

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

**KHOA AN TOÀN THÔNG TIN**





**MÔN KIỂM THỬ XÂM NHẬP**

**Stack Overflow**

**Giảng viên hướng dẫn :** Đinh Trường Duy

**Sinh viên thực hiện :** Hoàng Trung Kiên

**Lớp :**  D20CQAT02-B

**Mã sinh viên :** B20DCAT098

Hà nội – 4/2023

MỤC LỤC

[**I. Bài 1**](#_1r23bpmmgk8) 3

[**II. Bài 2:**](#_nuixb7athw1s) 4

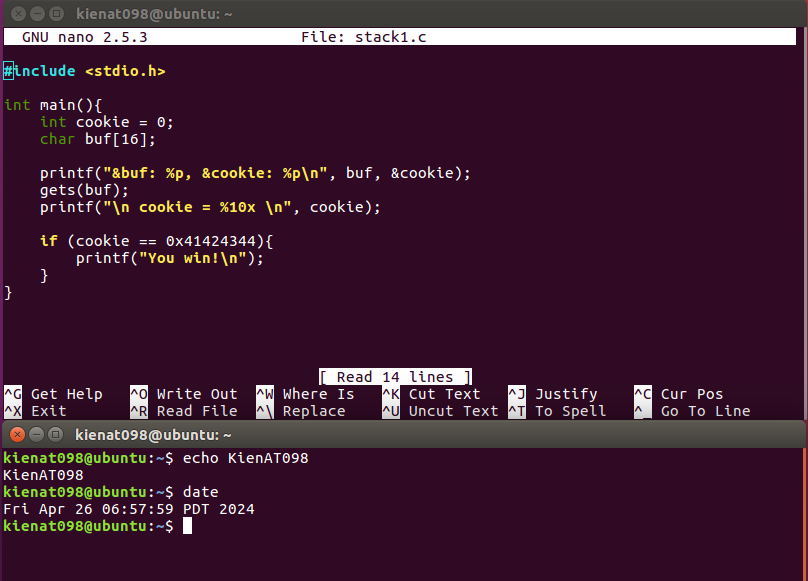
[**III. Bài 3**](#_vflo3858a3e2) 5

[**IV. Bài 4**](#_lkyooc93z3wq) 6

[**V. Bài 5:**](#_34c0scttqu73) 11

[**VI. Bài 6**](#_5c5u52wtq5xn) 17

# Bài 1



**Nhiệm vụ:** tận dụng lỗi tràn bộ đệm để in ra lỗi dòng chữ “You win!”

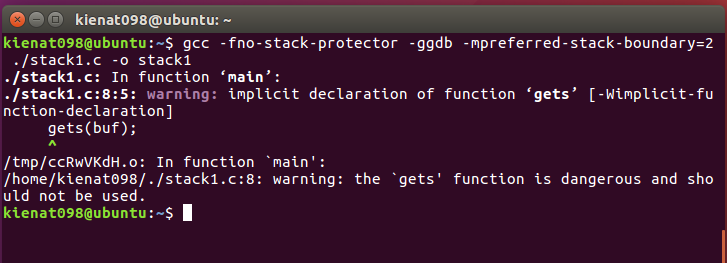
**Phân tích:**

Chúng ta sẽ làm tràn bộ đệm biến buf để ghi đè lên cookie giá trị 0x41424344. Để cookie có giá trị 41424344 thì các ô nhớ của biến cookie phải có giá trị lần lượt là 44, 43, 42, 41 theo như quy ước kết thúc nhỏ của bộ vi xử lý Intel x86. (ghi ngược lại)

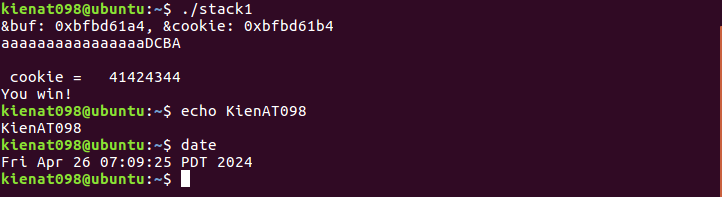
⇒ như vậy ta làm tràn bộ đệm buf bằng 16 ký tự bất kỳ tiếp theo là chuỗi ký tự DCBA ( 4 ký tự có mã ascii lần lượt là 44, 43,42,41)

**Các bước thực hiện:**

Biên dịch chương trình:

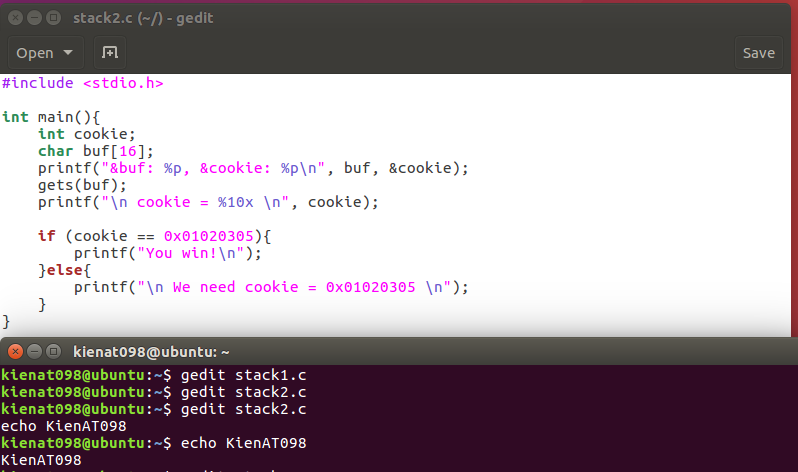


Chạy chương trình + nhập chuỗi tràn bộ đệm:



# Bài 2:

Tạo file stack2.c:



**Nhiệm vụ**: tấn công tràn bộ đệm để in ra dòng chữ “You win!”

**Phân tích:**

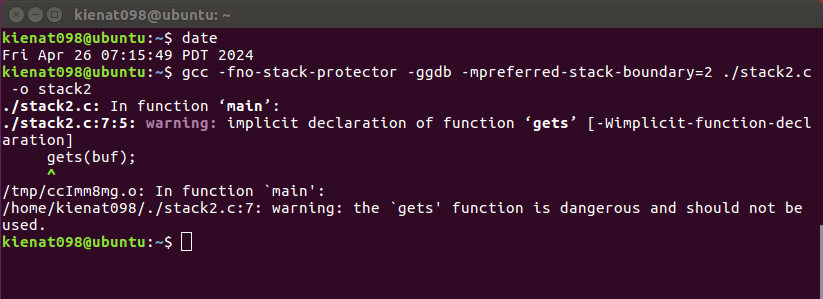
Ta cần tràn bộ đệm buf bằng 16 ký tự bất kỳ và ghi đè lên cookie giá trị 0x01020305. Để cookie có giá trị 01020305 thì các ô nhớ của biến cookie phải có giá trị lần lượt là 05, 03, 02, 01 theo như quy ước kết thúc nhỏ của bộ vi xử lý Intel x86 (ghi ngược lại)

⇒ 0x01020305, các ký tự này là những ký tự không in được, không có trên bàn phím nên cách nhập dữ liệu từ bàn phím sẽ không dùng được.

Dùng echo để truyền các ký tự đặc biệt vào qua ống (pipe)

**Các bước thực hiện:**

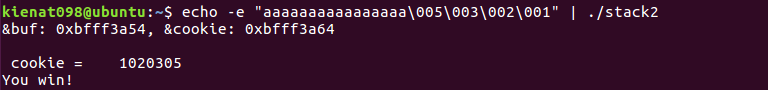
Biên dịch chương trình:



Run và nhập chuối gây tràn bộ đệm ghi đè lên cookie:

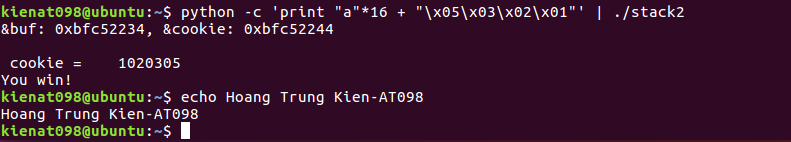
Sử dụng:

echo -e "aaaaaaaaaaaaaaaa\005\003\002\001" | ./stack 2



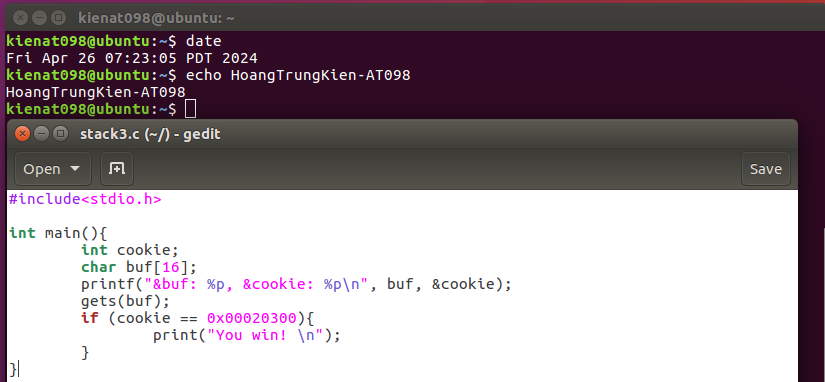
Chúng ta cũng có thể sử dụng:

python -c ’print "a"\*16 + "\x05\x03\x02\x01"’ | ./stack2



# Bài 3

Tạo file stack3.c



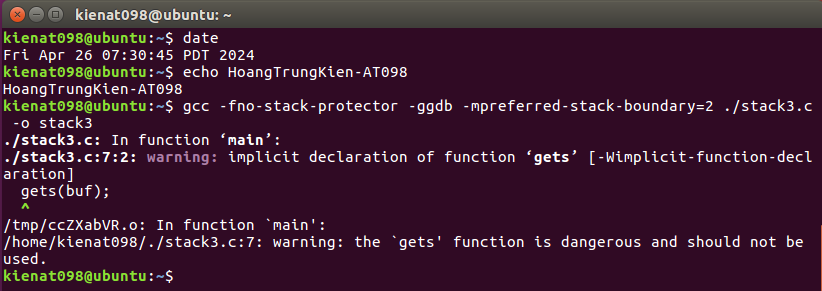
**Nhiệm vụ**: tấn công tràn bộ đệm và in ra chuỗi “You win!”

**Phân tích:**

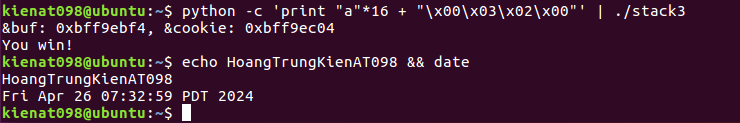
Ta sẽ gây tràn bộ đệm biến buf bằng 16 ký tự bất kỳ và ghi đè lên cookie giá trị 0x00020300. Để cookie có giá trị 00020300 thì các ô nhớ của biến cookie phải có giá trị lần lượt là 00, 03, 02, 00 theo như quy ước kết thúc nhỏ của bộ vi xử lý Intel x86 (tức ghi ngược lại). Bằng cách sử dụng python print hoặc echo như ở bài stack2.c

**Các bước thực hiện:**

Biên dịch chương trình:

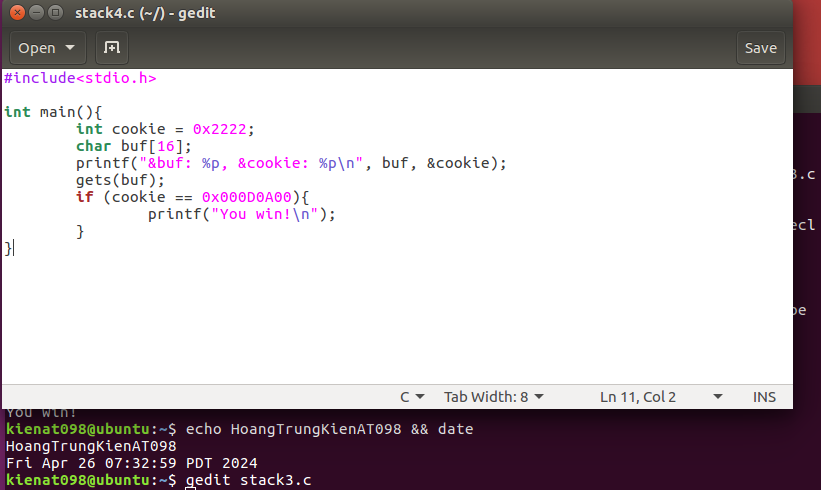


Run gây tràn bộ đệm buf và ghi đè giá trị lên cookie:



# Bài 4

Tạo file stack4.c

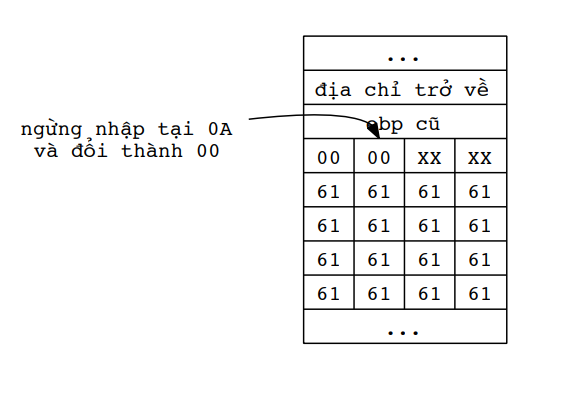


**Nhiệm vụ:** tấn công tràn bộ đệm và in ra dòng chữ You win

**Phân tích:**

Tìm hiểu tại sao không cho cookie bằng giá trị như mong muốn? (in ra biến cookie nhận được)

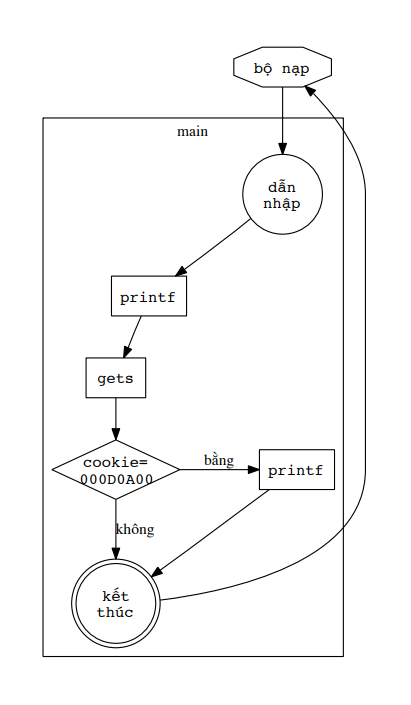
* Trả lời: ký tự dòng mới có mã ASCII là 0A. Ghi gặp ký tự này, gets sẽ ngừng việc nhận dữ liệu và thay ký tự này bằng ký tự có mã ASCII 0 (ký tự kết thúc chuỗi). Vì việc nhập dữ liệu bị ngắt tại ký tự dòng mới nên hai ký tự có mã ASCII 0D và 00 không được đưa vào cookie. Hơn nữa, bản thân ký tự dòng mới cũng bị đổi thành ký tự kết thúc chuỗi. Do đó giá trị của cookie sẽ không thể được gán bằng với giá trị mong muốn, và chúng ta cần một cách thức tận dụng lỗi khác



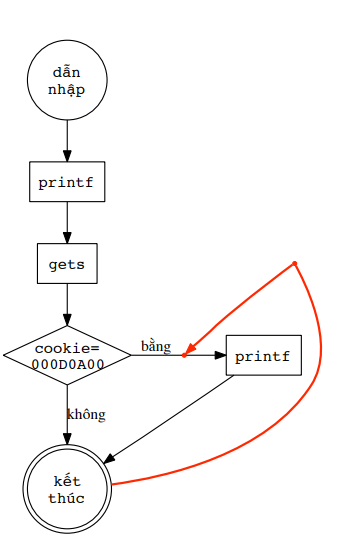
Ta sẽ thực hiện thay đổi luồng thực thị và tìm tìm địa chỉ nhánh bằng:

Quá trình thực hiện chương trình: Trước hết, hệ điều hành sẽ nạp chương trình vào bộ nhớ, và gọi hàm main. Việc đầu tiên hàm main làm là gọi tới printf để in ra màn hinh một chuỗi thông tin, sau đó main gọi tới gets để nhận dữ liệu từ bộ nhập chuẩn. Khi gets kết thúc, main sẽ kiểm tra giá trị của cookie với một giá trị xác định. Nếu hai giá trị này như nhau thì chuỗi “You win!” sẽ được in ra màn hình. Cuối cùng, main kết thúc và quay trở về bộ nạp (loader) của hệ điều hành.

Ở các ví dụ trước, chúng ta chuyển hướng luồng thực thi tại ô hình thoi để chương trình đi theo mũi tên “bằng” và gọi hàm printf in ra màn hình. Với ví dụ này, chúng ta không còn khả năng chọn nhánh so sánh đó nữa. Tuy nhiên chúng ta vẫn có thể sử dụng mã ở nhánh “bằng” ấy nếu như chúng ta có thể đưa con trỏ lệnh về vị trí của nhánh.



