Write up tjctf: sledshop + slice\_of\_pie

Mình quyết định viết write up 2 bài này vì qua 2 bài này có những vấn đề mới mà mình học được, đặc biệt là có aslr, stripped và pie enable.

Aslr là gì ?

Aslr là viết tắt của cụm từ Address space layout randomization, có nghĩa là bố trí không gian địa chỉ ngẫu nhiên, nó sẽ chọn những vùng ngẫu nhiên để chứa code của chương trình, nó sẽ nhớ những nơi đã chứa code để tiếp tục thực hiện chương trình. Mỗi lần chương trình được chạy thì địa chỉ các phân vùng stack, heap,shared\_libc cũng sẽ thay đổi, điều này sẽ giúp chống lại việc lỗi tràn bộ đệm vì không thể xác định chính xác địa chỉ của các biến, các thanh ghi ở đâu. Mà đặc điểm của khai thác lỗi tràn bộ đệm là phải xác định chính xác địa chỉ của các hàm và biến để tính toán lượng dữ liệu rác đưa vào.

Stripped là gì ?

Khi compile với no stripped, tức là bạn giữ lại tất cả thông tin debug bao gồm tên hàm, tên biến, các dữ liệu liên quan đến hàm trong source code. Còn đối với stripped, nó sẽ bỏ hết tất cả những thông tin trên nhằm giảm kích thước của file thực thi. Đối với pwnable, file stripped excute gây khó khăn trong việc debug vì khó có thể đặc breakpoint. Tuy vậy vẫn có cách, đó là sử dụng trình dịch ngược IDA.

Pie là gì?

Pie là viết tắt của cụm từ position independent excutables, có nghĩa là vị trí thực thi độc lập. Tức là với mỗi lần thực thi thì chương trình sẽ được load vào một địa chỉ ngẫu nhiên trong bộ nhớ ảo. Đối với pwn, pie binary excute gây khó khăn cho việc thực thi lỗi tràn bộ đệm( có thể nhiều hơn nữa…) vì chúng ta không thể xác định được địa chỉ chính xác qua các lần debug. Địa chỉ của hàm A ở lần debug thứ 1 sẽ khác hoàn toàn so với lần debug thứ 2 vì địa chỉ random mà.

Đây là những vấn đề mới mà mình gặp ở 2 bài trên.

Sledshop:

Bài này cũng không khó lắm.

Source code:

#include <stdio.h>

#include <stdlib.h>

void shop\_setup() {

gid\_t gid = getegid();

setresgid(gid, gid, gid);

setbuf(stdout, NULL);

}

void shop\_list() {

printf("The following products are available:\n");

printf("| Saucer | $1 |\n");

printf("| Kicksled | $2 |\n");

printf("| Airboard | $3 |\n");

printf("| Toboggan | $4 |\n");

}

void shop\_order() {

int canary = 0;

char product\_name[64];

printf("Which product would you like?\n");

gets(product\_name);

if (canary)

printf("Sorry, we are closed.\n");

else

printf("Sorry, we don't currently have the product %s in stock. Try again later!\n", product\_name);

}

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

shop\_setup();

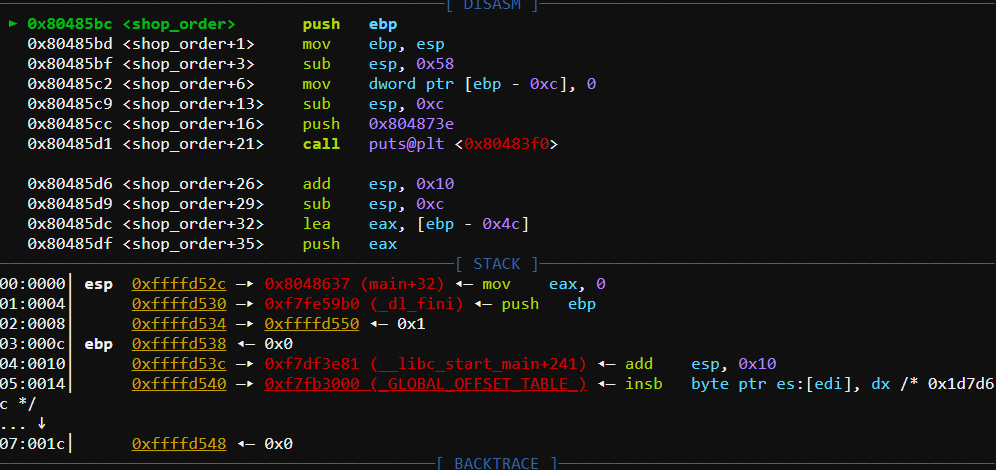
shop\_list();

shop\_order();

