Week1- Write-up

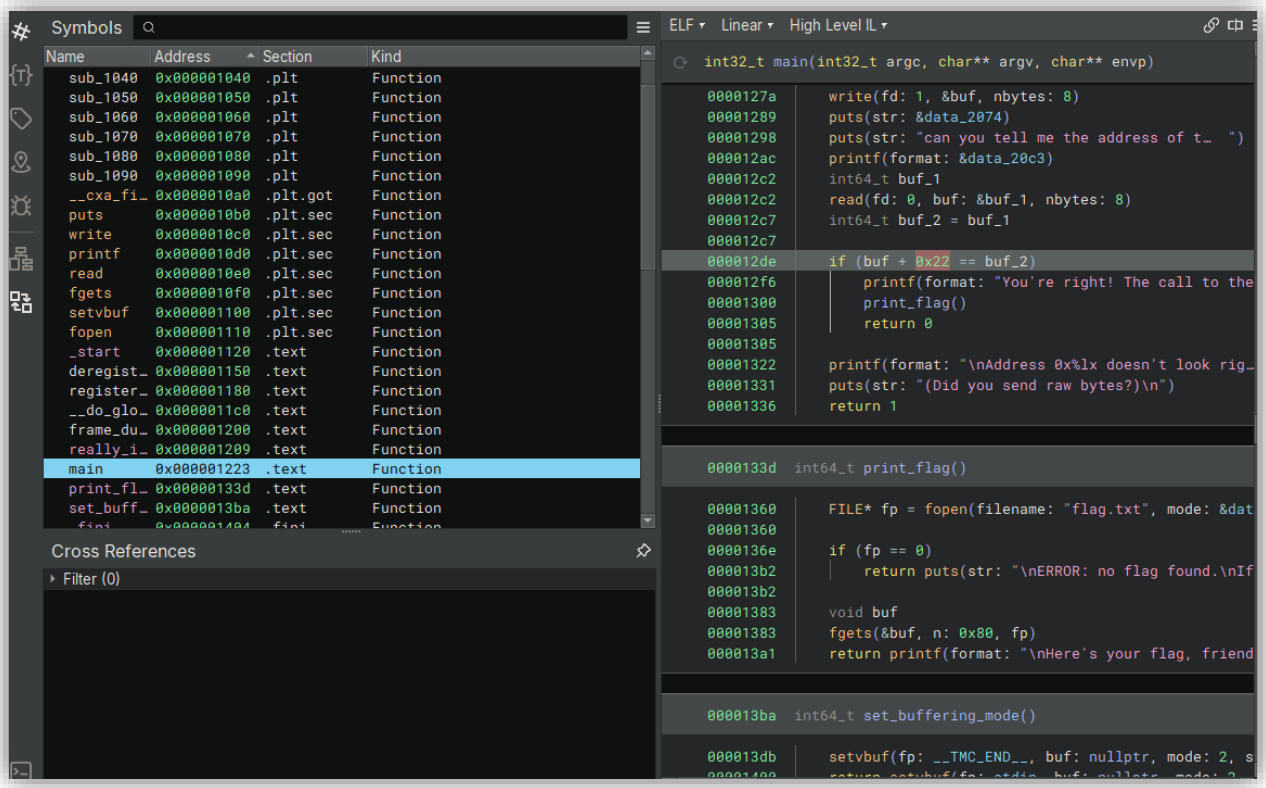
Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge – Directions

we are given the raw bytes address of the main() code and asked to figure out where the "really_important_function" is .

Solution steps -

- I used **pwntools** to connect to the server and sent my NetID (vc2499).
- The server sent a 6-byte raw address of main(), which I converted to a 64-bit integer using u64().
- I got that the value 0x22 which is added to main() by putting the given file into **Binary Ninja** tool .



- I added the value to the main() address to calculate the really_important_function address.
- I packed the calculated address into 6 bytes and sent it to the server.
- After sending the address, I used p.interactive() to get the flag.
- Submission -
  flag{st4t1c_4n4lys1s_g1v3s_us_s0_much_1nf0_4b0ut_4_b1n4ry!_8c85bdb794f4ab68}

```python
from pwn import *

conn = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1244)

print(conn.recvuntil(b'NetID (something like abc123): ').decode())
conn.sendline(b'vc2499')

print(conn.recvuntil(b'I found the raw bytes address of main() written somewhere: ').decode())
main_addr = u64(conn.recvn(6).ljust(8, b'\x00'))

func_addr = main_addr + 0x22
conn.send(p64(func_addr)[:6])

conn.interactive()

