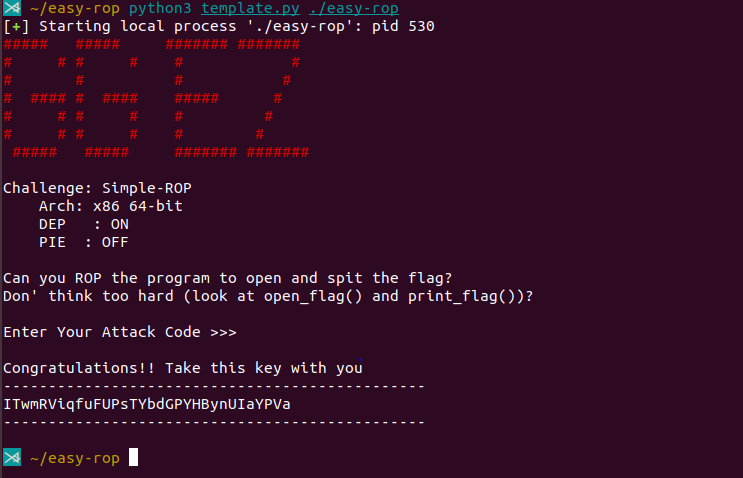
It seems that it takes no arguments

**4) Write exploit code**



0x18 is 24 bytes, so fill the buffer using 24bytes with nop, and then push open\_flag’s address and print\_flag’s address

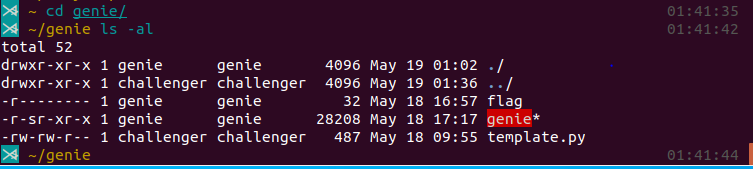
**5) Execute exploit code**



Found the flag

**2. Genie**

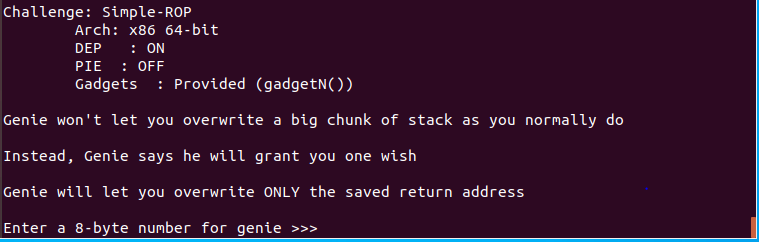
**1) Check folder structure and permission**



