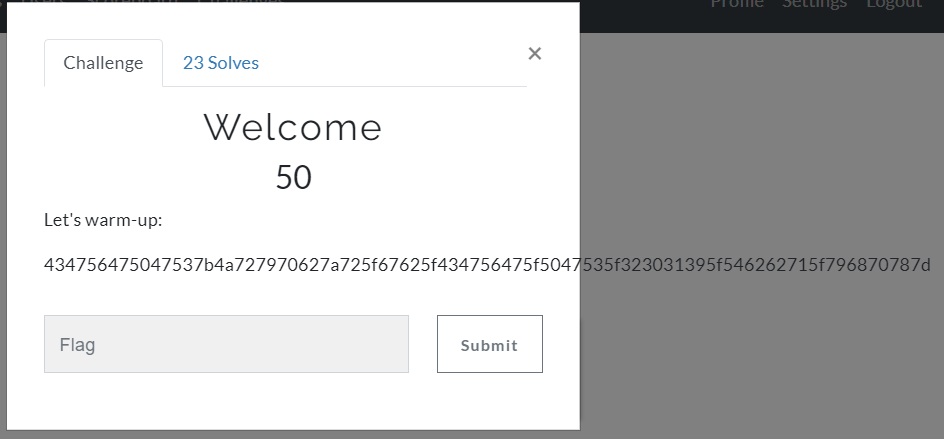
Write up PTITCTF 2019

Bài 1: welcome



Đây là bài mở đầu nên khá dễ. Mình đọc đề thì đoán đây là 1 chuỗi hex (ai học pwn với RE dễ đoán ra lắm) .

Nên mình dùng tool ở kt.gy để decode lại.

Sau khi decode thì mình nhận được chuỗi này :



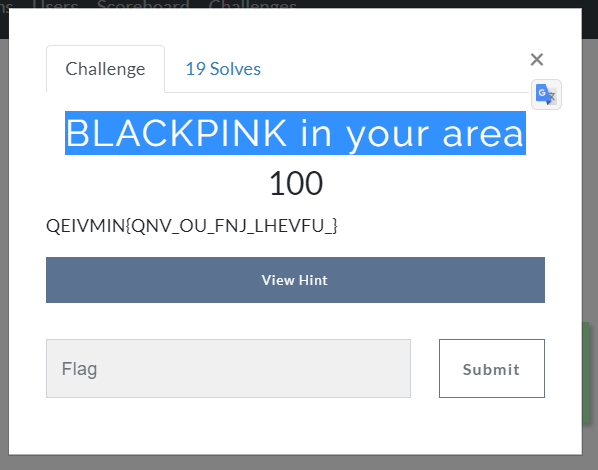
Flag có dạng: PTITCTF{

Và CGVGPGS{ . Nhìn kĩ thì đây là casear cipher. Dịch tuần tự các kí tự. Lại đưa vào brute-force ROT.

ở ROT+13 thì ta có:  

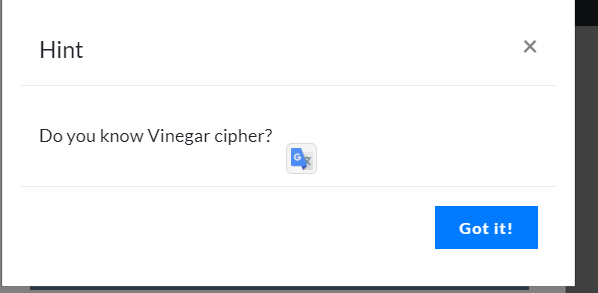

submit thôi !

Bài 2: BLACKPINK in your area



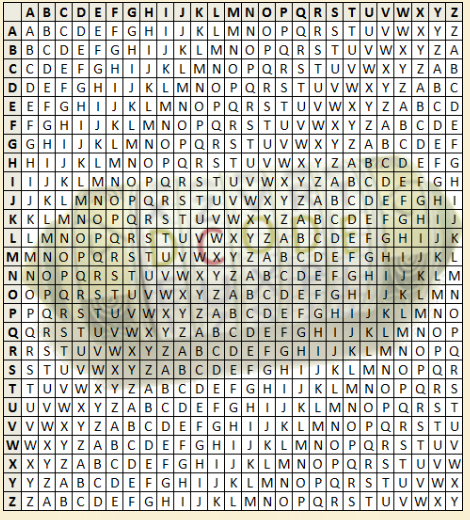
Ta nhận được 1 chuỗi mã hóa, nhìn vào dễ dàng nhận ra đây là flag bị mã hóa vì nó có dạng của flag.