```

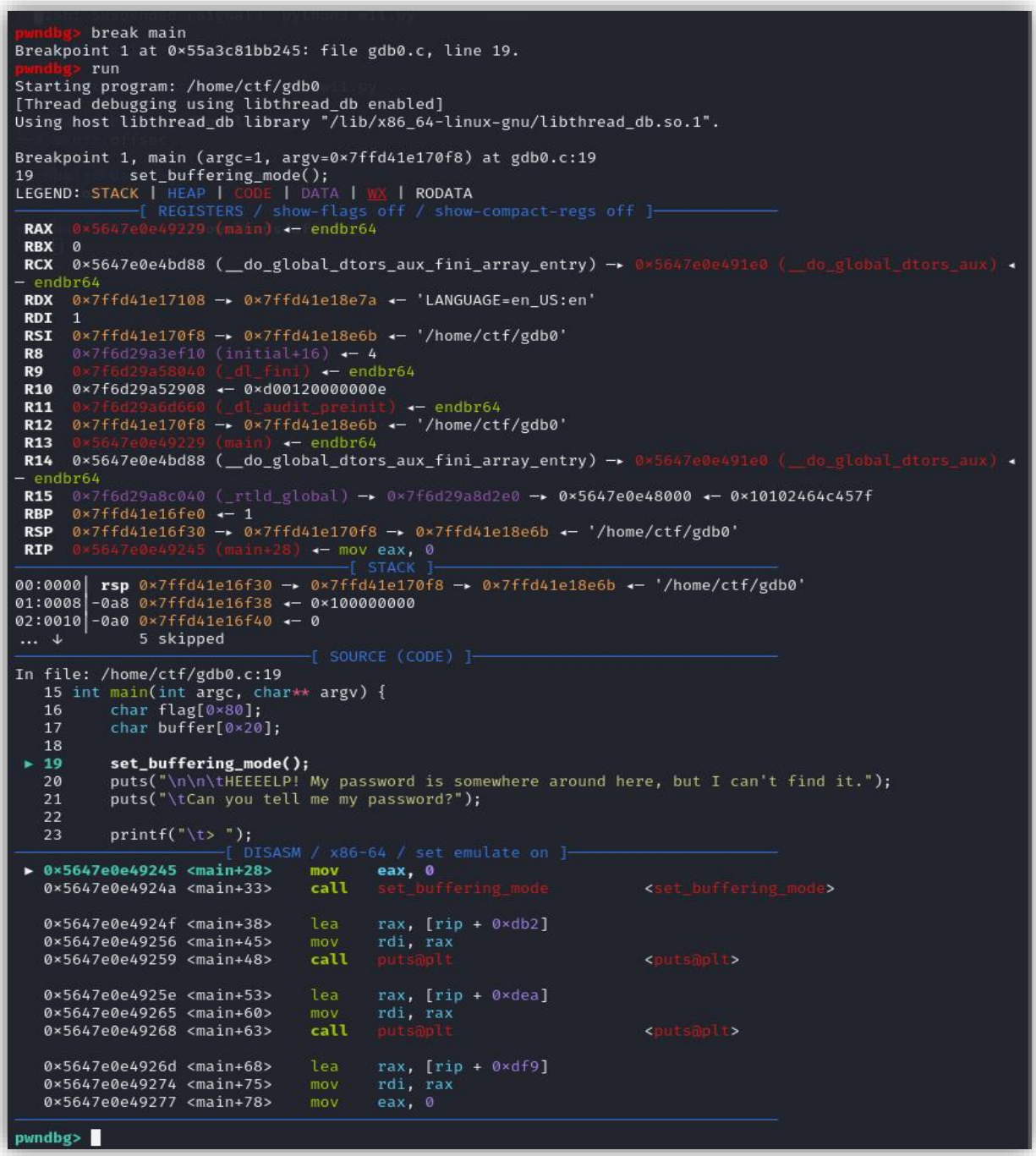
```
┌──(kali㉿kali)-[~/Downloads/offsec]
└─$ python3 directions.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1244: Done
[*] Switching to interactive mode
\x00\x00
can you tell me the address of the call to the really_important_function?
> You're right! The call to the really_important_function is at 0x562b23a46245!

Here's your flag, friend: flag{st4t1c_4n4lys1s_g1v3s_us_s0_much_1nf0_4b0ut_4_b1n4ry!_8c85bdb794f4ab68}

[*] Got EOF while reading in interactive
$
```

Challenge – GDB0

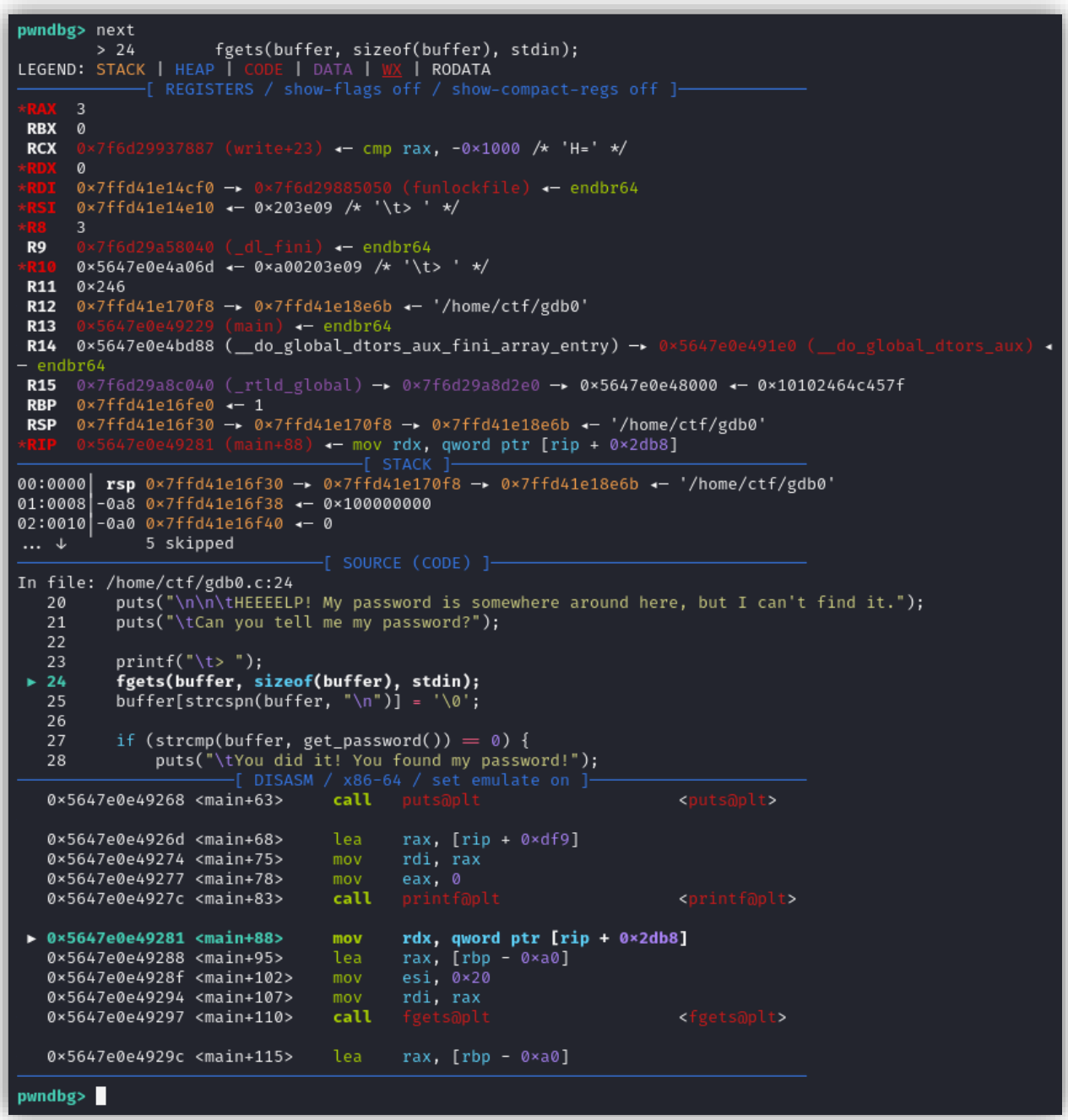
Challenge is to find the password by debugging the binary using GDB

