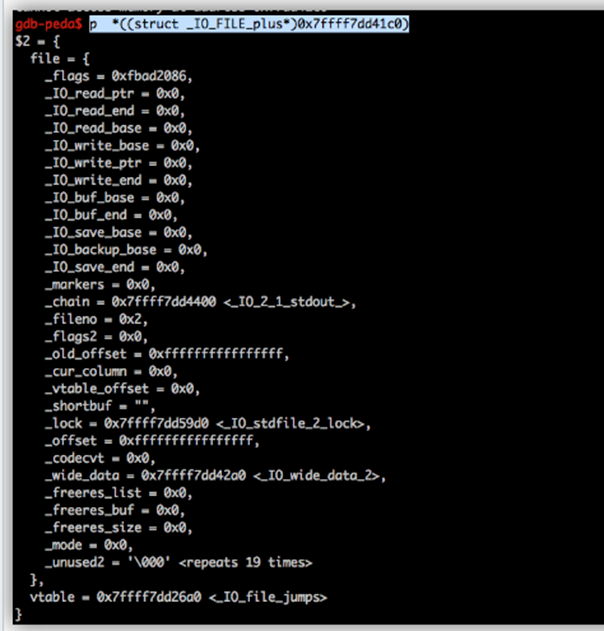
Io\_file\_all攻击

攻击进程时，堆上一个有趣的目标是拥有stream 函数(fopen()，fread()，fclose()，etc)的file结构体。File结构体指向stream使用的各种内存缓冲区。有趣的是这并非是全部structure。当fopen()分配一个新的FILE结构体并返回它的指针，实际上分配了一个内部结构体struct \_IO\_FILE\_plus，它包含struct \_IO\_FILE和一个指向struct \_IO\_jump\_t的指针，该指针包含一系列指向有关该file操作的函数的指针。这是它的虚表，就像C++的虚表，每当FILE的stream函数被调用时，他就会被使用。

IO\_FILE\_ALL是一个指向IO\_FILE\_plus结构的指针，IO\_FILE\_plus结构如下图所示:



IO\_file\_jumps包含各种函数地址，IO\_file\_jumps结构如下:



IO\_file\_jumps是IO\_FILE\_ALL指针值偏0xd8处的指针值，IO\_file\_jumps指针值索引3(第四个值)是overflow函数的地址。

总结:可通过溢出覆盖IO\_FILE\_ALL指针值为addr\_edit，从而触发错误时，就会执行addr\_edit+0xd8地址上地址+0x18处的函数(该处是函数的地址)了。

《1》总体流程

1》IO\_FILE\_ALL指针 ，指向IO\_FILE\_plus结构

//[ IO\_FILE\_ALL指针]是\_IO\_FILE\_plus结构的地址，即IO\_list\_all指针指向一个\_IO\_FILE\_plus结构；

//可调用该结构里的虚表函数，函数参数在IO\_FILE\_plus结构中；

//将 IO\_list\_all指针覆盖为可控制的地址，并在该地址构造伪\_IO\_FILE\_plus结构；

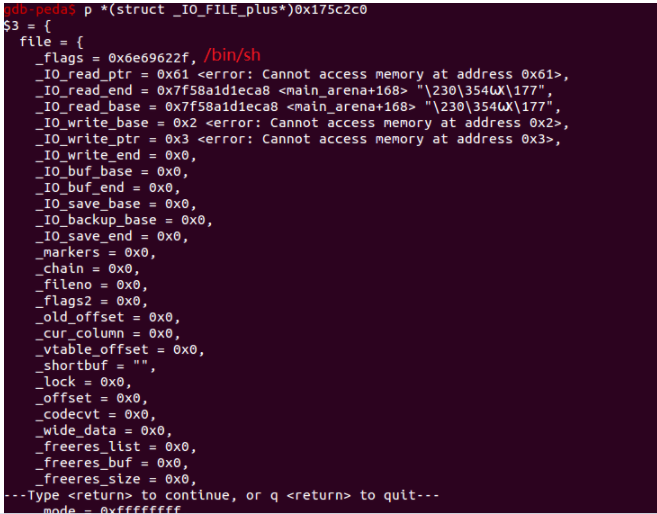
//伪\_IO\_FILE\_plus结构已edit需要的参数，且一般vtable指针(+0xd8)处覆盖为紧邻vtable指针地址处，便于直接在其后覆盖为伪造的IO\_file\_jumps结构(函数s地址，如system函数);

3》vtable指针，指向IO\_file\_jumps结构

// vtable指针=[ IO\_FILE\_ALL指针]+0xd8，即\_IO\_FILE\_plus结构+0xd8；

//[vtable指针]是IO\_file\_jumps结构的地址；

如下图，虚表偏移3的函数的参数位于io\_file\_plus结构偏移0的位置处:



《2》vtable函数使用条件

关于\_IO\_FILE结构体的构造我们结合源代码看:

while (fp != NULL)

{

…

fp = fp->\_chain;

...

if (((fp->\_mode <= 0 && fp->\_IO\_write\_ptr > fp->\_IO\_write\_base)

#if defined \_LIBC || defined \_GLIBCPP\_USE\_WCHAR\_T

|| (\_IO\_vtable\_offset (fp) == 0

&& fp->\_mode > 0 && (fp->\_wide\_data->\_IO\_write\_ptr

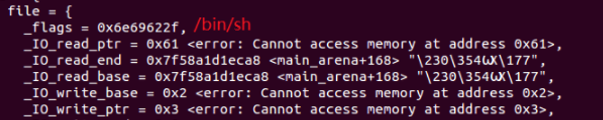
> fp->\_wide\_data->\_IO\_write\_base))

#endif

)

&& \_IO\_OVERFLOW (fp, EOF) == EOF)

所以：





<https://blog.csdn.net/weixin_40850881/article/details/80043934>

https://bbs.pediy.com/thread-222718.htm