Ban đầu mình không chuyên phần này nên chưa nhận ra. Sau đó có hint là



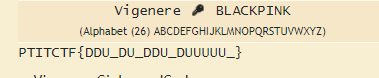
Thì mình search google vinegar cipher.

Mã hóa này tạo 1 bảng shift 26 chữ cái, với hàng đầu tiên là 26 chữ cái và mỗi hàng là dịch qua bên trái 1 ký tự so với hàng trên nó.

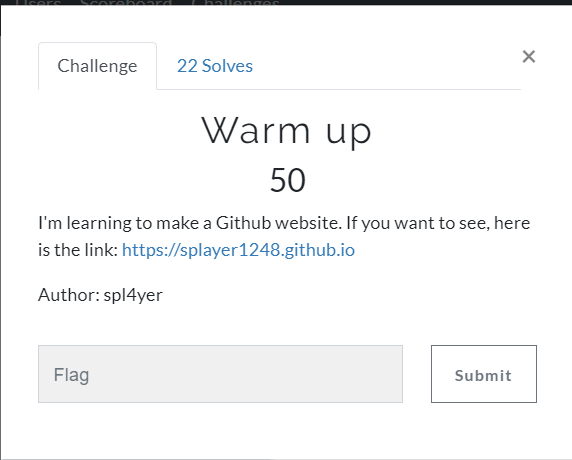


Để mã hóa thì chúng ta cần plaintex, key. Với mỗi kí tự mã hóa nhận được là kí tự ở hàng plaintext và cột ở key tức là công thức sẽ như thế này:   
- cipher[i]= word[plaintext[i]][key[i]]

Vì ta đã biết flag có dạng PTITCTF{…} nên ta sẽ lấy được 7 kí tự đầu tiên của key bằng cách đối chiếu với bảng trên. Tìm ra key = BLACKPINK. Vì mình ngáo nên không nhận ra đề cho sẵn key, chứ chả tự nhiên gì chữ BLACKPINK được in to đùng như vậy.

Mình dùng tools ở đây : <https://www.dcode.fr/vigenere-cipher> để decodes cipher trên và nhận được flag : 

Bài 3: web warm up



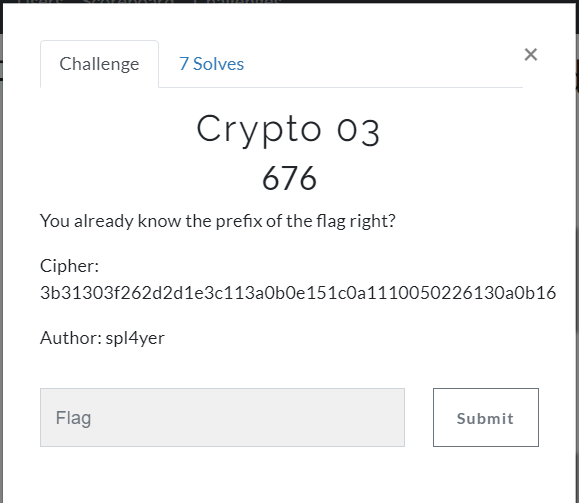
Mình không chuyên web nên chỉ có thể áp dụng câu nói thần thánh của các anh là “xem source code của bài web” và….



Hành trình làm web của mình đến đây là kết thúc, vì mấy bài sau thì xem source chả có gì.

