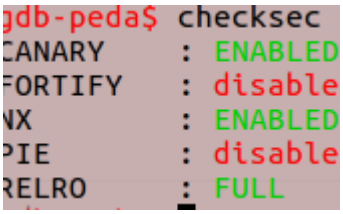
CTF Hw6 writeup

b05902127 劉俊緯, user ID: a127000555

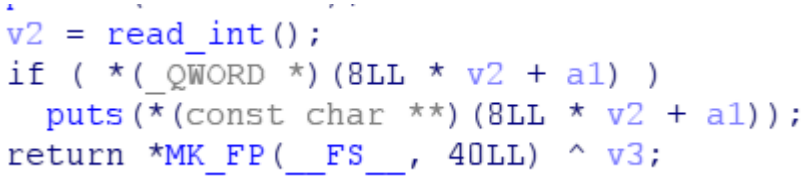
Wtfnote

FLAG{WTF!?!?D1d_y0u_r3411y_s0lv3D_THis???}

Checksec



Leak_stack



- 開IDA就發現，printe_note的函式中，並沒有做lim>=index>=0的防護，因此可以任由我們leak。
- Leak的規則： 假設要leak a，必須要存在一條chain使得c->b->a。因為在code中拿出array的是heap指標，而印出資料會是在heap指標的所指的值。
- 首先我們要leak stack。通常會形成這種長鍊的都是rbp，因此對目前指到的rbp的位置(<__libc_csu_init>)做refsearch，就會得到一條更長的rbp chain:

```
0x7fffffffda00 --> 0x7fffffffdae0 --> 0x7fffffffdb00 --> 0x7fffffffdbc0 -->
0x555555554e50 (<__libc_csu_init>:  push   r15)
```

- 我們抓中間三個好了(抓前三個在puts會炸)，因此我們要讓array讀到aslr + 0x7fffffffdae0，它就會吐給我們aslr+ 0x7fffffffdbc0，此時$rsp是0x7fffffffdb10，所以結論就是main的rsp = *ary[(0x7fffffffdae0-0x7fffffffdb10)/8] - (0x7fffffffdb10 - 0x7fffffffdbc0)。
    - 整理一下就是*ary[-6]-0xb0。

Leak_libc

- 起初我想用同樣的方式(refsearch)找一串chain然後leak出libc function的值，但是中間都會過一層ASLR，以失敗告終。

- 換句話說，我們只能自己做出一條chain了。重點在於read_int的buffer夠大，atoi不用吃掉整個buffer，我們就可以某種程度上的操縱stack。
- 因此，我們現在要先考慮的是：該leak哪一個值：
  - 由`info proc map`可以知道目前的libc的aslr到哪一個位置，並觀察stack上的資訊。
  - 在stack海中，只有難得的一條address會帶有資訊，以下是aslr得到的消息，而這個地址位在main_rsp+184：

```
# 0x7f14b890d830 <__libc_start_main+240>:   0x31000197f9e8c789
```

  - 因此，我們只要leak出這個值，就可以知道目前libc_base在哪裡。
- 接下來就是要觀察atoi的buffer到底會對應到main_rsp的哪個位置，這個就胡亂print一通就可以觀察到了。
  - 最後我們會得到read_int 的buffer扣掉前半段的index會是當時rsp*(-25*8),而這個值要指向上述的main_rsp+184。
- 如此一來，我們就可以leak __libc_start_main，扣掉offset後正式得到libc。

Leak_heap

- 同Leak_libc，比較輕鬆的是，在allocate過後，我們直接leak main_rsp就可以了。因為main_rsp剛好會存第一個heap的指標。

Double Free - Fast dup attack

- 不幸的，這題並沒有UAF。

```
free(*(void **)(8LL * v2 + a1));
*(_QWORD *)(8LL * v2 + a1) = 0LL;
}
```

- 更不幸的，它會檢查這格目前有沒有被free過。

```
if ( *(_QWORD *)(8LL * v2 + a1) )
{
```

- 因此再原本的routine: free0 -> free1 -> free0之中，我們需要騙過一次free，讓程式在fastbin中造成 01 loop。這個時候我們其實只需要在第三次free時，直接不由idx0指向main_rsp，而是自己製造一個idx指向main_rsp就夠了。而製作方法就會像上面的一樣，控制stack造成控制指標。
- 詳細就直接：

```
r.sendafter('> ','3'.ljust(8,b'\x00')+p64(leak_heap_start))
r.sendafter('index: ',str(-23))
```

BOF

- 我們已經leak出libc_base，也可任意寫特定位置了(因為Fast dup attack)，下一步就大概是BOF+ROP/one gadget囉！
- 要找適合的return address的話，new_node就是個不錯的選擇。因為當下read_int可碰到。至於要找到這個return address到底在哪，可以從gdb在new_node下breakpoint後，觀察rbp + `info frame`就可以知道

```
RBP: 0x7fff81cb7ed0 --> 0x7fff81cb7f90 --> 0x5581c1b2de50 (<__libc_csu_init>:   push
  r15)


0x7fff81cb7e78: 0x4646464646464646  0x0000000000000061
0x7fff81cb7e88: 0x13ebf44848d48a00  0x00007fff81cb7ed0
0x7fff81cb7e98: *0x00005581c1b2db8c*    0x0000000000000000
0x7fff81cb7ea8: 0x00007fff81cb7ee0  0x00007fff00000003
```

如果要從`7e78`開始寫，那麼這格得位置會是`main_rsp (7fff81cb7ee0) - 0x68`，
並且寫的時候要確保右上角是`0x61 / 0x60`。(`fastnote`會檢查大小。)
而要填入多少`overload`才可以到`7e78`，用測的就好了。因為這些位置都是相對固定的。

- 最後，在這個地方(alloc)隔著16個垃圾接上ROPgadget配上one_gadget就可以get shell了！