Flag is only read by genie

I can only execute genie and read/write template.py

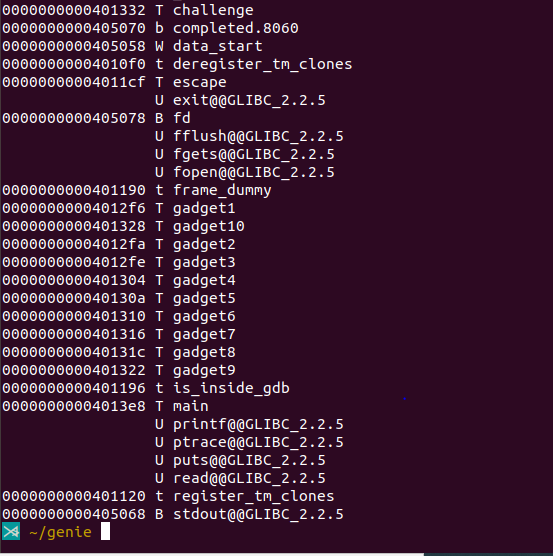
**2) Execute genie**



Genie says that it will let me overwrite only the saved return address

**3) Check symbols**

Enter $nm genie



Challenge is in 0x401332

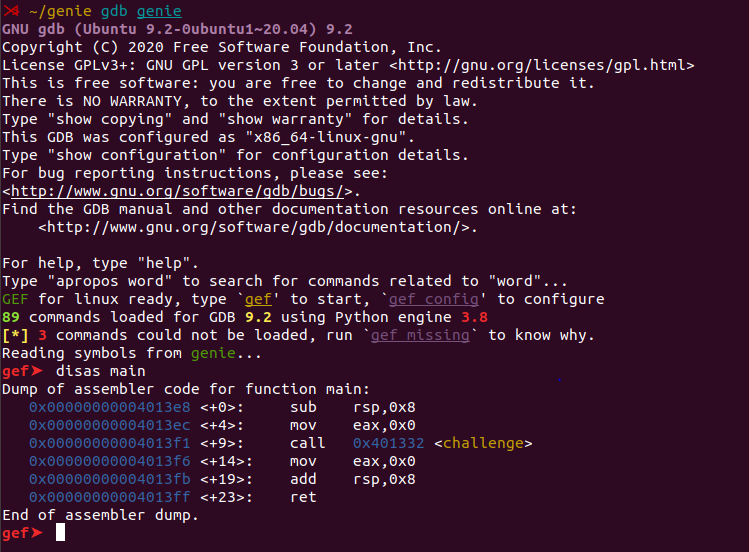
Escape is in 0x4011cf

Gadgets 1~10 are in 0x4012f6~0x401328

