

# LẬP TRÌNH HỆ THỐNG

---

ThS. Đỗ Thị Thu Hiền  
(hiendtt@uit.edu.vn)



**TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM**  
**KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG**  
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS

Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM  
Điện thoại: (08)3 725 1993 (122)

# Lab 6 - BUFFER OVERFLOW



# Nội dung

---

- **Làm bài TN trên Kahoot 😊**
- Review LT về buffer overflow
- Nội dung thực hành

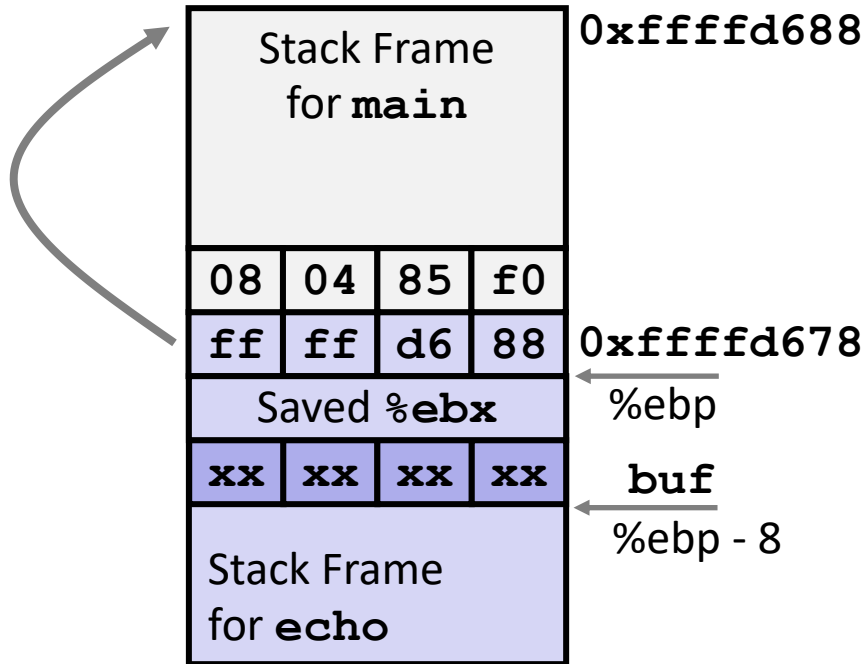
# Nội dung

---

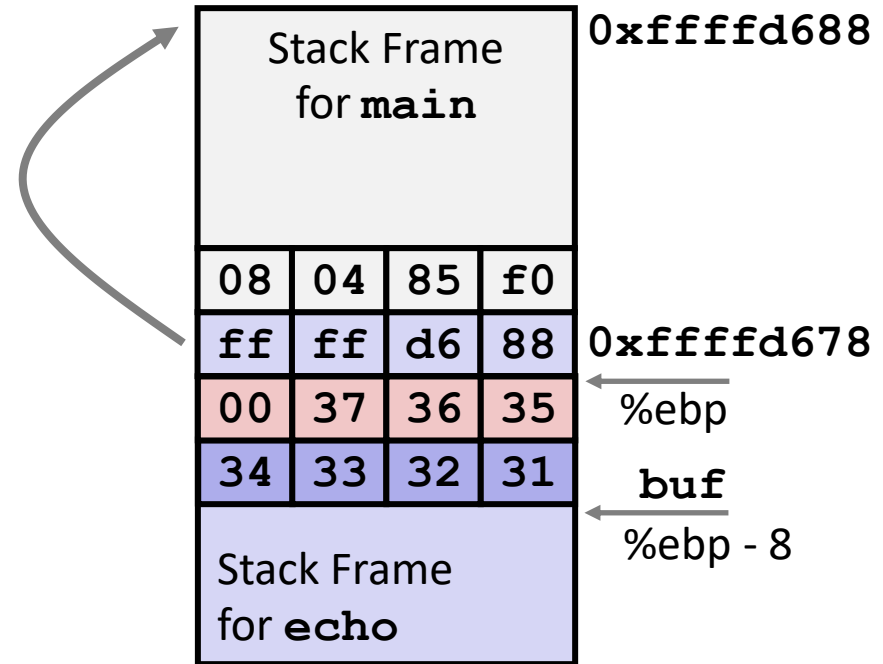
- Làm bài TN 😊
- **Review LT về buffer overflow**
- Nội dung thực hành