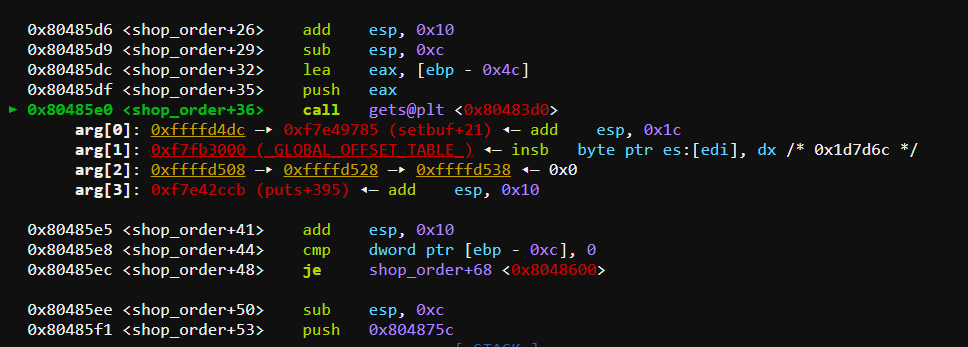
return 0;

}

Như thường lệ thì đầu tiên sẽ là phân tích source code, main gọi 3 hàm shop\_setup, shop\_list, shop\_order. 2 hàm shop\_setup và shop\_list hoàn toàn vô dụng, đến hàm shop\_order, ta thấy hàm này cho chúng ta nhập bằng hàm gets, ở đây ta đã thấy được 1 lỗi, vì hàm gets này không kiểm soát số lượng kí tự ta nhập vào cho nên có thể gây lỗi tràn bộ đệm. Chương trình đến đây là hết, ta sẽ debug hàm shop\_order để exploit.



Ta thấy khi hàm shop\_order được call thì nó sẽ push địa chỉ return của nó vào stack. Địa chỉ ret của nó nằm ở 0xffffd52c.



Hàm quan trọng của chúng là hàm gets, ta thấy nó nhận tham số đầu tiên ở 0xffffd4dc, đó là địa chỉ mà nó sẽ ghi dữ liệu vào.

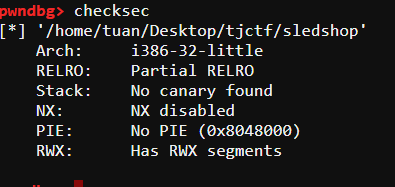
Ta biết địa chỉ return và địa chỉ ghi dữ liệu thì ta sẽ chọn hướng làm ghi đè địa chỉ return thành địa chỉ shellcode.

Vì chương trình này không có hàm gọi shell nên chúng ta sẽ ghi shellcode vào.

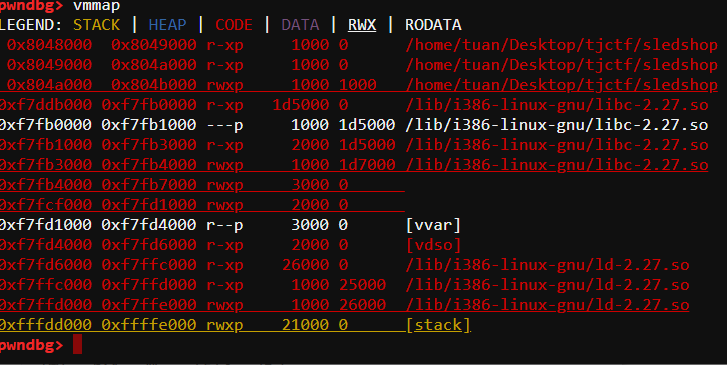
Ta tìm khoảng cách giữa 2 địa chỉ là 0xffffd52c – 0xffffd4dc = 0x50 = 80.

Vậy ta sẽ làm như sau: ta nhập shellcode, rồi cho return về địa chỉ shellcode mà ta đã nhập.

Ta thử checksec của file elf:



ở đây ta thấy có phân vùng full quyền rwx, và no pie, luôn bắt đầu từ địa chỉ 0x8048000.

Vậy giờ ta tìm địa chỉ để ghi shellcode vào. 

