

接上文，目前已知我们可以拿到的是任意地址写指针原语，想要弄出 dataonly 方式提权，那么用什么 dataonly 的方式，又怎么实现呢？笔者最先想到的是 dirtycred，但很遗憾的是内核的 file 和 cred 这两个 struct 的分配走的是 192，而写的指针是 256 对齐；后又想到一个惯用伎俩，劫持 pipe_buffer 的 page 指针实现 uaf 写物理页

那么首先最大的问题无疑就是如何去 leak 了，想要实现该利用方式无疑必须要 leak 出 page 指针，但是在上一个 exploit 中用的利用手法（越界读）是不可行的，因为 user_key_payload 走的是 kmalloc，而 pipe_buffer 走的是 kmalloc-cg，两者在 slab 层面有内存隔离，除非打 cross_cache attack 构造出相邻内存页或者找到一套同缓存泄露的结构体，否则只能看渺茫的运气。但是笔者灵光一现想到了一种方式，回想一下为什么 msg_msg 的任意读不可用呢？其实就是我们不能精确控制 m_ts 成员，但是有这么一种情况，我们分配的时候直接分配 4096 的 msg_msg 加上 512 的 msg_msgseg，然后劫持 msg_msgseg 指针为 pipe_buffer 不就好了？

事实如我所料，但是在其过程中遇到的一些问题和解决的方法给师傅们分享一下，首先第一个问题就是泄露完指针的内存并不能被 msg_msg 占用（因为想劫持 msg_msg 就必须把它分配到我们已知的地址上），一开始我以为是释放的内存太多堆喷不够导致的，结果我消耗部分内存依旧分配不过去，一开始笔者百思不得其解，后面想了又想，有了一个大致的猜想，那就是 msg_msg 自身不想分配释放的内存，为什么呢？其实就因为 4096 的 msg_msg 一个 obj 就占用了一张 4k 页，整个 slab 有 8 个 objs，虽然内核分配内存是随机的但是 slab 得是一块连续的内存，那么也就是说 msg_msg 的 slab 需要分配连续的八张 4k 页，而 uaf 对象只是个 1k 大小，对应的 slab 只需要四张 4k 页，地址连续要求不一样自然可能分配不出来（即 uaf 对象释放之后目标内存附近只有 4-7 张连续的 4k 页，不满足 msg_msg 的 slab），那么怎么解决呢？

其实解决方式很朴实无华，直接堆喷大量 4k 大小的 msg_msg，然后释放让 uaf 对象占用，再将 uaf 对象释放再堆喷 4k 大小的 msg_msg 就好了。但是又碰到了一个类似但不同的问题，释放的对象是 1k 大小，但是劫持目标是 512 大小的 pipe_buffer，这两者同样是地址连续要求不同的 slab，一个是四张 4k 页另一个是两张 4k 页，slab 向 buddy system 申请内存一般是按照阶来算的，不同阶的内存很难分配到一起，针对该问题笔者最终找到了一种解决方案，堆喷大量 512 大小的 skb 消耗低阶页，迫使 buddy system 把高阶页分配出来，最终也证实可以。问题都解决了但是最后还是没能任意读反而 kernel 崩溃了，经过分析发现 msg_msg 的任意读也有解引用要求（读取目标内存头八个字节必须是有效内核堆指针或者 null），由于初始原语就是能写两条指针的，笔者最终采用偏移读的方式绕过这一限制（能写两条指针，一条+0x200，一条+0x198，选用+0x198 这条覆盖 msg_msgseg 指针）

能实现内存读了就要考虑内存写了，但是在此之前还需要排除一个隐患，前面笔者在

第一个 exploit 中采用的是一次读多次写的方式避免任意写指针不稳定的问题（后面分配的对象地址小于存在 uaf 的对象的地址才能任意写指针，由于 kaslr 是开机的时候初始化的所以只需要泄露一次），但是这里明显不行，page 指针跟内核堆指针是每次都不同的，所以需要很稳定地实现任意写指针原语才能更好地展开漏洞利用工作，那么该怎么做呢？