# Buffer Overflow: Ví dụ #1

*Before call to gets*



*Input: 1234567*

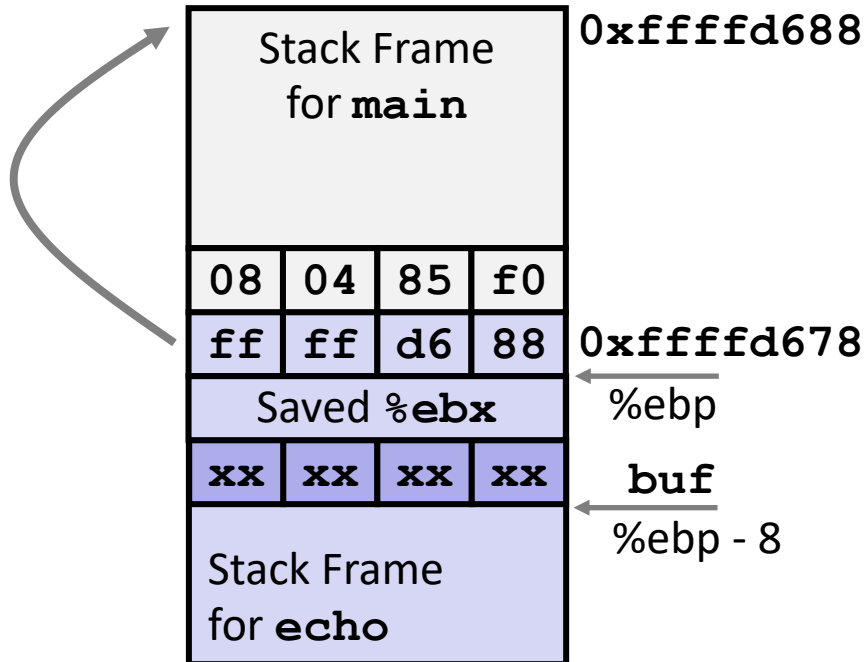


```
unix> ./bufdemo
Type a string: 1234567
1234567
```

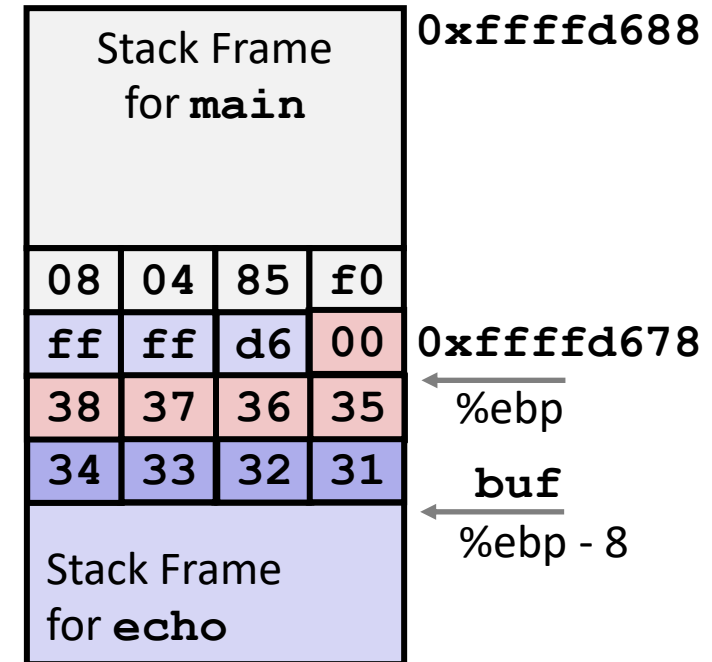
Vượt quá buf, ghi đè %ebx,  
nhưng không gây ra vấn đề gì  
→ Chỉ làm thay đổi 1 giá trị đã lưu

# Buffer Overflow: Ví dụ #2

*Before call to gets*



*Input: 12345678*



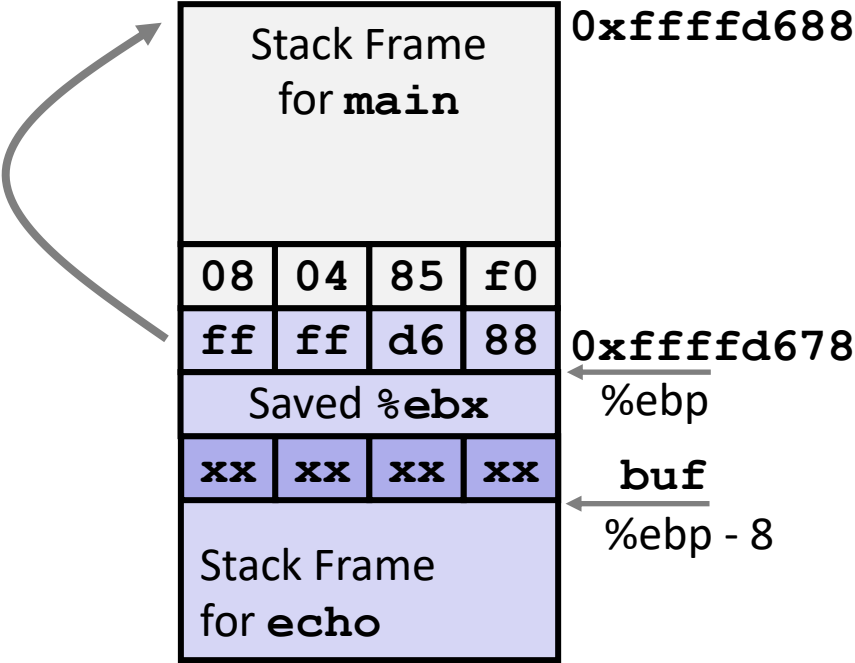
Ghi đè %ebp cũ → lỗi

```
unix> ./bufdemo
Type a string: 12345678
Segmentation Fault
```

