

LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Thu Hiền
(hiendtt@uit.edu.vn)

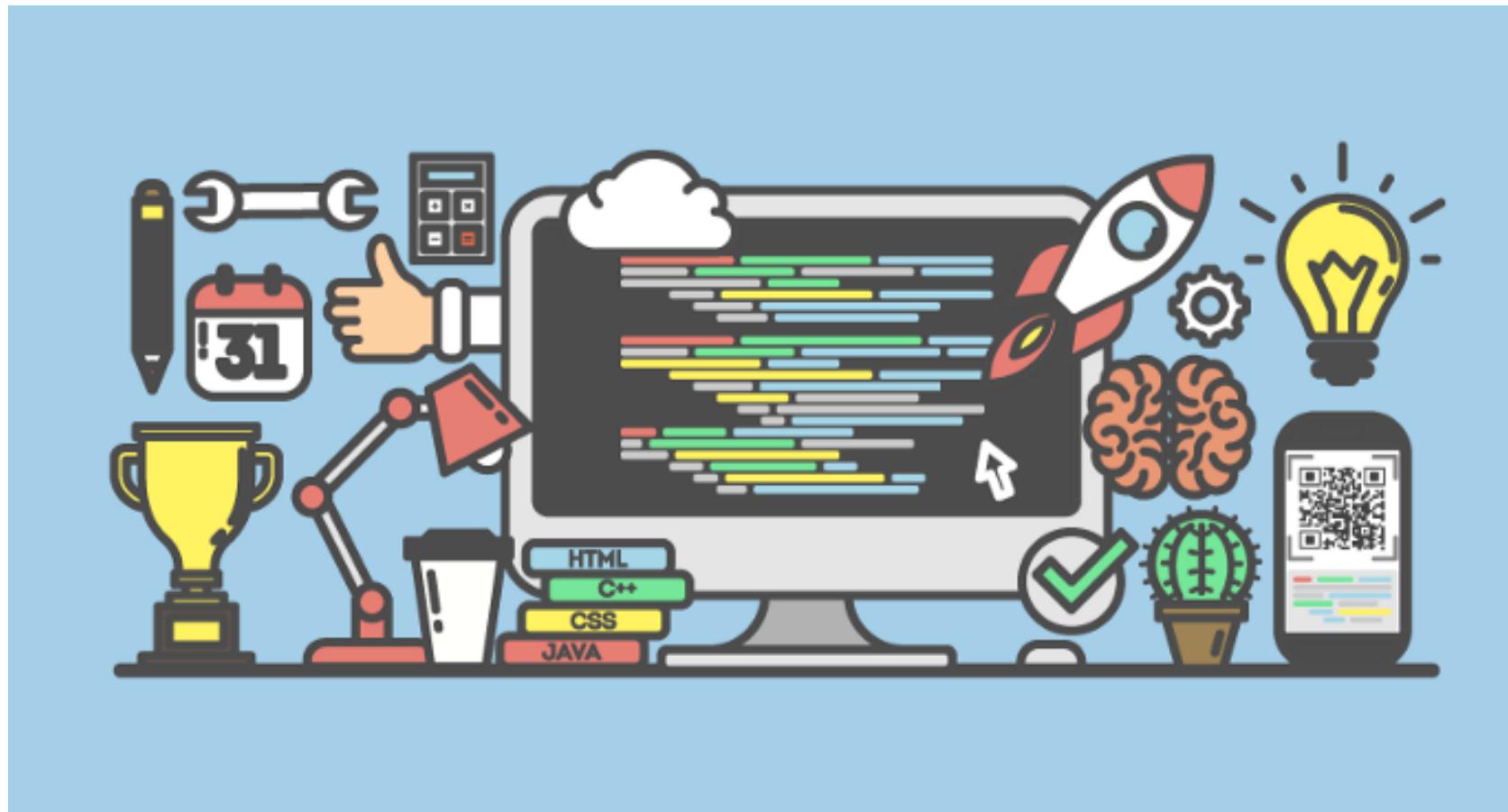


nc.uit.edu.vn

TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM
KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS

Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM
Điện thoại: (08)3 725 1993 (122)

Các chủ đề nâng cao



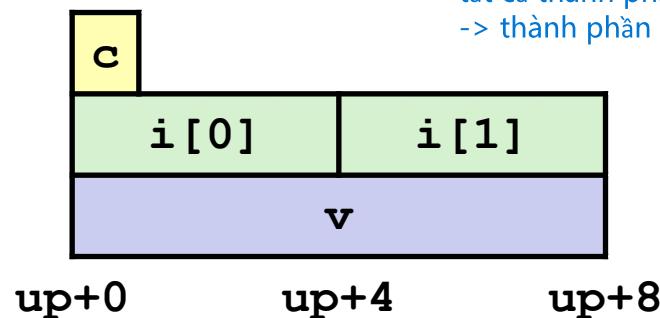
Nội dung

- Union
- Buffer overflow
 - Lỗ hổng
 - Biện pháp
- Switch (tụ tìm hiểu)

Union: Cấp phát

- Được cấp phát dựa trên thành phần lớn nhất
- Tại 1 thời điểm chỉ có thể sử dụng 1 field

```
union U1 {  
    char c;  
    int i[2];  
    double v;  
} *up;
```

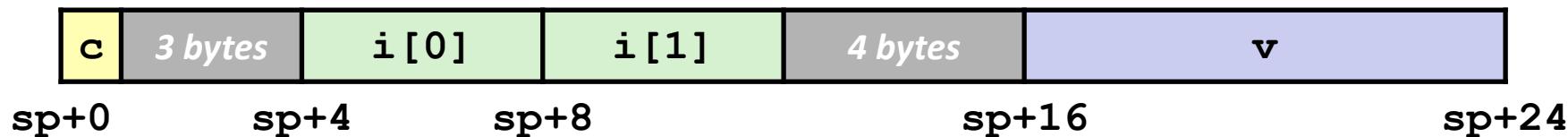


tất cả thành phần dùng chung vùng nhớ
-> thành phần có cùng địa chỉ bắt đầu - địa chỉ union

```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *sp;
```

Trong x86_64

Trường	Offset trong S1	Offset trong U1
c	0	0
i	4	0
v	16	0



Ví dụ: Dùng Union để truy xuất Bit Patterns

```
typedef union {
    float f;
    unsigned u;
} bit_float_t;
```



```
float bit2float(unsigned u)
{
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}
```

```
unsigned float2bit(float f)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

Giống như (`float`) `u` ?

Giống như (`unsigned`) `f` ?

Byte ordering: nhắc lại

■ Ý tưởng

- Short/long/quad words được lưu trong bộ nhớ như các khối 2/4/8 bytes liên tiếp
- Byte nào có trọng số thấp (cao) nhất?
- Có thể dẫn đến một số vấn đề khi trao đổi dữ liệu nhị phân giữa các máy tính.

■ BigEndian

- Byte trọng số cao nhất có địa chỉ thấp nhất
- Sparc

■ LittleEndian

- Byte trọng số thấp nhất có địa chỉ thấp nhất
- Intel x86, ARM Android và IOS

Byte Ordering: Ví dụ

```
union {  
    unsigned char c[8];  
    unsigned short s[4];  
    unsigned int i[2];  
    unsigned long l[1];  
} dw;
```

32-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]				
s[0]		s[1]		s[2]		s[3]					
i[0]				i[1]							
l[0]											

64-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]				
s[0]		s[1]		s[2]		s[3]					
i[0]				i[1]							
l[0]											

Code lấy giá trị trong Union

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
       dw.c[0], dw.c[1], dw.c[2], dw.c[3],
       dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

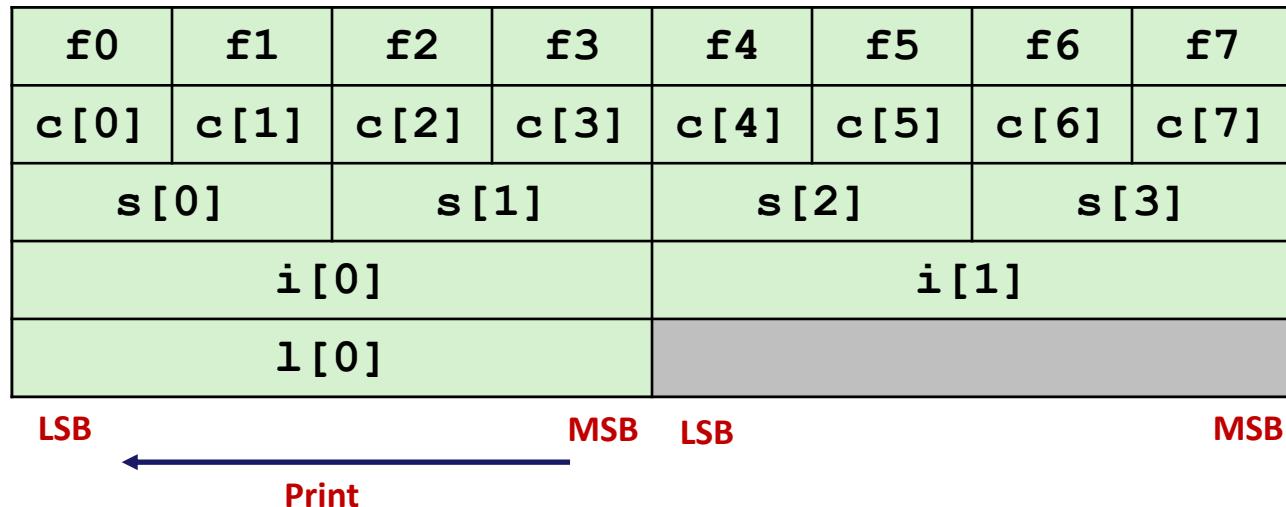
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
       dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
       dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
       dw.l[0]);
```