Main is in 0x4013e8

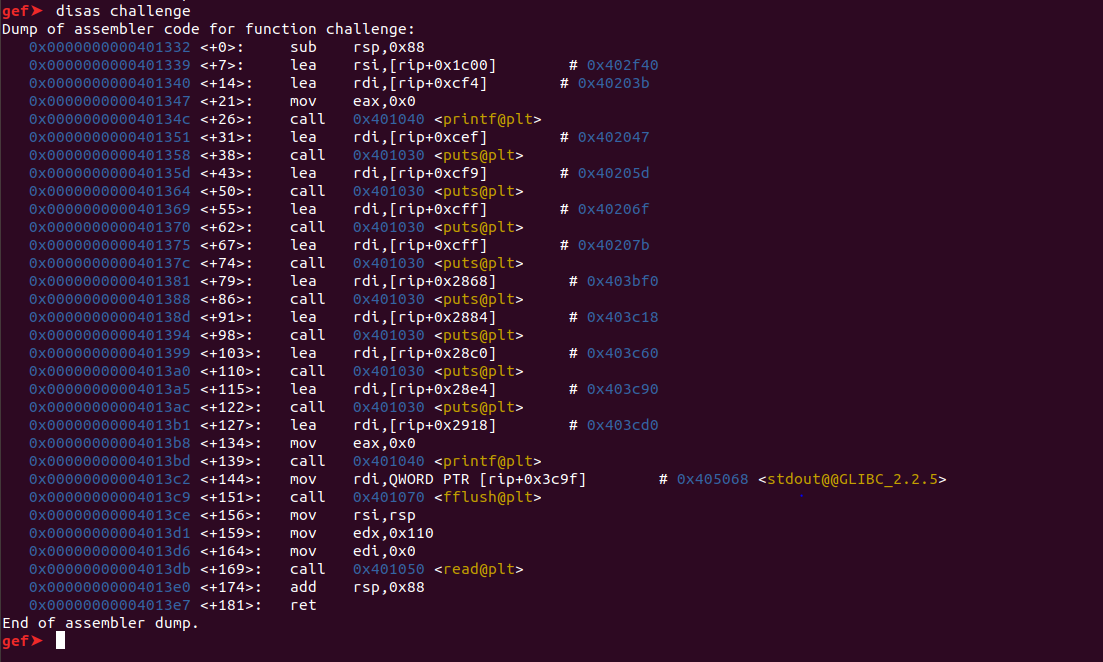
**4) Debug Genie**

*- disassemble main*



So challenge()’s saved return address is 0x4013f6

*-disassemble challenge*

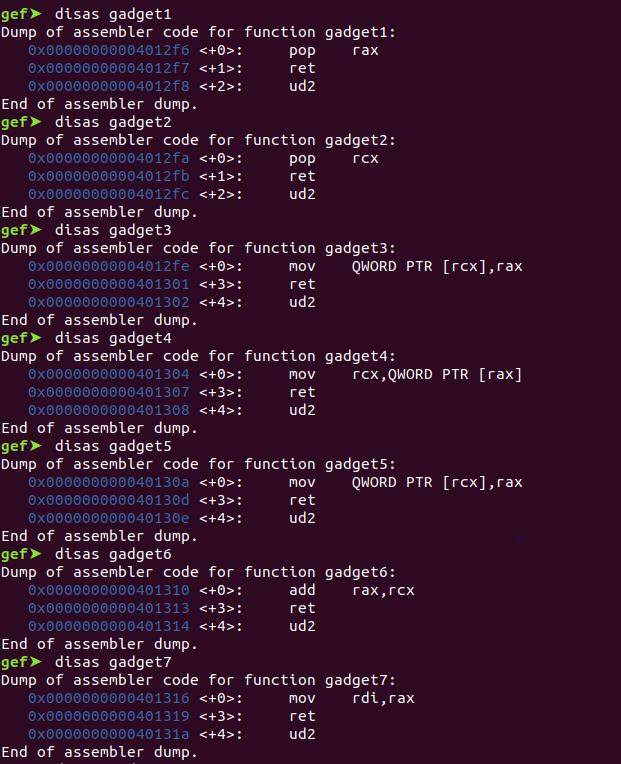


It’s stack frame is 0x88 bytes, and read() reads data and fills 0x110 bytes data from the top of the stack.

But genie says that it would allow me to fill only 0x88+0x8 bytes.