Solution steps -

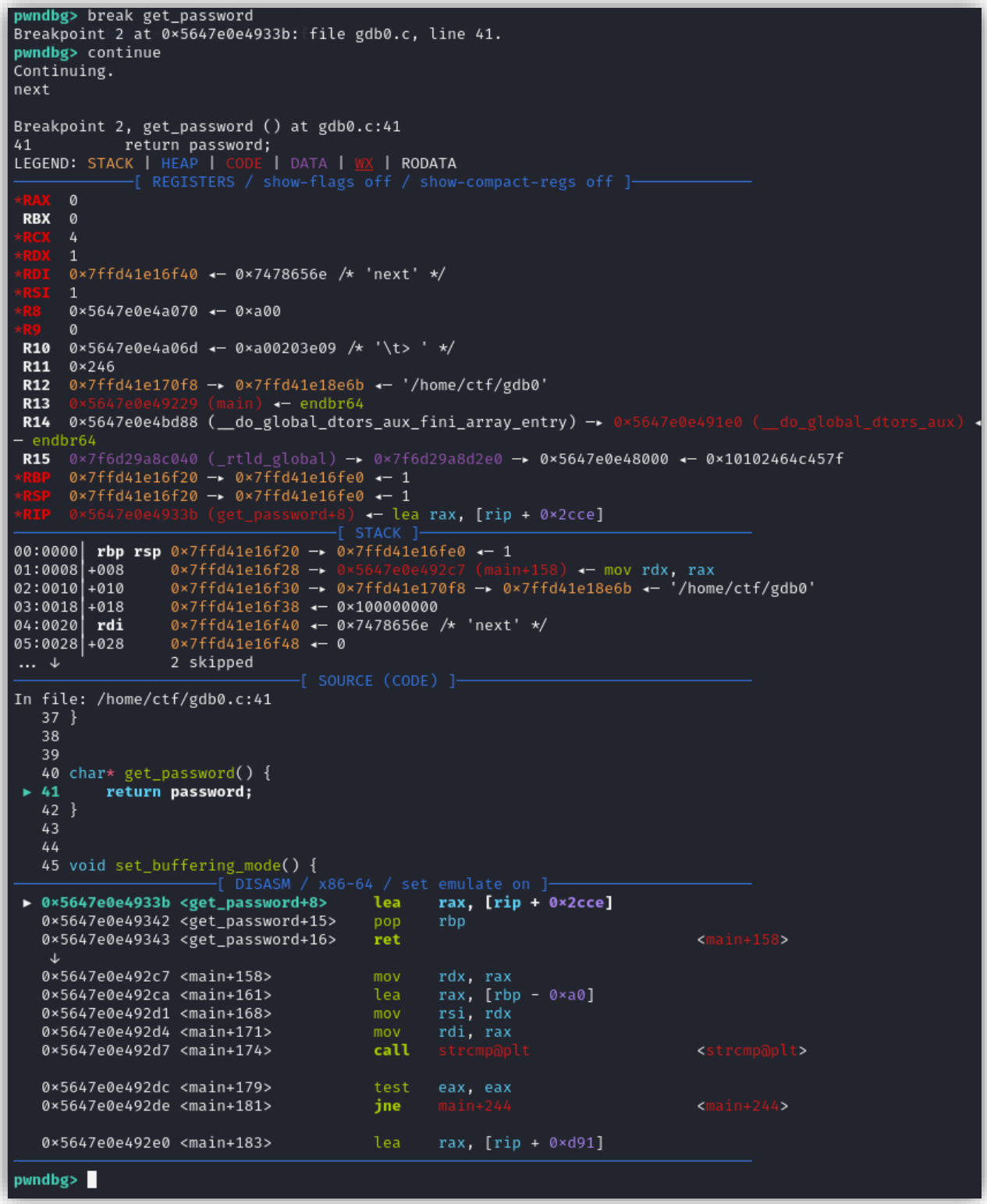
- I first set a breakpoint at the main() function to understand the flow of the program using
  - “break main”
  - “run”



- I stepped through the program to understand how it processes the password input. I identified that the program calls the get_password() function to retrieve the correct password for comparison.
- The code revealed that the program reads user input with fgets() and compares the input with the return value of get_password().