Biểu đồ luồng thực thi



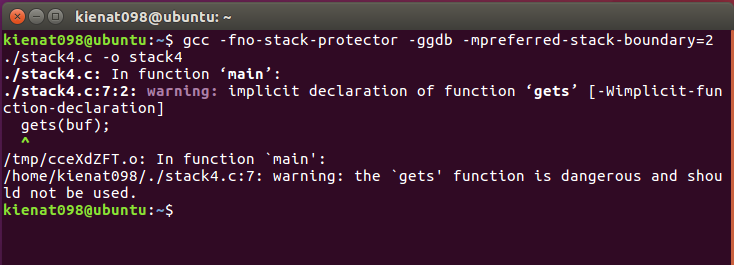
Trở về chính thân hàm

⇒ Tìm địa chỉ nhánh bằng

Để tìm địa chỉ nhánh bằng có thể sử dụng gdb hoặc objdump

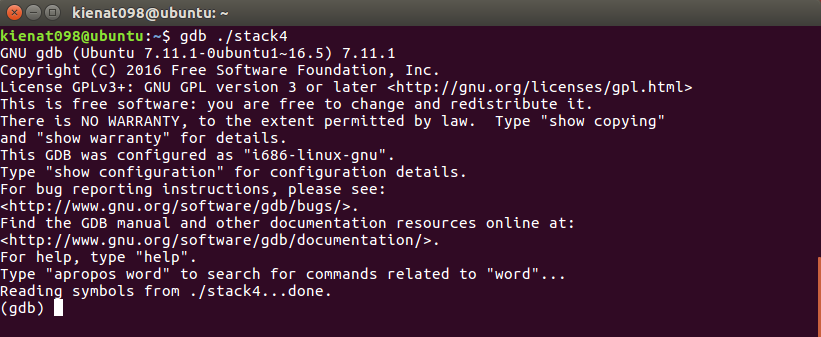
**Các bước thực hiện:**

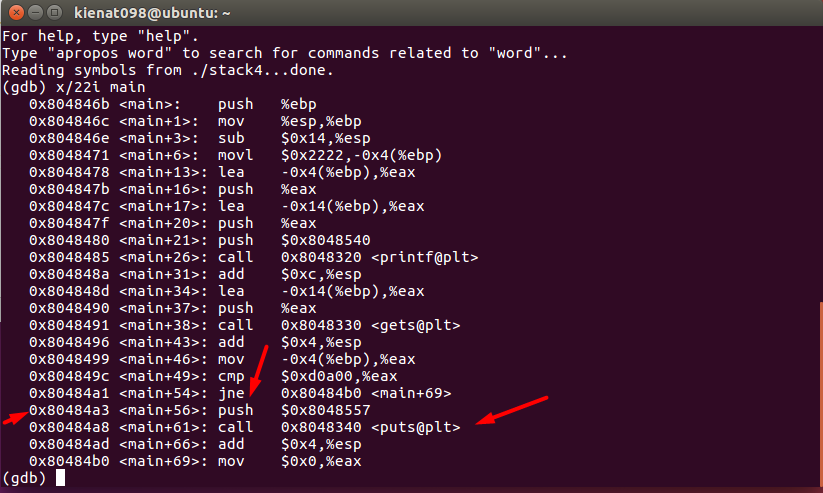
* Biên dịch chương trình:



* Sử dụng debug GDB:

GDB hiện ra dấu nhắc gdb$ chờ lệnh. Nếu ta nhập vào x/22i main thì GDB sẽ hiện (x) trên màn hình 22 (thập phân) lệnh hợp ngữ (i) đầu tiên của hàm main.

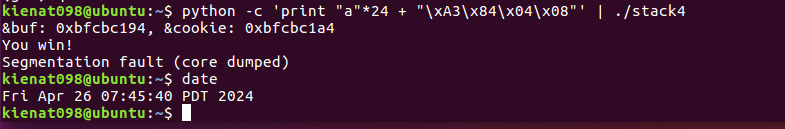




Dựa theo biều đồ luồng điều khiển, ta sẽ cần tìm tới nhánh có chứa lời gọi hàm printf thứ hai. Tại địa chỉ 0x80484a8 là lời gọi hàm printf thứ hai do đó nhánh “bằng” chính là nhánh có chứa địa chỉ này. Một vài dòng lệnh phía trên lời gọi hàm là một lệnh nhảy có điều kiện JNE, đánh dấu sự rẽ nhánh. Đích đến của lệnh nhảy này là một nhánh, và phần phía sau lệnh nhảy là một nhánh khác. Vì phần phía sau lệnh nhảy có chứa lời gọi hàm ta đang xét nên nhánh “bằng” bắt đầu từ địa chỉ 080484A3

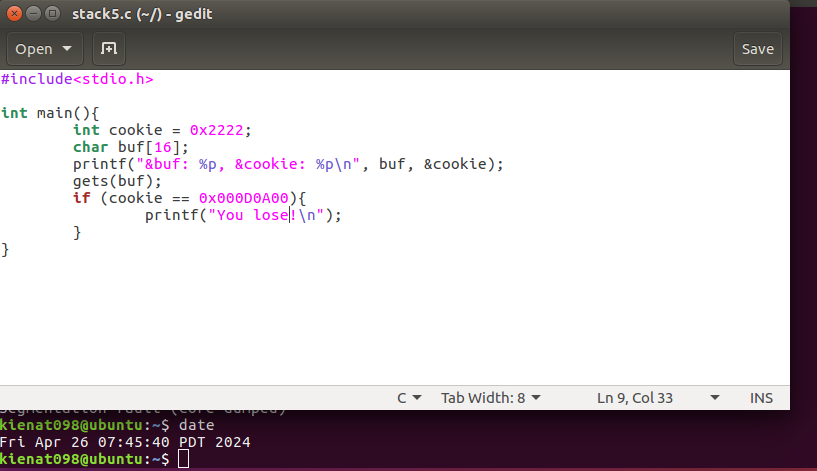
* Quay trở về thân hàm:

Để đạt được trạng thái này, chuỗi nhập vào phải đủ dài để lấp đầy biến buf (cần 16 byte), tràn qua biến cookie (cần 4 byte), vượt cả ô ngăn xếp chứa giá trị EBP cũ (cần 4 byte), và bốn ký tự cuối cùng phải có mã ASCII lần lượt là A3, 84, 04, và 08. May mắn cho chúng ta là trong bốn ký tự này, không có ký tự dòng mới. Như vậy ta sẽ cần 24 ký tự để lấp chỗ trống và 4 ký tự cuối như đã định.