So I should fill the last 8 bytes into the address of my attack code

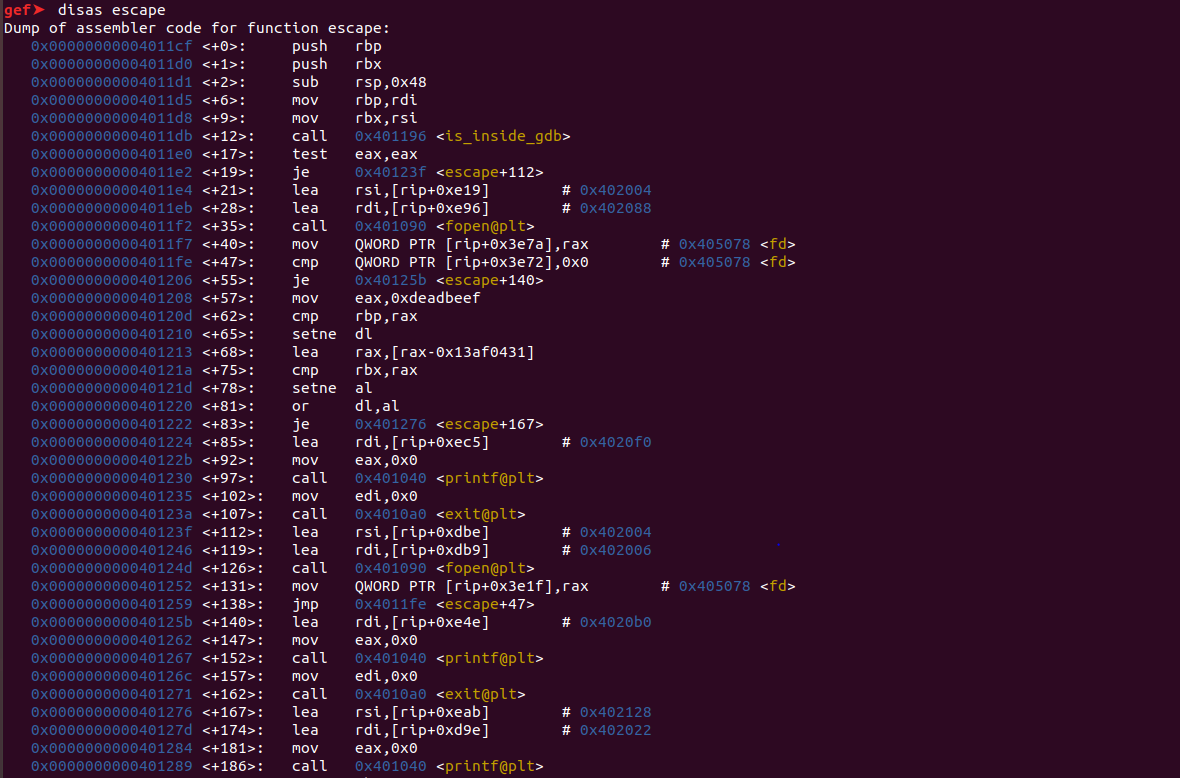
*-disassemble gadgets*

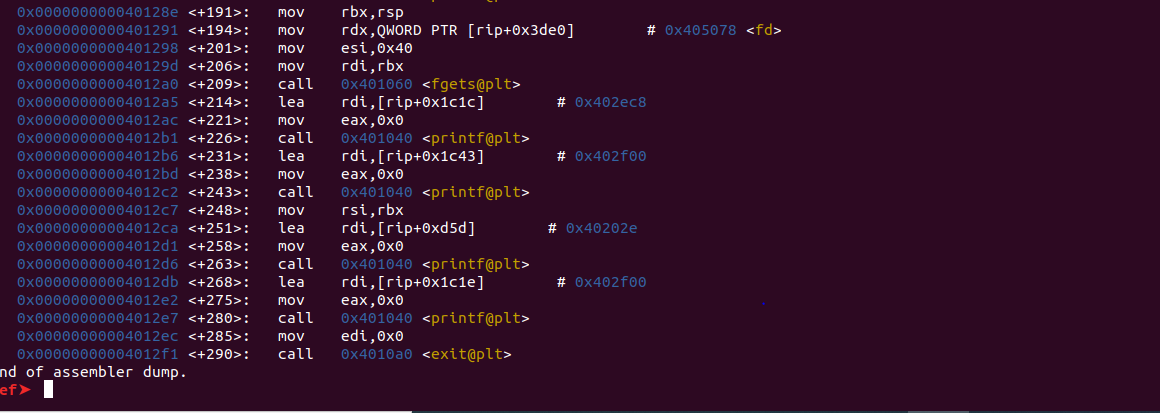




Gadget 10 changes rsp’s value

*-disassemble escape*





It saves rdi and rsi’s value into rbp, rbx. So it takes 2 arguments.

After it calls is\_inside\_gdb(), it compares that rbp and 0xdeadbeef, rbx and [0xdeadbeef – 0x13af0431] (=[0xcafebabe])

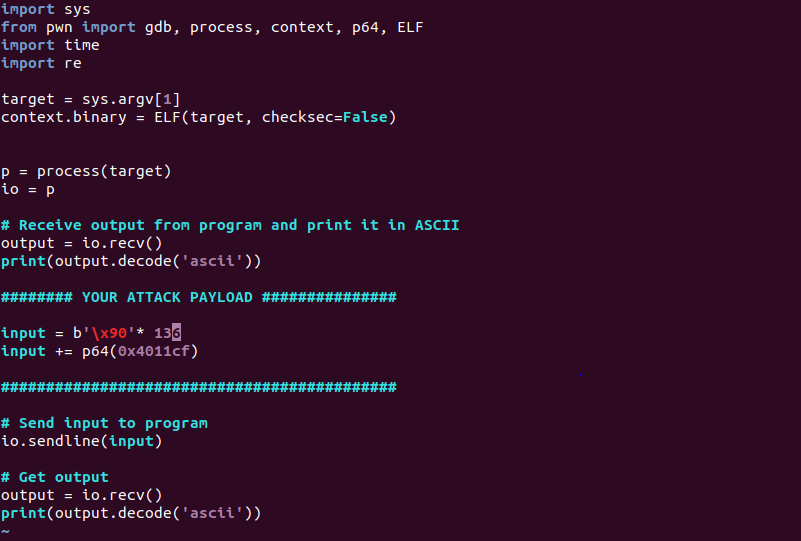
So, before call escape(), rdi should take 0xdeadbeef and rsi should take [0xcafebabe]

Attack code should be the order of **&gadget1, 0xcafebabe, &gadget9, &gadget8, &gadget1, 0xdeadbeef, &gadget7, &escape.**

