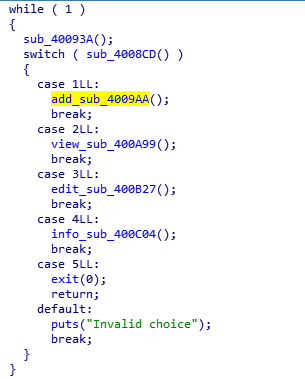
一个完整的house\_of\_orange解析

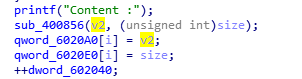
看了几篇相关分析后的总结

1. 分析程序功能：

1》整体功能:

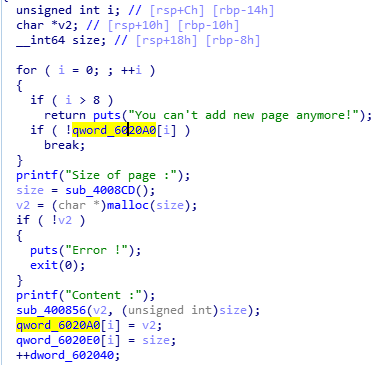


2》关键地址：



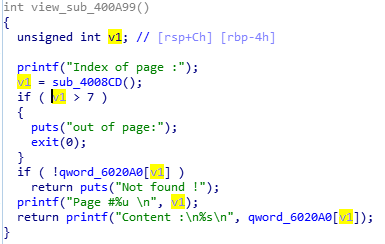
两个数组存放书页的地址和书页内容大小，两数组在BSS段上位置相邻。

3》add：



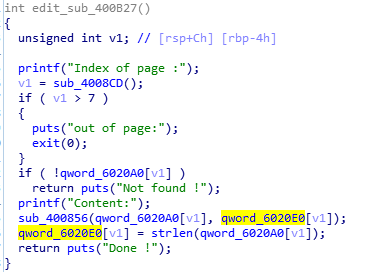
如上图，从0到8顺序查找的方式，进行堆块申请，堆块大小由用户输入，此时用户可获得一次输入堆块内容的机会。

3》view：

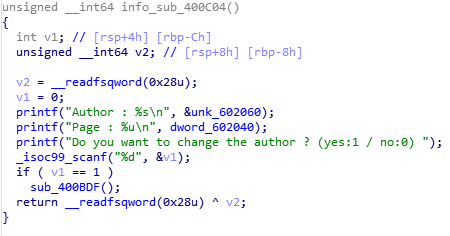


用户可输入0~7的数字查询书页内容。

4》edit：



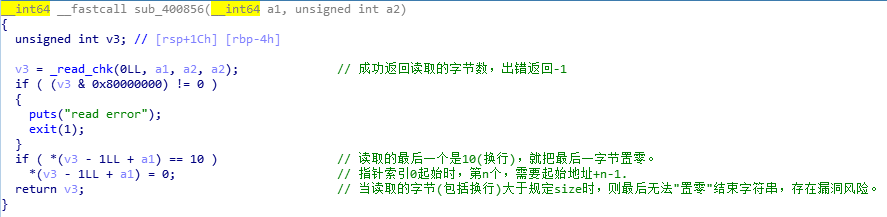
5》info\_show



2 漏洞分析

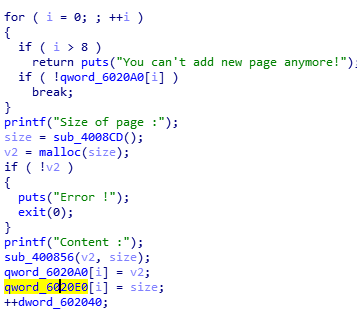
1》func\_sub\_read函数存在漏洞（字符串输入溢出）

由于strlen()更新size，进入edit函数，发现存在输入n个字符的情况，不能null断字符。



* 1. author name字符串溢出，导致泄露6020a0上的堆地址
  2. edit时内容溢出，导致链接到topchunk的size位置处，再次edit就可以修改topchunk的size了

2》索引溢出有效



当i=8时，&page\_addr[8]刚好是&page\_size[0],满足条件。好像没法用。

//有用第二次edit修改topchunk大小时，前用”/0”填充，strlen求edit后的str长度时就会得到chunk的新size为0.

3 漏洞利用

3.1》利用思路

整个程序中没有出现free函数，常规的UAF、Double free都不存在。House\_of\_orange攻击成功需要如下条件：

house\_of\_orange思路简介：

首先修改topchunk的size，然后申请一个较大的块，当size满足一定条件时，原来的top会被释放到unsorted bin。

通过堆溢出覆写原top(现unsort\_bin\_chunk)内容，主要是构造IO\_file\_plus指针中的函数虚表(在io\_file\_jump结构+0x18处中覆盖system函数地址)，并伪造bk指针为unsorted bin攻击做铺垫。

当再次申请内存时，造成unsorted bin attack，将\_\_IO\_list\_all覆写为unsort\_bin\_chunk地址(原top头地址,见unsorted\_bin\_attack)，由于unsorted bin结构的破坏(晚于unsort\_bin\_attack)，程序异常，会在malloc中调用malloc\_printerr函数进行错误打印，在malloc\_printerr中调用\_\_libc\_message，进一步调用abort，再调用 \_IO\_flush\_all\_lockp，在其中调用了\_IO\_OVERFLOW(fp,EOF)，这个函数是使用虚表调用，如果可以覆盖调用的虚表，就可以达到执行system('/bin/sh')。

3.2》利用实现

《1》泄露libc地址

因为程序中有堆的越界写，可以修改top\_chunk的大小。在malloc源码里面如果申请的堆块大小超过了top\_chunk的大小，将调用sysmalloc来进行分配。

sysmalloc里面针对这种情况有两种处理，一种是直接mmap出来一块内存，另一种是扩展top\_chunk

/\*

If have mmap, and the request size meets the mmap threshold, and

the system supports mmap, and there are few enough currently

allocated mmapped regions, try to directly map this request

rather than expanding top.

