PEKING UNIVERSITY

高版本glibc堆利用 - House_of_apple

2022.09.16 王晗

MORESHI POWERPOINT

目录

Glibc堆管理背景

MORESHI POWERPOINT

glibc高版本逐渐移除了__malloc_hook、__free_hook、__realloc_hook等等一众hook全局变量，ctf中pwn题对hook钩子的利用将逐渐成为过去式。而想要在高版本利用成功，基本上就离不开对IO_FILE结构体的伪造与IO流的攻击。之前很多师傅都提出了一些优秀的攻击方法，比如house of pig、house of kiwi 和 house of emma等。

其中，**house of pig**除了需要劫持IO_FILE结构体，还需要劫持tcache_perthread_struct结构体或者能控制任意地址分配；**house of kiwi**则至少需要修改三个地方的值：_IO_helper_jumps + 0xA0和_IO_helper_jumps + 0xA8，另外还要劫持_IO_file_jumps + 0x60处的_IO_file_sync指针；而**house of emma**则至少需要修改两个地方的值，一个是tls结构体的point_guard(或者想办法泄露出来)，另外需要伪造一个IO_FILE或替换vtable为xxx_cookie_jumps的地址。

总的来看，如果想使用上述方法成功地攻击IO，至少需要两次写或者一次写和一次任意地址读。而在只给一次任意地址写（如一次largebin attack）的情景下是很难利用成功的。

House_of_apple

MORESHI POWERPOINT

largebin attack（Tcache Stashing Unlink Attack…）是高版本中为数不多的可以任意地址写一个堆地址的方法，并常常和上述三种方法结合起来利用。House_of_apple是一种新的利用方法，在仅使用一次largebin attack并限制读写次数的条件下进行FSOP利用。

顺便说一下，house of banana 也只需要一次largebin attack，但是其攻击的是rtld_global结构体，而不是IO流。

上述方法利用成功的前提均是已经泄露出libc地址和heap地址。House_of_apple的方法也不例外。

House_of_apple利用条件

MORESHI POWERPOINT

利用条件

使用house of apple的条件为：
1、程序从main函数返回或能调用exit函数或libc执行abort流程时
2、能泄露出heap地址和libc地址
3、 能使用一次largebin attack（一次即可）

利用原理

MORESHI POWERPOINT

当程序从main函数返回或者执行exit函数的时候，均会调用fcloseall函数，该调用链为：

- exit
  - fcloseall
    - _IO_cleanup
      - _IO_flush_all_lockp
        - _IO_OVERFLOW

最后会遍历_IO_list_all存放的每一个**IO_FILE**结构体，如果满足条件的话，会调用每个结构体中vtable->_overflow函数指针指向的函数。

IO_FILE 背景介绍

这是FILE结构体的内容，FILE结构会通过_chain连接形成一个链表，链表头部用全局变量_IO_list_all表示。默认情况下依次链接了stderr,stdout,stdin三个文件流，并将新建的流插入到头部。

vtable这个函数表中有19个函数，分别完成IO相关的功能，由IO函数调用，如fwrite最终会调用__write函数，fread会调用__doallocate来分配IO缓冲区等。

接下来看一个函数的源码：

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

  /* The following pointers correspond to the C++ streambuf protocol. */
  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;
  int _flags2;
  __off_t _old_offset; /* This used to be _offset but it's too small.  */

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

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
  struct _IO_FILE _file;
#endif
  __off64_t _offset;
  /* Wide character stream stuff.  */
  struct _IO_codecvt *_codecvt;
  struct _IO_wide_data *_wide_data;
  struct _IO_FILE *_freeres_list;
  void *_freeres_buf;
  size_t __pad5;
  int _mode;
  /* Make sure we don't get into trouble again.  */
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
};
```

```c
struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};
```

```c
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
};
```

IO_flush_all_lockp

MORESHI POWERPOINT

如果构造条件让程序运行到for循环的if中，就会调用_IO_OVERFLOW，最终调用到vtable中的__overflow。

```
_IO_OVERFLOW (fp, EOF)
#define _IO_OVERFLOW(FP, CH) JUMP1 (__overflow, FP, CH)
#define JUMP1(FUNC, THIS, X1) (_IO_JUMPS_FUNC(THIS)->FUNC) (THIS, X1)
#define _IO_JUMPS_FUNC(THIS) (IO_validate_vtable (_IO_JUMPS_FILE_plus (THIS)))
#define _IO_JUMPS_FILE_plus(THIS) \
  _IO_CAST_FIELD_ACCESS ((THIS), struct _IO_FILE_plus, vtable)
```

什么时候会调用_IO_flush_all_lockp呢？

1.libc执行abort流程时

2.执行exit函数时

3.执行流从main函数返回时

libio > C genops.c > _IO_flush_all_lockp(int)