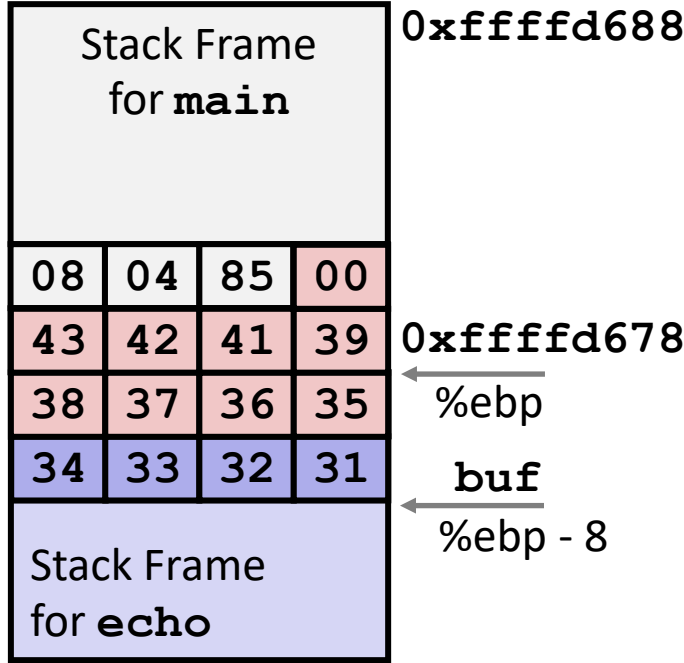
```
. . .
80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Set %ebp to corrupted value
80485f1: c3 ret
```

# Buffer Overflow: Ví dụ #3

Before call to gets



Input: 123456789ABC



```
unix> ./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```

Return address bị ghi đè

```
80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point
```

# Nội dung

---

- Làm bài TN 😊
- Review LT về buffer overflow
- **Nội dung thực hành**



# Buffer Overflow

## THỰC HÀNH

---

- **Cá nhân**
  - simple-buffer
- **Nhóm**
  - bufbomb (4 level: 0, 1, 2, 3)

# Môi trường thực hành

---

- **01 máy Linux để thực thi file cần phân tích**
  - 32 hoặc 64 bit
  - Ubuntu, Kali, CentOS...
  - VMWare, Virtualbox...
  - Có thể thực thi file 32-bit
- **Các disassembler**
  - Windows: IDA Pro
  - Linux: objdump, GDB

# Các yêu cầu thực hành

---

- **Thực hành cá nhân: simple-buffer**
  - Khai thác lỗ hổng buffer overflow
  - Ghi đè thay đổi 1 số biến cục bộ nằm cùng stack frame với chuỗi input
- **Thực hành nhóm: bufbomb**
  - Khai thác lỗ hổng buffer overflow
  - Ghi đè địa chỉ trả về để điều hướng thực thi của chương trình

# Thực hành Cá nhân

- **simple-buffer**: File thực thi Linux 32 bit, không canary
- **Hàm smash\_my\_buffer**: đọc 1 input lưu vào **buf** với **gets**

```
int student_id;           # biến lưu giá trị số nguyên của MSSV từ tham số khi chạy
char student_name[8];     # biến lưu tên của SV từ tham số khi chạy
void smash_my_buffer()
{
... unsigned int var = 0x12345678; ...
  int another_var = 0x10; ...
  char my_name[8] = "student";
  ....
  char buf[20];
  gets(buf);
  if (strcmp(my_name, "student") || var != 0x12345678 || another_var != 0x10){
    printf("You changed my local variables.\n");
    if (strcmp(my_name, student_name) == 0)
      printf("Level 1: DONE..."); ...
    if (another_var == 0x4165)
      printf("Level 2: DONE..."); ...
    if (var == student_id)
      printf("Level 3: DONE..."); ...
  }
}
```

# Buffer Overflow: (cá nhân)

- **Mục tiêu:** nhập input `buf` sao cho gây ra buffer overflow làm thay đổi biến cục bộ:
  - `my_name` thành **tên SV**
  - `another_var` thành **0x4165**
  - `var` thành giá trị số nguyên của **MSSV**
- **Chạy file:** cần cung cấp MSSV và tên SV làm tham số cho file  
`./simple-buffer <MSSV> <ten>`
- **Vẽ stack:** xác định vị trí tương đối giữa `buf` và các biến cục bộ trong stack.  
→ Suy ra được độ dài `buf` và vị trí các biến cục bộ tương ứng với phần nào trong `buf`?

# Buffer Overflow: (cá nhân)

- **Mục tiêu:** nhập input **buf** sao cho gây ra buffer overflow làm thay đổi biến cục bộ:
  - **my\_name** thành **tên SV**
  - **another\_var** thành **0x4165**
  - **var** thành giá trị số nguyên của **MSSV**
- Gợi ý:
  - Chú ý khi lưu các kiểu dữ liệu lớn hơn 1 byte trên Linux (Little Endian).
  - Chuỗi kết thúc bằng byte NULL (0).
  - Dùng file hỗ trợ **hex2raw** hoặc **code python** để truyền các byte không gõ được từ bàn phím.

Lưu ý: check phiên bản python trước khi viết code!

