Devils Are in the File Descriptors: It Is Time To Catch Them All

Le Wu from Baidu Security

About me

Le Wu(@NVamous)

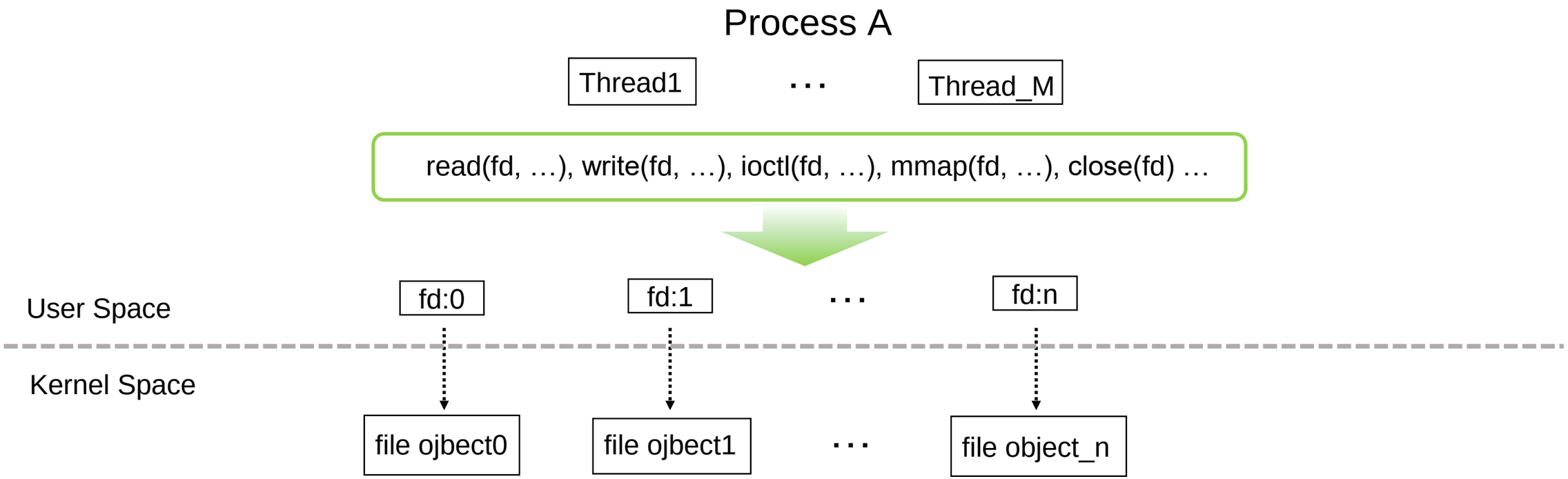
- Focus on Android/Linux bug hunting and exploit
- Found 200+ vulnerabilities in the last two years
- Blackhat Asia 2022 speaker

Outline

- Background
- Diving into issues in the fd **export** operations
- Diving into issues in the fd **import** operations
- Conclusion & Future work