笔者一番思索找到了一种解决方案，虽然分配的内存是随机的，但是地址增长其实是由低到高的，那么不难想到，假如我们提前分配一段低地址内存，然后分配 uaf 对象，再释放这段低地址内存让后面分配的对象占用，就能比较稳定地实现任意写指针原语（注：这里说的还是比较稳定而不是 100%，并且需要提前清洁内存避免有部分释放的高地址内存被后面分配的对象占用）。解决了该问题就是考虑怎么实现内存写了，这里笔者最先采用的是跟之前写的 exploit 同一种手法，通过 msg_msg 任意释放原语，将一个 pipe_buffer 非法释放然后分配 skb 占用完成对 pipe_buffer 的劫持，但是在进行可行性验证（使用 gdb 修改目标内存看是否能实现想要的结果）的时候没有通过，最后分析发现想要劫持 pipe_buffer 实现 uaf 写物理页，pipe_inode_info 的 head 成员不能为 0 否则其会自己分配物理页覆盖我们伪造的。但是 msg_msg 的任意释放原语有一个致命约束，目标内存头八个字节得是 null，但理想的情况必须要我们在 pipe_buffer 分配物理页之后再劫持，不难想到我们必须找到一个没有条件任意释放的结构体或者寻找其它方式，这里笔者最终决定采用劫持 pipe_inode_info 的 pipe_buffer 指针这种方式实现，但是怎么实现呢？

在实践的过程中碰到一个最棘手的问题就是，怎么确保 pipe_inode_info 占用 uaf 对象？（笔者一开始就想采用这种方式，但最终因为没有解决这个问题从而考虑采用任意释放原语），slab 地址连续要求不一致在这甚至只是一个小问题，最大的问题是分配管道除了有 pipe_inode_info 和 pipe_buffer，还有大量噪声结构体以及系统可能恰好分配的，而 uaf 对象和要劫持的 pipe_inode_info 不在同一个缓存池则必须把 uaf 对象释放回 buddy system，那么 uaf 对象所在的内存极有可能被其它 slab 分配到（lsm_file_cache、filp、vm_area……），那么怎么解决该问题呢？

笔者最终想到一种比较可行的解决方案，利用在漏洞利用中经常阻碍我们的机制：内存隔离，不难想到，我们先把 uaf 对象内存分配给跟劫持目标对象同缓存的 slab，等分配 pipe_inode_info 时释放给其占用就好了，这里需要选用相对干净的对象，笔者直接使用 msg_msgseg（难点转变为怎么较为精确地控制 uaf 对象分配给 msg_msgseg）

后续技术细节比较简单笔者这里就不赘述了，简单总结一下该 exploit 用到的原语：

1. 利用任意写指针原语结合 msg_msg，事先解决 m_ts，并解决内存分配问题，转为任意读原语

2. 利用任意写指针原语结合 pipe_inode_info, 事先解决 head, 并解决内存分配问题, 转为物理页 uaf 写原语

除此之外还有一个零碎问题, 我们最终的目的是篡改 filp, 那么怎么确保 filp 落入 page 指针指向的内存, 这里其实比较好解决, 在释放 page 指针指向的物理页之后堆喷一块内存占用它, 然后在堆喷 filp 之前释放它就好了

笔者在 exploit 中主要进行了三次插桩, 可以在断点在 single_open 依次调试, 首先在 check 任意读是否正确

```
pwndbg> x/10gx 0xffff888109546000
0xffff888109546000: 0xffff8881070acac0 0xffff8881070acac0
0xffff888109546010: 0x00000000000001f4 0x000000000000011c8
0xffff888109546020: 0xffff888107080d98 0x0000000000000000
0xffff888109546030: 0xffff888107080e00 0x0000000000000000
0xffff888109546040: 0x0000000000000000 0x0000000000000000
pwndbg> x/10gx 0xffff888107080d98
0xffff888107080d98: 0x0000000000000000 0x0000000000000000
0xffff888107080da8: 0x0000000000000000 0x0000000000000000
0xffff888107080db8: 0x0000000000000000 0x0000000000000000
0xffff888107080dc8: 0x0000000000000000 0x0000000000000000
0xffff888107080dd8: 0x0000000000000000 0x0000000000000000
pwndbg>
0xffff888107080de8: 0x0000000000000000 0x0000000000000000
0xffff888107080df8: 0x0000000000000000 0xfffffea00041d8600
0xffff888107080e08: 0x00000000400000000 0xfffffffff8264be80
0xffff888107080e18: 0x0000000000000010 0x0000000000000000
0xffff888107080e28: 0x0000000000000000 0x0000000000000000
```