```
input.py
exploit_str = "0"*44
exploit_str += "\x0A\x0B\x0C\x0D"
print exploit_str
```

**python 2.x.x**

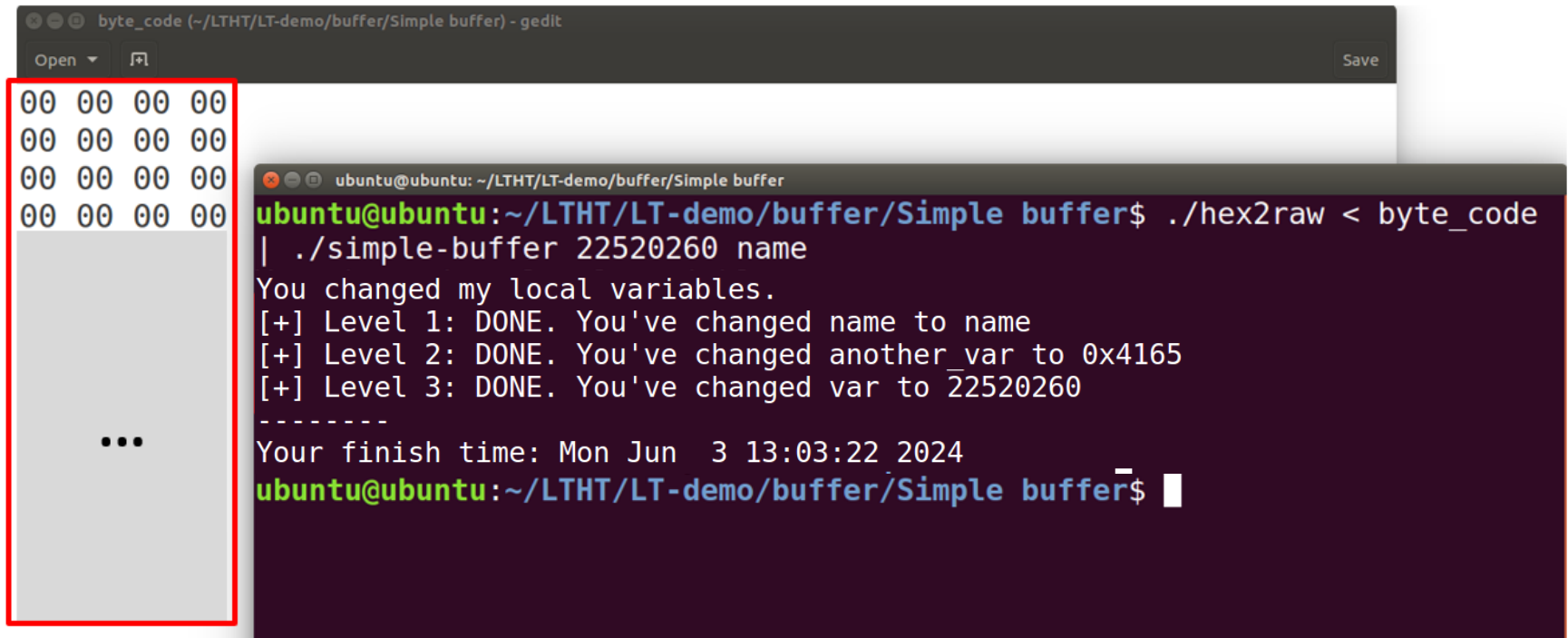
**\$ python <python file> | ./simple-buffer <MSSV> <name>**

```
input3.py
import sys
str = 'a'*44
sys.stdout.buffer.write(str.encode())
x = bytes.fromhex('0A 0B 0C 0D')
sys.stdout.buffer.write(x)
```

**python 3.x.x**

# Nộp bài – Simple buffer

Hình ảnh minh chứng khai thác thành công 3 level có kèm theo nội dung chuỗi/code



The image shows a terminal window and a hex editor. The hex editor displays a file named `byte_code` with four rows of hex data, all consisting of `00 00 00 00`. The terminal window shows the execution of a program named `simple-buffer` with the following output:

```
ubuntu@ubuntu: ~/LTHT/LT-demo/buffer/Simple buffer
ubuntu@ubuntu:~/LTHT/LT-demo/buffer/Simple buffer$ ./hex2raw < byte_code
| ./simple-buffer 22520260 name
You changed my local variables.
[+] Level 1: DONE. You've changed name to name
[+] Level 2: DONE. You've changed another_var to 0x4165
[+] Level 3: DONE. You've changed var to 22520260
-----
Your finish time: Mon Jun  3 13:03:22 2024
ubuntu@ubuntu:~/LTHT/LT-demo/buffer/Simple buffer$
```

# Buffer Overflow: (Nhóm)

---

- **Buffer Overflow Attack (Buffer Bomb)**
- File cần khai thác: **bufbomb** – Linux 32 bit
  - Một số file hỗ trợ: **hex2raw**, **makecookie**



# File bufbomb

- File thực thi **Linux 32 bit**
- Khi thực thi, cần truyền 1 tham số là userid với option `-u`  
`./bufbomb -u <userid>`
- File khi hoạt động sẽ nhận vào 1 chuỗi input từ người dùng.