Bật vmmap lên, ta thấy có 6 phân vùng cho phép write, nhưng ta không nên ghi vào các phân vùng libc.

Ta thấy phân vùng stack cũng cho phép write nhưng vì môi trường thực thi có bật aslr cho nên nếu ta ghi vào stack, ta sẽ không thể lấy lại địa chỉ của shellcode mà quay về, vì nó luôn random. Vậy ta phải chọn vùng nào đó để nó không thay đổi qua mỗi lần chương trình được thực thi, lúc nãy ta đã thấy chương trình no pie , luôn bắt đầu từ địa chỉ 0x804000 nên địa chỉ này sẽ cố định.

Ta chọn vùng từ 0x804a000 đến 0x804b000, vì vùng này cố định và cho phép ta ghi dữ liệu vào.

Vậy làm sao để điều khiển được vị trí ghi vào của hàm gets?

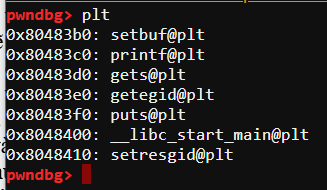
Ta sẽ phải gọi nó 1 lần nữa và đưa địa chỉ tham số cho nó là địa chỉ mà ta cần ghi, sau đó cho nó return về địa chỉ shellcode vừa ghi luôn .

Payload của ta sẽ có dạng:

‘a’ \* 0x50 + [địa chỉ của gets] + [địa chỉ tham số của gets để ghi shellcode] + [địa chỉ để gets return về = địa chỉ shellcode vừa ghi]

Vậy ta sẽ tìm địa chỉ hàm gets. Trong chương trình, khi file được thực thi, nó sẽ tìm địa chỉ các hàm trong thư viện và lưu vào 1 vùng nhớ tên là plt (Procedure Linkage Table) .

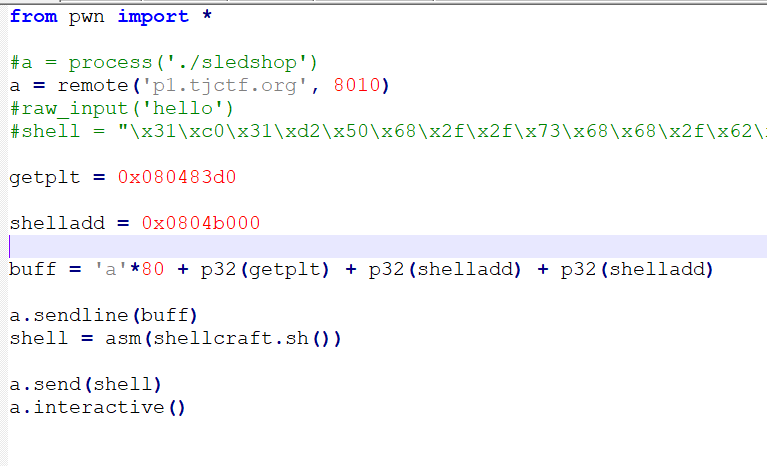
Vùng nhớ này chứa địa chỉ các hàm cần dùng trong chương trình để khi chương trình gọi lại thì nó sẽ không cần tốn thời gian tìm trong thư viện libc nữa.



Ta thấy địa chỉ của hàm gets nằm ở 0x80483d0.

Ta chọn vị trí ghi vào ở địa chỉ 0x804a000.

Payload của ta sẽ là: ‘a’+0x50 + 0x80483d0 + 0x804a000 + 0x804a000

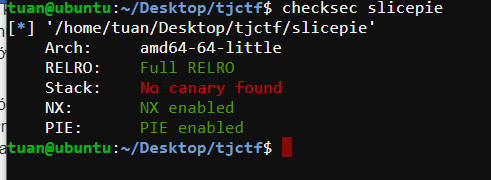


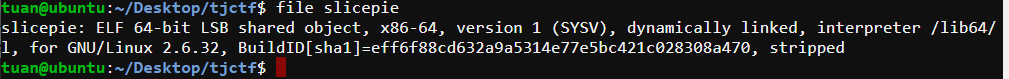
Và đến đây ta đã hoàn thành bài này.