其次是 uaf 对象的内存是否被 kmalloc-cg-192 这个 slab 占用

```
pwndbg> slab contains 0xffff888107b90000
0xffff888107b90000 @ kmalloc-cg-192
Did not finding containing slab.
```

最后是是否成功实现第二次任意写指针, 指针指向的内存是否是伪造 pipe_buffer 的 msg_msg, 伪造的 pipe_buffer 的 page 指针对应的内存是否在 filp 这个 slab, 三次调试下来哪怕最终 kernel panic 了也能成功篡改/etc/passwd 了

```
pwndbg> x/10gx 0xffff888107b90000
0xffff888107b90000: 0x0000000000000000 0x0000000000000000
0xffff888107b90010: 0xffff888107b90010 0xffff888107b90010
0xffff888107b90020: 0x0000000000000000 0xffff888107b90028
0xffff888107b90030: 0xffff888107b90028 0x0000000000000000
0xffff888107b90040: 0xffff888107b90040 0xffff888107b90040
pwndbg>
0xffff888107b90050: 0x0000000000000001 0x0000000020000002
0xffff888107b90060: 0x0000000010000002 0x0000000020000001
0xffff888107b90070: 0x0000000010000001 0x0000000000000000
0xffff888107b90080: 0x0000000000000000 0x0000000000000000
0xffff888107b90090: 0x0000000000000000 0xffff888107859600
pwndbg>
0xffff888107b900a0: 0xffff88810555c240 0x0000000000000000
0xffff888107b900b0: 0xffff888107859598 0x0000000000000000
0xffff888107b900c0: 0x0000000000000000 0x0000000000000000
0xffff888107b900d0: 0xffff888107b900d0 0xffff888107b900d0
0xffff888107b900e0: 0x0000000000000000 0xffff888107b900e8
pwndbg> x/10gx 0xffff888107859600
0xffff888107859600: 0xfffffea00041d8600 0x0000001000000000
0xffff888107859610: 0xfffffffff8264be80 0x0000000000000010
0xffff888107859620: 0x0000000000000000 0x0000000000000000
0xffff888107859630: 0x0000000000000000 0x0000000000000000
0xffff888107859640: 0x0000000000000000 0x0000000000000000
pwndbg> pageinfo 0xfffffea00041d8600
Virtual address: 0xffff888107618000
page @ 0xfffffea00041d8600 [slab, refcount: 1]
pwndbg> slab contains 0xffff888107618000
0xffff888107618000 @ filp
Did not finding containing slab.
```

最终可以看到成功写入 hacker 用户到/etc/passwd 中实现权限提升

```
root@syzkaller:~# dmesg -n 1
root@syzkaller:~# su test
test@syzkaller:/root$ /exp
[.]leak_ptr failed!
test@syzkaller:/root$ /exp
idx 4: uaf_ptr = 0xffff888109546000
done
done
[+]Found page_ptr: 0xffffea00041d8600 at idx_0x1f3
idx 14: uaf_ptr = 0xffff888107b90000
done1
done1
done2
done2
hacker done
test@syzkaller:/root$ head /etc/shadow
head: cannot open '/etc/shadow' for reading: Permission denied
test@syzkaller:/root$ su hacker
hacker@syzkaller:~# id
uid=0(hacker) gid=0(root) groups=0(root)
hacker@syzkaller:~# uname -a
Linux syzkaller 6.12.56 #7 SMP PREEMPT_DYNAMIC Mon Jan 26 22:43:00 CST 2026 x86_64 GNU/Linux
hacker@syzkaller:~# head /etc/shadow
root:*:20453:0:99999:7:::
daemon:*:20453:0:99999:7:::
bin:*:20453:0:99999:7:::
sys:*:20453:0:99999:7:::
sync:*:20453:0:99999:7:::
games:*:20453:0:99999:7:::
man:*:20453:0:99999:7:::
```

不难看出在这期间其实最难解决的是内存分配、内存布局的问题，希望笔者遇到的问题以及对应的解决方法能给师傅们提供参考。这里笔者本来是想弄出一个比较稳定的 exploit，但是年关将至没什么时间，后续内存布局的配平以提高 exploit 成功率稳定性的工作感兴趣的师傅可以自行研究