```
ubuntu@ubuntu:~/LTHT/Lab5$ ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string: hello world
Dud: getbuf returned 0x1
Better luck next time
ubuntu@ubuntu:~/LTHT/Lab5$
```

# File bufbomb

- Chuỗi gọi hàm bình thường: **test() → getbuf() → gets()**
- Chuỗi input sẽ được lưu trong biến cục bộ **buf** của hàm getbuf():

```
1 /* Buffer size for getbuf */
2 #define NORMAL_BUFFER_SIZE 32
3 int getbuf()
4 {
5     char buf[NORMAL_BUFFER_SIZE];
6     Gets(buf);
7     return 1;
8 }
```

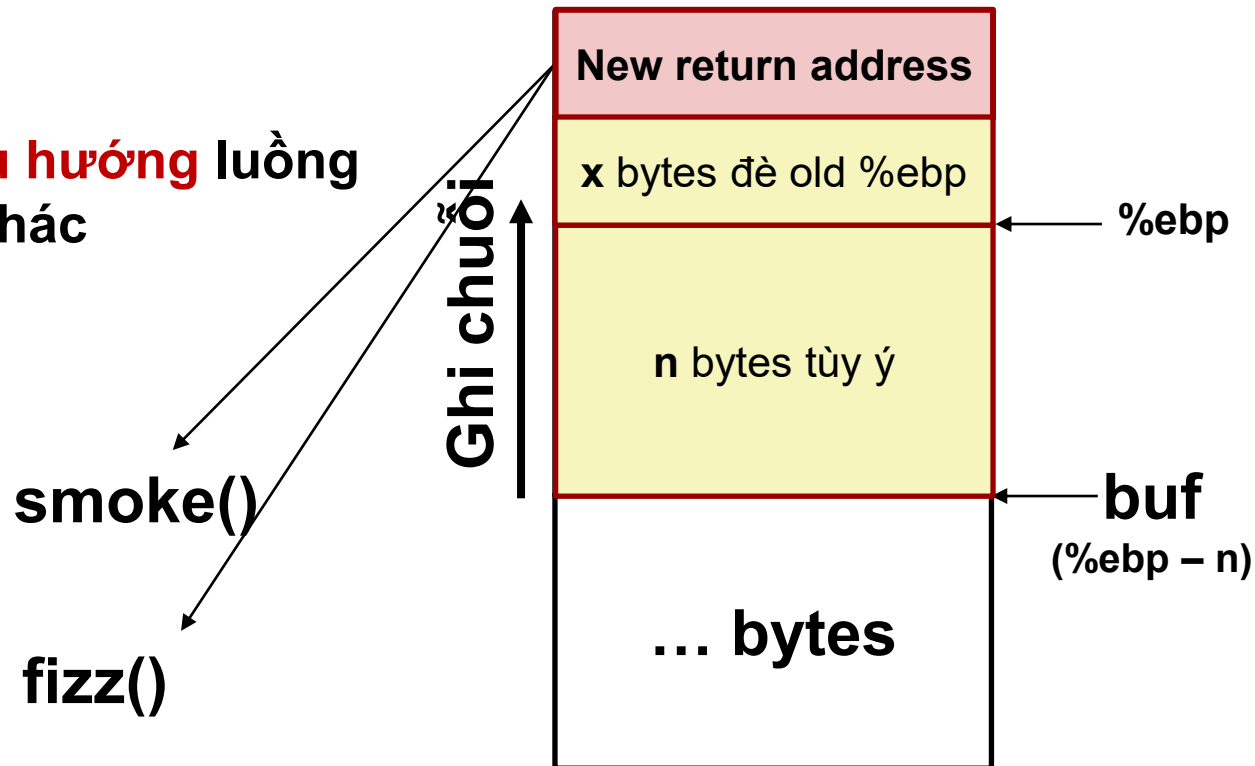
*Gets() có vấn đề không kiểm soát/giới hạn kích thước chuỗi tương tự gets()*

- Chuỗi buf có thể vượt quá không gian đã cấp trong stack getbuf()
- Ảnh hưởng đến vùng nhớ lân cận (trong stack của getbuf)
- Mục tiêu: truyền các chuỗi có thể làm tràn buf và thay đổi địa chỉ trả về để chuyển luồng hoạt động (+ ...) - **chuỗi exploit**

# Khai thác File bufbomb

- Chuỗi gọi hàm bình thường: **test() → getbuf() → gets()**
- Chuỗi input sẽ được lưu trong biến buf trong hàm getbuf()
- Khai thác bufbomb:

**Phần 1 (bắt buộc): Điều hướng** luồng thực thi đến các hàm khác



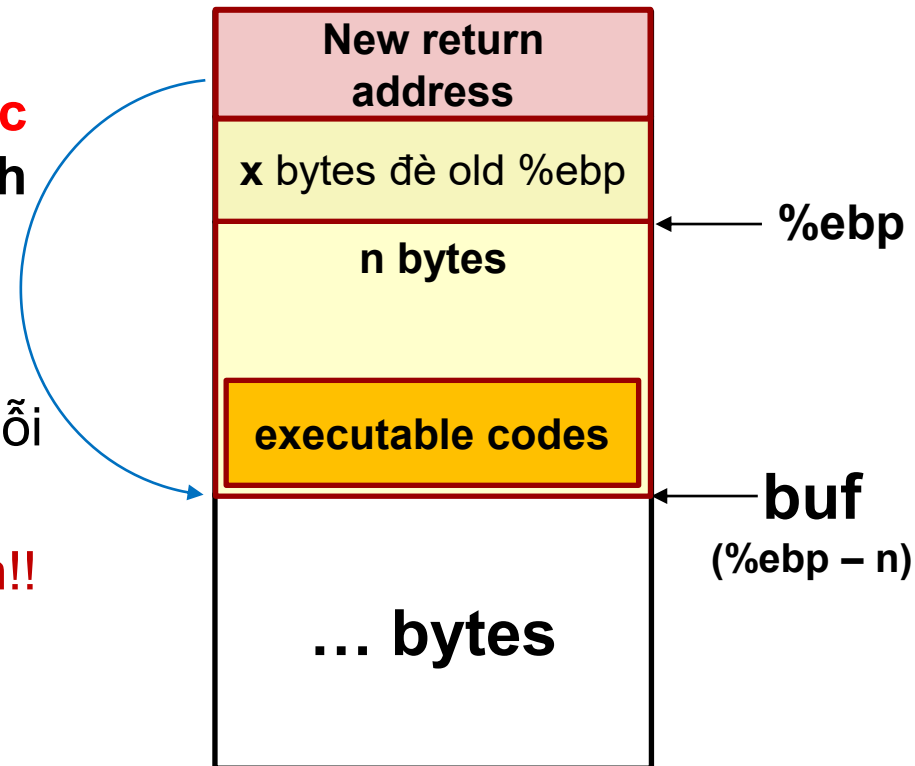
# Khai thác File bufbomb (tt)

- Chuỗi gọi hàm bình thường: **test() → getbuf() → gets()**
- Chuỗi input sẽ được lưu trong biến buf trong hàm getbuf()
- Khai thác bufbomb:

**Phần 2 (tùy chọn): truyền **code thực thi** vào và **điều hướng** chương trình để thực thi**

- Có executable code trong chuỗi
- Địa chỉ trả về mới = địa chỉ lưu chuỗi buf trong stack

**Chỉ xác định khi chạy chương trình!!**



# Các bước thực hiện tấn công file bufbomb

## 1. Chọn 1 userid và tạo cookie tương ứng

- Luôn phải cung cấp **userid** khi thực thi bufbomb.
- **Quy ước**: ghép từ MSSV của các TV trong nhóm  
*Xem quy ước trong file hướng dẫn*
- Cố định userid từ đầu, không thay đổi khi phân tích và chạy file
- Xem cookie tương ứng:

`./makecookie <userid>`