Slice\_of\_pie:

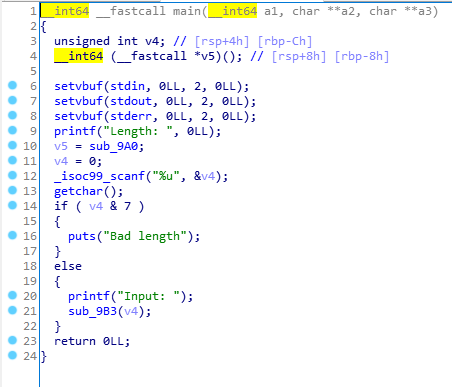
Bài này mình ăn hành khá nhiều. Gần như là full sự giúp đỡ từ các anh. Nên viết ra để học tập thôi.

Sau khi tải file về, mình sẽ kiểm tra file.

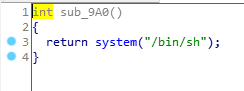




File elf 64 bit và có stripped,có pie, có nx(not excute) , khá là căng và ta không thể debug bằng gdb được vì nó đã xóa hết tên hàm. Dùng phần mềm IDA để debug chương trình.



Phân tích hàm main, đầu tiên là khai báo một biến không âm v4, sau đó nó gán địa chỉ của hàm sub\_9A0 cho v5 , ta xem hàm sub\_9A0 thực hiện cái gì.



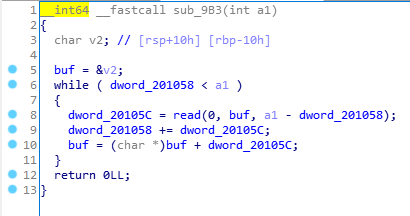
Hàm này trả về địa chỉ của hàm system gọi shell.

Tức là v5 sẽ là địa chỉ hàm call shell.

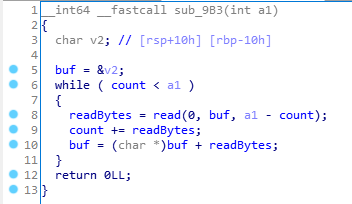
Tiếp theo chương trình sẽ cho nhập vào v4 1 số nguyên không âm. Rồi AND v4 với 7 , nếu !=0 thì nó sẽ in ra “Bad length” và kết thúc chương trình, nếu == 0 thì nó sẽ gọi hàm sub\_9B3 với tham số là số v4.

Tức là chương trình này bắt chúng ta phải nhập số bội với 8, vì 8&7 == 0.

Phân tích hàm sub\_9B3:



Mình sửa 1 số biến để dễ phân tích hơn



Tham số hàm này là v4, vào trong hàm nó biến thành a1.

Đầu tiên buf = địa chỉ của chuỗi char v2.

Hàm này sẽ so sánh biến count = 0 với a1, nếu nhỏ hơn thì nó sẽ cho chúng ta nhập a1- count vào biến buf và trả về số lượng bytes ta đã nhập thành công vào biến readBytes.

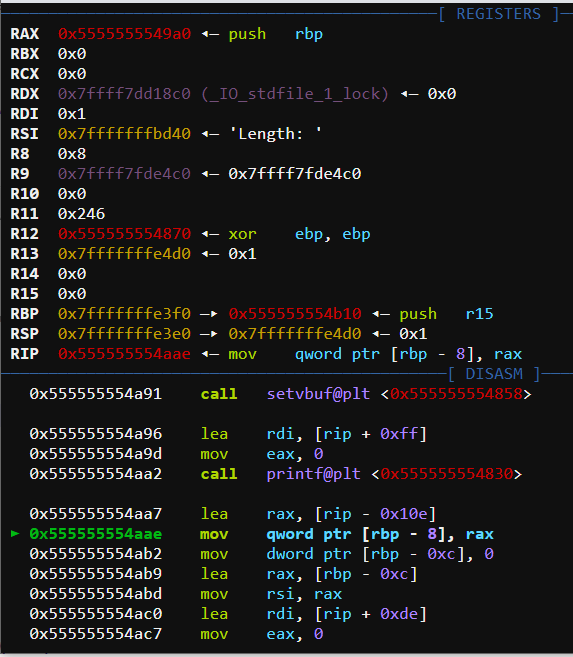
Tiếp theo nó cộng số kí tự đã nhập cho biến count để đếm được bao nhiêu bytes đã nhập.

Tiếp theo nó tăng địa chỉ của buf lên.

Rồi quay lại kiểm tra count < a1. Hàm này dừng lại khi count >= a1. Ta thấy ở đây a1 = v4 , là giá trị mà ta nhập vào, tức là ta có thể kiểm soát số lượng kí tự nhập vào, nên có thể exploit lỗi tràn bộ đệm.