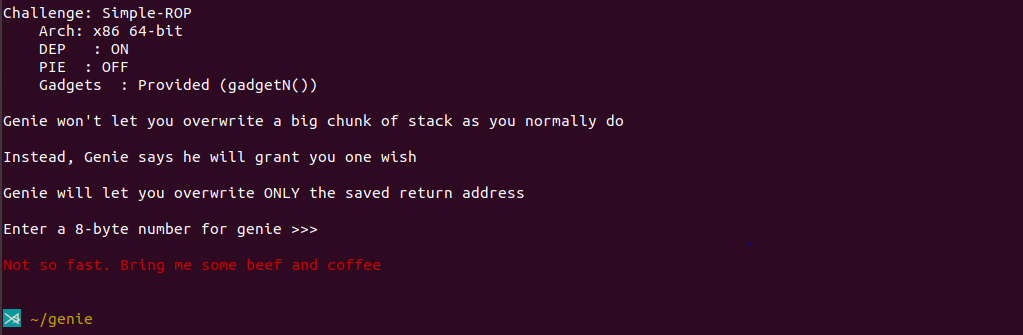
And then overwrite saved return address with the address of where the above attack code is saved

**5) Write exploit code**

*1. return address -> escape*



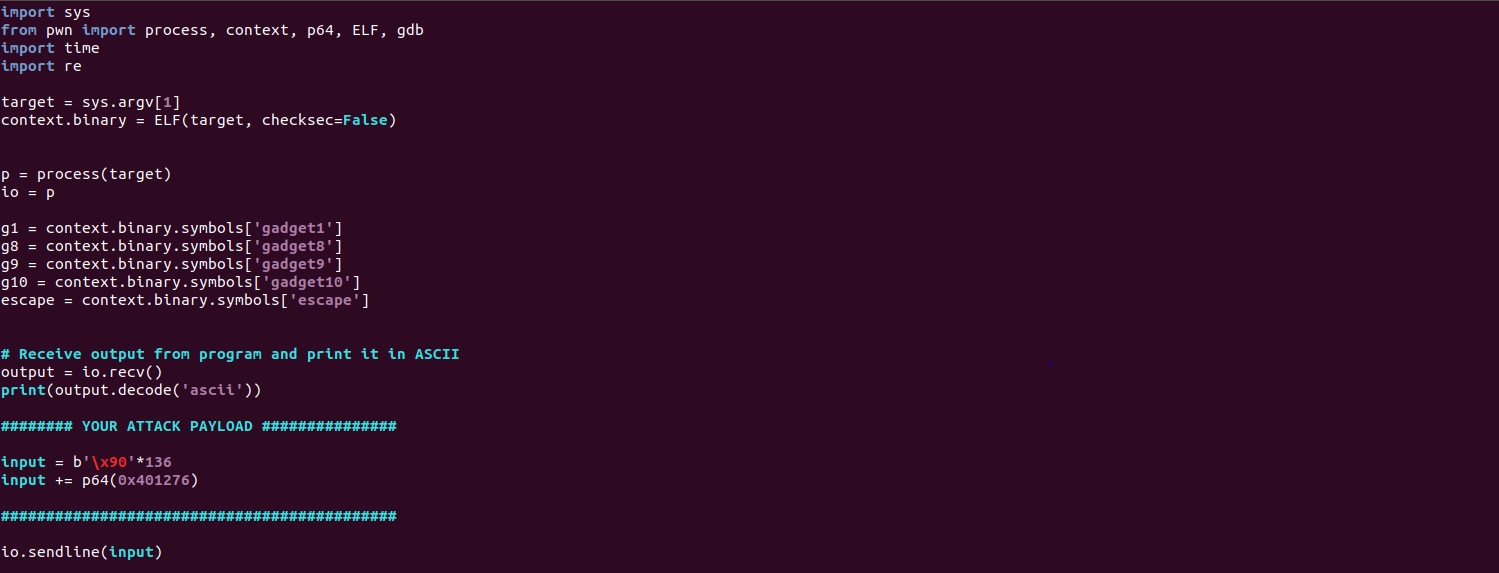
Fills the saved return address into escape()’s address