```
ubuntu@ubuntu:~/LTHT/ $ ./makecookie 02600143
0x3df64b09
ubuntu@ubuntu:~/LTHT/ $
```

# Các bước thực hiện tấn công file bufbomb

## 2. Phân tích mã assembly của bufbomb xác định độ dài và nội dung chuỗi exploit

- Xem mã assembly của file bufbomb để hiểu các hoạt động cấp phát bộ nhớ bên dưới hệ thống (hàm getbuf).
- Xác định vị trí lưu chuỗi input (buf), và khoảng cách so với ô nhớ cần thay đổi chứa địa chỉ trả về → Xác định chuỗi exploit
  - Bao nhiêu byte (ký tự)?
  - Những byte nào sẽ ghi đè lên địa chỉ trả về?

# Các bước thực hiện tấn công file bufbomb

## 3. Truyền chuỗi exploit cho file bufbomb

- Một số byte đặc biệt không thể truyền bằng cách gõ từ bàn phím.
- Ví dụ: Giá trị 0x01020304 → Các byte 01 02 03 04?
- Sử dụng file **hex2raw**
  - Định nghĩa sẵn các byte của chuỗi exploit trong 1 file text  
Ví dụ: exploit.txt
  - Truyền file exploit.txt cho hex2raw và truyền kết quả cho bufbomb  
\$ ./hex2raw < exploit.txt | ./bufbomb -u <userid>

# Level 0

Khai thác lỗ hổng buffer overflow trong bufbomb tại hàm **getbuf**, sao cho sau khi **getbuf** thực thi xong, chương trình sẽ thực thi đoạn code của hàm **smoke** thay vì thực thi tiếp hàm mẹ là **test**

**smoke là 1 hàm có sẵn trong bufbomb nhưng không được gọi**