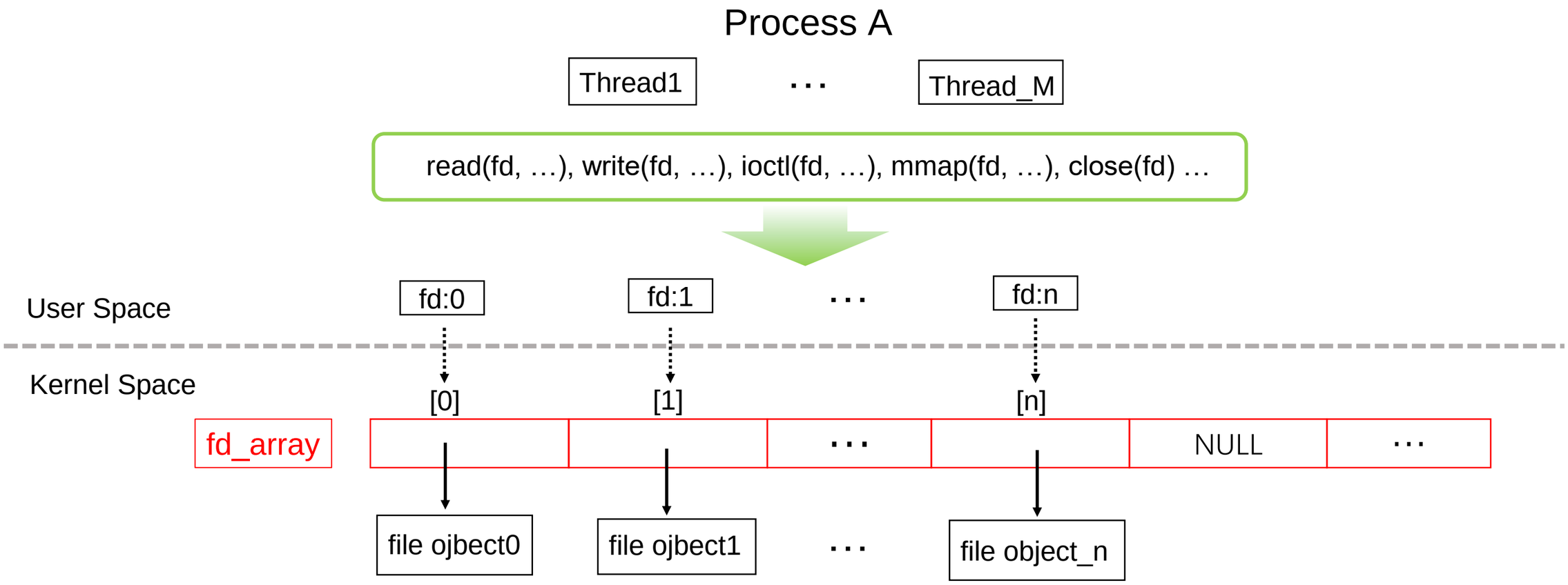
Background

Introduction to file descriptor—— An integer in a process



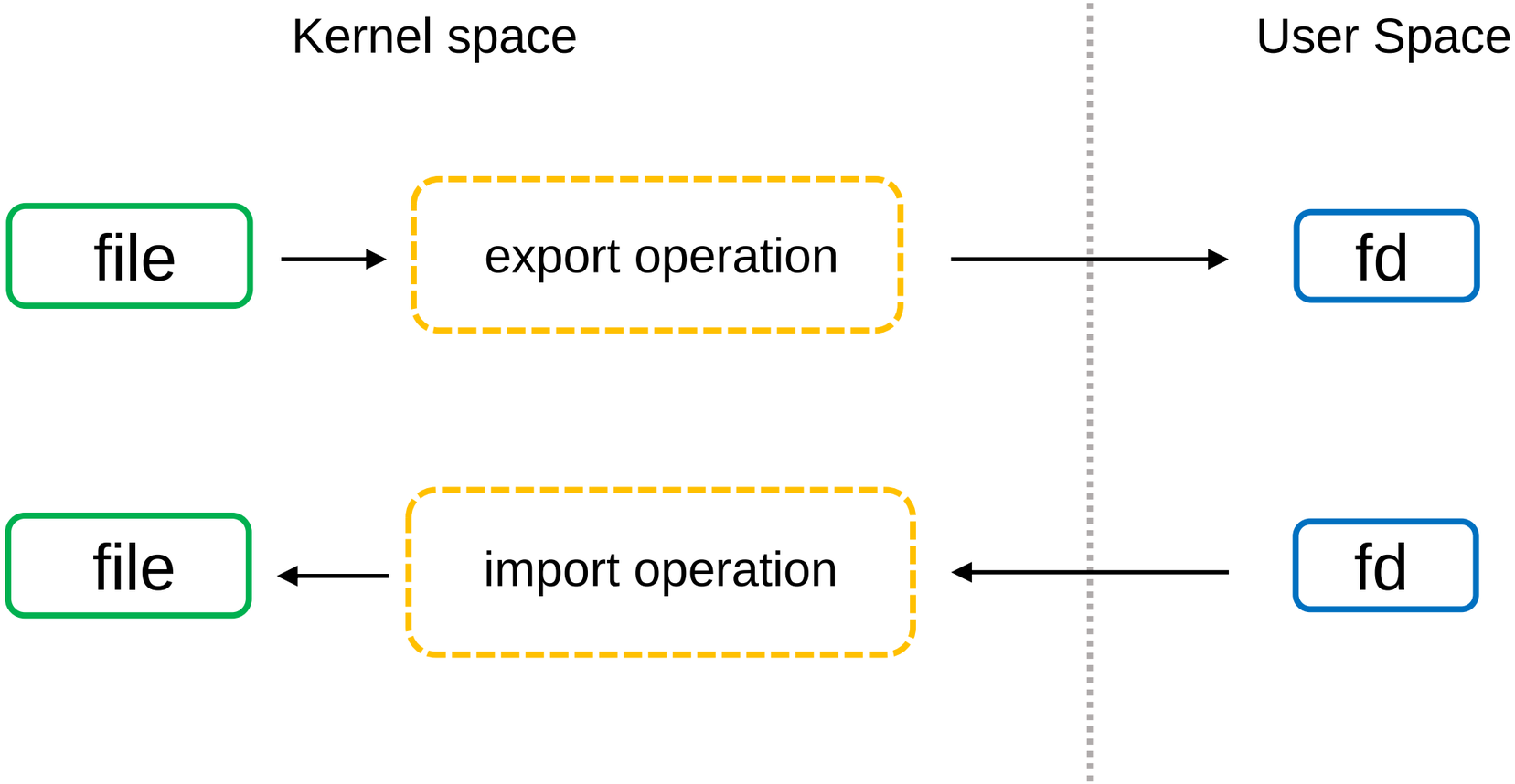
Background