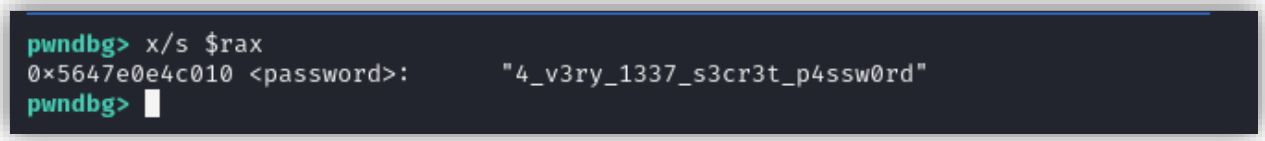


- Since the get_password() function holds the actual password, I set a breakpoint at this function to inspect the return value.
    - break get_password
    - continue

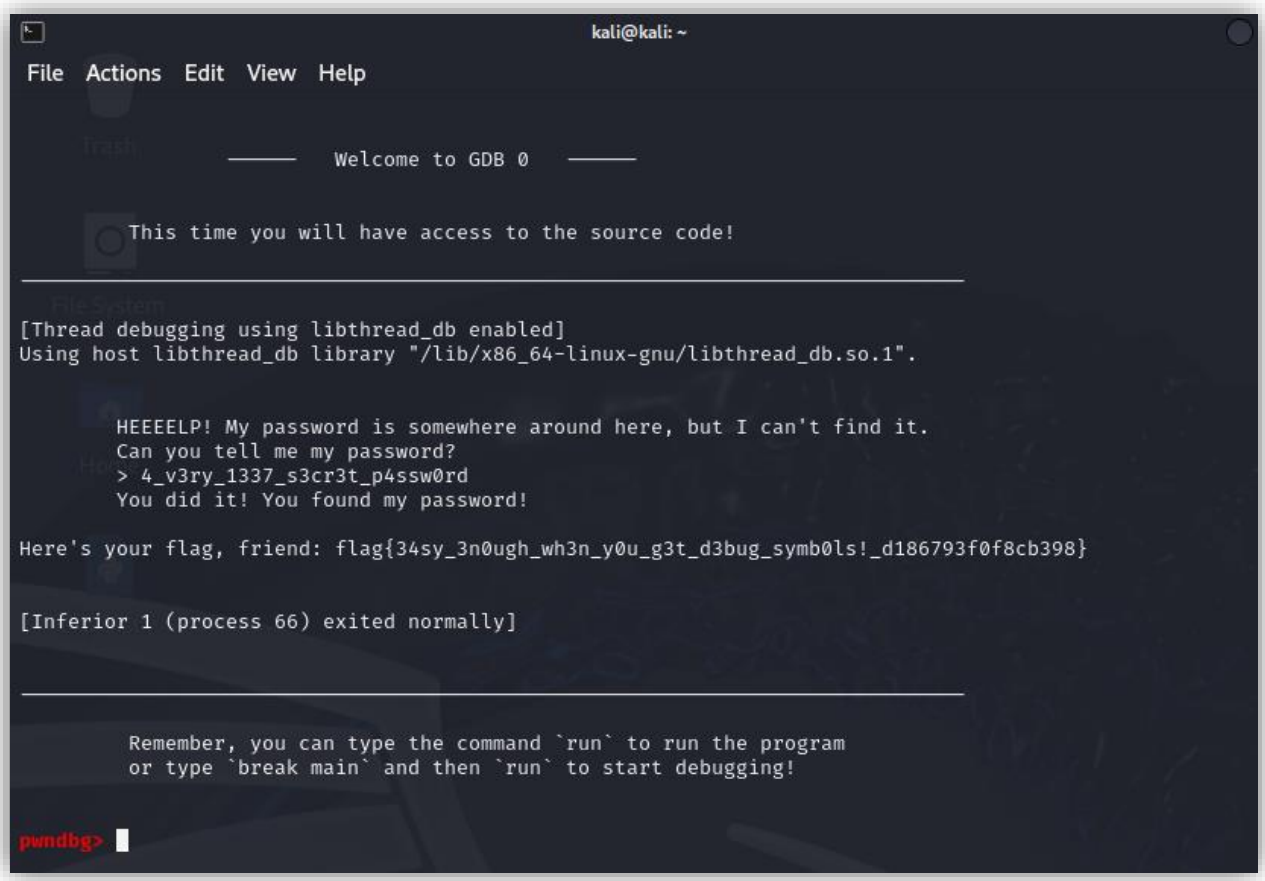


- When I hit the breakpoint at get_password(), I stepped through the function using “next” until the RAX register (which holds the return value) contained the address of the password.

- I used “x/s $rax” to get the password:
- The password was revealed to be: 4_v3ry_1337_s3cr3t_p4ssw0rd



- After retrieving the password, I entered it when prompted by the program, which resulted in successfully solving the challenge and receiving the flag.
  flag{34sy_3n0ugh_wh3n_y0u_g3t_d3bug_symb0ls!_d186793f0f8cb398}