```
int
_IO_flush_all_lockp (int do_lock)
{
  int result = 0;
  FILE *fp;

#ifdef _IO_MTSAFE_IO
  _IO_cleanup_region_start_noarg (flush_cleanup);
  _IO_lock_lock (list_all_lock);
#endif

  for (fp = (FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
    {
      run_fp = fp;
      if (do_lock)
	_IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
	   || (_IO_vtable_offset (fp) == 0
	       && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
				    > fp->_wide_data->_IO_write_base))
	   )
	  && _IO_OVERFLOW (fp, EOF) == EOF)
	result = EOF;

      if (do_lock)
	_IO_funlockfile (fp);
      run_fp = NULL;
    }

#ifdef _IO_MTSAFE_IO
  _IO_lock_unlock (list_all_lock);
  _IO_cleanup_region_end (0);
#endif

  return result;
}
```

House_of_apple

MORESHI POWERPOINT

使用largebin attack可以劫持_IO_list_all变量，将其替换为伪造的IO_FILE结构体，而在此时，我们其实仍可以继续利用某些IO流函数去修改其他地方的值。要想修改其他地方的值，就离不开_IO_FILE的一个成员_wide_data的利用。

```
struct _IO_FILE_complete
{
  struct _IO_FILE _file;
  __off64_t _offset;
  /* Wide character stream stuff.  */
  struct _IO_codecvt *_codecvt;
  struct _IO_wide_data *_wide_data; // 劫持这个变量
  struct _IO_FILE *_freeres_list;
  void *_freeres_buf;
  size_t __pad5;
  int _mode;
  /* Make sure we don't get into trouble again.  */
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)]
};
```

House_of_apple

MORESHI POWERPOINT

我们在伪造_IO_FILE结构体的时候，伪造_wide_data变量，然后通过某些函数，比如_IO_wstrn_overflow就可以将已知地址空间上的某些值修改为一个已知值。

分析一下这个函数，首先将fp强转为_IO_wstrnfile *指针，然后判断fp->_wide_data->_IO_buf_base != snf->overflow_buf是否成立（一般肯定是成立的），如果成立则会对fp->_wide_data的_IO_write_base、_IO_read_base、_IO_read_ptr和_IO_read_end赋值为snf->overflow_buf或者与该地址一定范围内偏移的值；最后对fp->_wide_data的_IO_write_ptr和_IO_write_end赋值。

也就是说，只要控制了fp->_wide_data，就可以控制从fp->_wide_data开始一定范围内的内存的值，也就等同于**任意地址写已知地址。**

```c
static wint_t
_IO_wstrn_overflow (FILE *fp, wint_t c)
{
  /* When we come to here this means the user supplied buffer is
     filled.  But since we must return the number of characters which
     would have been written in total we must provide a buffer for
     further use.  We can do this by writing on and on in the overflow
     buffer in the _IO_wstrnfile structure.  */
  _IO_wstrnfile *snf = (_IO_wstrnfile *) fp;

  if (fp->_wide_data->_IO_buf_base != snf->overflow_buf)
    {
      _IO_wsetb (fp, snf->overflow_buf,
     snf->overflow_buf + (sizeof (snf->overflow_buf)
          / sizeof (wchar_t)), 0);

      fp->_wide_data->_IO_write_base = snf->overflow_buf;
      fp->_wide_data->_IO_read_base = snf->overflow_buf;
      fp->_wide_data->_IO_read_ptr = snf->overflow_buf;
      fp->_wide_data->_IO_read_end = (snf->overflow_buf
          + (sizeof (snf->overflow_buf)
        / sizeof (wchar_t)));
    }

  fp->_wide_data->_IO_write_ptr = snf->overflow_buf;
  fp->_wide_data->_IO_write_end = snf->overflow_buf;

  /* Since we are not really interested in storing the characters
     which do not fit in the buffer we simply ignore it.  */
  return c;
}
```

House_of_apple构造注意点

MORESHI POWERPOINT

这里有时候需要绕过_IO_wsetb
函数里面的free：

```
void
_IO_wsetb (FILE *f, wchar_t *b, wchar_t *eb, int a)
{
  if (f->_wide_data->_IO_buf_base && !(f->_flags2 & _IO_FLAGS2_USER_WBUF))
    free (f->_wide_data->_IO_buf_base); // 其不为0的时候不要执行到这里
  f->_wide_data->_IO_buf_base = b;
  f->_wide_data->_IO_buf_end = eb;
  if (a)
    f->_flags2 &= ~_IO_FLAGS2_USER_WBUF;
  else
    f->_flags2 |= _IO_FLAGS2_USER_WBUF;
}
```

```
/* Bits for the _flags2 field.  */
#define _IO_FLAGS2_MMAP 1
#define _IO_FLAGS2_NOTCANCEL 2
#define _IO_FLAGS2_USER_WBUF 8
#define _IO_FLAGS2_NOCLOSE 32
#define _IO_FLAGS2_CLOEXEC 64
#define _IO_FLAGS2_NEED_LOCK 128
```

Glibc内存管理概述

MORESHI POWERPOINT

_IO_wstrnfile涉及到的结构体如下：

其中，overflow_buf相对于_IO_FILE结构体的偏移为0xf0，在vtable后面。

```c
typedef struct
{
  _IO_strfile f;
  /* This is used for the characters which do not fit in the buffer
     provided by the user.  */
  wchar_t overflow_buf[64]; // overflow_buf在这里*********
} _IO_wstrnfile;
```

```c
typedef struct _IO_strfile_
{
  struct _IO_streambuf _sbf;
  struct _IO_str_fields _s;
} _IO_strfile;
```

```c
struct _IO_streambuf
{
  FILE _f;
  const struct _IO_jump_t *vtable;
};
```

```c
struct _IO_str_fields
{
  /* These members are preserved for ABI compatibility.  The glibc
     implementation always calls malloc/free for user buffers if
     _IO_USER_BUF or _IO_FLAGS2_USER_WBUF are not set.  */
  _IO_alloc_type _allocate_buffer_unused;
  _IO_free_type _free_buffer_unused;
};
```

IO_FILE 相关结构体

MORESHI POWERPOINT

而struct _IO_wide_data结构体如下

总的来说，假如此时在堆上伪造一个_IO_FILE结构体并已知其地址为A。

将A + 0xd8(vtable的偏移)替换为_IO_wstrn_jumps地址
将A + 0xa0(_wide_data的偏移)设置为B，并设置其他成员以便能调用到_IO_OVERFLOW。

exit函数则会一路调用到_IO_wstrn_overflow函数，并将B至B + 0x38的地址区域的内容都替换为A + 0xf0(snf->overflow_buf的偏移)或者A + 0x1f0（ snf->overflow_buf + (sizeof (snf->overflow_buf) / sizeof (wchar_t))的大小）。

```c
/* Extra data for wide character streams.  */
struct _IO_wide_data
{
  wchar_t *_IO_read_ptr;  /* Current read pointer */
  wchar_t *_IO_read_end;  /* End of get area. */
  wchar_t *_IO_read_base; /* Start of putback+get area. */
  wchar_t *_IO_write_base;  /* Start of put area. */
  wchar_t *_IO_write_ptr; /* Current put pointer. */
  wchar_t *_IO_write_end; /* End of put area. */
  wchar_t *_IO_buf_base;  /* Start of reserve area. */
  wchar_t *_IO_buf_end;   /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  wchar_t *_IO_save_base; /* Pointer to start of non-current get area. */
  wchar_t *_IO_backup_base; /* Pointer to first valid character of
                backup area */
  wchar_t *_IO_save_end;  /* Pointer to end of non-current get area. */

  __mbstate_t _IO_state;
  __mbstate_t _IO_last_state;
  struct _IO_codecvt _codecvt;

  wchar_t _shortbuf[1];

  const struct _IO_jump_t *_wide_vtable;
};
```

ptmalloc源代码分析

MORESHI POWERPOINT

简单写一个demo程序进行验证：

从输出中可以看到，已经成功修改了sdterr->_wide_data所指向的地址空间的内存。

由上可以，在只给了1次largebin attack的前提下，能利用_IO_wstrn_overflow函数将任意地址空间上的值修改为一个已知地址，并且这个已知地址通常为堆地址。那么，当我们伪造两个甚至多个_IO_FILE结构体，并将这些结构体通过chain字段串联起来就能进行组合利用。基于此，house of apple至少有以下几种利用思路。

```
hanwang@hanwang:~/tools/pwn-exercise-mine/house_of_apple$ ./demo

[*] allocate a 0x100 chunk
===========================old value=======================
[0x560e74b7c2a0]: 0x1122334455667788  0x1122334455667788
[0x560e74b7c2b0]: 0x1122334455667788  0x1122334455667788
[0x560e74b7c2c0]: 0x1122334455667788  0x1122334455667788
[0x560e74b7c2d0]: 0x1122334455667788  0x1122334455667788
===========================old value=======================
[*] puts address: 0x7f25ba713420
[*] stderr address: 0x7f25ba87c5c0
[*] stderr->_IO_write_ptr address: 0x7f25ba87c5e8
[*] stderr->_flags2 address: 0x7f25ba87c634
[*] stderr->_wide_data address: 0x7f25ba87c660
[*] stderr->vtable address: 0x7f25ba87c698
[*] _IO_wstrn_jumps address: 0x7f25ba877c60
[+] step 1: change stderr->_IO_write_ptr to -1
[+] step 2: change stderr->_flags2 to 8
[+] step 3: replace stderr->_wide_data with the allocated chunk
[+] step 4: replace stderr->vtable with _IO_wstrn_jumps
[+] step 5: call fcloseall and trigger house of apple
===========================new value=======================
[0x560e74b7c2a0]: 0x00007f25ba87c6b0  0x00007f25ba87c7b0
[0x560e74b7c2b0]: 0x00007f25ba87c6b0  0x00007f25ba87c6b0
[0x560e74b7c2c0]: 0x00007f25ba87c6b0  0x00007f25ba87c6b0
[0x560e74b7c2d0]: 0x00007f25ba87c6b0  0x00007f25ba87c7b0
===========================new value=======================
```

利用思路

MORESHI POWERPOINT

思路一：修改tcache线程变量

该思路需要借助house of pig的思想，利用**_IO_str_overflow**中的**malloc**进行任意地址分配，memcpy进行任意地址覆盖，利用步骤如下：

伪造至少两个_IO_FILE结构体

第一个_IO_FILE结构体执行_IO_OVERFLOW的时候，利用_IO_wstrn_overflow函数修改tcache全局变量为已知值，也就控制了tcache bin的分配

第二个_IO_FILE结构体执行_IO_OVERFLOW的时候，利用_IO_str_overflow中的malloc函数任意地址分配，并使用memcpy使得能够**任意地址写任意值**

利用两次任意地址写任意值修改pointer_guard和IO_accept_foreign_vtables的值绕过_IO_vtable_check函数的检测，利用一个_IO_FILE，随意伪造vtable劫持程序控制流即可。（或者利用一次任意地址写任意值修改libc.got里面的函数地址，如果got表可写的话）

因为可以已经任意地址写任意值了，所以这可以控制的变量和结构体非常多，也非常地灵活，需要结合具体的题目进行利用。

利用思路

MORESHI POWERPOINT

思路二：修改mp_结构体

该思路与上述思路差不多，不过对tcachebin分配的劫持是通过修改mp_.tcache_bins这个变量。打这个结构体的好处是在攻击远程时不需要爆破地址，因为线程全局变量tls结构体的地址本地和远程并不一定是一样的，有时需要爆破。

利用步骤如下：

伪造至少两个_IO_FILE结构体
第一个_IO_FILE结构体执行_IO_OVERFLOW的时候，利用_IO_wstrn_overflow函数修改mp_.tcache_bins为很大的值，使得很大的chunk也通过tcachebin去管理
接下来的过程与上面的思路是一样的

利用思路

MORESHI POWERPOINT

思路三：修改pointer_guard线程变量之house of emma

该思路其实就是house of apple + house of emma。

利用步骤如下：

伪造两个_IO_FILE结构体
第一个_IO_FILE结构体执行_IO_OVERFLOW的时候，利用_IO_wstrn_overflow函数修改tls结构体pointer_guard的值为已知值
第二个_IO_FILE结构体用来做house of emma利用即可控制程序执行流

题目示例

MORESHI POWERPOINT

https://bbs.pediy.com/thread-273418.htm

其他利用思路

MORESHI POWERPOINT

house of pig

house of kiwi

house of emma

house of banana

house of apple2

house of apple3

北京大學
PEKING UNIVERSITY

Thank You！

2022.9.16 王晗