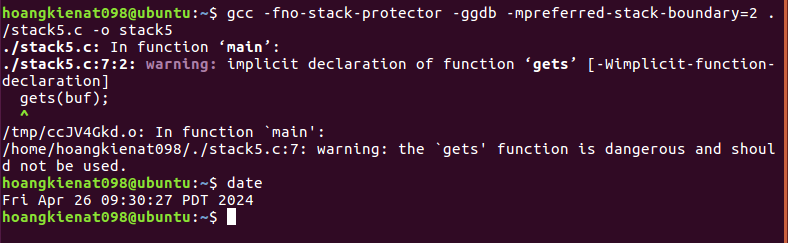


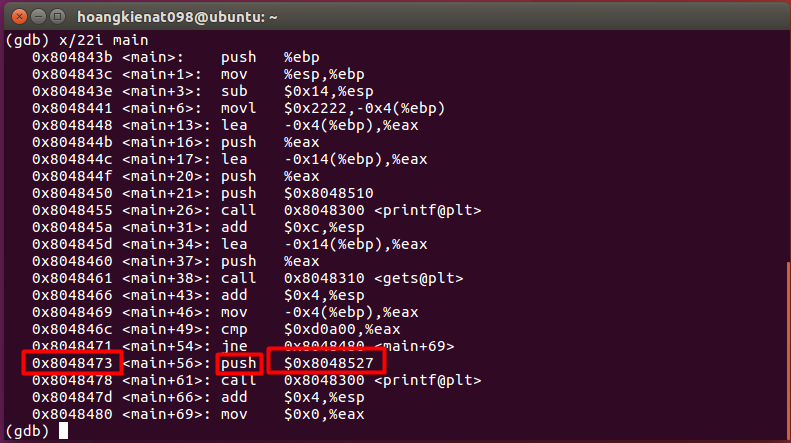
# Bài 5:

Tạo file stack5.c

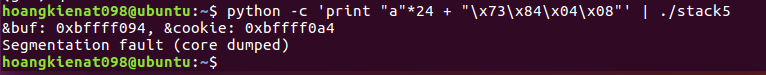


Thực thi chương trình theo cách giải stack4.c





Quit

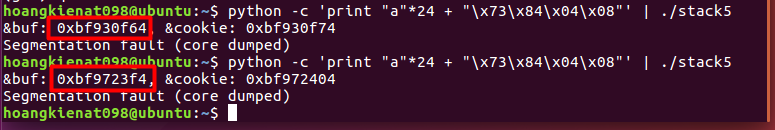


Cần truyền biến “You win!” vào chương trình, sau đó gọi hàm print với chuỗi “You win!” làm tham số đầu vào

Chèn vào chính biến buffer chuỗi “You win!”, sau đó đặt biến buffer làm tham số cho lệnh print cuối cùng của stack5.c

Theo cách giải bài stack04.c, nhưng đặt con trỏ trả về (old EIP) với địa chỉ của chính lệnh call printf, khi đó đỉnh của ngăn xếp chứa địa chỉ của buffer và lệnh call sẽ in ra buffer

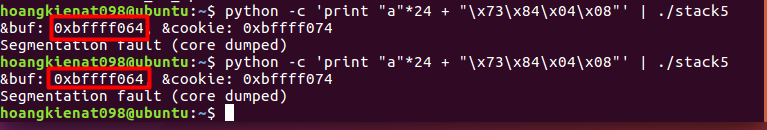
Vấn đề: địa chỉ của buffer thay đổi mỗi lần gọi do cơ chế bảo vệ bộ nhớ ASLR



Địa chỉ 2 buf khác nhau

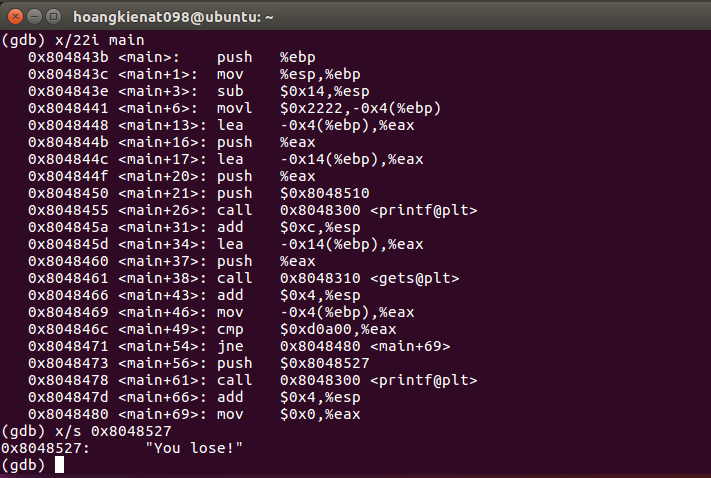
Loại bỏ ASLR:





⇒ Địa chỉ 2 buf đã giống nhau

Thực hiện ý tưởng:



Trước khi thực hiện lệnh CALL, tại địa chỉ 08048473, lệnh PUSH đưa địa chỉ của chuỗi “You lose!” vào ngăn xếp. Chúng ta có thể kiểm tra chính xác chuỗi gì được đặt tại 08048527 thông qua lệnh x/s.

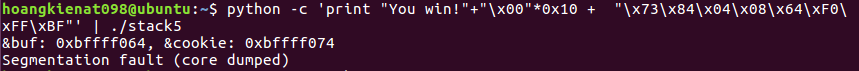


Như vậy, trước khi đến lệnh CALL tại 08048473, đỉnh ngăn xếp sẽ phải chứa địa chỉ chuỗi cần in. Nhận xét này đem lại cho chúng ta ý tưởng quay trở về thẳng địa chỉ 08048473 nếu như ta có thể gán địa chỉ chuỗi “You win!” vào đỉnh ngăn xếp

Việc còn lại chúng ta cần làm là tìm vị trí đỉnh ngăn xếp sau khi hàm main đã quay về địa chỉ 08048473. Để main quay về CALL printf thì trước khi lệnh RET ở phần kết thúc của hàm main được thực hiện, con trỏ ngăn xếp phải chỉ tới ô ngăn xếp chứa địa chỉ trở về.