```
1 void smoke()
2 {
3     printf("Smoke!: You called smoke()\n");
4     validate(0);
5     exit(0);
6 }
```



# Level 0 – Các yêu cầu

1. Sinh viên vẽ stack của hàm **getbuf()** với mô tả như trên để xác định vị trí của chuỗi **buf** sẽ lưu chuỗi input?

*Cần thể hiện trong stack các vị trí: return address của getbuf, vị trí của buf*

2. Xác định các đặc điểm của chuỗi exploit:

- Chuỗi exploit cần có **kích thước bao nhiêu bytes?**
- **4 bytes ghi đè** lên 4 bytes địa chỉ trả về sẽ **nằm ở vị trí nào** trong chuỗi exploit?

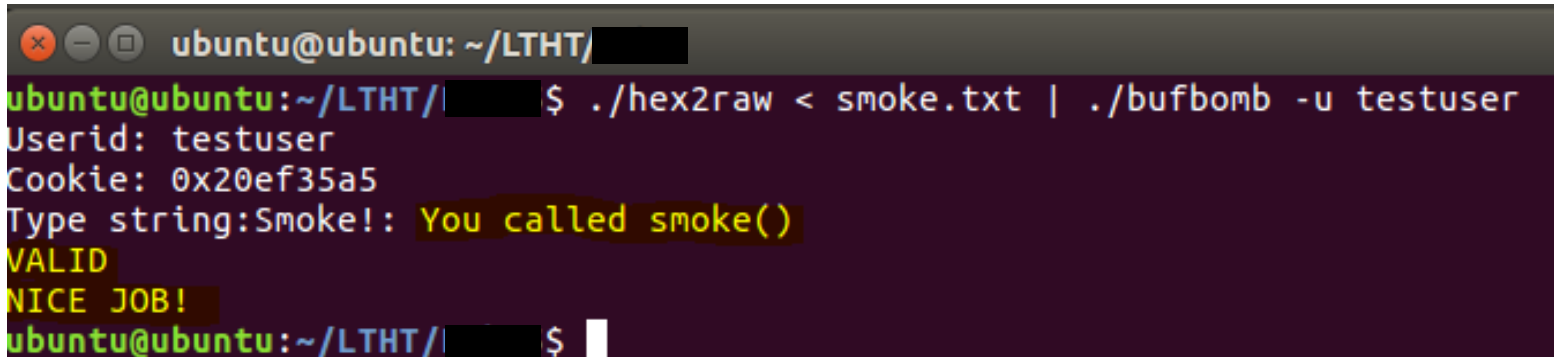
3. Xác định địa chỉ trả về mới

4. Xây dựng chuỗi exploit có độ dài và nội dung đã xác định.

5. Truyền chuỗi exploit cho bufbomb và báo cáo kết quả

# Level 0 – Kết quả đúng

```
1 void smoke ()
2 {
3     printf("Smoke!: You called smoke() \n");
4     validate(0);
5     exit(0);
6 }
```



A terminal window titled 'ubuntu@ubuntu: ~/LTHT/' shows the execution of a command: `./hex2raw < smoke.txt | ./bufbomb -u testuser`. The output displays the user 'testuser', a cookie '0x20ef35a5', and the string 'Smoke!: You called smoke()' which is highlighted in yellow. Below this, the words 'VALID' and 'NICE JOB!' are also highlighted in yellow, indicating a successful exploit.

```
ubuntu@ubuntu:~/LTHT/ $ ./hex2raw < smoke.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
ubuntu@ubuntu:~/LTHT/ $
```

# Level 1

Khai thác lỗ hổng buffer overflow trong bufbomb tại hàm **getbuf**, sao cho sau khi **getbuf** thực thi xong, chương trình sẽ thực thi đoạn code của hàm **fizz** kèm theo tham số là cookie

**fizz là 1 hàm có sẵn trong bufbomb nhưng không được gọi**

```
1 void fizz(int val)
2 {
3     if (val == cookie) {
4         printf("Fizz!: You called fizz(0x%x)\n", val);
5         validate(1);
6     } else {
7         printf("Misfire: You called fizz(0x%x)\n", val);
8         exit(0);
9     }
10 }
```

# Level 1 – Kết quả đúng

```
1 void fizz(int val)
2 {
3     if (val == cookie) {
4         printf("Fizz!: You called fizz(0x%x)\n", val);
5         validate(1);
6     } else {
7         printf("Misfire: You called fizz(0x%x)\n", val);
8         exit(0);
9     }
10 }
```