Bài 4: crypto/acm

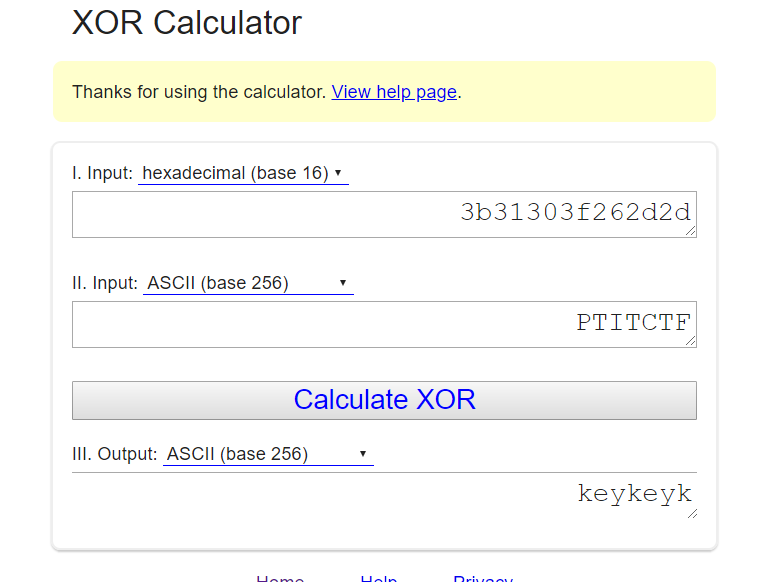
Bài này thì mình đã gặp trong lần thi pis 1. Cái này là đặc sản của anh K0M4ng luôn.



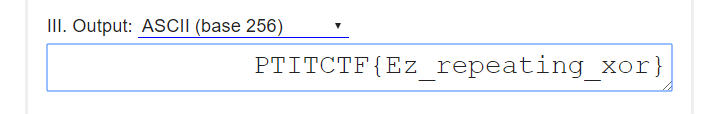
The prefix of flag = PTITCTF{ .Sau khi mình search hết tất cả các kiểu cipher, các kiểu hash mà vẫn bí. Một chút gợi ý thì mình biết đây là bài XOR với key repeat. Key sẽ lặp lại cho đến hết length của flag.

Ta có flag[i] xor key[i] = cipher[i], thì flag[i] xor cipher[i] = key[i] . Nên ta có thể lấy chuỗi cipher xor lại với prefix của flag để tìm 1 đoạn key.

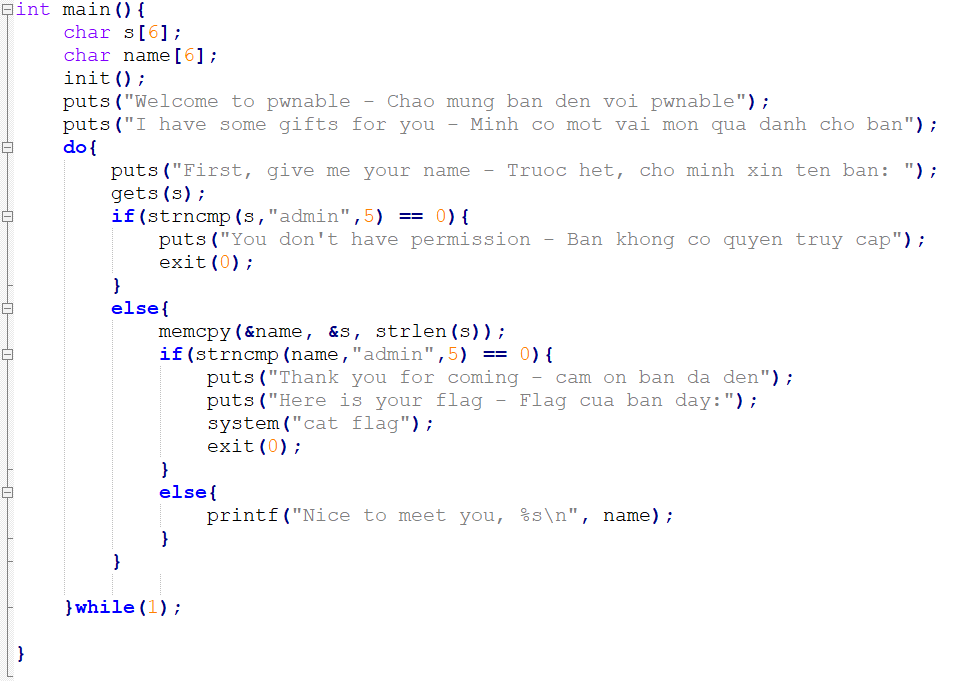
[http://xor.pw/#](http://xor.pw/) tools



Đây là kết quả. Ta thấy output = keykeyk… tức là đã lặp lại, ta có được từ khóa là “key” . giờ chỉ cần đếm length của chuỗi cipher rồi điền key cho đủ và xor là sẽ ra flag.