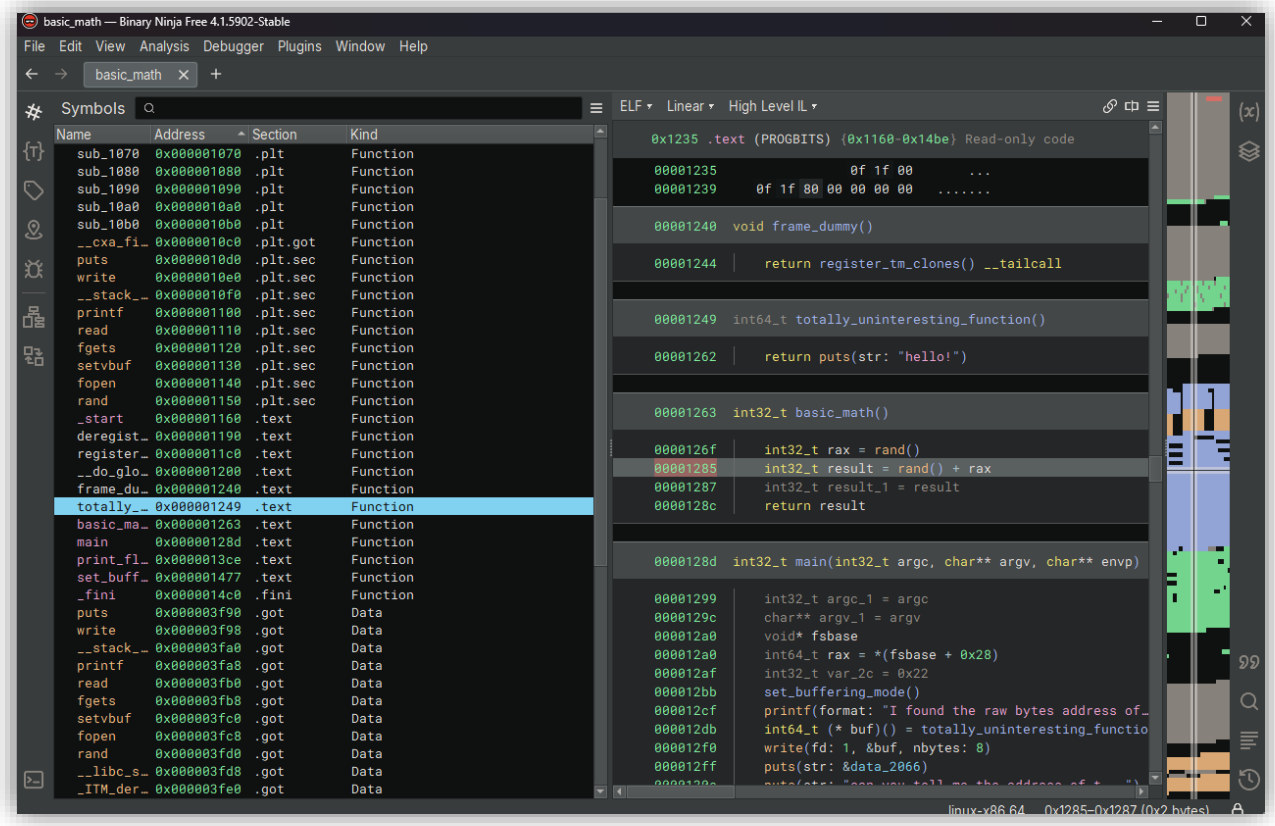


Challenge – Basic Math

We are given the raw bytes address of the totally_uninteresting_function and are asked to figure out where the ADD instruction is located in the basic_math function.

Solution Steps:

- I used **pwntools** to connect to the server and sent my NetID (vc2499).
- The server sent a 6-byte raw address of the totally_uninteresting_function, which I converted to a 64-bit integer using u64().
- I analyzed the given binary using **Binary Ninja**, where I found:
  - The offset of totally_uninteresting_function is 0x1249.
  - The offset of the ADD instruction in basic_math is 0x1285.



- To calculate the base address, I subtracted the offset 0x1249 from the raw address of totally_uninteresting_function that the server provided.
- I then added the offset 0x1285 to the base address to compute the exact memory address of the ADD instruction.
- I packed the calculated ADD instruction address into 6 bytes and sent it to the server.
- After sending the correct address, I used p.interactive() to interact with the server and retrieve the flag.
  flag{R34d1ng_4ss3mbly_l4ngu4ge_w4snt_th4t_h4rd!_ac16eabc5199b51b}

Open | basicMath.py ~/Downloads/offsec | Save

```python
from pwn import *

HOST = "offsec-chalbroker.osiris.cyber.nyu.edu"
PORT = 1245
NETID = b"vc2499"

TOTALLY_UNINTERESTING_FUNC_OFFSET = 0x1249  # Offset of totally_uninteresting_function
ADD_INSTRUCTION_OFFSET = 0x1285  # Offset of the ADD instruction in basic_math

p = remote(HOST, PORT)
p.recvuntil(b"NetID (something like abc123): ")
p.sendline(NETID)

p.recvuntil(b'I found the raw bytes address of `totally_uninteresting_function` written somewhere: ')
raw_func_addr = p.recvn(6)

func_addr = u64(raw_func_addr.ljust(8, b'\x00'))

base_addr = func_addr - TOTALLY_UNINTERESTING_FUNC_OFFSET

add_instruction_addr = base_addr + ADD_INSTRUCTION_OFFSET
raw_add_instruction_addr = p64(add_instruction_addr)[:6]

p.send(raw_add_instruction_addr)

p.interactive()

```

```
┌──(kali㉿kali)-[~/Downloads/offsec]
└─$ python3 basicMath.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1245: Done
[*] Switching to interactive mode
\x00\x00
can you tell me the address of the ADD instruction in basic_math?
> You're right! The call to the add instruction is at 0x55c17de21285!

Here's your flag, friend: flag{R34d1ng_4ss3mbly_l4ngu4ge_w4snt_th4t_h4rd!_ac16eabc5199b51b}

[*] Got EOF while reading in interactive
$ c
```