Byte Ordering trong IA32

LittleEndian

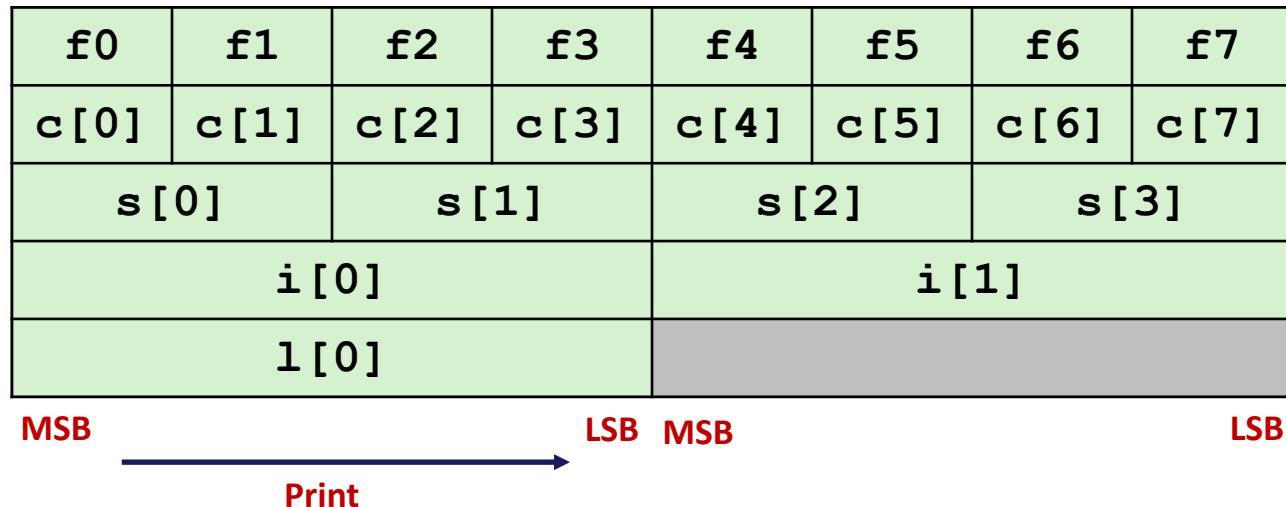


Output:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]

Byte Ordering trong Sun

Big Endian

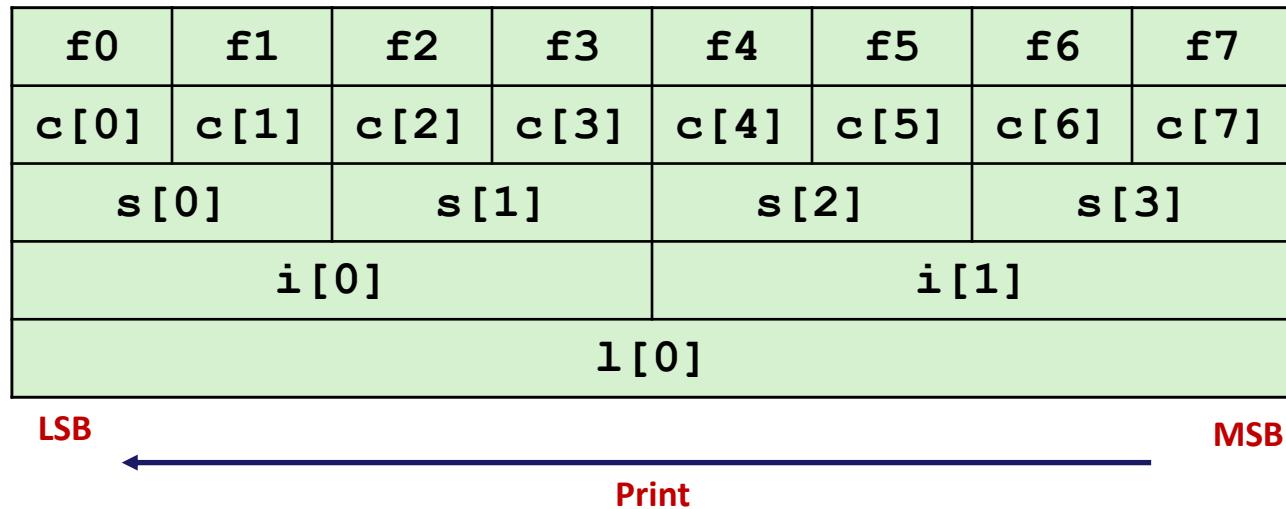


Output on Sun:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0-1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]

Byte Ordering trong x86_64

LittleEndian



Output on x86-64:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

Nội dung

- Union
- Buffer overflow
 - Lỗ hổng
 - Biện pháp
- Switch (tụ tìm hiểu)

Nhắc lại: Ví dụ bug khi truy xuất bộ nhớ

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* 0x4000000 - Possibly out of bounds */
    return s.d;
}
```

fun(0)	→	3.14
fun(1)	→	3.14
fun(2)	→	3.1399998664856
fun(3)	→	2.00000061035156
fun(4)	→	3.14
fun(6)	→	Segmentation fault

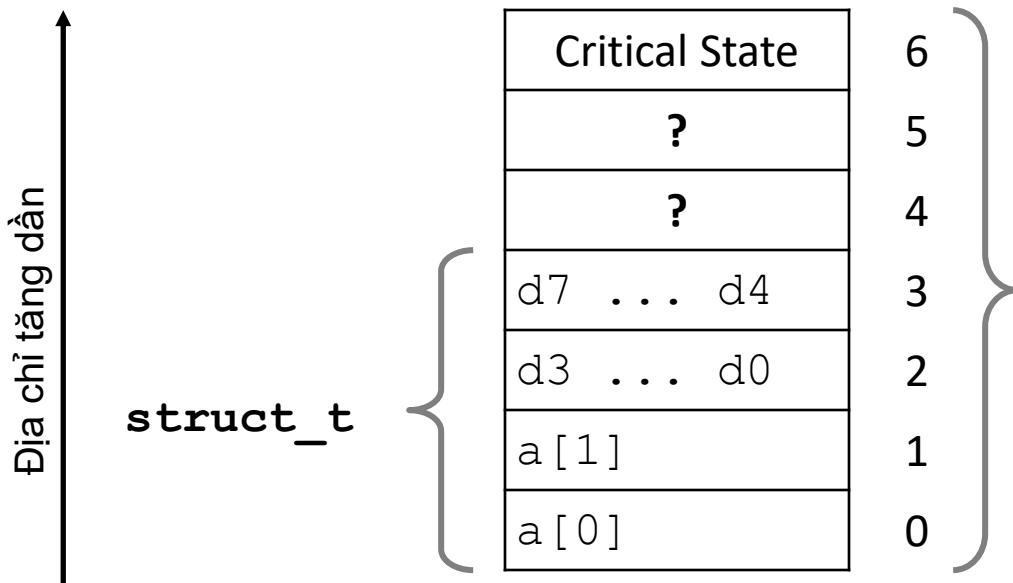
- Kết quả tuỳ vào hệ thống

Ví dụ bug khi truy xuất bộ nhớ: Giải thích

```
typedef struct {  
    int a[2];  
    double d;  
} struct_t;
```

fun(0) →	3.14
fun(1) →	3.14
fun(2) →	3.1399998664856
fun(3) →	2.00000061035156
fun(4) →	3.14
fun(5) →	3.14
fun(6) →	Segmentation fault

Giải thích:



Các vị trí được truy xuất trong
`fun(i)` để gán
`s.a[i] = 1073741824`
`(0x40000000)`

Vấn đề

- **Thường được gọi là “buffer overflow”**
 - Khi ghi vượt quá không gian bộ nhớ được cấp phát cho một mảng
- **Vì sao là 1 vấn đề lớn?**
 - Là nguyên nhân kỹ thuật #1 của các lỗ hổng bảo mật
 - nguyên nhân chung #1 là social engineering / người dùng thiếu hiểu biết
- **Dạng phổ biến nhất**
 - Không kiểm tra kích thước của chuỗi input
 - Riêng với trường hợp chuỗi ký tự giới hạn trong stack
 - còn được gọi là stack smashing

Thư viện String

■ Hàm trong Unix: gets ()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- Không có cơ chế giới hạn số ký tự sẽ đọc
- Vấn đề tương tự cũng xảy ra với các hàm thư viện
 - **strcpy, strcat**: Sao chép các chuỗi có kích thước tùy ý.
 - **scanf, fscanf, sscanf**, khi sử dụng %s

Code có lỗ hổng buffer overflow

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Nơi lưu chuỗi input */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo
Type a string: 1234567
1234567
```

```
unix>./bufdemo
Type a string: 12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

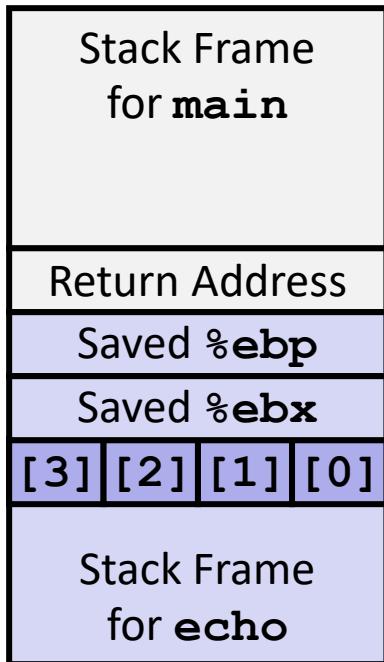
80485c5: 55	push %ebp	vị trí bắt
80485c6: 89 e5	mov %esp, %ebp	đầu lưu buf
80485c8: 53	push %ebx	
80485c9: 83 ec 14	sub \$0x14, %esp	
80485cc: 8d 5d f8	lea 0xffffffff8(%ebp), %ebx	
80485cf: 89 1c 24	mov %ebx, (%esp)	
80485d2: e8 9e ff ff ff	call 8048575 <gets>	
80485d7: 89 1c 24	mov %ebx, (%esp)	
80485da: e8 05 fe ff ff	call 80483e4 <puts@plt>	
80485df: 83 c4 14	add \$0x14, %esp	
80485e2: 5b	pop %ebx	
80485e3: 5d	pop %ebp	
80485e4: c3	ret	

call_echo:

80485eb: e8 d5 ff ff ff	call 80485c5 <echo>
80485f0: c9	leave
80485f1: c3	ret

Buffer Overflow Stack

Trước khi gọi gets



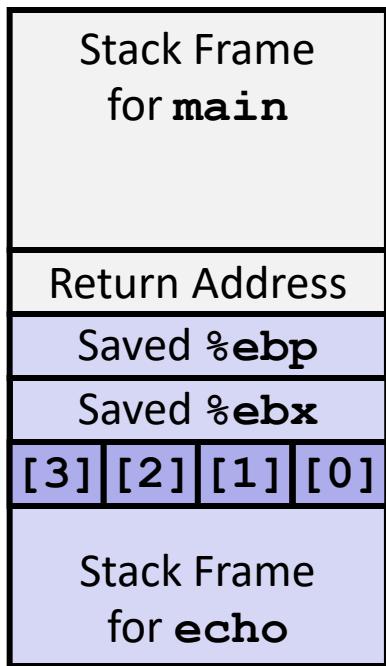
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp          # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx          # Save %ebx
    subl $20, %esp      # Allocate stack space
    leal -8(%ebp), %ebx # Compute buf as %ebp-8
    movl %ebx, (%esp)   # Push buf on stack
    call gets           # Call gets
    . . .
```