Bài 5: pwn01

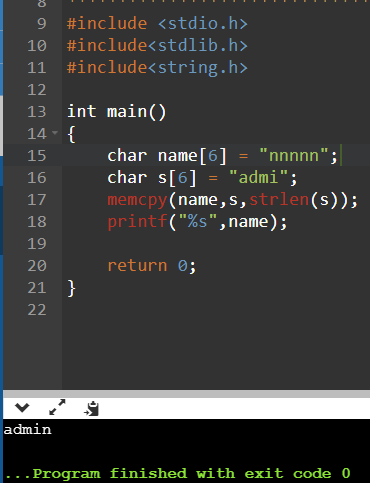


Đầu tiên mình sẽ phân tích về code:

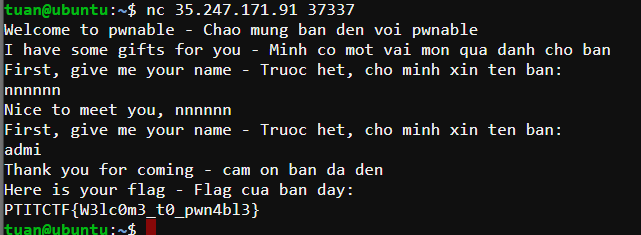
Chương trình khai báo hai mảng kí tự s[6] và name[6] . Vào vòng lặp do while, chương trình sẽ cho nhập chuỗi s với hàm gets() . Sau đó so sánh 5 kí tự đầu của chuỗi s với chuỗi “admin” . Nếu giống nhau thì chương trình sẽ exit(0) tức là thoát. Còn nếu khác thì chương trình sẽ copy chuỗi s vào chuỗi name rồi lại so sánh 5 kí tự đầu của chuỗi name với chuỗi “admin” , nếu bằng thì cho chúng ta “cat flag” và exit(0) còn nếu không bằng thì lặp lại vòng lặp.

Đề bài khá là kì lạ vì ở trên chuỗi s thì không cho bằng “admin” nhưng sau khi copy chuỗi s qua chuỗi name thì lại bắt bằng “admin” mới cho cat flag. Tức là logic của nó sẽ thế này:

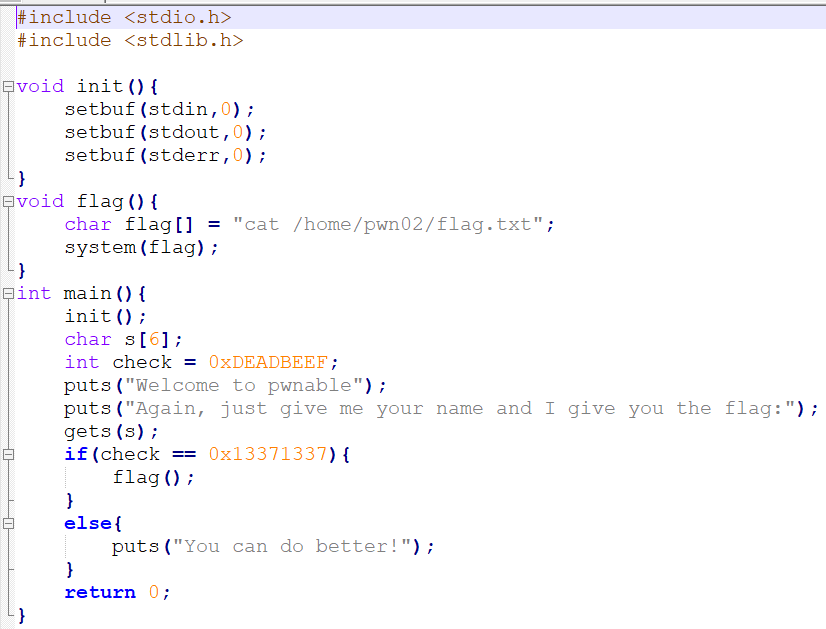
s != “admin” = name = s => s !=s , qúa là sai trái luôn. Mình suy nghĩ cách bypass chỗ này. Mà thực ra nó cũng đơn giản lắm. Mẹo ở bài này là chương trình copy bằng hàm memcpy chứ không phải hàm strcpy() . Hàm strcpy(dest, source) sau khi copy chuỗi source vào dest , nó tự động thêm 1 kí tự kết thúc vào cuối chuỗi. Còn với hàm memcpy() chỉ copy n bytes từ bộ nhớ chuỗi source qua bộ nhớ chuỗi dest. Mình có code 1 đoạn nhỏ bên dưới để so sánh 2 hàm strcpy và memcpy.