Challenge – GDB 2

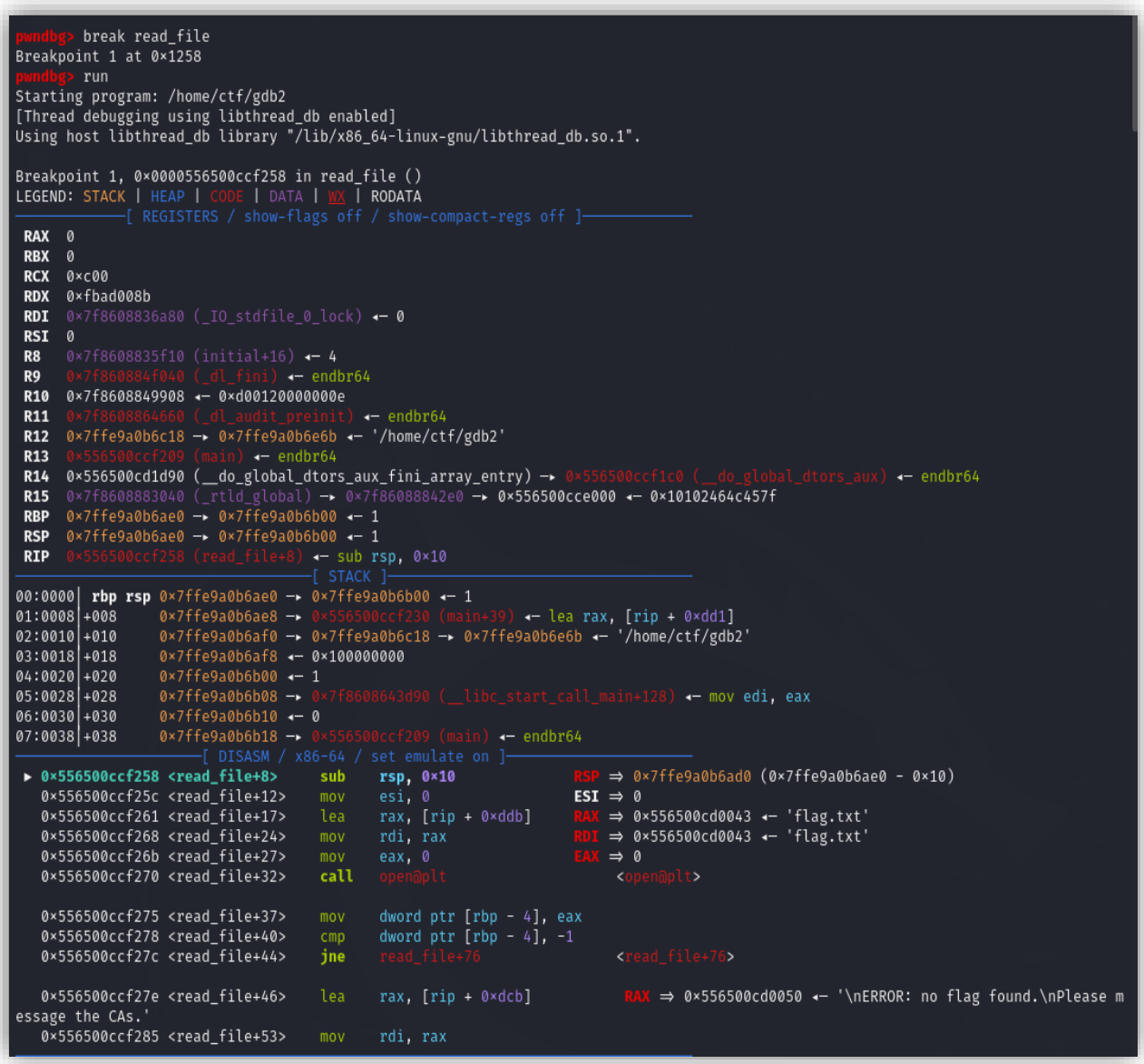
Challenge is to extract a flag from a binary called gdb2. The binary didn't require any input and the challenge hinted that the flag could be obtained through debugging.

Solution Steps:

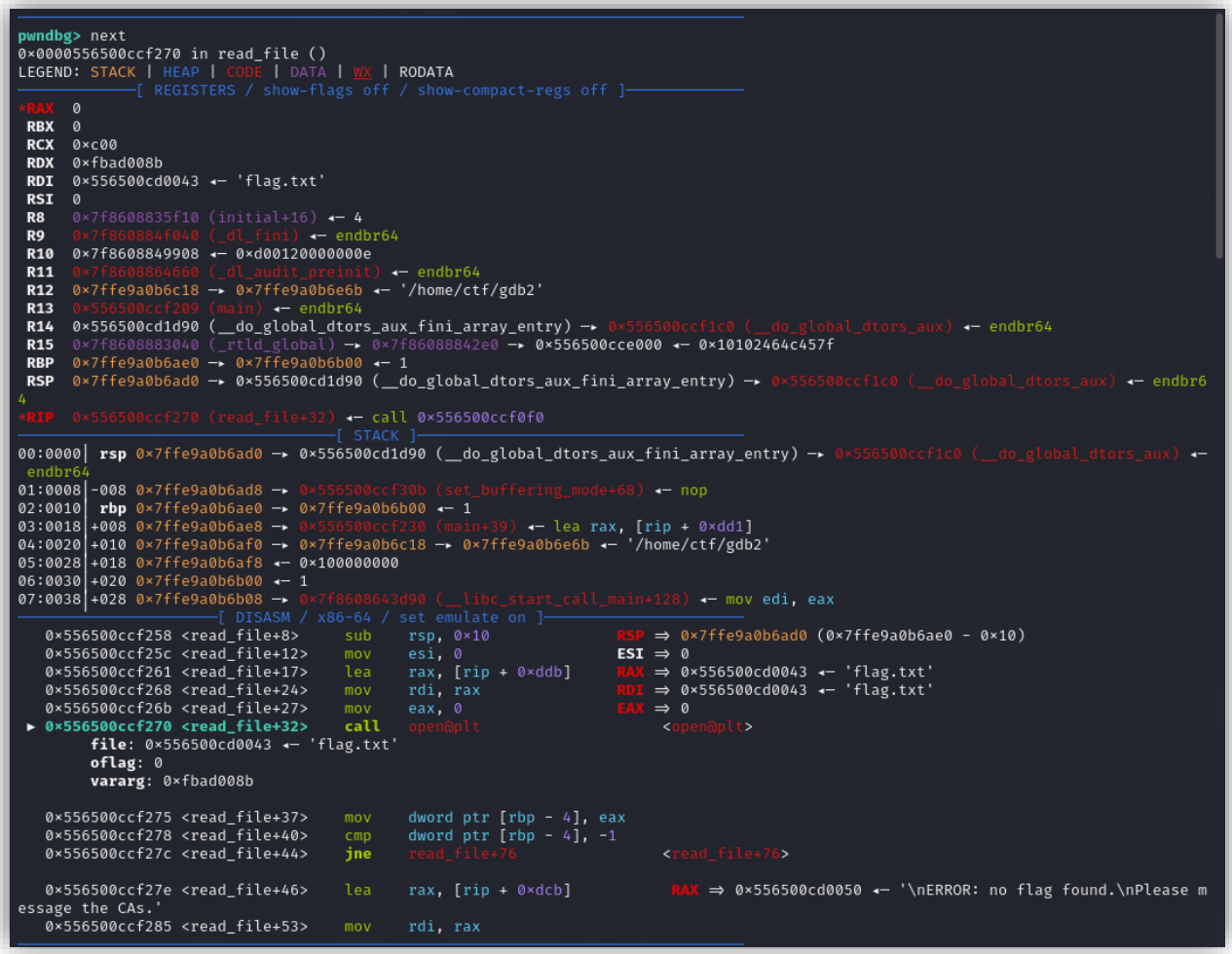
- I began by disassembling the main() function to get a sense of how the program works.
    - disassemble main