\*/

if ((unsigned long) (nb) >= (unsigned long) (mp\_.mmap\_threshold) &&

(mp\_.n\_mmaps < mp\_.n\_mmaps\_max))

{

char \*mm; /\* return value from mmap call\*/

try\_mmap:

就是如果申请大小>=mp\_.mmap\_threshold,就会mmap。我们质只要申请不要过大，一般不会触发这个，这个mmap\_threshold的值为128\*1024。

不过下面还有两个assert需要检查，如下：

old\_top = av->top;

old\_size = chunksize (old\_top);

old\_end = (char \*) (chunk\_at\_offset (old\_top, old\_size));

brk = snd\_brk = (char \*) (MORECORE\_FAILURE);

/\*

If not the first time through, we require old\_size to be

at least MINSIZE and to have prev\_inuse set.

\*/

assert ((old\_top == initial\_top (av) && old\_size == 0) ||

((unsigned long) (old\_size) >= MINSIZE &&

prev\_inuse (old\_top) &&

((unsigned long) old\_end & pagemask) == 0));

/\* Precondition: not enough current space to satisfy nb request \*/

assert ((unsigned long) (old\_size) < (unsigned long) (nb + MINSIZE));

第一个assert就是要求修改后的top\_chunk\_size必须满足：

top\_chunk\_size>MINSIZE(MINISIZE)没有查到是多少，反正不要太小都行；

top\_chunk需要有pre\_in\_use的标志，就是最后一个比特为1；

还有就是(old\_end &pagemask ==0)

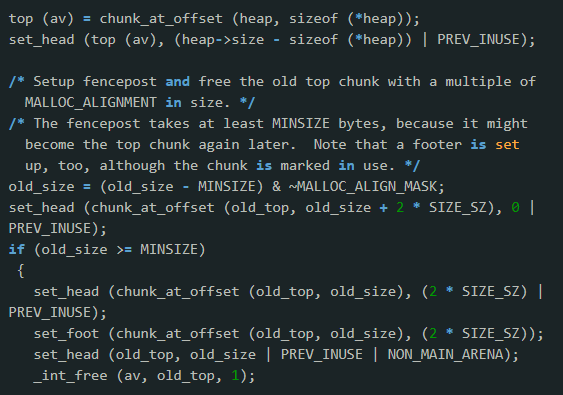
#define chunk\_at\_offset(p, s) ((mchunkptr) (((char \*) (p)) + (s)))

这里没有太深究，应该就是top\_chunk需要和原来的堆页在一个页上吧。

第二个assert就是要求

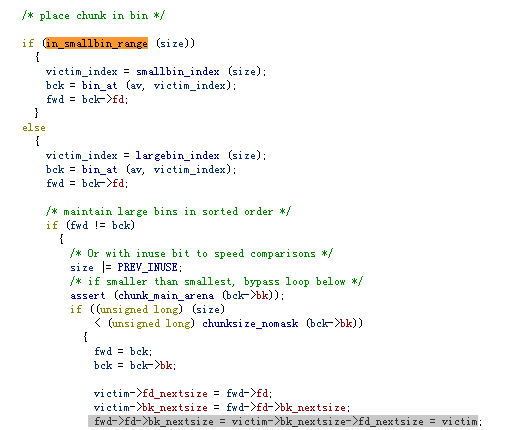
top\_chunk\_size小于申请分配的内存即可

满足以上四个条件之后，继续往下执行最后把原先的那个old\_top给释放掉了，如下：



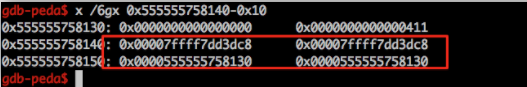
显然，这样的free操作的话，我们就可以得到一个unsort\_bin，然后之后再次分配时候如果是符合unsort\_bin大小的话，就会从unsort\_bin里面切出来。

如果我们分配的大小是large\_chunk的话。malloc源码中还把old\_top的堆地址放到了堆里面（没有细究原因，但是好像是large bin没有区分大小，需要有个字段来保存大小的原因吧），源码如下：

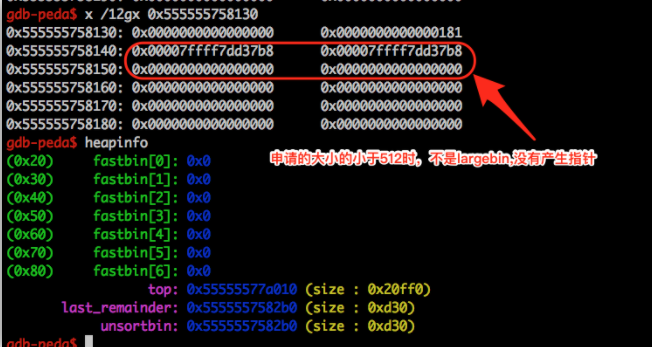


https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#1661

所以如果再次分配时候如果分配大小为largebin(也就是大于512字节)的chunk的话，就是可以既泄露libc又可以泄露heap。如下



而如果分配大小不到512字节时候是无法泄露堆地址的：



。