Và đến đây thì ta đã có hướng bypass chuỗi if vô lý bên trên bằng cách nhập 2 lần qua 2 vòng while, vòng while đầu tiên ta nhập một chuỗi ‘n’ \*x với x >= 6 kí tự vào s để sau khi qua hàm memcpy thì chuỗi name[6] = “nnnnnn” . Vì name != ‘admin’ nên chương trình sẽ lặp lần 2 và cho chúng ta nhập lại 1 lần nữa, lần này ta nhập chuỗi s = “admi” , vì “admi” != “admin” nên chương trình vẫn cho ta tiếp tục, sau khi qua hàm memcpy() , copy 4 bytes của chuỗi s vào chuỗi name, thì khi đó chuỗi name[6] = “adminn” . tiếp theo strcpy(name, “admin” ,5) == 0 , nó sẽ so sánh 5 bytes đầu của chuỗi name nên ta không cần lo lắng về “n” bị dư. Và thế là ta đã bypass được chương trình và cat được flag.



Bài pwn02:

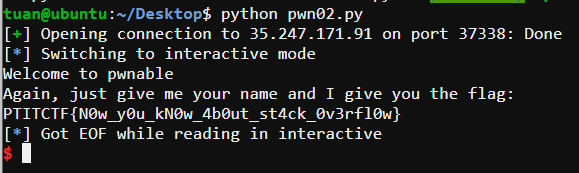


Đầu tiên mình sẽ phân tích code:

Trong hàm main, ta thấy chương trình khai báo 1 mảng kí tự s[6] và 1 biến check = 0Xdeadbeef, sau đó chương trình cho ta nhập vào s qua hàm gets( ) , sau đó kiểm tra nếu check = 0x13371337, nếu bằng sẽ cho call hàm flag. Vậy mục đích rõ ràng là làm sao cho biến check = 0x13371337. Nhưng chương trình đã khởi tạo sẵn giá trị cho check và không cho nhập lại giá trị của check thì làm sao mà bằng được.

Ta thấy ở hàm main, chương trình cho nhập s bằng hàm gets( ) , hàm này không kiểm tra chúng ta đã nhập bao nhiêu ký tự vào chuỗi nên hàm này gây ra lỗi tràn dữ liệu (buffer overflow) . Ta thấy chương trình khai báo mảng s[6] ngay sau biến check, tức là nếu s bị tràn thì dữ liệu sẽ tràn ngay xuống check, lúc đó check sẽ chứa dữ liệu bị tràn. Vậy ta chỉ cần nhập full 6 kí tự của s và giá trị cần thay đổi của check = 0x13371337.

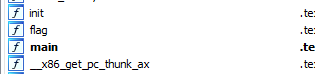
Chuỗi buffer = ‘a’\*6 + ‘\x37\x13\x37\x13’ . và ta đã có flag



Bài pwn03:

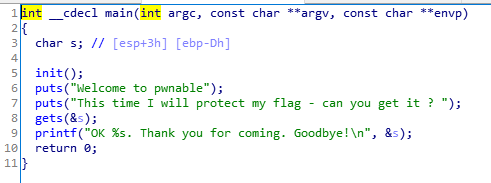
Bài này khiến mình tốn bao nhiêu bánh, nước của BTC mới làm ra đấy .

Bài này không cho source code nên chúng ta sẽ xem mã giả bằng phần mềm IDA.

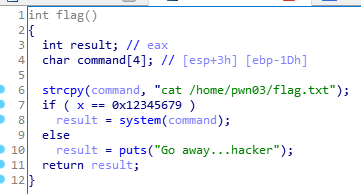


Sau khi mở với IDA, chúng ta thấy chương trình có 2 hàm là flag( ) và main.

Hàm main( ) :



Hàm flag() :



Đầu tiên mình sẽ phân tích code:

Trong hàm main cho chúng ta nhập vào mảng ký tự s bằng hàm gets() , sau đó … không có sau đó nữa vì chỉ nhiêu đó là đã kết thúc main. Nó không hề gọi hàm flag( ) thì làm sao lấy flag.