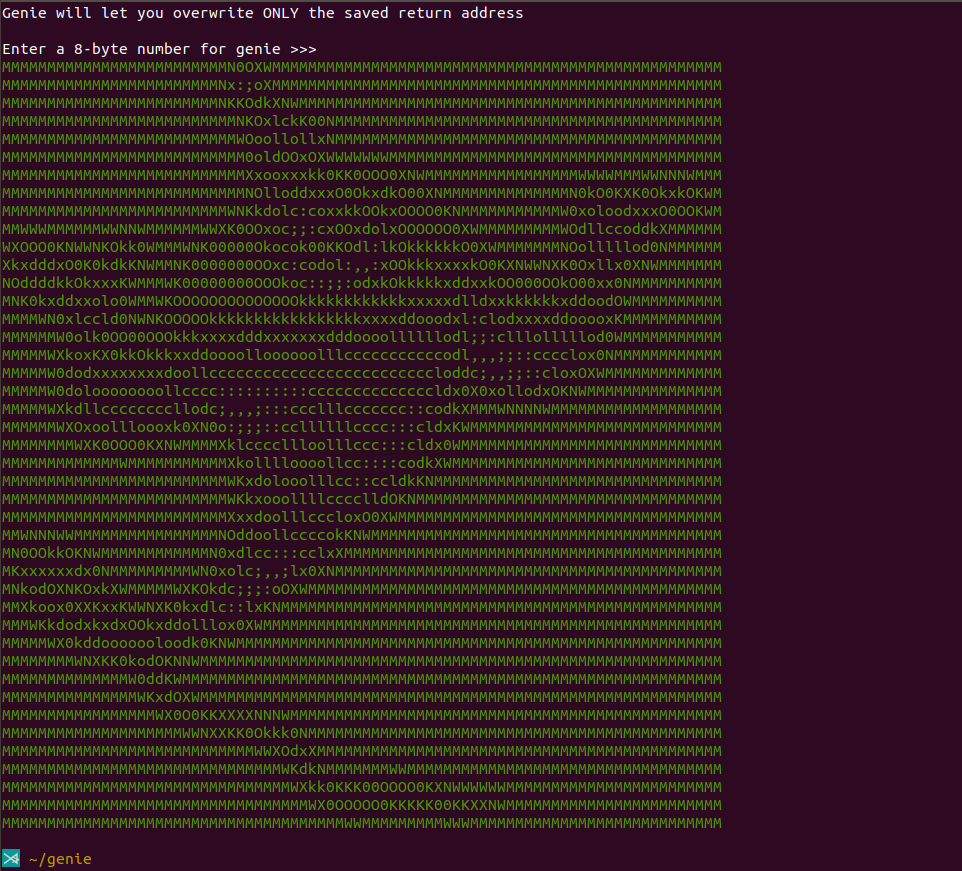
- It says no so fast. “bring me some beef and coffee” means that it takes two arguments and its value should be ‘0xdeadbeef’ and ‘[0xcafebabe]’

*2. return address -> past escape’s branch (0x401276)*

What if just jump the argument checking process?



Just fill the saved return address into after escape() finishes comparing arguments

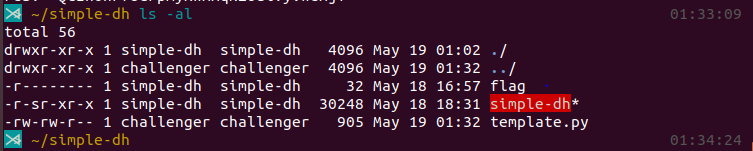


It seems that genie welcomes me, but the flag isn’t shown.

I know I should write my attack code in the memory somewhere I can control, but I don’t know how.