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x0000000000001209 <+0>:     endbr64
   0x000000000000120d <+4>:     push   rbp
   0x000000000000120e <+5>:     mov    rbp,rsp
   0x0000000000001211 <+8>:     sub    rsp,0x10
   0x0000000000001215 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000000000001218 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x000000000000121c <+19>:    mov    eax,0x0
   0x0000000000001221 <+24>:    call   0x12c7 <set_buffering_mode>
   0x0000000000001226 <+29>:    mov    eax,0x0
   0x000000000000122b <+34>:    call   0x1250 <read_file>
   0x0000000000001230 <+39>:    lea    rax,[rip+0xdd1]        # 0x2008
   0x0000000000001237 <+46>:    mov    rdi,rax
   0x000000000000123a <+49>:    call   0x10b0 <puts@plt>
   0x000000000000123f <+54>:    mov    edi,0x3
   0x0000000000001244 <+59>:    call   0x1110 <sleep@plt>
   0x0000000000001249 <+64>:    mov    eax,0x0
   0x000000000000124e <+69>:    leave
   0x000000000000124f <+70>:    ret
End of assembler dump.
```

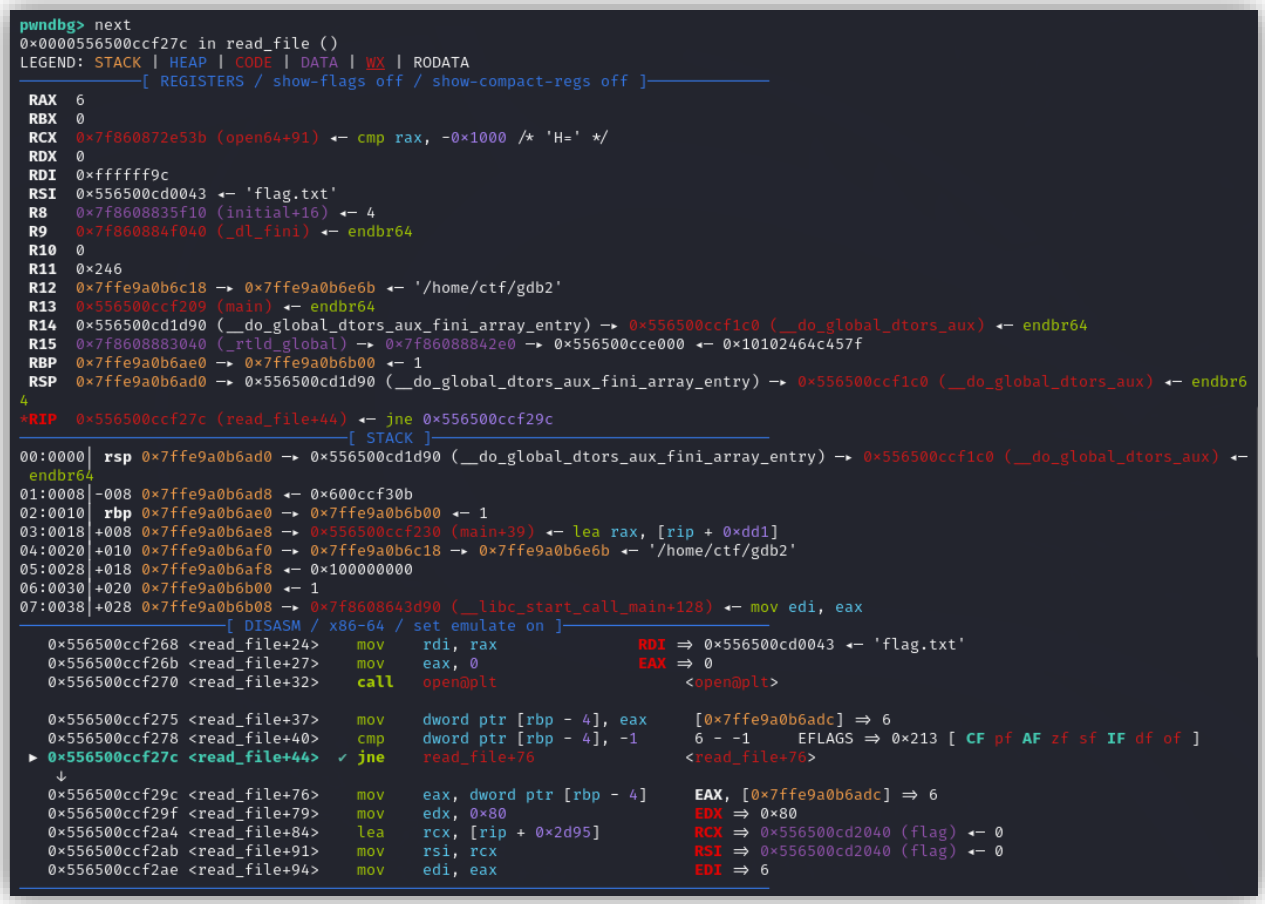