Vậy mục tiêu của ta là làm sao để call được cái hàm sub\_9A0. Ta sẽ lợi dụng return của hàm sub\_9B3, ghi đè thành địa chỉ của hàm sub\_9A0, để làm được vậy, ta cần tính toán địa chỉ của buf và địa chỉ return của sub\_9B3, sau đó làm tràn địa chỉ return thành địa chỉ sub\_9A0.

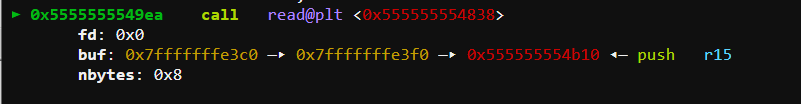
Tìm địa chỉ sub\_9A0:



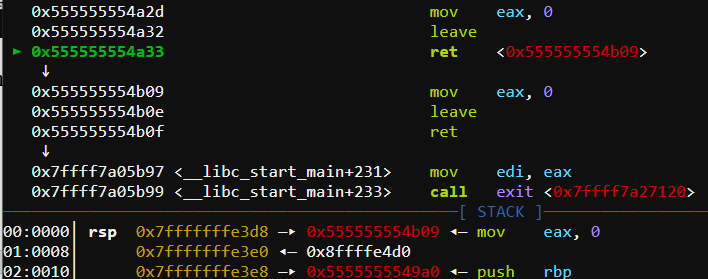
Ta thấy sau khi call print, nó sẽ lea rax, [rip – 0x10e] , tức là sẽ đưa địa chỉ hàm sub\_9A0 cho rax, rax = 0x5555555549a0. Tức là hàm call shell nằm ở địa chỉ 0x5555555549a0.

Tiếp tục debug để tìm offset giữa buf và ret của sub\_9B3.

Ta debug và thấy địa chỉ biến buf nằm ở: 0x7fffffffe3c0



Ta tiếp tục để tìm địa chỉ return của hàm sub\_9B3:



Đây là địa chỉ return của hàm nằm trong stack vị trí 0x7fffffffe3d8.

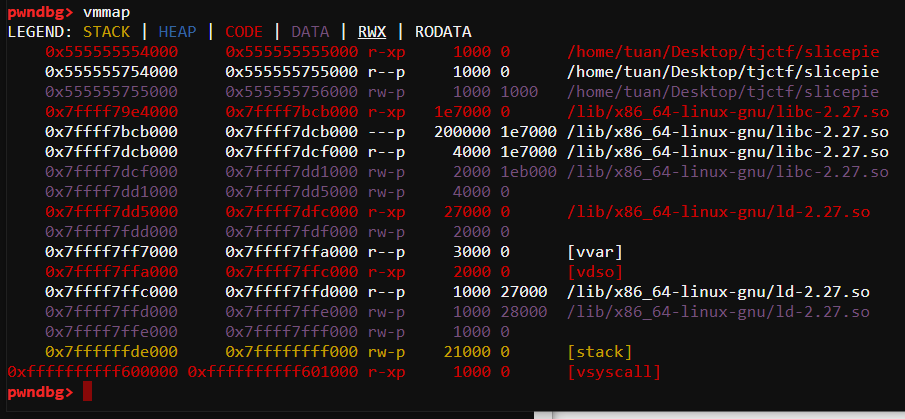
Ta tìm khoảng cách giữa 2 vị trí: 0xe3d8 – 0xe3c0 = 0x18 = 24.

Vậy ta có thể làm tràn địa chỉ return của sub\_9b3 và ghi thành địa chỉ của sub\_9a0, nhưng chương môi trường có bật aslr nên mỗi lần chạy thì địa chỉ của hàm sẽ hoàn toàn khác nhau.

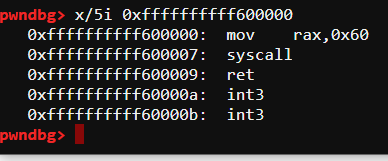
Ta để ý thấy địa chỉ của hàm sub\_9A0 nằm ngay dưới địa chỉ return của hàm sub\_9b3. Vậy ta phải làm sao để return về địa chỉ dưới thay vì địa chỉ trên? Nếu ta có thể pop rsp ra thêm 1 lần nữa và return thì nó sẽ cho chúng ta return về sub\_9a0 mà không cần phải biết chính xác địa chỉ của hàm, nhưng làm sao có thể chèn địa chỉ của câu lệnh, ta cũng không có câu lệnh có sẵn, nghe có vẻ giống rop chain, ta sẽ build 1 rop\_chain. Nhưng dù cho có build rop chain thì nó vẫn bị thay đổi vì aslr.

Tức là ta không cách nào xác định địa chỉ của rop chính xác qua mỗi lần chạy.

Vậy ta thử tìm có khoảng nào mà sẽ không đổi, mở vmmap lên:

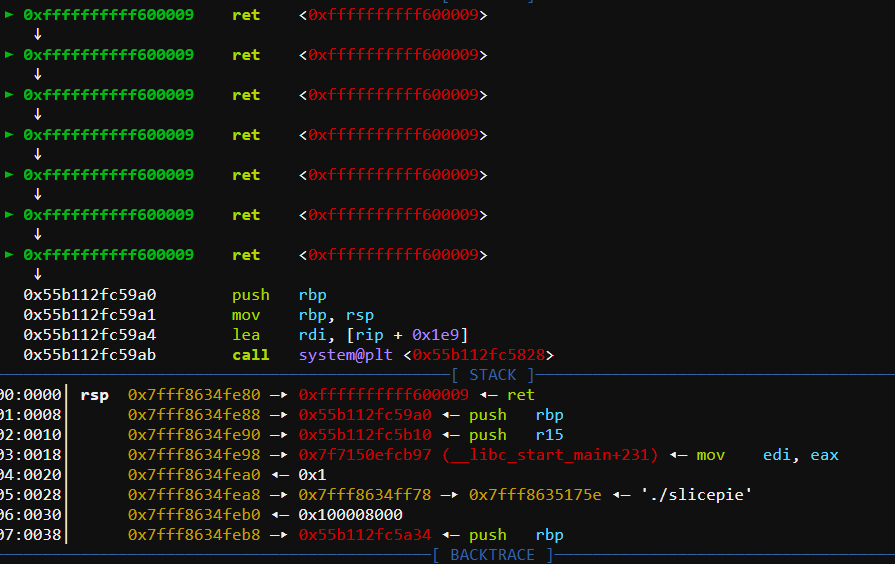


Ta thấy có 1 vùng tên là vsyscall, sau khi search google thì mình thấy hàm này sẽ không thay đổi dù ta có thực thi bao nhiêu lần và vùng này chứa được nhiều nhất 4 lệnh.



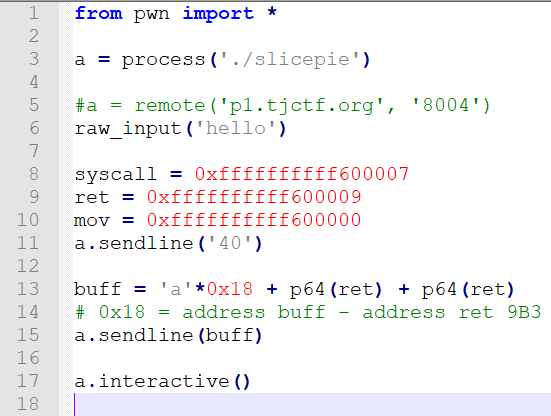
Ta đã có 3 dòng ROP của vsyscall, và các lệnh này luôn cố định.

Ta thử ghi đè địa chỉ return của hàm sub\_9b3 bằng địa chỉ hàm ret trong vsyscall, để bỏ qua lần ret thứ nhất, lần ret thứ hai sẽ là địa chỉ của sub\_9A0.



Mình bị lỗi gì đó. Thử với lệnh mov trong vsyscall thì chạy được.

Payload:



Printf\_polyglot:

Bài này là kiến thức mới. Áp dụng lỗi tên là format string bug.

Đọc đề ta có thể thấy trong hàm newsletter có gọi 2 lần câu lệnh printf(email) . Mà email là chuỗi chúng ta được phép nhập = > format string.

Ý tưởng bài này là ghi đè địa chỉ plt của printf.plt thành hàm system để lần gọi thứ printf(email) thứ 2 , thay vì gọi printf nó sẽ gọi thành system(email) . nếu chuỗi email = ‘sh;…..’ thì chúng ta sẽ có lệnh system(“sh”) và gọi được shell.

<tạm ghi lại ý tưởng chứ chưa có tgian viết kĩ>