```
ubuntu@ubuntu: ~/LTHT/Lab5/Demo
ubuntu@ubuntu:~/LTHT/ $ ./hex2raw < fizz.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:Fizz!: You called fizz(0x20ef35a5)
VALID
NICE JOB!
ubuntu@ubuntu:~/LTHT/ $
```

# Level 2

Khai thác lỗ hổng buffer overflow để truyền vào bufbomb một chuỗi exploit **có chứa code thực thi** sao cho:

- Thay đổi được giá trị của **global\_value** thành **cookie**
- Gọi được hàm **bang** thay vì trở về hàm test

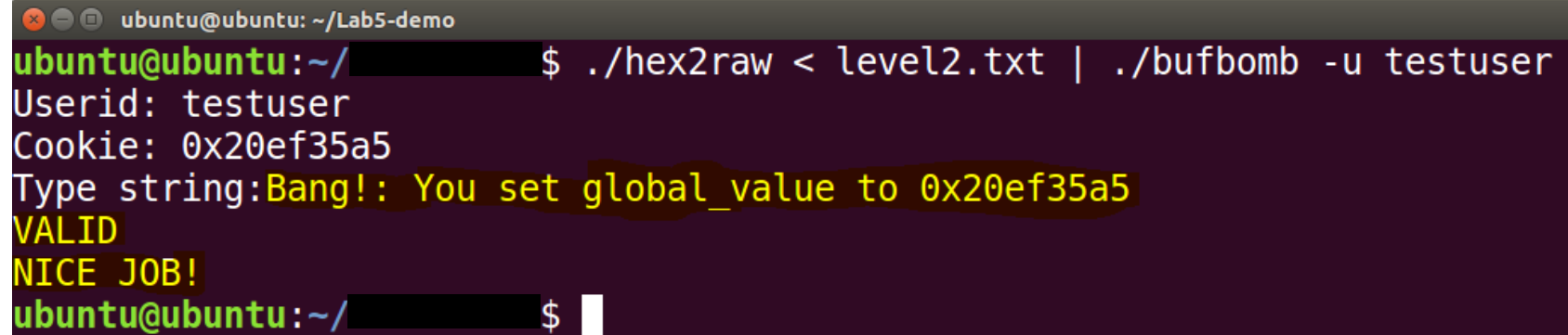
**global\_value** là 1 biến toàn cục, lưu tại 1 địa chỉ trong .bss

**bang** là hàm có sẵn trong bufbomb nhưng không được gọi

```
1  int global_value = 0;
2  void bang(int val)
3  {
4      if (global_value == cookie) {
5          printf("Bang!: You set global_value to 0x%x\n", global_value);
6          validate(2);
7      } else {
8          printf("Misfire: global_value = 0x%x\n", global_value);
9          exit(0);
10     }
11 }
```

# Level 2 – Kết quả đúng

```
1  int global_value = 0;
2  void bang(int val)
3  {
4      if (global_value == cookie) {
5          printf("Bang!: You set global_value to 0x%x\n", global_value);
6          validate(2);
7      } else {
8          printf("Misfire: global_value = 0x%x\n", global_value);
9          exit(0);
10     }
11 }
```



```
ubuntu@ubuntu: ~/Lab5-demo
ubuntu@ubuntu:~/ $ ./hex2raw < level2.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string: Bang!: You set global_value to 0x20ef35a5
VALID
NICE JOB!
ubuntu@ubuntu:~/ $
```

# Level 3

Khai thác lỗ hổng buffer overflow để truyền vào một chuỗi exploit **chứa code thực thi** sao cho **getbuf** khi thực thi xong, **giá trị trả về sẽ là cookie** tương ứng với userid cho hàm test, thay vì trả về 1.

# Level 3 – Kết quả đúng

```
ubuntu@ubuntu: ~/Lab5-demo
ubuntu@ubuntu:~/ $ ./hex2raw < level3.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:Boom!: getbuf returned 0x20ef35a5
VALID
NICE JOB!
ubuntu@ubuntu:~/ $
```



# Một số lưu ý

- Chuỗi exploit không được chứa byte **0x0A** ở các vị trí trung gian, vì 0x0A là dấu hiệu kết thúc chuỗi.
- Mỗi byte truyền cho **hex2raw** là 2 số hexan, kể cả byte 0 cũng phải ghi rõ 00.
- Cần lưu ý đến **byte ordering** trong Linux khi truyền giá trị lớn hơn 1 byte. Trong bộ nhớ 0x12ABCDEF sẽ được lưu là EF CD AB 12 từ địa chỉ thấp đến cao.

# Yêu cầu

---

## ■ Trên lớp: khai thác file simple-buffer

- Làm cá nhân
- Nộp hình ảnh minh chứng kết quả khai thác file kèm theo code/nội dung các byte trên **course**
- Bắt buộc: cả 3 level

## ■ Về nhà: khai thác file bufbomb

- Làm theo nhóm
- Nộp hình ảnh minh chứng kết quả thực thi file kèm theo nội dung chuỗi exploit/code trên **course**
- Bắt buộc: Level 0 – 1
- Không bắt buộc: Level 2 - 3