Buffer Overflow Stack

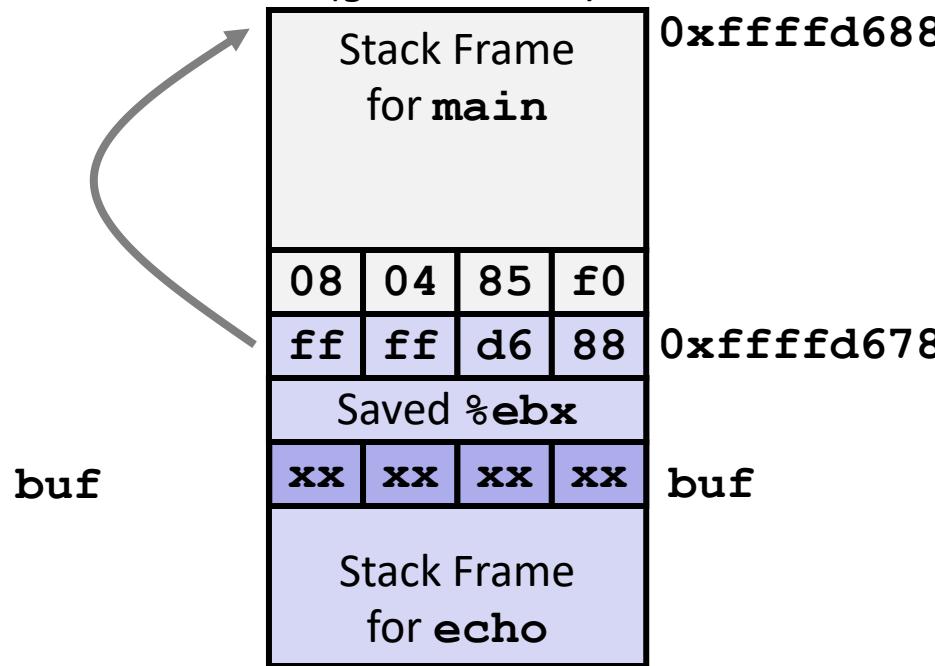
Ví dụ chạy thực tế

Trước khi gọi gets
(tên đại diện)



```
80485eb: e8 d5 ff ff ff    call  80485c5 <echo>
80485f0: c9                 leave
```

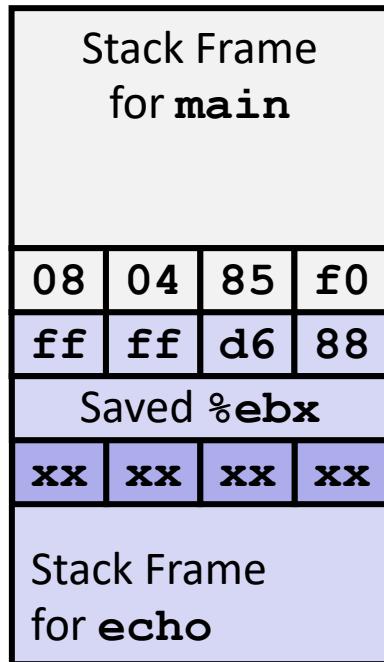
Trước khi gọi gets
(giá trị cụ thể)



```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```

Buffer Overflow: Ví dụ #1

Before call to gets

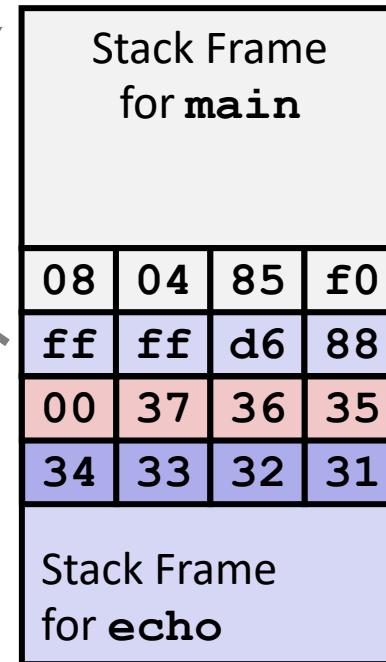


0xffffd688

0xffffd678

buf

Input: 1234567



0xffffd688

0xffffd678

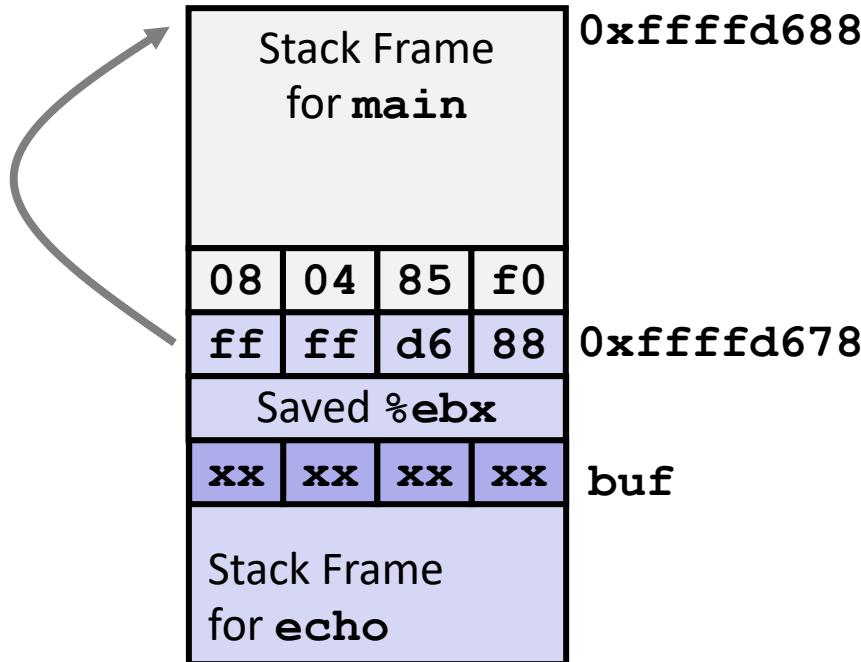
buf

```
unix> ./bufdemo
Type a string: 1234567
1234567
```

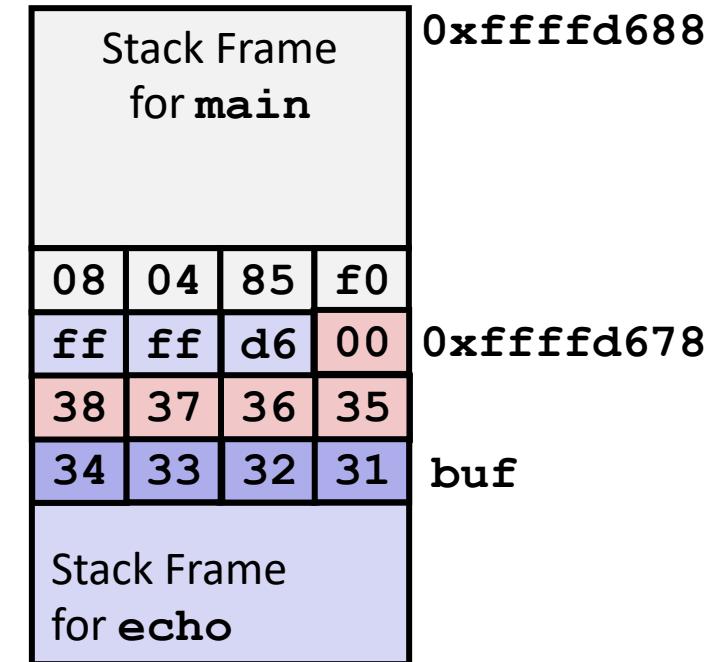
Vượt quá buf, ghi đè %ebx,
nhưng không gây ra vấn đề gì
→ Chỉ làm thay đổi 1 giá trị đã lưu

Buffer Overflow: Ví dụ #2

Before call to gets



Input: 12345678



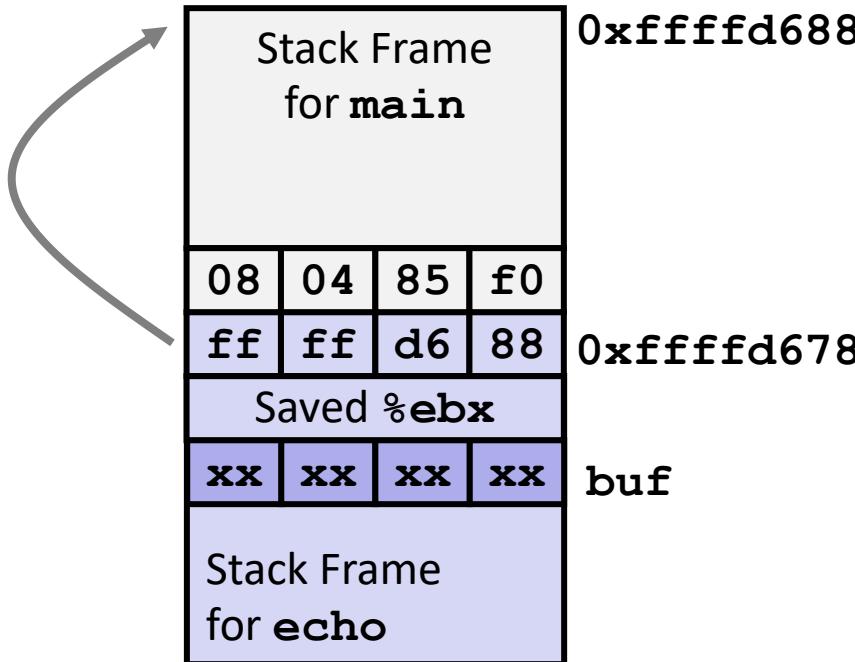
```
unix>./bufdemo
Type a string: 12345678
Segmentation Fault
```

Ghi đè %ebp cũ → lỗi

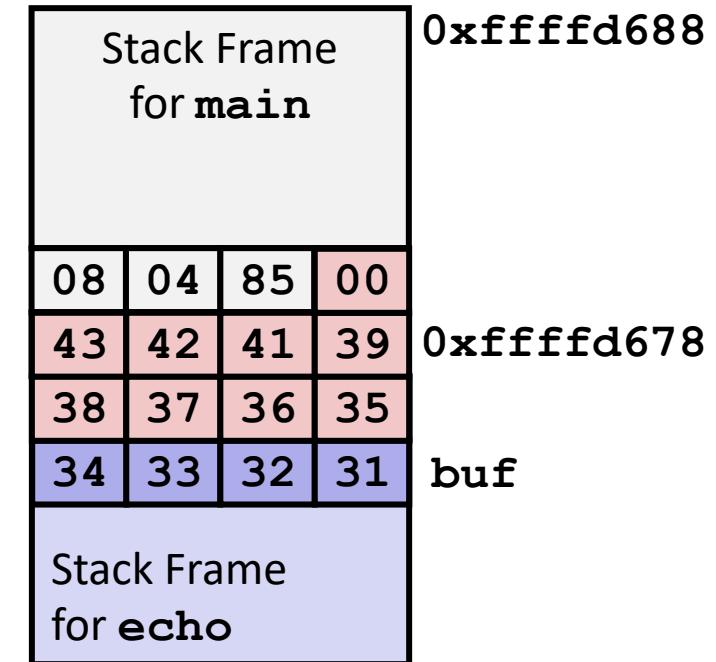
```
80485eb: e8 d5 ff ff ff    call  80485c5 <echo>
80485f0: c9                  leave # Set %ebp to corrupted value
80485f1: c3                  ret
```

Buffer Overflow: Ví dụ #3

Before call to gets



Input: 123456789ABC



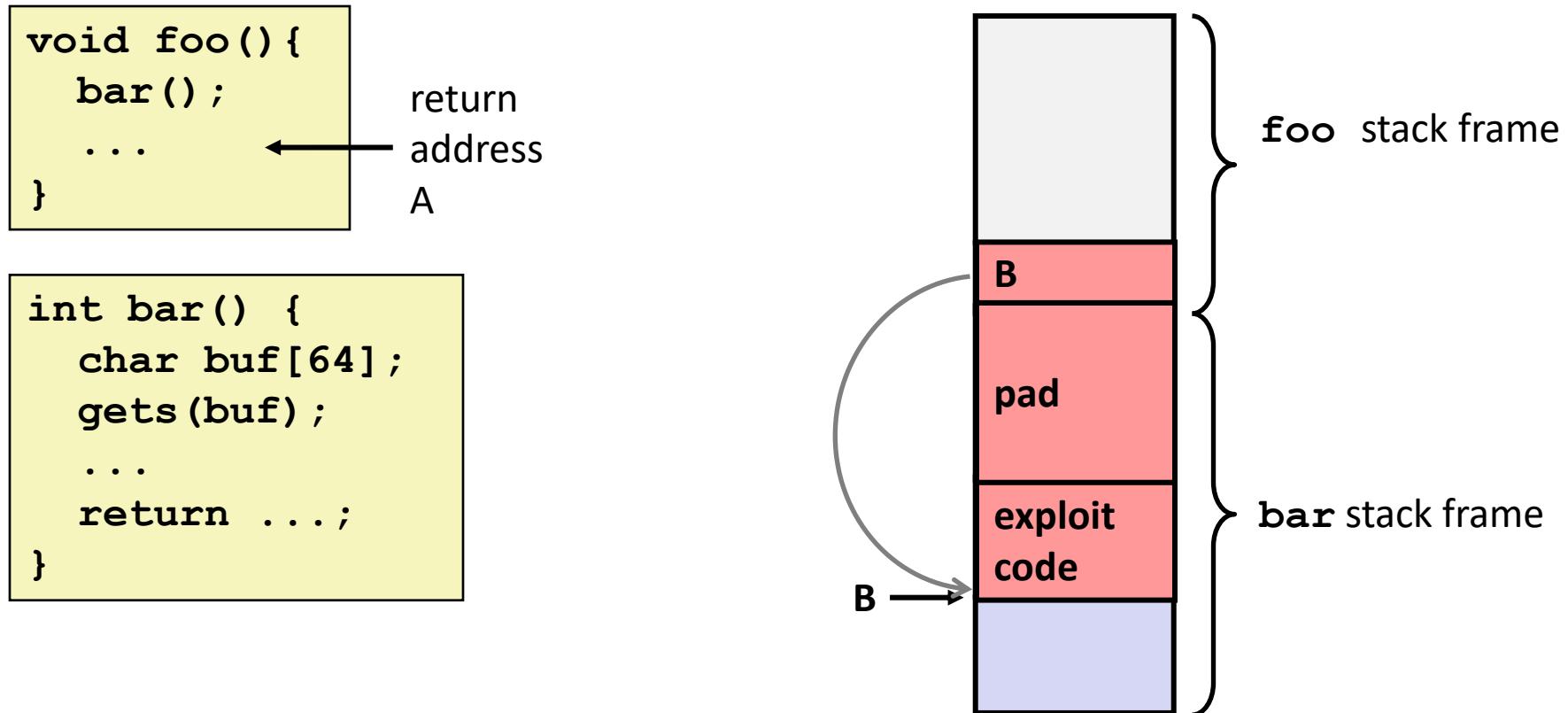
```
unix>./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```

```
80485eb: e8 d5 ff ff ff    call
80485f0: c9                  leave
```

Return address bị ghi đè

```
80485c5 <echo>
# Desired return point
```

Lợi dụng Buffer Overflow với mục đích gây hại



- Chuỗi nhập vào chứa các byte biểu diễn của code thực thi
- Ghi đè `return address A` bằng địa chỉ của **buffer B** (chuỗi đã nhập), chính xác hơn là vị trí bắt đầu của những byte code thực thi
- Khi `bar()` thực thi lệnh `ret`, sẽ nhảy đến vị trí B để thực thi code

Các tấn công dựa trên Buffer Overflows (1)

- *Buffer overflow cho phép các máy tính từ xa thực thi các code mong muốn trên máy tính nạn nhân*
- Internet worm
 - Các phiên bản đầu tiên của finger server (fingerd) sử dụng `gets()` để đọc các tham số gửi từ phía client:
 - `finger droh@cs.cmu.edu`
 - Worm tấn công fingerd server bằng cách gửi tham số như sau:
 - `finger "exploit-code padding new-return-address"`
 - exploit code: thực thi một root shell trên máy tính nạn nhân với kết nối TCP đến attacker.

Các tấn công dựa trên Buffer Overflows (2)

- *Buffer overflow cho phép các máy tính từ xa thực thi các code mong muốn trên máy tính nạn nhân*
- IM War
 - AOL khai thác lỗ hổng buffer overflow trên AIM clients
 - exploit code: trả về signature 4-bytes cho server.
 - Khi Microsoft sửa code để khớp signature, AOL thay đổi vị trí signature.

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

....

Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!

Tránh lỗ hổng buffer overflow

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Sử dụng các hàm thư viện có giới hạn độ dài chuỗi
 - **fgets** thay cho **gets**
 - **strncpy** thay cho **strcpy**
 - Không dùng **scanf** với định dạng **%s**
 - Dùng **fgets** để đọc chuỗi
 - Hoặc dùng **%ns** với giá trị **n** phù hợp

Bảo vệ ở mức hệ thống (system-level)

■ Stack offsets ngẫu nhiên

- Khi chạy một chương trình, cấp phát một không gian có kích thước ngẫu nhiên trong stack
- Gây khó khăn cho hacker để đoán được vị trí bắt đầu của exploit code đã thêm

■ Những code segment không thực thi được

- Trong hệ thống x86, có thể đánh dấu cho các vùng nhớ là “read-only” hay “writeable”
 - Có thể thực thi bất cứ thứ gì đọc được
- x86-64 thêm quyền thực thi “execute” trên các vùng nhớ

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xfffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xfffffb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xfffffc6a8
```

Stack Canaries

■ Ý tưởng

- Đặt một giá trị đặc biệt (canary) trong stack nằm bên ngoài buffer
- Kiểm tra có bị ghi đè hay không trước khi thoát hàm

■ Hỗ trợ trong GCC

- **-fstack-protector**
- **-fstack-protector-all**

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

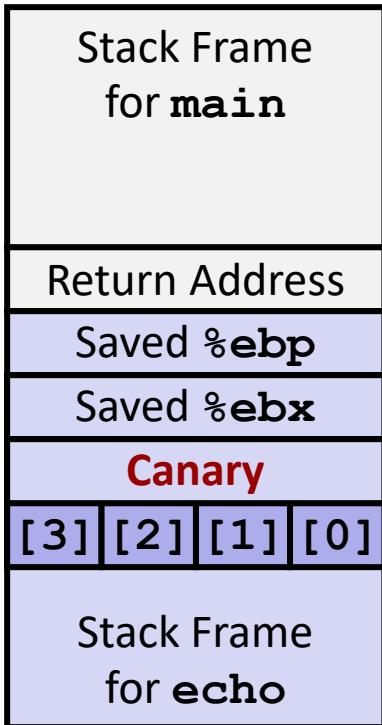
Buffer được bảo vệ: Disassembly

echo:

804864d:	55	push	%ebp
804864e:	89 e5	mov	%esp, %ebp
8048650:	53	push	%ebx
8048651:	83 ec 14	sub	\$0x14, %esp
8048654:	65 a1 14 00 00 00	mov	%gs:0x14, %eax
804865a:	89 45 f8	mov	%eax, 0xfffffffff8(%ebp)
804865d:	31 c0	xor	%eax, %eax
804865f:	8d 5d f4	lea	0xfffffffff4(%ebp), %ebx
8048662:	89 1c 24	mov	%ebx, (%esp)
8048665:	e8 77 ff ff ff	call	80485e1 <gets>
804866a:	89 1c 24	mov	%ebx, (%esp)
804866d:	e8 ca fd ff ff	call	804843c <puts@plt>
8048672:	8b 45 f8	mov	0xfffffffff8(%ebp), %eax
8048675:	65 33 05 14 00 00 00	xor	%gs:0x14, %eax
804867c:	74 05	je	8048683 <echo+0x36>
804867e:	e8 a9 fd ff ff	call	804842c <FAIL>
8048683:	83 c4 14	add	\$0x14, %esp
8048686:	5b	pop	%ebx
8048687:	5d	pop	%ebp
8048688:	c3	ret	

Thiết lập Canary

Before call to gets



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

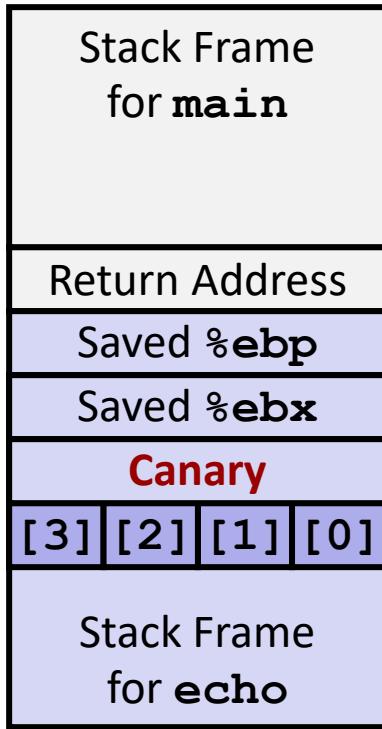
buf

`echo:`

```
    . . .
    movl    %gs:20, %eax      # Get canary
    movl    %eax, -8(%ebp)    # Put on stack
    xorl    %eax, %eax       # Erase canary
    . . .
```

Kiểm tra Canary

Before call to gets

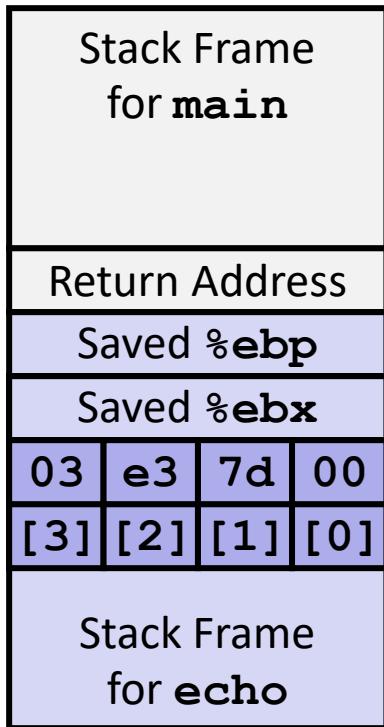


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

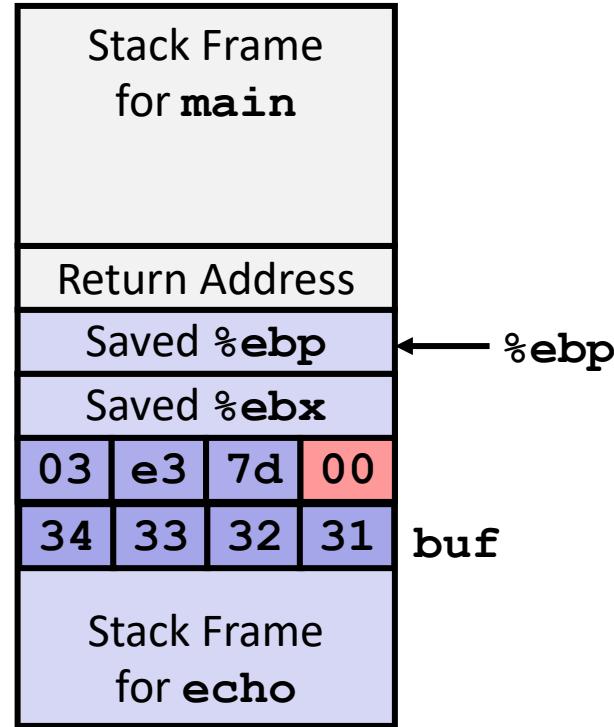
```
echo:
    . . .
    movl    -8(%ebp), %eax      # Retrieve from stack
    xorl    %gs:20, %eax       # Compare with Canary
    je     .L24                 # Same: skip ahead
    call    __stack_chk_fail   # ERROR
.L24:
    . . .
```

Canary – Có chắc luôn an toàn?

Before call to gets



Input 1234



```
(gdb) break echo  
(gdb) run  
(gdb) stepi 3  
(gdb) print /x *((unsigned *) $ebp - 2)  
$1 = 0x3e37d00
```

Canary bị ghi đè nhưng vẫn
bình thường!

Nội dung

- Union
- Buffer overflow
 - Lỗ hổng
 - Biện pháp
- Switch (tụ tìm hiểu)

```
int switch_eg(int x, int y, int z)
{
    int w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Ví dụ về Switch

- Case thông thường:
 - Case 1, 3
- Nhiều case cùng chung đoạn code:
 - Case 5 & 6
- Các case thực thi fallthrough (không break)
 - Case 2
- Case bị khuyết
 - Case 4?

```
int switch_eg(int x, int y, int z)
{
    int w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

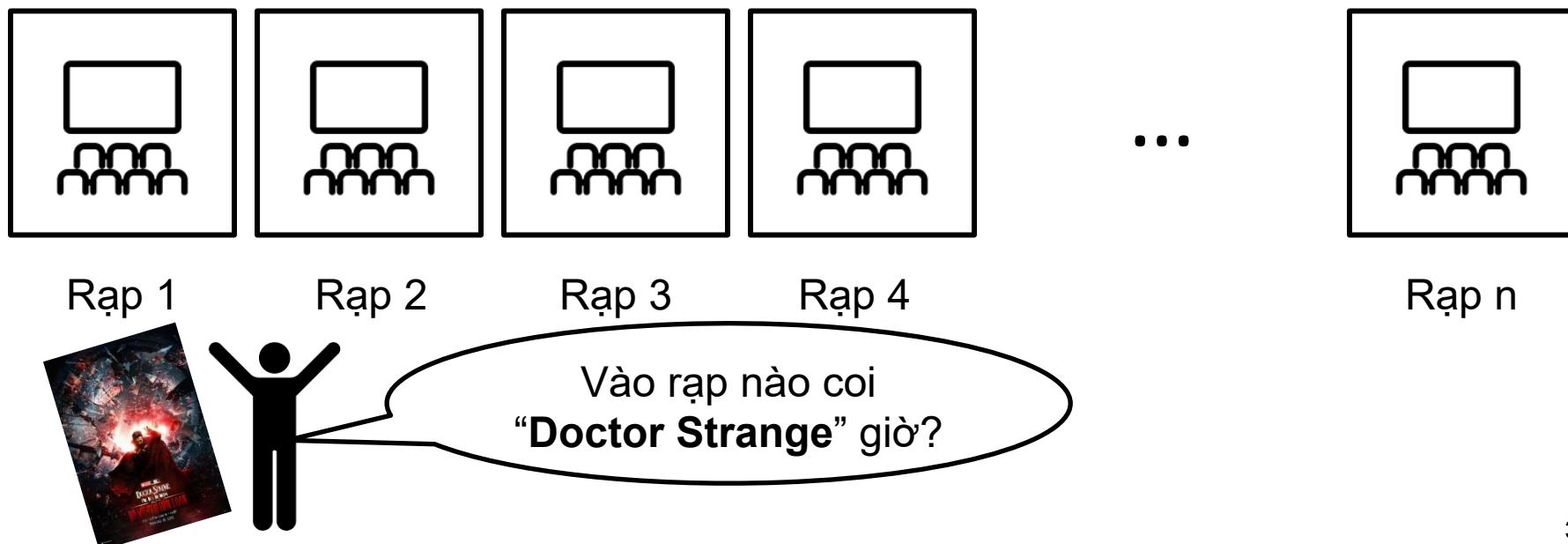
Switch sang If?

```
int switch_eg(int x, int y,
int z)
{
    int w = 1;
    if (x==1)
        w = y*z;
    else if (x == 2)
    {
        w = y/z;
        w += z;
    }
    else if (x==3)
        w += z;
    else if (x==5 || x == 6)
        w -= z;
    else
        w = 2;
    return w;
}
```

Ví dụ: Kiểm soát vé tại rạp chiếu phim

■ Ngũ cảnh: 1 rạp chiếu phim có n rạp

- Mỗi rạp chiếu 1 phim
- Vé chỉ ghi tên phim (không ghi tên rạp)
- Cần có người kiểm tra và hướng dẫn người xem đến đúng rạp

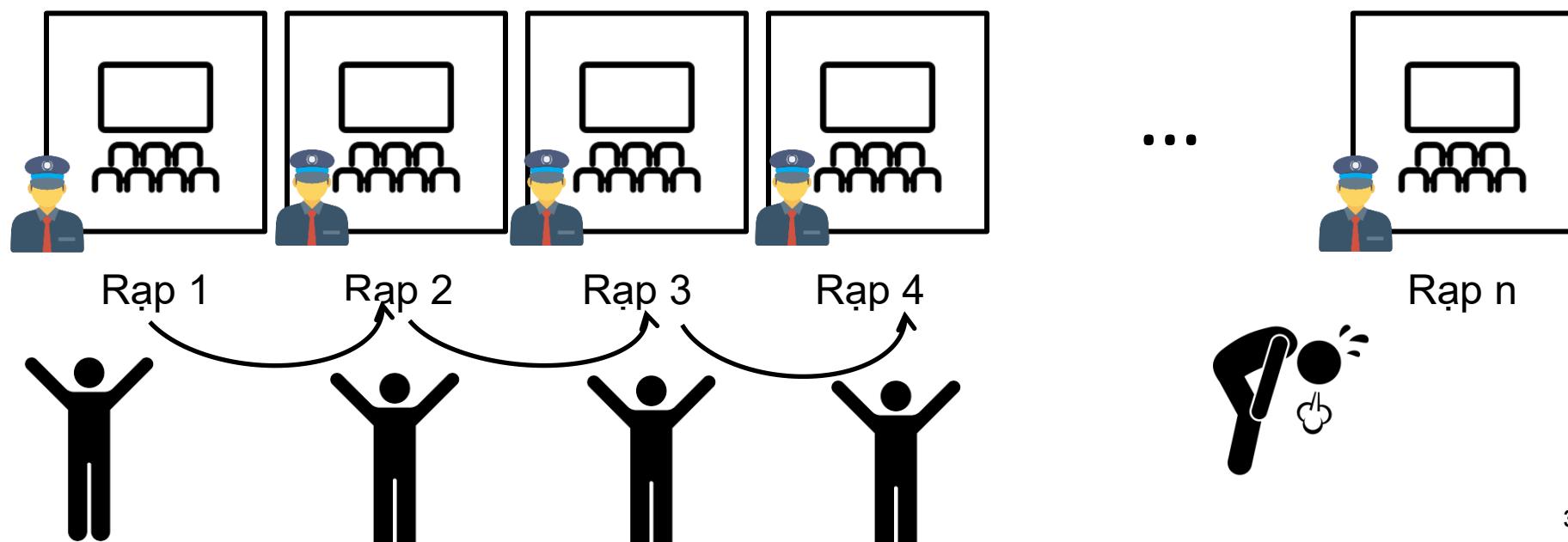


Ví dụ: Kiểm soát vé tại rạp chiếu phim

■ Giải pháp 1:

- Thuê ở mỗi rạp 1 nhân viên kiểm vé
- Người xem đến mỗi rạp xuất trình vé, nếu đúng tên phim của rạp thì vào xem, không thì sang rạp kế tiếp.
- Kiểm tra đến khi nào đến đúng rạp.

Trường hợp tệ nhất, hỏi hết n rạp

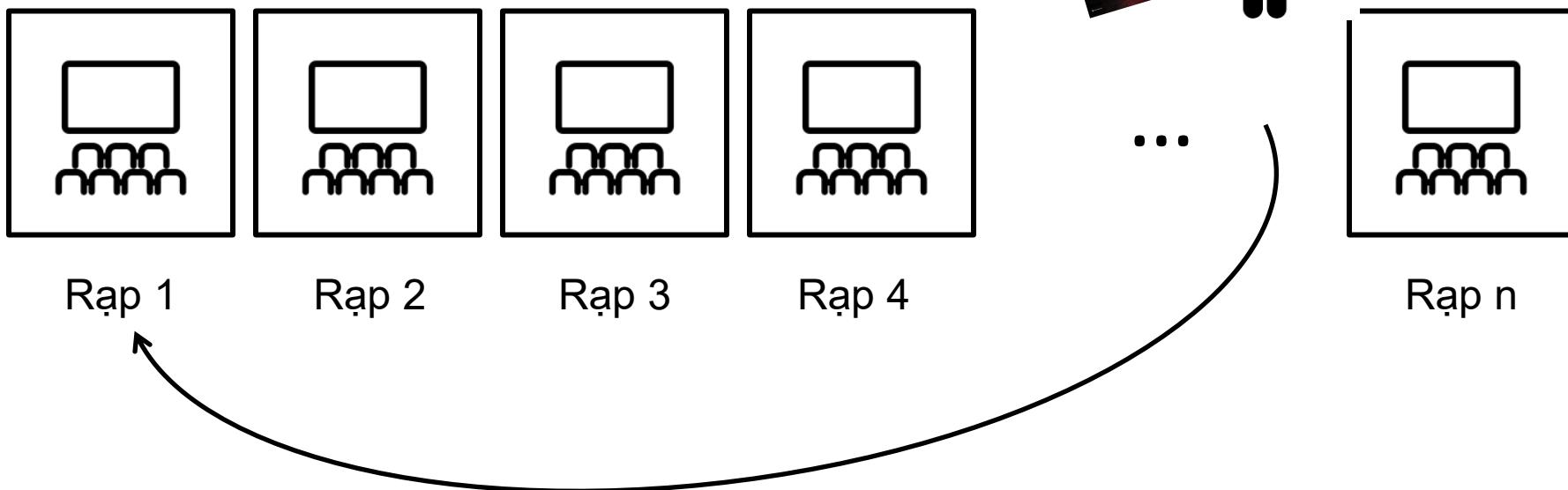


Ví dụ: Kiểm soát vé tại rạp chiếu phim

■ Giải pháp 2:

- Chỉ thuê 1 nhân viên soát vé ở cổng vào.
- Có 1 bảng ánh xạ tên phim – rạp để hướng dẫn người xem vào rạp nào.
- Người xem có vé sau đó có thể vào trực tiếp rạp đã được hướng dẫn.

Doctor Strange	Rạp 1
Thỏ gà	Rạp 2
Ngôi đền kỳ quái	Rạp 3



Ví dụ: Kiểm soát vé tại rạp chiếu phim

■ Giải pháp 1:

- Có thể thích hợp nếu số rạp ít.
- Phải kiểm tra nhiều lần
- Giống if/else với tất cả case của switch

■ Giải pháp 2:

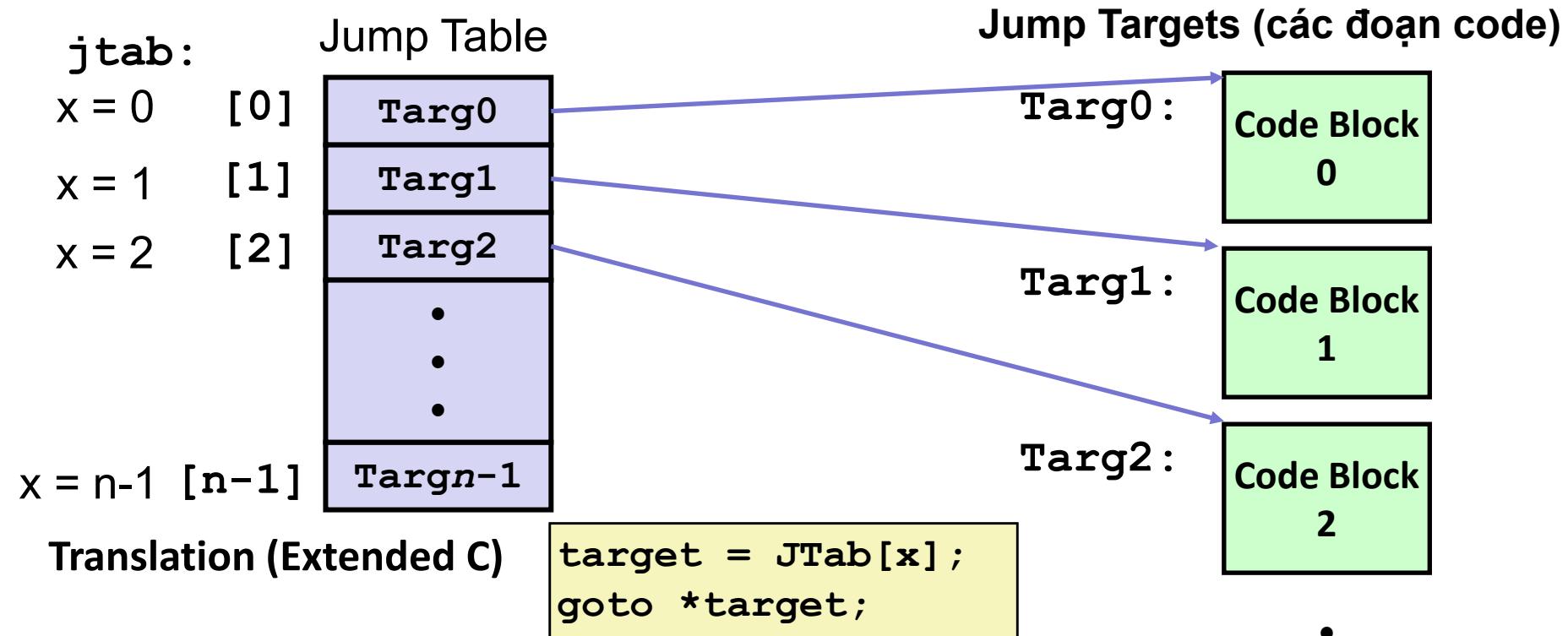
- “Thông minh hơn”
- Chỉ kiểm tra 1 lần
- Cần có bảng ánh xạ giữa phim – rạp, có thể mở rộng dễ dàng

→ Giải pháp 2 tương tự ý tưởng của switch trong assembly

Từ kiểm soát vé sang switch

- Kiểm tra vé ở cổng vào \Leftrightarrow Kiểm tra giá trị một biến với switch(x)
- Các rạp chiếu phim \Leftrightarrow Vị trí đoạn code cần thực thi trong từng trường hợp
- Bảng ánh xạ phim - rạp \Leftrightarrow Bảng ánh xạ các giá trị x với vị trí đoạn code tương ứng cần thực thi (**Jump table**)

Jump Table trong IA32



- Jump Table là 1 mảng, mỗi phần tử là địa chỉ của các block code ứng với các trường hợp của switch.

Các case của x được dùng như “index” để truy Targn-1 : xuất đến đúng entry tương ứng trong mảng.

Code Block
n-1

Switch: Ví dụ (IA32)

Translation (Extended C)

```
target = JTab[x];  
goto *target;
```

Assembly code:

switch_eg:

```
...  
movl 8(%ebp), %eax # %eax = x  
cmpl $6, %eax # Compare x:6  
ja .L2 # If unsigned > goto default  
jmp * .L7(,%eax,4) # Goto *JTab[x]  
.....
```

*Indirect
jump*

Jump table

.section	.rodata
.align 4	
.L7:	
.long	.L2 # x = 0
.long	.L3 # x = 1
.long	.L4 # x = 2
.long	.L5 # x = 3
.long	.L2 # x = 4
.long	.L6 # x = 5
.long	.L6 # x = 6

Label của block code

Khoảng giá trị nào
ứng với case default?

Lấy địa chỉ thứ x trong mảng jump
table (JTab)

Switch trong assembly: Giải thích

■ Jump table

- Mỗi địa chỉ đích cần 4 bytes
- Địa chỉ nền (base address) ở .L7

■ Các lệnh nhảy

- Direct: **jmp .L2**
- Vị trí cần nhảy đến là label .L2
- Indirect: **jmp *.L7(%eax,4)**
- Vị trí bắt đầu của jump table: .L7
- Phải nhân với hệ số 4 (kích thước mỗi địa chỉ là 32-bits = 4 Bytes trong IA32)
- Lấy ra một địa chỉ đích từ .L7 + eax*4
 - Chỉ trong các trường hợp $0 \leq x \leq 6$

Jump table

```
.section    .rodata
.align 4
.L7:
.long     .L2 # x = 0
.long     .L3 # x = 1
.long     .L4 # x = 2
.long     .L5 # x = 3
.long     .L2 # x = 4
.long     .L6 # x = 5
.long     .L6 # x = 6
```

Jump Table

Jump table

```
.section .rodata
.align 4
.L7:
.long .L2 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L2 # x = 4
.long .L6 # x = 5
.long .L6 # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L4
        w = y/z;
        /* Fall Through */
    case 3:          // .L5
        w += z;
        break;
    case 5:
    case 6:          // .L6
        w -= z;
        break;
    default:         // .L2
        w = 2;
}
```

Xử lý trường hợp Fall-Through

```
int w = 1;  
. . .  
switch(x) {  
. . .  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
. . .  
}
```

case 3:

```
w = 1;  
goto merge;
```

case 2:

```
w = y/z;
```

merge:

```
w += z;
```

Ví dụ các Code Blocks (1)

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
    . . .  
    case 3:          // .L5  
        w += z;  
        break;  
    . . .  
    default:         // .L2  
        w = 2;  
}
```

The diagram illustrates the control flow from the switch statement to three separate code blocks. A central point of entry branches into three paths: one leading to .L2 (Default), one to .L3 (# x == 1), and one to .L5 (# x == 3). The .L2 path contains code for w = 2. The .L3 path contains code for calculating w = y*z. The .L5 path contains code for w = 1.

```
.L2:      # Default  
    movl $2, %eax # w = 2  
    jmp  .L8       # Goto done  
  
.L5:      # x == 3  
    movl $1, %eax # w = 1  
    jmp  .L9       # Goto merge  
  
.L3:      # x == 1  
    movl 16(%ebp), %eax # z  
    imull 12(%ebp), %eax # w = y*z  
    jmp  .L8       # Goto done
```

```
.section  .rodata  
.align 4  
.L7:  
.long   .L2 # x = 0  
.long   .L3 # x = 1  
.long   .L4 # x = 2  
.long   .L5 # x = 3  
.long   .L2 # x = 4  
.long   .L6 # x = 5  
.long   .L6 # x = 6
```

Ví dụ các Code Blocks (2)

```
switch(x) {  
    . . .  
    case 2: // .L4  
        w = y/z;  
        /* Fall Through */  
    merge: // .L9  
        w += z;  
        break;  
    case 5:  
    case 6: // .L6  
        w -= z;  
        break;  
}
```

```
.L4: # x == 2  
    movl 12(%ebp), %edx  
    movl %edx, %eax  
    sarl $31, %edx  
    idivl 16(%ebp) # w = y/z  
  
.L9: # merge:  
    addl 16(%ebp), %eax # w += z  
    jmp .L8 # goto done  
  
.L6: # x == 5, 6  
    movl $1, %eax # w = 1  
    subl 16(%ebp), %eax # w = 1-z
```

```
.section .rodata  
.align 4  
.L7:  
.long .L2 # x = 0  
.long .L3 # x = 1  
.long .L4 # x = 2  
.long .L5 # x = 3  
.long .L2 # x = 4  
.long .L6 # x = 5  
.long .L6 # x = 6
```

Switch Code (Kết thúc)

```
return w;
```

```
.L8:    # done:  
        popl %ebp  
        ret
```

■ Lưu ý

- Sử dụng jump table để xử lý các trường hợp khuyết hoặc trùng
- Sử dụng trình tự chương trình để xử lý trường hợp fall-through
- Không khởi tạo $w = 1$ trừ khi cần thiết

Switch trong x86-64?

- Chung ý tưởng hiện thực, tùy chỉnh code theo 64-bit
- Jump Table chứa các địa chỉ 64 bits (pointers)

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
        . . .  
}
```

```
.L3:  
    movq    %rdx, %rax  
    imulq   %rsi, %rax  
    ret
```

Jump Table

```
.section .rodata  
.align 8  
.L7:  
.quad    .L2      # x = 0  
.quad    .L3      # x = 1  
.quad    .L4      # x = 2  
.quad    .L5      # x = 3  
.quad    .L2      # x = 4  
.quad    .L6      # x = 5  
.quad    .L6      # x = 6
```

IA32 Object Code

■ Setup

- Label .L2 (default) chuyển thành địa chỉ 0x8048422
- Label .L7 (base của jump table) chuyển thành địa chỉ 0x8048660

Assembly Code

```
switch_eg:  
    . . .  
    ja      .L2          # If unsigned > goto default  
    jmp    * .L7(,%eax,4) # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:  
    . . .  
8048419: 77 07          ja      8048422 <switch_eg+0x12>  
804841b: ff 24 85 60 86 04 08 jmp    *0x8048660(,%eax,4)
```

IA32 Object Code (tt)

■ Jump Table

- Không hiển thị trong disassembled code
- Có thể xem được với lệnh GDB

`gdb switch`

`(gdb) x/7xw 0x8048660`

- Examine 7 hexadecimal format “words” (mỗi word 4 bytes)
- Sử dụng lệnh “`help x`” để biết thêm format

`0x8048660:` `0x08048422` `0x08048432` `0x0804843b` `0x08048429`

`0x8048670:` `0x08048422` `0x0804844b` `0x0804844b`

IA32 Object Code (tt)

■ Phân tích Jump Table

0x8048660 : 0x08048422 **0x8048670 :** 0x08048422 **0x8048660 :** 0x08048432 **0x8048664 :** 0x08048432 **0x8048668 :** 0x0804843b
0x8048670 : 0x0804844b **0x8048674 :** 0x0804844b **0x8048678 :** 0x0804844b **0x804867c :** 0x0804844b

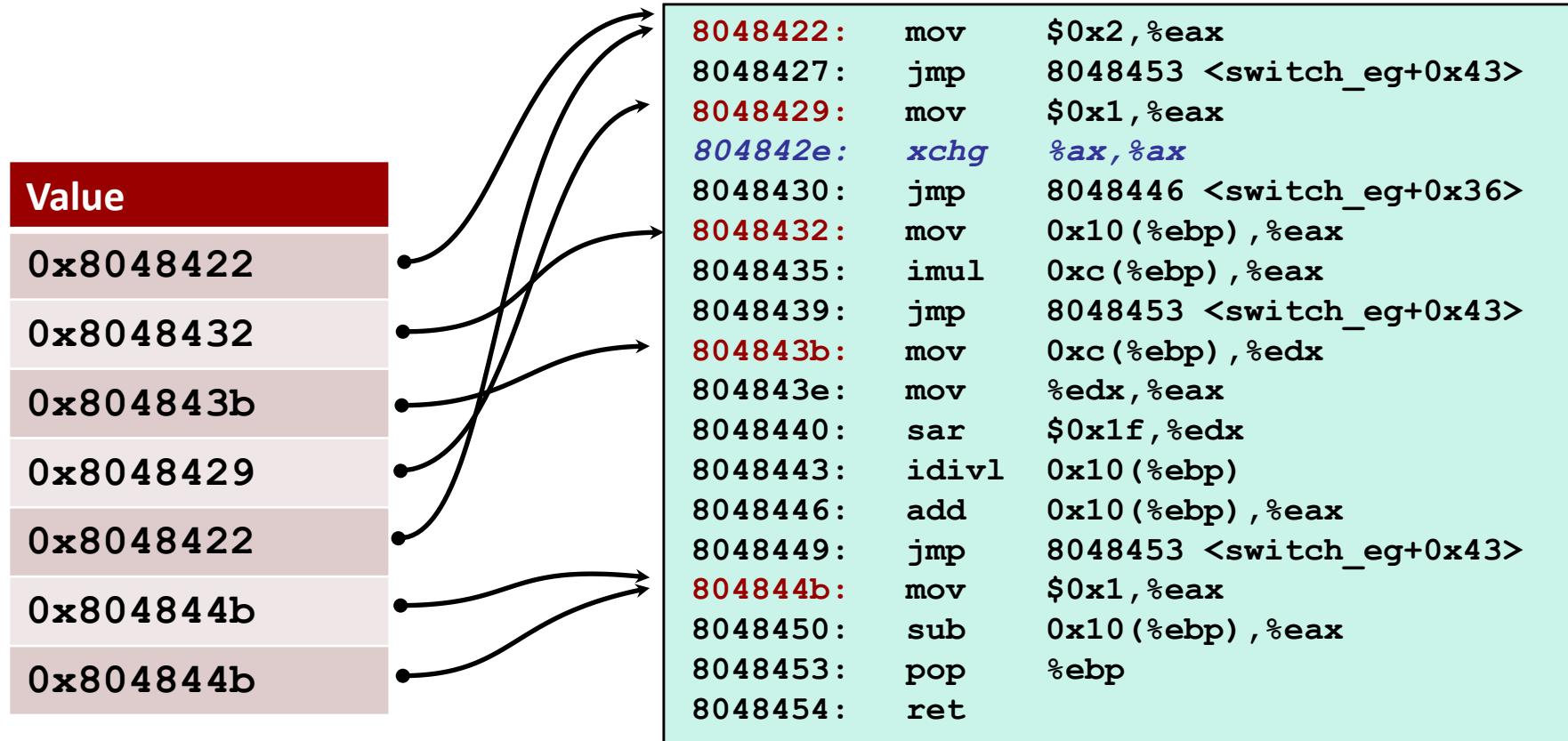
Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	1
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

Disassembled Targets – Các block code

0x8048660: 0x08048422
0x8048670: 0x08048422 0x08048432 0x0804843b 0x08048429

→	8048422: b8 02 00 00 00	mov \$0x2,%eax
→	8048427: eb 2a	jmp 8048453 <switch_eg+0x43>
→	8048429: b8 01 00 00 00	mov \$0x1,%eax
→	804842e: 66 90	xchg %ax,%ax # noop
→	8048430: eb 14	jmp 8048446 <switch_eg+0x36>
→	8048432: 8b 45 10	mov 0x10(%ebp),%eax
→	8048435: 0f af 45 0c	imul 0xc(%ebp),%eax
→	8048439: eb 18	jmp 8048453 <switch_eg+0x43>
→	804843b: 8b 55 0c	mov 0xc(%ebp),%edx
→	804843e: 89 d0	mov %edx,%eax
→	8048440: c1 fa 1f	sar \$0x1f,%edx
→	8048443: f7 7d 10	idivl 0x10(%ebp)
→	8048446: 03 45 10	add 0x10(%ebp),%eax
→	8048449: eb 08	jmp 8048453 <switch_eg+0x43>
→	804844b: b8 01 00 00 00	mov \$0x1,%eax
→	8048450: 2b 45 10	sub 0x10(%ebp),%eax
→	8048453: 5d	pop %ebp
→	8048454: c3	ret

Matching Disassembled Targets



Switch: Thêm

■ Giả sử x trong khoảng từ 10 đến 14?

- Bao nhiêu entry trong Jump Table?

? 15 entry cho các trường hợp từ 0 đến 14
(tính cả trường hợp khuyết)??

Assembly code:

```
subl    $10, %eax
cmpb    $4, %eax
ja     .L2
movl    .L4(%eax,4), %eax
jmp    *%eax
```

.L4:

```
.long   .L8
.long   .L7
.long   .L6
.long   .L5
.long   .L3
```

```
int result = 0;
switch(x)
{
    case 10: ...
    case 11: ...
    case 12: ...
    case 13: ...
    case 14: ...
    default: ...
}
```

Sử dụng phép trừ để chuyển khoảng giá trị của x về khoảng index có thể truy xuất jump table
 $x \in [10, 14] \rightarrow x \in [0, 4]$

Switch: Thêm

■ Tìm các giá trị của các case trong switch?

```
x at %ebp+8
```

1	movl 8(%ebp), %eax	Jump table for switch2		
	<i>Set up jump table access</i>	1	.L8:	switch(x)
2	addl \$2, %eax	2	.long .L3	{
3	cmpl \$6, %eax	3	.long .L2	case -2:...
4	ja .L2	4	.long .L4	case 0:...
5	jmp * .L8(,%eax,4)	5	.long .L5	case 1:...
		6	.long .L6	case 2:...
		7	.long .L6	case 3:...
		8	.long .L7	case 4:...
				default:...

- Dòng assembly 2: **index = x + 2**
- Dòng assembly 3 & 4: nếu index lớn hơn 6 thì nhảy đến .L2 → **.L2 là trường hợp default**
- Truy xuất jump table với index
 - 7 entry trong jump table → 7 trường hợp giá trị của index từ 0 – 6
 - Tuy nhiên index = 1 có chung nhãn .L2 với trường hợp default → index = 1 là trường hợp khuyết

Các case của index: 0, 2, 3, 4, 5, 6

Các case của x: -2, 0, 1, 2, 3, 4

Assembly code

```
//x at %ebp+12, y at %ebp+16, n at %ebp+8
1.      movl    8(%ebp), %eax          // n
2.      movb    %al, -20(%ebp)        // ?
3.      movl    $0, -4(%ebp)         // z
4.      movsbl  -20(%ebp), %eax
5.      subl    $42, %eax
6.      cmpl    $7, %eax
7.      ja     .L2
8.      movl    .L4(%eax,4), %eax
9.      jmp    *%eax
10.     .L5:
11.     movl    12(%ebp), %edx
12.     movl    16(%ebp), %eax
13.     addl    %edx, %eax
14.     movl    %eax, -4(%ebp)
15.     jmp    .L10
16.     .L6:
17.     movl    12(%ebp), %eax
18.     subl    16(%ebp), %eax
19.     movl    %eax, -4(%ebp)
20.     jmp    .L10
21.     .L3:
22.     movl    12(%ebp), %eax
23.     imull  16(%ebp), %eax
24.     movl    %eax, -4(%ebp)
25.     jmp    .L10
```

Switch (*)

```
26. .L7:
27.     movl    12(%ebp), %eax
28.     cltd
29.     idivl   16(%ebp)
30.     movl    %eax, -4(%ebp)
31.     jmp    .L10
32. .L8:
33.     movl    12(%ebp), %eax
34.     movl    %eax, -4(%ebp)
35.     jmp    .L10
36. .L9:
37.     movl    16(%ebp), %eax
38.     movl    %eax, -4(%ebp)
39.     jmp    .L10
40. .L2:
41.     movl    $1, -4(%ebp)
42. .L10:
43.     movl    -4(%ebp), %eax
44.     leave
45.     ret
```

.L4:

```
.long   .L3
.long   .L5
.long   .L2
.long   .L6
.long   .L2
.long   .L7
.long   .L8
.long   .L9
```

Thử viết code C tương ứng?

```
int function(char n,int x, int y)
```

```
{
```

```
    int z = 0;
    switch(n)
    ....
```

Nội dung

■ Các chủ đề chính:

- 1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
- 2) Ngôn ngữ assembly
- 3) Điều khiển luồng trong C với assembly
- 4) Các thủ tục/hàm (procedure) trong C ở mức assembly
- 5) Biểu diễn mảng, cấu trúc dữ liệu trong C
- 6) Một số topic ATTT: reverse engineering, bufferoverflow
- 7) Phân cấp bộ nhớ, cache
- 8) Linking trong biên dịch file thực thi

■ Lab liên quan

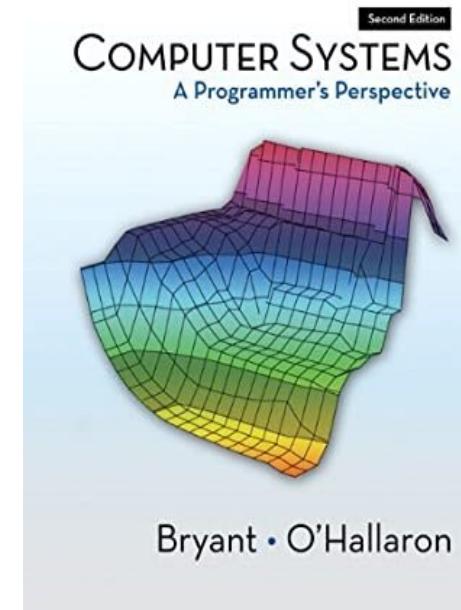
- Lab 1: Nội dung 1
- Lab 2: Nội dung 1, 2, 3
- Lab 3: Nội dung 1, 2, 3, 4, 5, 6
- Lab 4: Nội dung 1, 2, 3, 4, 5, 6
- Lab 5: Nội dung 1, 2, 3, 4, 5, 6
- Lab 6: Nội dung 1, 2, 3, 4, 5, 6

Giáo trình

■ Giáo trình chính

Computer Systems: A Programmer's Perspective

- Second Edition (CS:APP2e), Pearson, 2010
- Randal E. Bryant, David R. O'Hallaron
- <http://csapp.cs.cmu.edu>



■ Tài liệu khác

- *The C Programming Language*, Second Edition, Prentice Hall, 1988
 - Brian Kernighan and Dennis Ritchie
- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 1st Edition, 2008
 - Chris Eagle
- *Reversing: Secrets of Reverse Engineering*, 1st Edition, 2011
 - Eldad Eilam



KEEP
CALM
AND
ENJOY YOUR
SEMESTER :)