**3. Simple-dh**

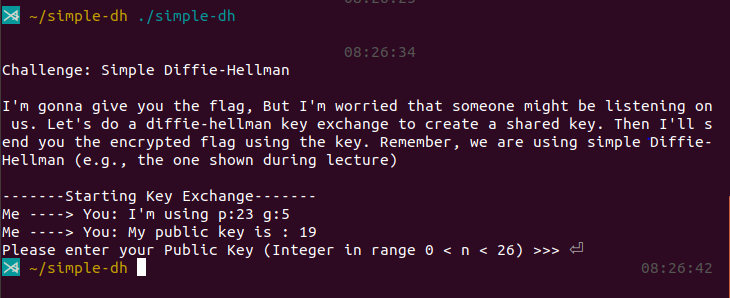
**1) Check folder structure and permissons**



Flag is only read by simple-dh

I can only execute simple-dh and read/write template.py

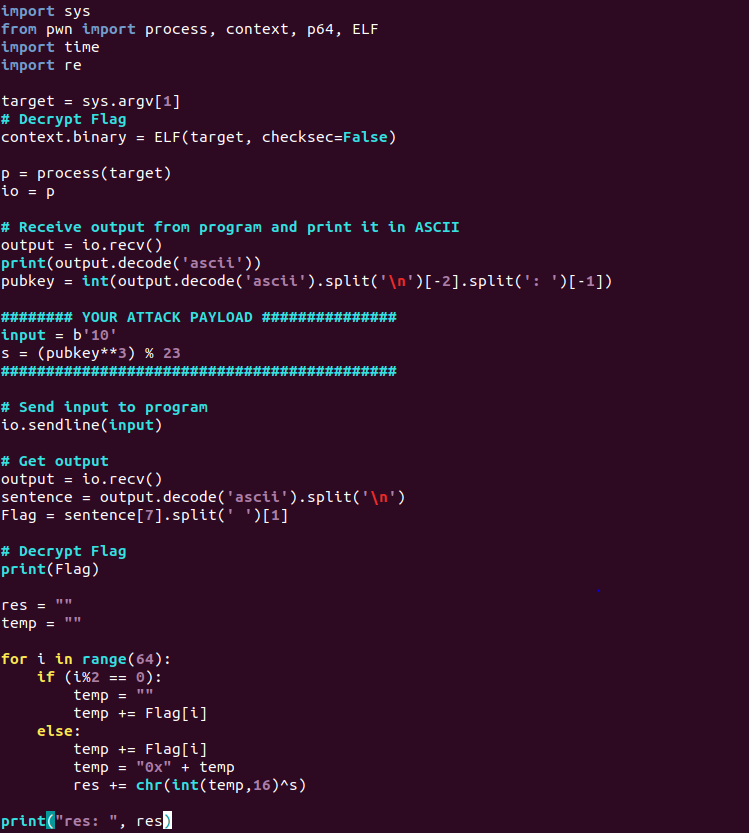
**2) Execute simple-dh**



It seems that it uses p: 23, g:5. And it’s public key is changing every time.

And if I enter my public key, it shows the encrypted flag. So I should find the shared key and decrypt given flag.

**3) Write exploit code**



From the printed string, extract its public key and save it as pubkey.

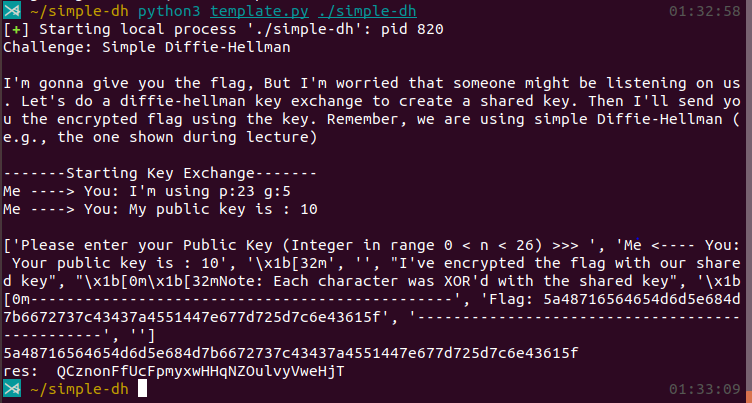
My public key is 10, and shared key is the value of (its pubkey\*\*3)%23

If I send my public key, it sends the encrypted flag. From the given string, extract the encrypted flag and save it as Flag.

Every character is xor’d with shared key, so after one more xor operation, it would decrypted.

One byte is two hex character, so save each one byte in temp, and xor it with shared key s, and convert it as character.

**4) Execute exploit code**



Finally I get the flag