Chúng ta đã xác định được đỉnh ngăn xếp nên chúng ta sẽ đặt tại vị trí đó địa chỉ chuỗi “You win!”. Để đơn giản hóa vấn đề tìm địa chỉ, chúng ta sẽ truyền chuỗi “You win!” vào chương trình thông qua việc nhập vào biến buf. Do đó địa chỉ chuỗi sẽ là BFFFF064

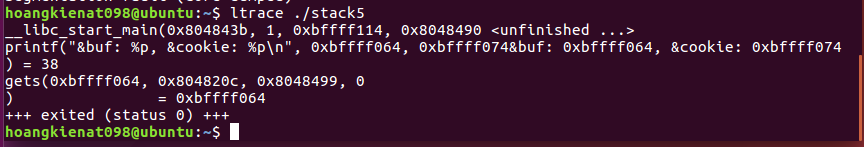
Tóm lại, chúng ta sẽ cần một chuỗi bắt đầu với “You win!”, theo sau bởi ký tự kết thúc chuỗi, rồi tới 7 ký tự bất kỳ để lấp đầy buf, sau đó 4 ký tự để lấp cookie, 4 ký tự khác để lấp giá trị EBP cũ, địa chỉ của dòng lệnh CALL printf, và kết thúc với địa chỉ của biến buf. Hay nói cách khác, ta sẽ cần 1 + 7 + 4 + 4 = 10 ký tự kết thúc chuỗi (mã ASCII 00) để lấp chỗ trống



⇒ Chúng ta không nhận được chuỗi “You win!” trên màn hình!

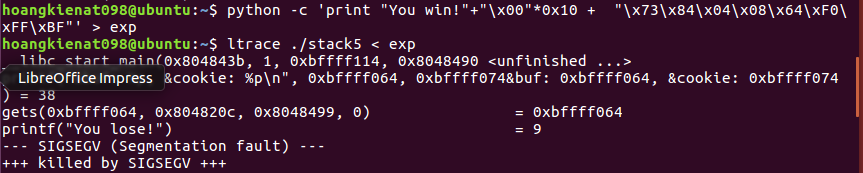
Dòng chữ “You win!” cũng lặng lẽ biến mất y như vấn đề chúng ta gặp phải. Phải chăng hàm printf đã bị bỏ qua? Để kiểm tra xem hàm printf có được gọi với tham số chính xác hay không, chúng ta có thể dùng công cụ ltrace. Công cụ ltrace theo dõi mọi lời gọi thư viện động trong quá trình thực thi của một ứng dụng.

Chúng ta sẽ dùng ltrace để kiểm chứng liệu hàm printf có được gọi với tham số mong muốn không. Trước tiên, chúng ta phải tìm địa chỉ của biến buf khi chạy chương trình bị lỗi trong ltrace.



Địa chỉ biến buf được in ra là BFFFF064

Thay vì dùng ống để truyền thẳng vào chương trình, chúng ta sẽ chuyển chuỗi này vào một tập tin để sử dụng với ltrace.

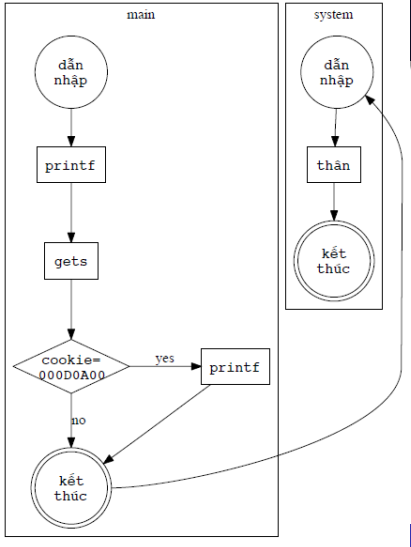


Chuỗi you win chưa được truyền vào,

⇒ Thử với các khác

Ý tưởng:

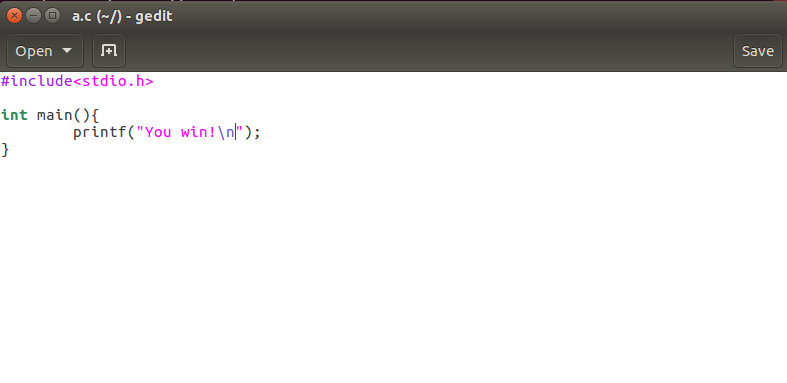
* Quay về hàm system vì không thể dùng printf
* Hàm system thực thi một lệnh trong shell

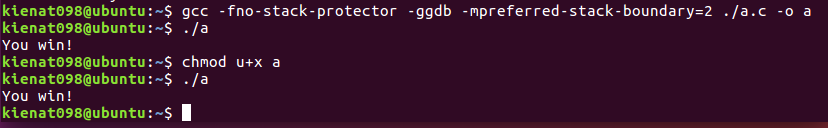


Chú ý:

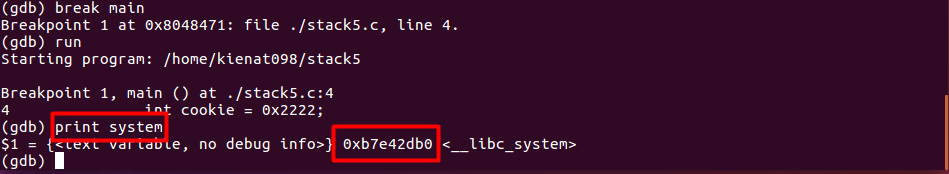
* Chương trình không có lời gọi tới hàm system
* Lệnh CALL: đưa địa chỉ trở về vào (đỉnh) ngăn xếp và nhảy tới địa chỉ của đối số (thay đổi con trỏ lệnh)
* Tìm địa chỉ hàm system để thay thế cho địa chỉ hàm printf (là đối số của lệnh CALL)

Tạo a.c với nội dung như sau:





Tìm địa chỉ system:

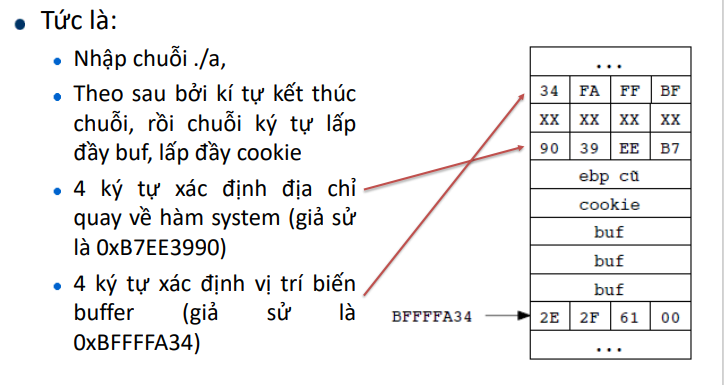


Lệnh break main đặt một điểm dừng tại hàm main của chương trình. Lệnh run thực thi chương trình. Khi chương trình chạy, hàm main sẽ được gọi, và điểm dừng đã thiết lập sẽ chặn chương trình ngay đầu hàm main. Tại thời điểm này, chương trình đã được tải lên bộ nhớ hoàn toàn cho nên bộ thư viện chuẩn cũng đã được tải. Cuối cùng chúng ta dùng lệnh print system để in địa chỉ hàm system. Địa chỉ của hàm system là b7e43da0.

Chúng ta có thể gán địa chỉ này vào ô ngăn xếp chứa địa chỉ trở về của hàm main để khi main kết thúc thì nó sẽ nhảy tới system trực tiếp. Vấn đề còn lại là ta cần xác định ô ngăn xếp nào chứa tham số của system.

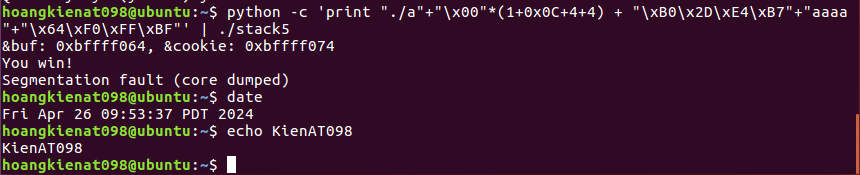
Các bước làm tiếp theo:

* Gán địa chỉ của hàm system trong bộ nhớ vào ngăn xếp chứa địa chỉ trở về của hàm main để khi main kết thúc nó nhảy về system
* Xác định ô ngăn xếp chứa tham số của system: là đỉnh của ngăn xếp khi quay trở về hàm gọi



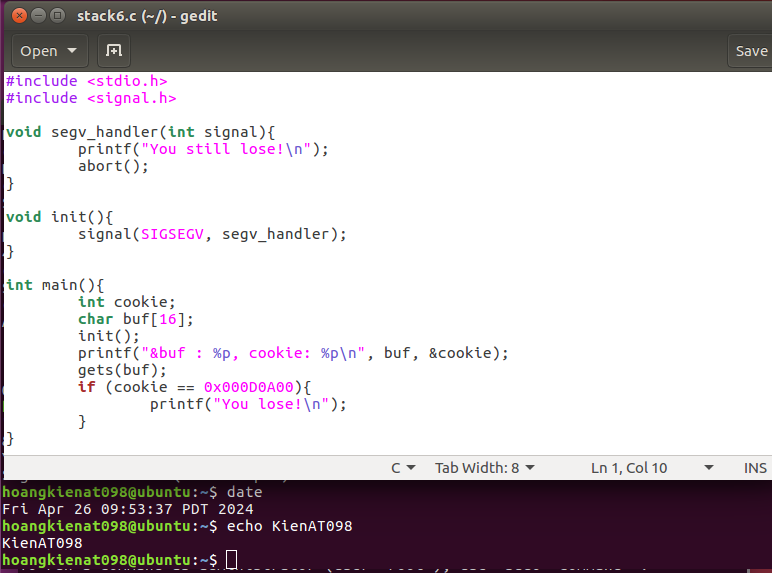
Thực hiện khai thác lỗi:

* Nhập chuỗi ./a và 1 ký tự kết thúc chuỗi
* Thêm n ký tự lấp đầy buffer, cookie
* 4 ký tự lấp đầy EPB cũ
* 4 ký tự xác định địa chỉ của system
* 4 ký tự bất kỳ để lấp địa chỉ trả về của system
* 4 giá trị xác định địa chỉ biến buffer



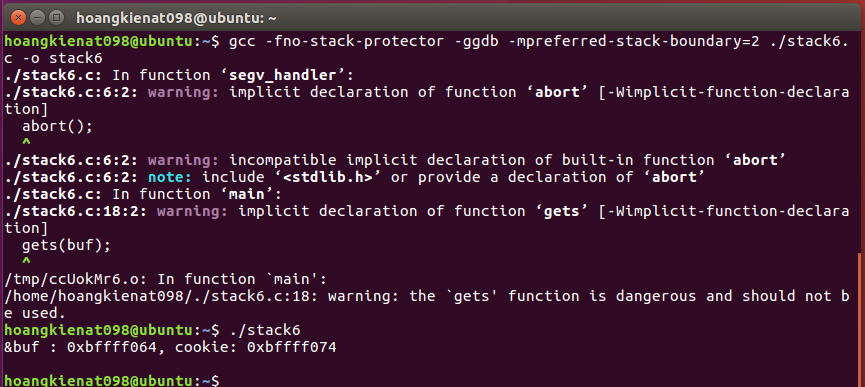
# Bài 6

Tạo file stack6.c với nội dung như sau:

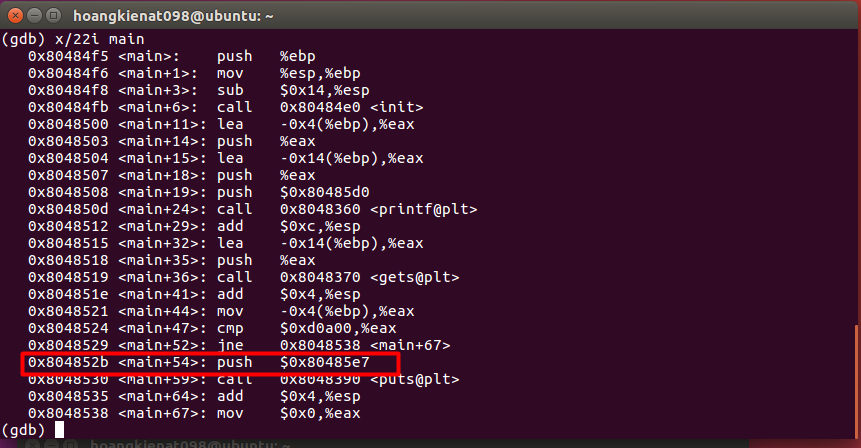


Thử thực hiện như bài 5

Biên dịch:

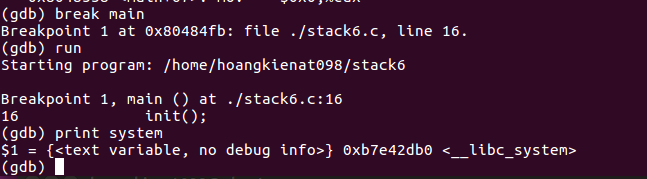


Tìm địa chỉ bằng:

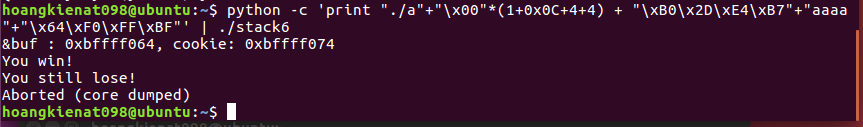


⇒ 804852b

Tìm địa chỉ system?



Thực hiện tấn công:



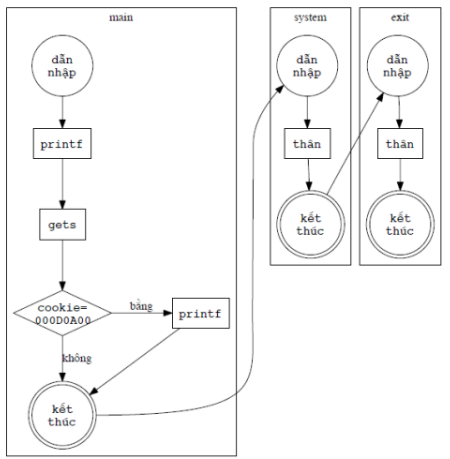
Có một điểm không hay ở những câu lệnh tận dụng mà chúng ta đã xem qua. Tuy chúng ta vẫn in được chuỗi cần in nhưng đồng thời chúng ta cũng làm chương trình gặp lỗi phân đoạn (segmentation fault). Ở những chương trình được thiết kế tốt, việc gặp phải một lỗi tương tự như thế này sẽ khiến cho người quản trị được cảnh báo và dẫn đến việc tận dụng lỗi gặp nhiều khó khăn hơn

Thật ra chương trình này chỉ thêm vào một phần xử lý tín hiệu (signal handler) SIGSEGV để in ra màn hình dòng chữ “You still lose!”. Tín hiệu SIGSEGV được hệ điều hành gửi tới chương trình khi chương trình mắc phải lỗi phân đoạn.

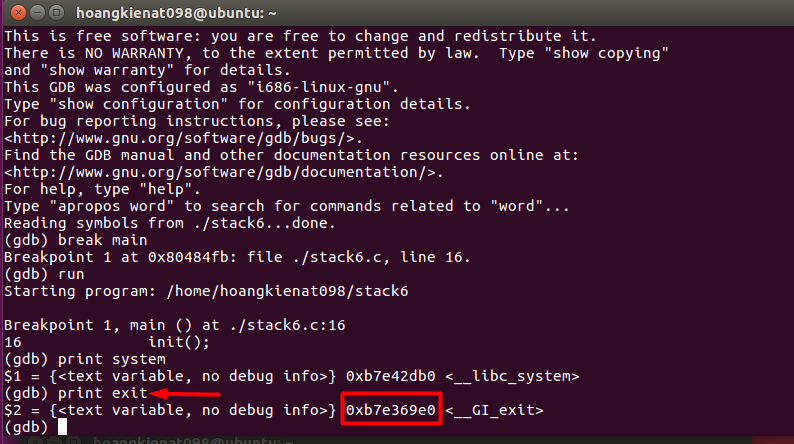
Trước hết, chúng ta cần phải tìm hiểu tại sao các câu lệnh tận dụng lỗi của chúng ta lại làm cho chương trình mắc phải lỗi phân đoạn. Nhớ lại rằng chúng ta đã sử dụng cách quay về thư viện chuẩn để ép hàm main khi thoát sẽ nhảy tới hàm system. Hàm system cũng như nhiều hàm khác, khi thực hiện xong tác vụ cũng sẽ phải trở về hàm gọi nó.

Địa chỉ trở về của system thông thường không được ánh xạ vào bộ nhớ nên khi con trỏ lệnh quay về địa chỉ đó, chương trình không thể đọc lệnh từ bộ nhớ, gây ra lỗi phân đoạn

Để khắc phục lỗi, chúng ta phải ép hàm system quay về một lệnh, hoặc một hàm nào đó để chấm dứt chương trình, không tiếp tục quay về hàm gọi nó. Một trong những hàm không quay về là hàm exit. Hàm exit chấm dứt hoạt động của một chương trình với mã kết thúc là tham số được truyền vào. Vì chúng ta không quan tâm tới mã kết thúc của chương trình nên chúng ta cũng không cần quan tâm đến tham số của hàm



Để không còn vướng lỗi phân đoạn thì chúng ta chỉ cần cho system quay về exit. Công việc cần làm sẽ bao gồm tìm địa chỉ exit và thay địa chỉ này vào vị trí của 4 ký tự a trong câu lệnh tận dụng lỗi của chúng ta. Địa chỉ của hàm exit (là một hàm trong bộ thư viện chuẩn) có thể được tìm thông qua GDB tương tự như khi chúng ta tìm địa chỉ của system..



Thực hiện tấn công:

