

## BÀI 10. AN TOÀN VÙNG NHỚ TIỀN TRÌNH

---

Bùi Trọng Tùng,  
Viện Công nghệ thông tin và Truyền thông,  
Đại học Bách khoa Hà Nội

1

1

## Nội dung

- Lỗi hỏng tràn bộ đệm (Buffer Overflow)
- Lỗi hỏng tràn số nguyên
- Lỗi hỏng xâu định dạng
- Cơ bản về lập trình an toàn

2

2

## 2020 CWE Top 25

- Danh sách 25 lỗi hỏng phần mềm nguy hiểm nhất: 4 trong số Top 10 là dạng lỗi hỏng truy cập bộ nhớ
  - +1 lỗi hỏng liên quan: CWE-20

Rank	ID	Name	Score
[1]	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	<a href="#">CWE-787</a>	Out-of-bounds Write	46.17
[3]	<a href="#">CWE-20</a>	Improper Input Validation	33.47
[4]	<a href="#">CWE-125</a>	Out-of-bounds Read	26.50
[5]	<a href="#">CWE-119</a>	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	<a href="#">CWE-200</a>	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	<a href="#">CWE-416</a>	Use After Free	18.87
[9]	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)	17.29
[10]	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44

3

3

## 1. TỔNG QUAN VỀ TIỀN TRÌNH (NHẮC LẠI)

Bùi Trọng Tùng,  
Viện Công nghệ thông tin và Truyền thông,  
Đại học Bách khoa Hà Nội

4

4

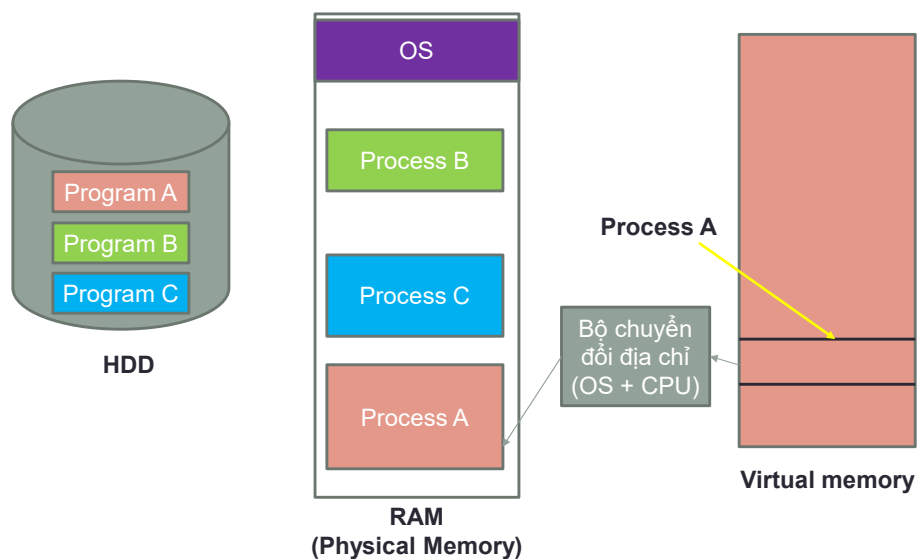
## Tiến trình(process) là gì?

- Tiến trình(process)  $\neq$  chương trình(program)
- Là chương trình đang được thực hiện
- Các tài nguyên tối thiểu của tiến trình:
  - Vùng nhớ được cấp phát
  - Con trỏ lệnh(Program Counter)
  - Các thanh ghi của CPU
- Khối điều khiển tiến trình(Process Control Block-PCB):  
Cấu trúc chứa thông tin của tiến trình

5

5

## Cấp phát bộ nhớ cho tiến trình ntn?

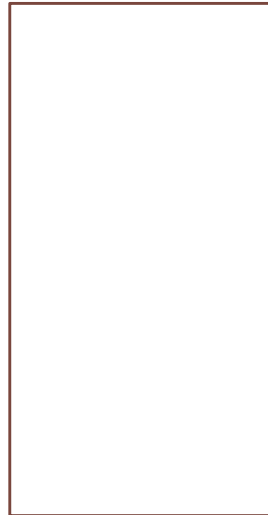


6

6

## Bộ nhớ của tiến trình(Linux 32-bit)

Tiến trình coi bộ nhớ thuộc toàn bộ sở hữu của nó



0xffffffff

Thực tế đây là bộ nhớ ảo với địa chỉ ảo, sẽ được HĐH/CPU ánh xạ sang địa chỉ vật lý

0x00000000

7

7

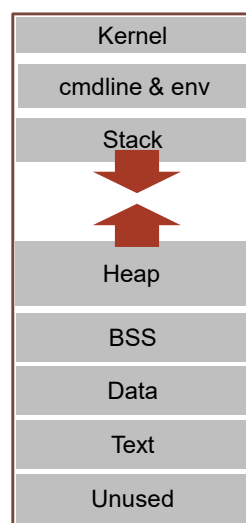
## Bộ nhớ của tiến trình(Linux 32-bit)

Thiết lập khi tiến trình bắt đầu

Thay đổi khi thực thi

Xác định ở thời điểm biên dịch

Không gian địa chỉ của thiết bị vào-ra



0xffffffff

0xc0000000

0x08048000

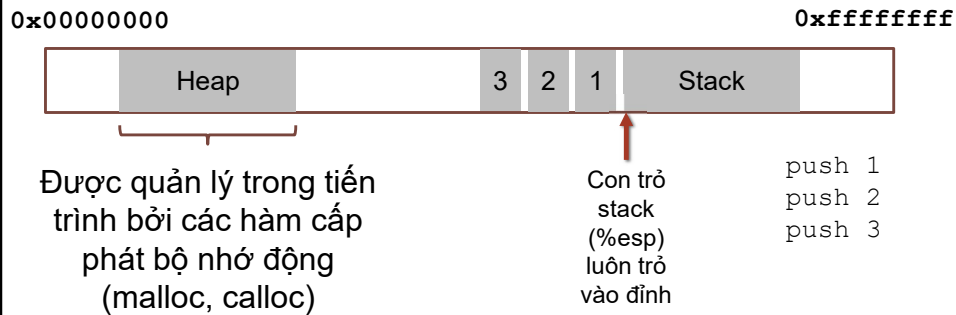
0x00000000

8

8

## Vùng nhớ stack và heap

Trình biên dịch cung cấp các hàm làm thay đổi kích thước vùng nhớ stack khi thực thi chương trình



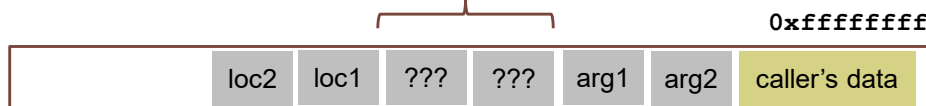
9

9

## Stack - Thực hiện lời gọi hàm

```
void func(char *arg1, int arg2)
{
    char loc1[4];
    int loc2;
}
```

8 byte giữa các tham số và các biến



Các cục bộ được đưa vào stack theo thứ tự

Các tham số đưa vào stack theo thứ tự ngược

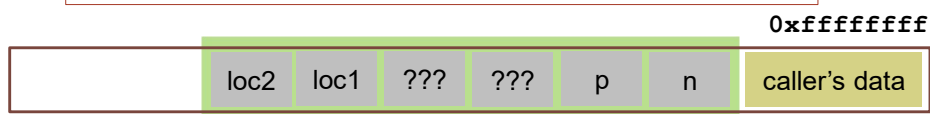
10

10

## Stack frame

```
void func(char *arg1, int arg2)
{
    char loc1[4];
    int loc2;
}
```

```
caller(...)
{
    func(p, n);
}
```



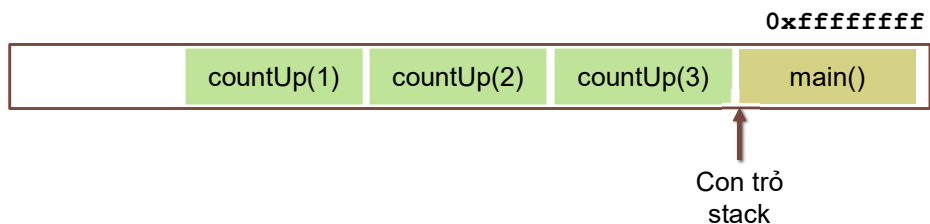
Stack frame: Một phần của vùng nhớ stack tương ứng với lời gọi của một hàm

11

11

## Stack frame

```
void main(){ countUp(3); }
void countUp(int n)
{
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```



12

12

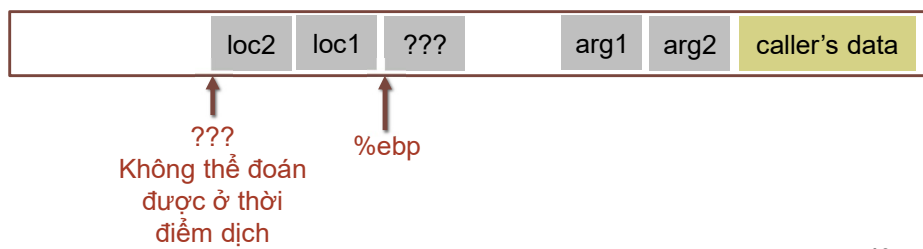
## Stack frame

```
void func(char *arg1, int arg2)
{
    char loc1[4];
    int loc2;
    loc2++;
}
```

Q: loc2 nằm ở đâu?

- %ebp: con trỏ cơ sở để xác định địa chỉ ô nhớ trong stack
- (%ebp): nội dung vùng nhớ trỏ bởi %ebp

0xffffffff



13

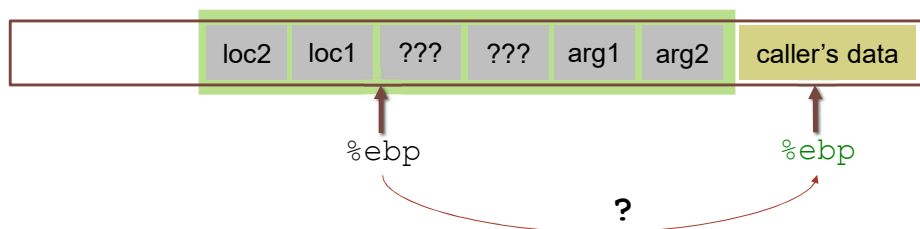
13

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi

0xffffffff



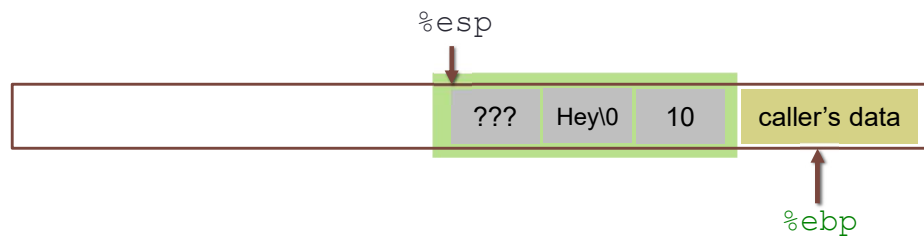
14

14

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



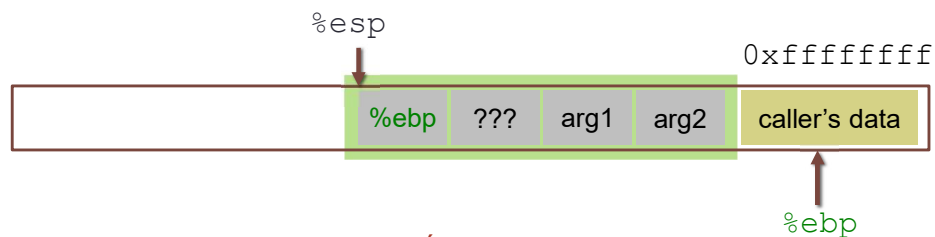
15

15

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



1. Đưa `%ebp` vào stack trước biến cục bộ (`pushl %ebp`)

16

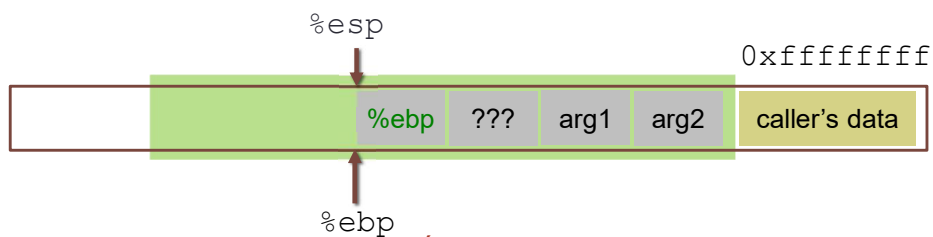
16



## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



1. Đưa %ebp vào stack trước biến cục bộ (pushl %ebp)
2. Thiết lập %ebp bằng với %esp (movl %esp %ebp)

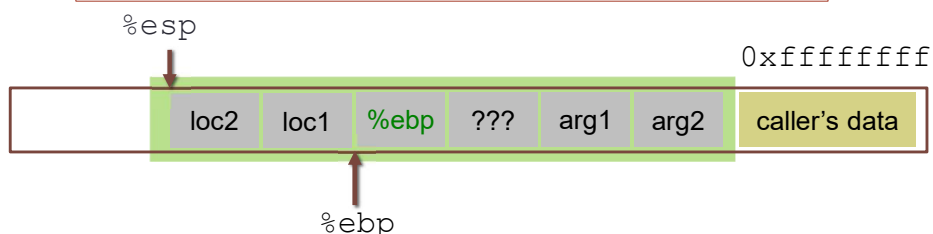
17

17

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



1. Đưa %ebp vào stack trước biến cục bộ (pushl %ebp)
2. Thiết lập %ebp bằng với %esp (movl %esp %ebp)

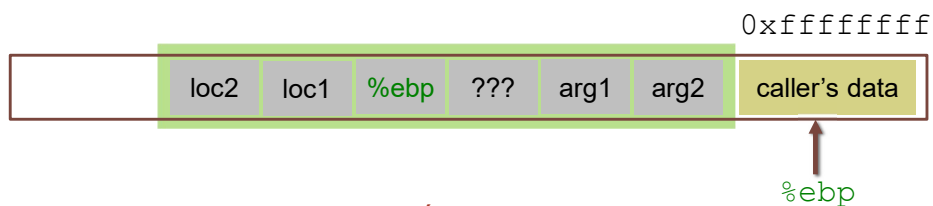
18

18

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để thực thi tiếp lệnh sau khi hàm trả về

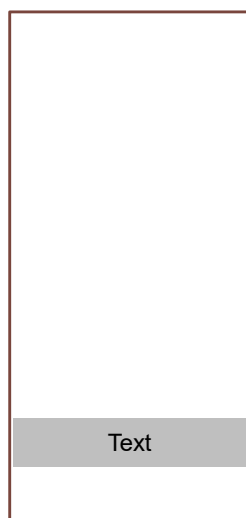


1. Đưa %ebp vào stack trước biến cục bộ (`pushl %ebp`)
2. Thiết lập %ebp bằng với %esp (`movl %esp %ebp`)
3. Khi hàm trả về, thiết lập %ebp bằng (%ebp) (`movl (%ebp) %ebp`)

19

19

## Con trỏ lệnh - %eip



```
...
0x5bf mov %esp,%ebp
0x5be push %ebp
...
```

```
...
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..., (%esp)
0x493 movl $0xa,0x4(%esp) ← %eip
...
```

20

20

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



Đưa %eip của lệnh tiếp theo vào stack trước khi gọi hàm

21

21

## Stack – Trả về từ hàm

```
int main()
{
    ...
    func("Hey", 10);
    ...
}
```

Q: Làm cách nào để khôi phục %ebp của hàm gọi



Thiết lập %eip bằng 4(%ebp) khi trả về

Đưa %eip của lệnh tiếp theo vào stack trước khi gọi hàm

22

22

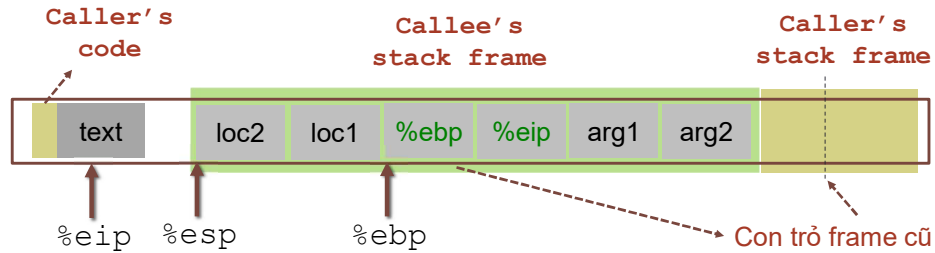
## Stack – Trả về từ hàm

Trong C

```
return;
```

Mã assembly sau khi dịch

```
leave:  mov %ebp %esp
        pop %ebp
ret:    pop %eip
```



23

23

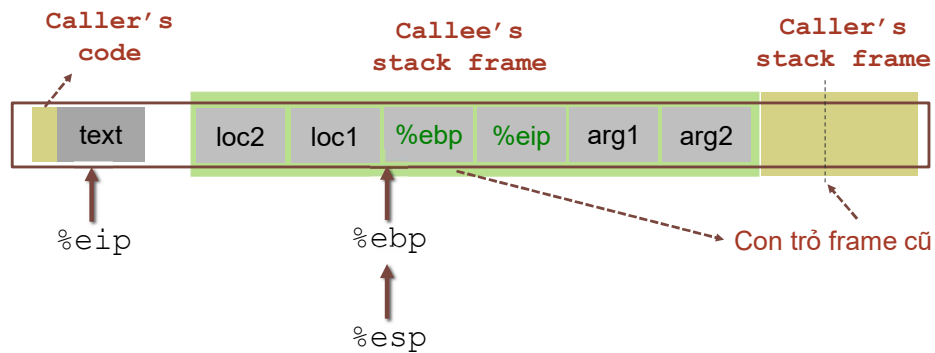
## Stack – Trả về từ hàm

Trong C

```
return;
```

Mã assembly sau khi dịch

```
leave:  ➡ mov %ebp %esp
        pop %ebp
ret:    pop %eip
```



24

24

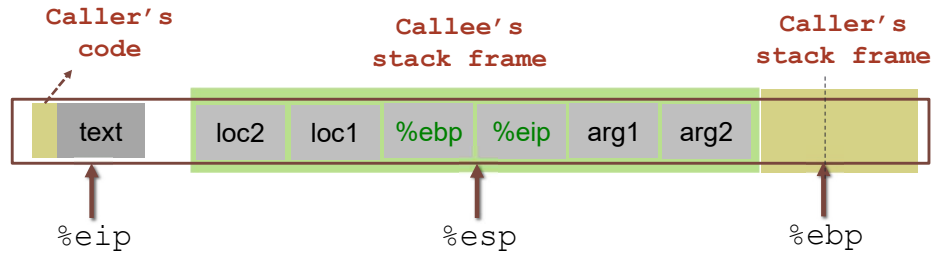
## Stack – Trả về từ hàm

Trong C

```
return;
```

Mã assembly sau khi dịch

```
leave:  mov %ebp %esp
        ➡ pop %ebp
ret:    pop %eip
```



25

25

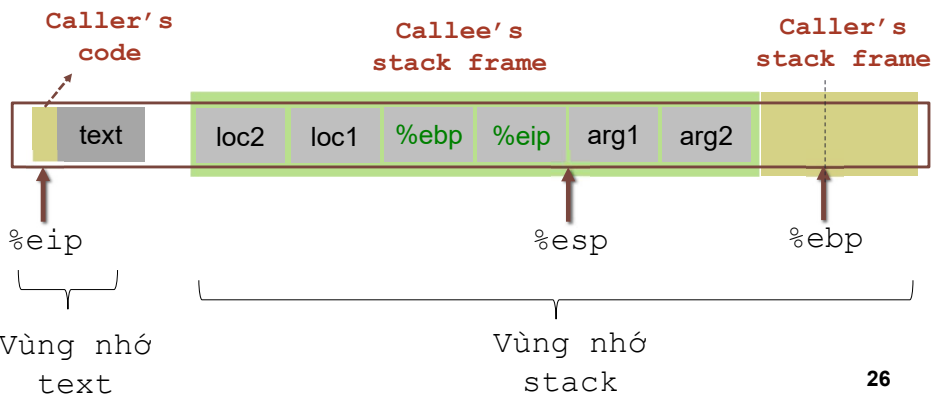
## Stack – Trả về từ hàm

Trong C

```
return;
```

Mã assembly sau khi dịch

```
leave:  mov %ebp %esp
        pop %ebp
ret:    ➡ pop %eip
```



26

26

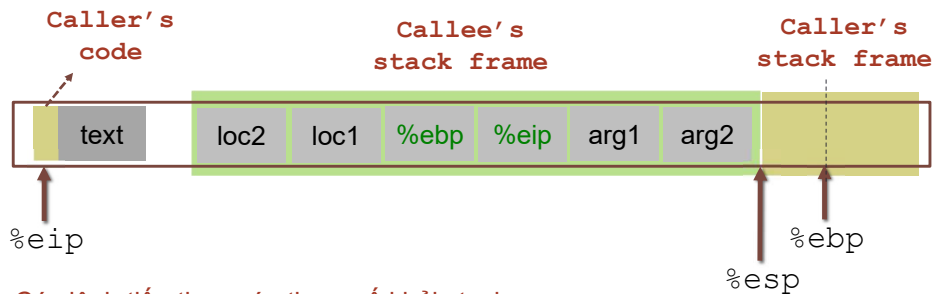
## Stack – Trả về từ hàm

Trong C

```
return;
```

Mã assembly sau khi dịch

```
leave:  mov %ebp %esp
        pop %ebp
ret:    ➡ pop %eip
```



Các lệnh tiếp theo xóa tham số khỏi stack

27

27

## Tổng kết

Gọi hàm

### Hàm gọi(trước khi gọi):

1. Đẩy các tham số vào stack theo thứ tự ngược
2. Đẩy địa chỉ trả về của con trỏ lệnh vào stack, ví dụ %eip + 2
3. Con trỏ lệnh nhảy tới địa chỉ ô nhớ chứa lệnh của hàm được gọi

### Hàm được gọi:

4. Đẩy giá trị của %ebp cũ(%ebp của hàm gọi) vào stack
5. Thiết lập %ebp trở tới đỉnh của stack
6. Đẩy các biến cục bộ vào stack truy cập theo độ lệch từ %ebp

### Hàm được gọi trả về:

7. Thiết lập lại %ebp cũ để %ebp trở vào khung stack của hàm gọi
8. Con trỏ %eip nhảy tới địa chỉ trả về(lệnh tiếp theo sau lệnh gọi hàm trên hàm gọi)

### Hàm gọi:

9. Xóa các tham số khỏi stack

28

28

## 2. TẤN CÔNG TRÀN BỘ ĐỆM

Bùi Trọng Tùng,  
Viện Công nghệ thông tin và Truyền thông,  
Đại học Bách khoa Hà Nội

29

29

## Khái niệm

- Bộ đệm (Buffer): tập hợp liên tiếp các phần tử có kiểu dữ liệu xác định
  - Ví dụ: Trong ngôn ngữ C/C++, xâu là bộ đệm của các ký tự
  - Có thể hiểu theo nghĩa rộng: bộ đệm = vùng nhớ chứa dữ liệu
- Tràn bộ đệm (Buffer Overflow): Đưa dữ liệu vào bộ đệm nhiều hơn khả năng chứa của nó
- Lỗi hỏng tràn bộ đệm: Không kiểm soát kích thước dữ liệu đầu vào.
- Tấn công tràn bộ đệm: Phần dữ liệu tràn ra khỏi bộ đệm làm thay đổi luồng thực thi của tiến trình.
  - Dẫn tới một kết quả ngoài mong đợi
- Ngôn ngữ bị ảnh hưởng: C/C++

30

30

## C/C++ vẫn rất phổ biến(2020)

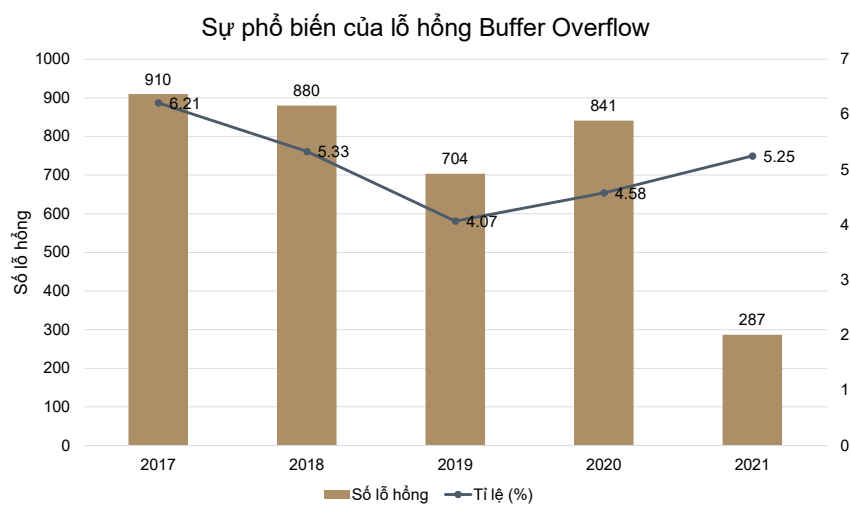
### Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python▼	🌐 🖥️ ⚙️	100.0
2	Java▼	🌐 📱 🖥️	95.3
3	C▼	📱 🖥️ ⚙️	94.6
4	C++▼	📱 🖥️ ⚙️	87.0
5	JavaScript▼	🌐	79.5

31

31

## Sự phổ biến của lỗi hỏng BoF



32

32



## Ví dụ về tràn bộ đệm

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    return;
}
int main()
{
    char *mystr = "AuthMe!";
    ➡ func(mystr);
    ...
}
```

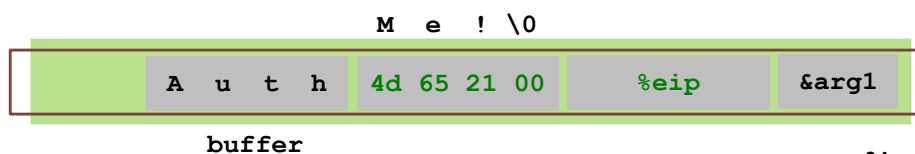


33

33

## Ví dụ về tràn bộ đệm

```
void func(char *arg1)
{
    char buffer[4];
    ➡ strcpy(buffer, arg1);
    return;
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



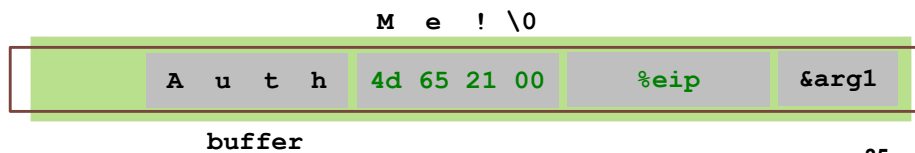
34

34

## Ví dụ về tràn bộ đệm

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ➡ return;
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

**pop %ebp**    **%ebp = 0x0021654d**  
➔ **SEGMENTATION FAULT**



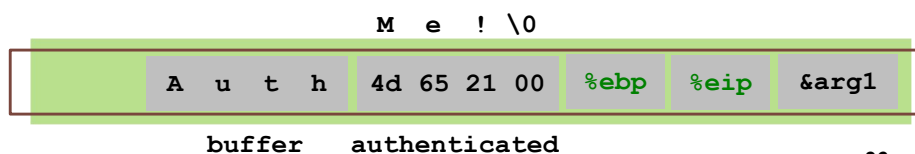
35

35

## Tràn bộ đệm – Ví dụ khác

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated){/*privileged execution*/}
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

**Hàm được thực thi như thế nào?**



36

36

## Tràn bộ đệm – Ví dụ khác

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated){//privileged execution}
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Người dùng có thể ghi đè dữ liệu tùy ý tới các vùng nhớ khác



37

37

## Khai thác lỗi hỏng tràn bộ đệm

- Lỗi hỏng tràn bộ đệm cho phép kẻ tấn công truy cập (read/write/execute) tùy ý vào vùng nhớ khác
- Phương thức khai thác phổ biến nhất: chèn mã nguồn thực thi (code injection)
- Ý tưởng



38

38

## Code Injection

- Vấn đề 1: Nạp mã độc(malcode) vào stack
  - Phải là mã máy
  - Không chứa byte có giá trị 0
  - Không sử dụng bộ nạp (loader)
  - Không sử dụng vùng nhớ stack
- Vấn đề 2: Nạp đúng các địa chỉ lệnh thực thi sau khi kết thúc lời gọi hàm → Xác định đúng %eip
  - Mức độ khó khi xác định giá trị %eip phụ thuộc vị trí của malcode
- Vấn đề 3: Nạp đúng địa chỉ trả về → Xác định đúng %ebp

39

39

## Buffer Overflow – Phòng chống

- **Secure Coding:** sử dụng các hàm an toàn có kiểm soát kích thước dữ liệu đầu vào.
  - fgets(), strncpy(), strcat()...
- **Stack Shield:**
  - Lưu trữ địa chỉ trả về vào vùng nhớ bảo vệ không thể bị ghi đè
  - Sao chép địa chỉ trả về từ vùng nhớ bảo vệ
- **Stack Guard:** sử dụng các giá trị canh giữ (canary) để phát hiện mã nguồn bị chen
- **Non-executable stack:** Không cho phép thực thi mã nguồn trong stack
  - Linux: sysctl -w kernel.exec-shield=0
  - Vẫn bị khai thác bởi kỹ thuật return-to-libc

40

40

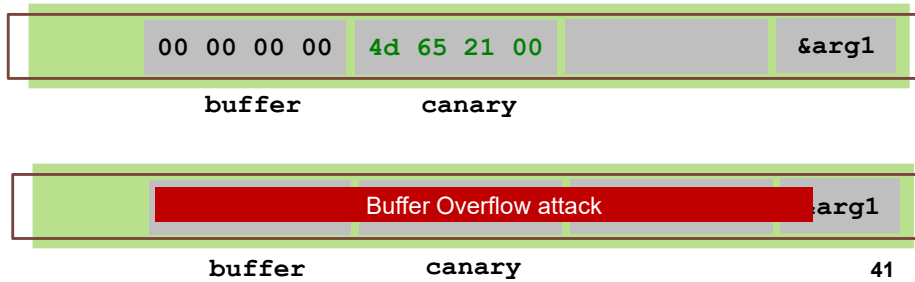
## Sử dụng giá trị canh giữ - Ví dụ

```

callee()
{
    int canary = random;
    char buffer[];
    ...
    if(canary!=random)
        //detect attack
    else return;
}
    
```

```

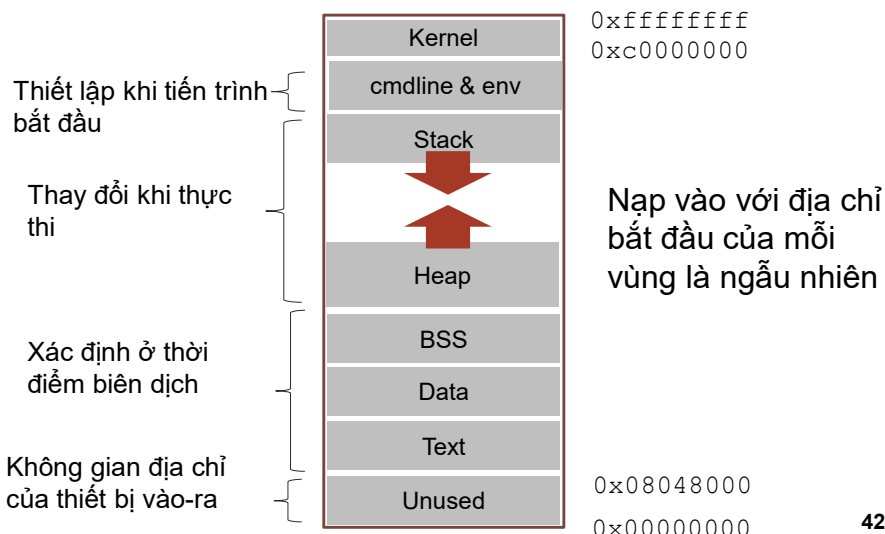
static int random;
caller()
{
    random = rand();
    callee();
}
    
```



41

## Buffer Overflow – Phòng chống

- Address Space Layout Randomization



42

## Buffer Overflow – Phòng chống

- Mã xác thực con trỏ(PAC - Pointer Authentication Code)
- Bộ xử lý 64-bit: sử dụng con trỏ có kích thước 64 bit
  - Định địa chỉ được cho  $2^{64}$  ô nhớ ~ 18 tỉ GB
  - Các hệ thống máy tính hiện đại: chỉ cần tối đa 42 bit để định địa chỉ
  - Số bit không được sử dụng: 22 bit
- PAC được gán cho 22 bit không được sử dụng:
  - Sử dụng các thuật toán để sinh PAC từ giá trị bí mật do CPU sinh
  - Các chương trình không thể truy cập giá trị bí mật này
- Đã được triển khai trên kiến trúc ARM v8.3

43

43

### 3. MỘT SỐ LỖ HỒNG TRUY CẬP BỘ NHỚ KHÁC

Bùi Trọng Tùng,  
Viện Công nghệ thông tin và Truyền thông,  
Đại học Bách khoa Hà Nội

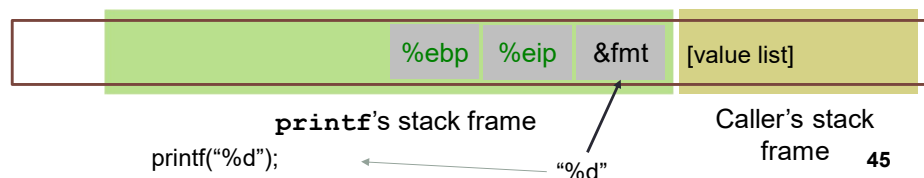
44

44

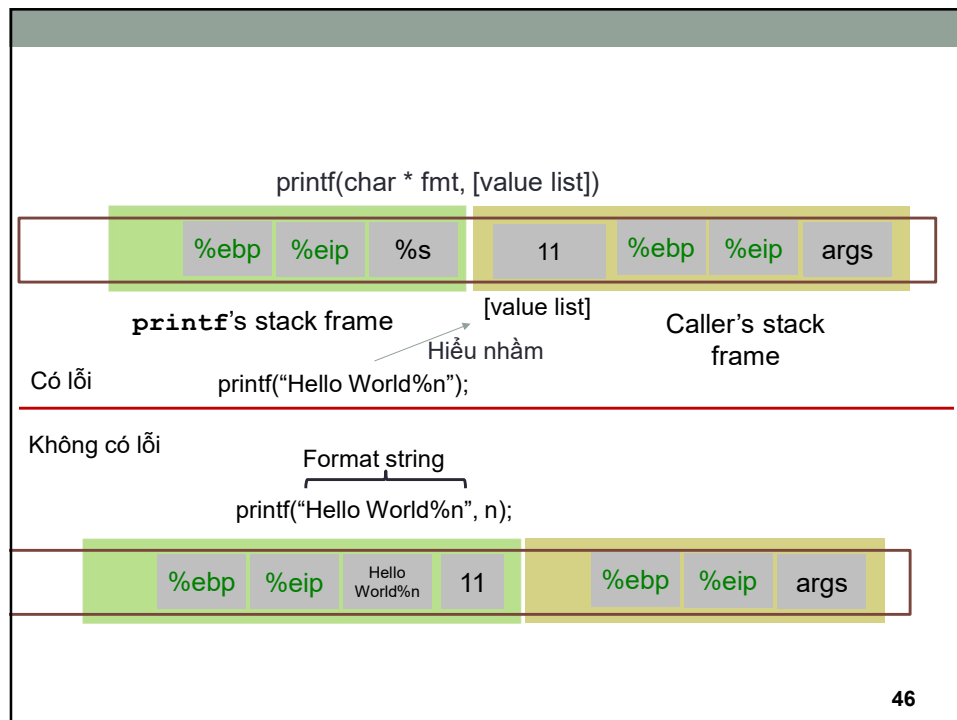
## Lỗi hỏng chuỗi định dạng

- Format String: Chuỗi định dạng vào ra dữ liệu
- Lỗi hỏng Format String: chuỗi định dạng không phù hợp với danh sách tham số `printf(char * fmt, [value list])`
- Ví dụ

```
void func()
{
    char buf[32];
    if(fgets(buf, sizeof(buf), stdin) == NULL)
        return;
    printf(buf); //Sửa: printf("%s", buf);
}
```



45



46

46

## Lỗi hỏng khâu định dạng

- `buf = "%d" → thực thi lệnh printf("%d");`
- Hiển thị 4 byte phía trước địa chỉ đầu tiên của stack frame của hàm `printf`
- `buf = "%s" → thực thi lệnh printf("%s");`
- Hiển thị các byte cho tới khi gặp ký tự kết thúc chuỗi
- `buf = "%d%d%d..." → thực thi lệnh printf("%d%d%d...");`
- Hiển thị chuỗi byte dưới dạng số nguyên
- `printf("%x%x%x...");`
- Hiển thị chuỗi byte dưới dạng hexa
- `printf("...%n");`
- Ghi số byte đã hiển thị vào vùng nhớ

47

47

## Lỗi hỏng tràn số nguyên

- Trong máy tính, số nguyên được biểu diễn bằng trục số tròn. Dải biểu diễn:
  - Số nguyên có dấu:  $[-2^{n-1}, 2^{n-1} - 1]$
  - Số nguyên không dấu:  $[0, 2^n - 1]$
- Integer Overflow: Biến số nguyên của chương trình nhận một giá trị nằm ngoài dải biểu diễn. Ví dụ
  - Số nguyên có dấu:  $0x7f..f + 1 = 0x80..0$ ,
  - Số nguyên không dấu:  $0xff..f + 1 = 0x0, 0x0 - 1 = 0xff...f$
- Ngôn ngữ bị ảnh hưởng: Tất cả
- Việc không kiểm soát hiện tượng tràn số nguyên có thể dẫn đến các truy cập các vùng nhớ mà không thể kiểm soát.

48

48



## Lỗi tràn số nguyên – Ví dụ 1

- Lỗi hỏng nằm ở đâu?

```
#define MAX 1024
void vul_func1()
{
    char buff[1024];
    int len = recv_len_from_client();//len is length of
                                   //message from client
    char *mess = recv_mess_from_client();
    if (len > 1024)
        printf ("Too large");
    else
        memcpy(buff, mess, len);
}
```

```
len = -1 = 0xffffffff < 1024
memcpy(buff, mess, 0xffffffff) → buffer overflow
```

49

49

## Lỗi tràn số nguyên – Ví dụ 2

- Lỗi hỏng nằm ở đâu?

```
int main()
{
    int *arr;
    int len;
    printf("Number of items: "); scanf("%d", &len);
    arr = (int *) malloc(len * sizeof(int));
    for(int i = 0; i < len; i++)
        scanf("%d", arr[i]);
    return 0;
}
```

Khi nào thì vùng nhớ có kích thước  $4*n$  không đủ chỗ chứa cho  $n$  phần tử số nguyên?  
Có xảy ra  $4*n = 0$  khi  $n \neq 0$ ?  
 $n = 0100\ 0000\ 0000\ 0000 = 0x4000$   
 $4*n = 0x0000$

50

50

## 4. LẬP TRÌNH AN TOÀN

---

Bùi Trọng Tùng,  
Viện Công nghệ thông tin và Truyền thông,  
Đại học Bách khoa Hà Nội

51

51

## Lập trình an toàn

- Yêu cầu: Viết mã nguồn chương trình để đạt được các mục tiêu an toàn bảo mật
- Bao gồm nhiều kỹ thuật khác nhau:
  - Kiểm soát giá trị đầu vào
  - Kiểm soát truy cập bộ nhớ chính
  - Che giấu mã nguồn
  - Chống dịch ngược
  - Kiểm soát kết quả đầu ra
  - Kiểm soát quyền truy cập
  - ...
- Bài này chỉ đề cập đến một số quy tắc và nhấn mạnh vào vấn đề truy cập bộ nhớ một cách an toàn

52

52

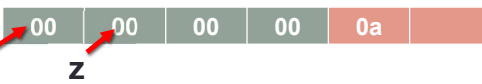
## An toàn truy cập bộ nhớ

- An toàn không gian(Spatial safety): thao tác chỉ nên truy cập vào đúng vùng nhớ đã xác định
- Nếu gọi:
  - b: địa chỉ ô nhớ đầu tiên của vùng nhớ được chỉ ra
  - p: địa chỉ cần truy cập tới
  - e: địa chỉ ô nhớ cuối cùng của vùng nhớ được chỉ ra
  - s: kích thước vùng nhớ mà con trỏ p truy cập tới
- Thao tác truy cập bộ nhớ chỉ an toàn khi và chỉ khi:
$$b \leq p \leq e - s$$
- Lưu ý: Các toán tử tác động trên p không làm thay đổi b và e.

53

53

## An toàn không gian – Ví dụ

x(chiếm 4 byte): 

```
int x = 0;
int *y = &x; // b = &x, e = &x + 4, s = 4
int *z = y + 1; // b = &x, e = &x + 4, s = 4
*y = 10; //OK: &x ≤ p = &x ≤ (&x + 4) - 4
*z = 10; //Fail: &x ≤ p = &x + 1  $\nless$  (&x + 4) - 4
```

```
char str[10]; //b = &str, e = &str + 10
str[5] = 'A'; //OK: &str ≤ p = &str + 5 ≤ (&str + 10) - 1
str[10] = 'F'; //Fail: &str ≤ p = &str + 10  $\nless$  (&str + 10) - 1
```

- Lỗi truy cập không an toàn về không gian gây ra các lỗi hổng như đã biết

54

54

## An toàn truy cập bộ nhớ

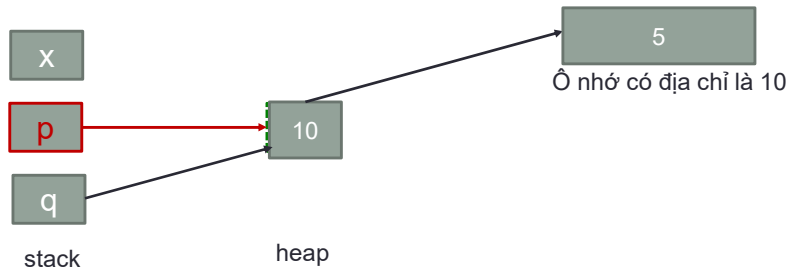
- An toàn thời gian(): thao tác chỉ truy cập vào vùng nhớ mà đã được khởi tạo:
  - Đã cấp phát bộ nhớ
  - Đã được khởi tạo giá trị
- Ví dụ: Vi phạm an toàn về thời gian

```
int n;  
printf("%d", n);      // Fail  
int m = n + 1;  
if (n) ...  
int *p;  
*p = 0;               // Fail  
p = (int *) malloc(sizeof(int));  
*p = 0;               // OK  
free(p);  
p = 0;                //Tại sao thao tác này là cần thiết?
```

55

55

```
int x = 5;  
int *p;  
p = (int *) malloc(sizeof(int));  
*p = 0;               // OK  
free(p);  
int **q = malloc(sizeof(int*)); //q có thể sẽ được cấp phát  
                                //vùng nhớ vừa được giải phóng  
*q = &x;  
*p = 10;  
**q = 5; //runtime error
```



56

56

## Điều kiện truy cập bộ nhớ

- Tiền điều kiện(precondition): điều kiện để câu lệnh/hàm được thực thi đúng đắn
- Hậu điều kiện(postcondition): khẳng định trạng thái đúng đắn của các đối tượng khi lệnh/hàm kết thúc
- Ví dụ: Xác định các điều kiện truy cập bộ nhớ

```
void displayArr(int a[], int n)
{
    for(int i = 0; i < n; i += 2){
        printf("%d", a[i]);
    }
}
```

- Tiền điều kiện:  $n \leq a.size()$ ,  $a \neq \text{null}$
- Hậu điều kiện:  $i < n$
- $a[i] \sim a + i$

57

57

## Các nguyên tắc lập trình an toàn

- Không tin cậy những thứ mà không do bạn tạo ra
- Người dùng chỉ là những kẻ gốc gác  
    > Hàm gọi (Caller) = Người dùng
- Hạn chế cho kẻ khác tiếp cận những gì quan trọng. Ví dụ: thành phần bên trong của một cấu trúc/đối tượng  
    > Ngôn ngữ OOP: nguyên lý đóng gói  
    > Ngôn ngữ non-OOP: sử dụng token
- Không bao giờ nói “không bao giờ”
- Sau đây sẽ đề cập đến một số quy tắc trong C/C++
- Về chủ đề lập trình an toàn, tham khảo tại đây:  
<https://security.berkeley.edu/secure-coding-practice-guidelines>

58

58

## Kiểm tra mọi dữ liệu đầu vào

- Các giá trị do người dùng nhập
- File được mở
- Các gói tin nhận được từ mạng
- Các dữ liệu thu nhận từ thiết bị cảm biến (Ví dụ: QR code, âm thanh, hình ảnh,...)
- Thư viện của bên thứ 3
- Mã nguồn được cập nhật
- Khác...

59

59

## Sử dụng các hàm xử lý xâu an toàn

- Sử dụng các hàm xử lý xâu an toàn thay cho các hàm thông dụng
  - `strcat`, `strncat` → `strlcat`
  - `strcpy`, `strncpy` → `strlcpy`
  - `gets` → `fgets`, `fprintf`
- Luôn đảm bảo xâu được kết thúc bằng `'\0'`
- Nếu có thể, hãy sử dụng các thư viện an toàn hơn
  - Ví dụ: `std::string` trong C++

60

60

## Sử dụng con trỏ một cách an toàn

- Hiểu biết về các toán tử con trỏ: +, -, sizeof
- Cần xóa con trỏ về NULL sau khi giải phóng bộ nhớ

```
int x = 5;
int *p = (int *)malloc(sizeof(int));
free(p);
p = NULL;
int **q = (int **)malloc(sizeof(int*));
*q = &x;
*p = 5;           //Crash → OK
**q = 3;
```

61

61

## Cẩn trọng khi sử dụng lệnh goto

- Ví dụ:

```
int foo(int arg1, int arg2) {
    struct foo *pf1, *pf2;
    int retc = -1;

    pf1 = malloc(sizeof(struct foo));
    if (!isok(arg1)) goto DONE;

    ...

    pf2 = malloc(sizeof(struct foo));
    if (!isok(arg2)) goto FAIL_ARG2;

    ...

    retc = 0;
FAIL_ARG2:
    free(pf2); //fallthru
DONE:
    free(pf1);
    return retc;
}
```

62

62

## Sử dụng các thư viện an toàn hơn

- Nên sử dụng chuẩn C/C++11 thay cho các chuẩn cũ
- Sử dụng `std::string` trong C++ để xử lý chuỗi
- Truyền dữ liệu qua mạng: sử dụng Google Protocol Buffers hoặc Apache Thrift

63