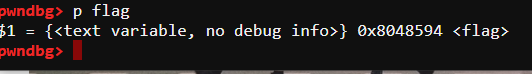
Trong hàm flag( ) , chúng ta thấy nó khởi tạo mảng kí tự command[4] rồi copy chuỗi lệnh vào ,để chứa kệnh “cat …”. Sau đó chương trình kiểm tra nếu x == 0x12345679 thì mới call system. Còn nếu sai thì chương trình sẽ in ra chuỗi “Go away…hacker” .

Lại 1 thứ kỳ lạ nữa, x còn chưa được khai báo , cũng không được nhập vào.

Vì chương trình cho nhập chuỗi s bằng gets( ) mà hàm này không kiểm tra lượng dữ liệu nhập vào nên mình có hướng làm là ghi đè địa chỉ return của hàm main thành địa chỉ của hàm flag, để sau khi chạy xong main thì chương trình sẽ chạy tới hàm flag. Để làm được vậy thì mình cần tìm khoảng cách giữa chuỗi s và địa chỉ return trên stack, cần địa chỉ của hàm flag( ) .

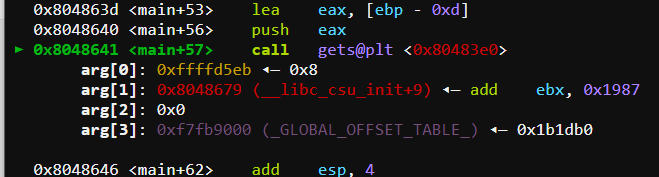
Mình debub bằng pwndbg.

Để lấy địa chỉ hàm flag, mình tìm hàm flag trên dbg.

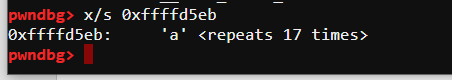


Ta đã có địa chỉ flag\_address = 0x8048594. Tiếp tục debug main để tìm khoảng cách.

Đến hàm gets( )



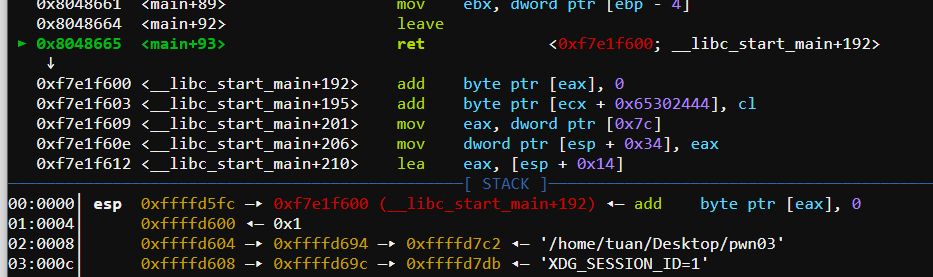
Ta thấy hàm gets( ) nhận tham số ở địa chỉ 0xffffd5eb. Mình nhập thử 1 chuỗi “aaaaaaaaaaaaaaa” để xem có cho chắc chắn.



Đúng rồi, chuỗi mình nhập vào gồm 17 chữ ‘a’, dbg nó hiển thị rút gọn thôi.

Gọi gets\_add = 0xffffd5eb

Tiếp theo tìm địa chỉ return của hàm main.



Hàm main return về địa chỉ chứa ở địa chỉ 0xffffd5fc. Gọi return\_add = 0xffffd5fc

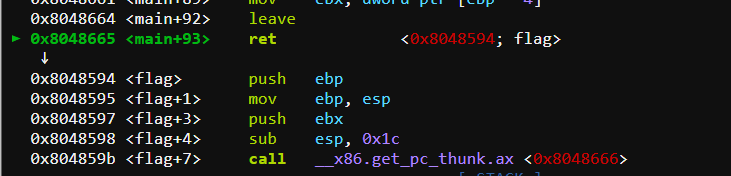
Vậy là ta đã tìm được địa chỉ return và cả địa chỉ chuỗi s. Tính khoảng cách (offset) :

Offset = return\_add – gets\_add = 0x11

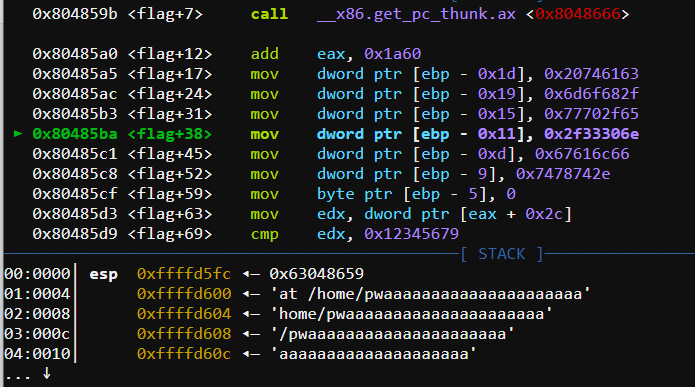
Vậy để ghi đè được địa chỉ return của main, ta cần 0x11 ký tự + địa chỉ mới ta cần. vì ta cần return về hàm flag( ) nên:

Ta có buffer = 0x11\* ‘a’ + flag\_address

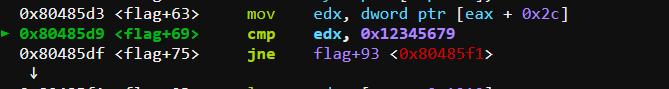
Thử xem đã đúng chưa.



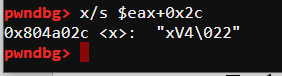
Ta đã thấy sau main nó ret về hàm flag rồi, đúng như ta mong muốn, giờ debug flag xem nó làm gì.

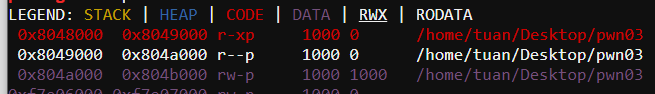


Đây là chương trình đang copy chuỗi cat flag vào command. Tiếp tục debug.



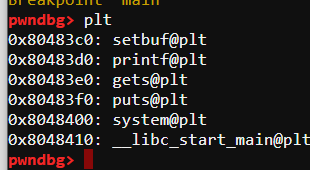
Ta thấy ở đoạn này chương trình cmp (so sánh) edx với giá trị 0x12345679. Nhưng  . Thế là đã khác rồi. Ta sẽ không lấy được flag nếu không làm cho edx = 0x12345679. Ta thấy trước lệnh cmp, chương trình mov giá trị tại eax+0x2c cho edx, ta cũng xem tại đó là gì.

 ta thấy eax+ 0x2c = 0x804a02c <x> (biến x lúc nãy đọc mã giả ta đã nói) và nó không chưa giá trị ta cần. Xem địa chỉ trong vmmap

 ta có thể thấy biến x nằm trong vùng có địa chỉ từ 0x804a000 đến 0x804b000, đó là vùng DATA. Và vùng này cho phép chúng ta có quyền write ( w) .

Tức là ta có thể ghi dữ liệu vào vùng này. Mình dùng hàm gets( ) để ghi dữ liệu mình vào vùng nhớ này để khi so sánh trong hàm flag thì cho chúng ta call system .

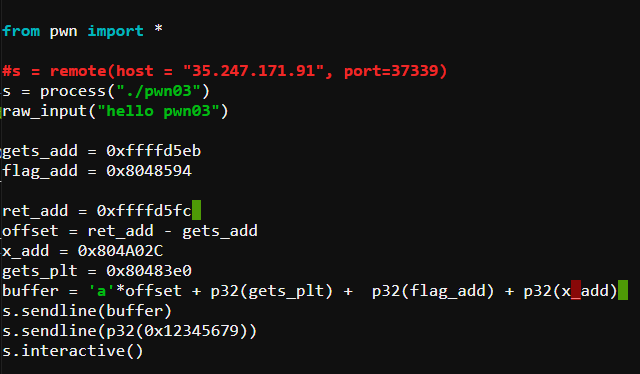
Vậy giờ thay vì return thằng về flag như ban nãy thì chúng ta return về hàm gets( ) truyền tham số cho hàm gets là địa chỉ của biến x = 0x804a02c và sau đó truyền địa chỉ return sau khi hàm gets chạy xong là hàm flag.

Để call được hàm gets thì ta phải lấy địa chỉ của gets trong bảng plt. 

Chuỗi buffer = 0x11\*’a’ + gets\_plt + flag + x

Sau đó chúng ta sẽ gửi giá trị vào x là 0x12345679.

Code exploit:



Run and get flag.