- While looking through the code, I saw that main() calls a function called read_file(). Since this sounded like it might be where the flag is being loaded, I decided to set a breakpoint at the start of read_file() to take a closer look at what it does.
    - break read_file
    - run



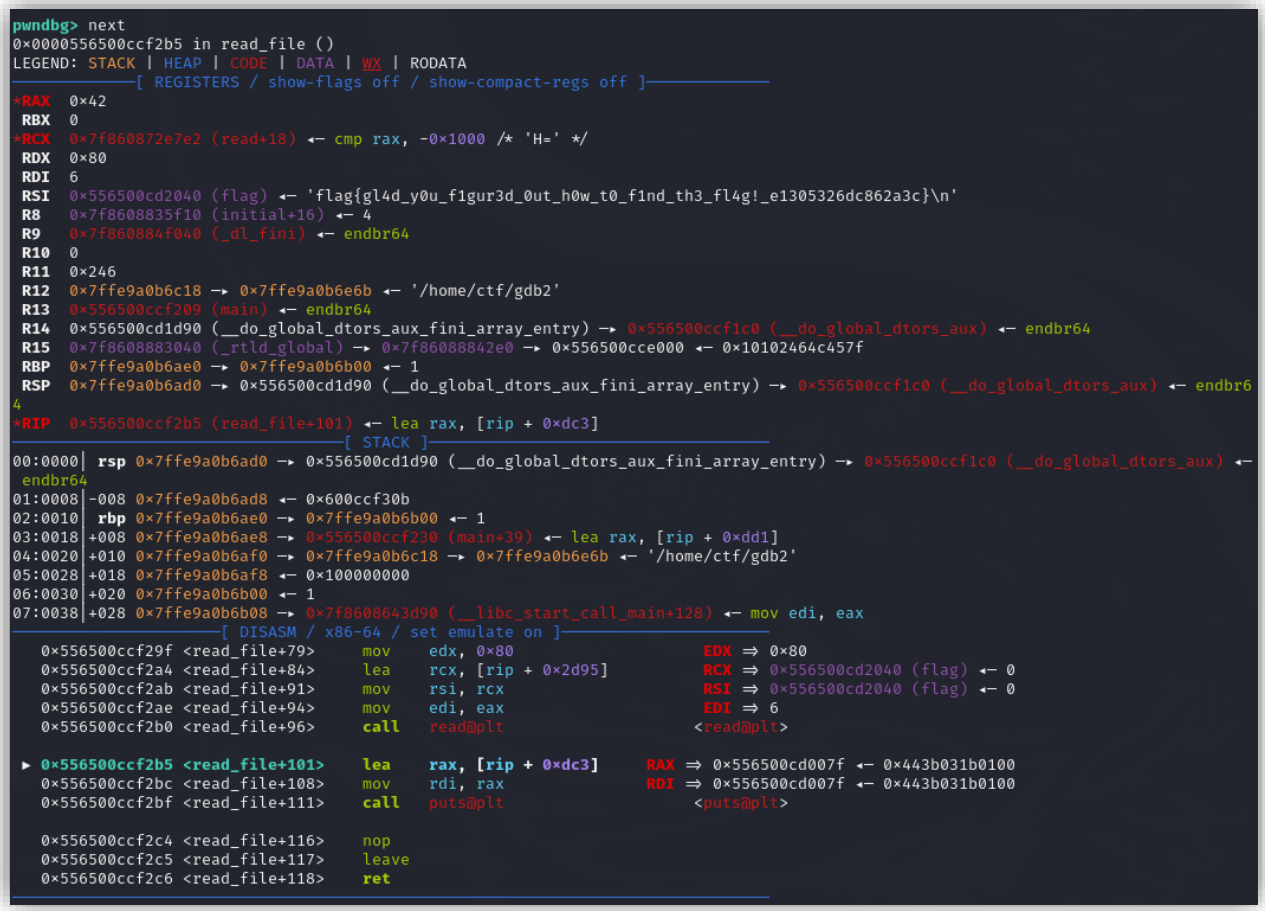
- Once the program hit the breakpoint, I started stepping through the instructions using next. It didn't take long to see that read_file() calls the open@plt function, trying to open a file called "flag.txt". That confirmed my suspicion that this file contains the flag.



- After opening the file, the program then moves on to read@plt, which reads the contents of flag.txt into a memory buffer. I carefully followed the execution and found the address where this buffer is located.



- To get the flag, I simply inspected the contents of this buffer after the read@plt call. The flag was sitting right there in memory.



- I used “x/s 0x556500cd2040” to print the flag as it is a string.
- And like that, I had the flag:
  flag{gl4d_y0u_f1gur3d_0ut_h0w_t0_f1nd_th3_fl4g!_e1305326dc862a3c}

```
pwndbg> x/s 0×556500cd2040
0×556500cd2040 <flag>:  "flag{gl4d_y0u_f1gur3d_0ut_h0w_t0_f1nd_th3_fl4g!_e1305326dc862a3c}\n"
pwndbg>
```