Introduction to file descriptor—— An integer in a process



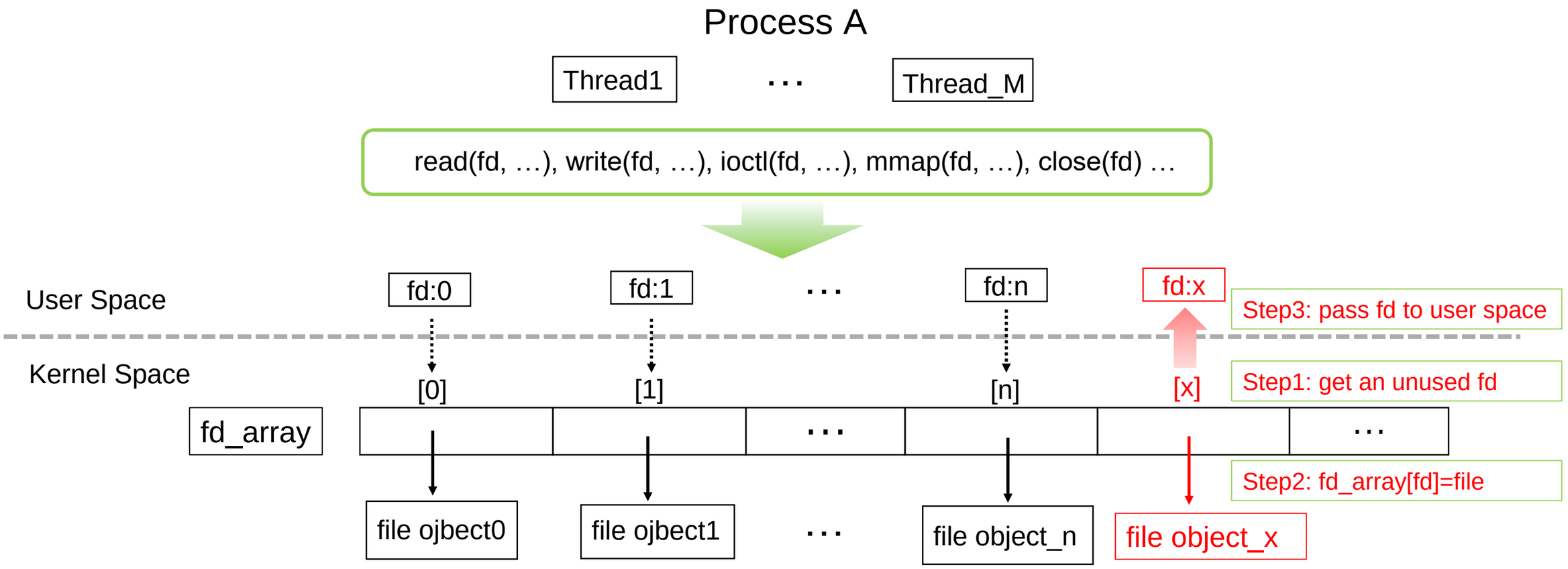
Background

Introduction to file descriptor——fd export operation and import operation in kernel



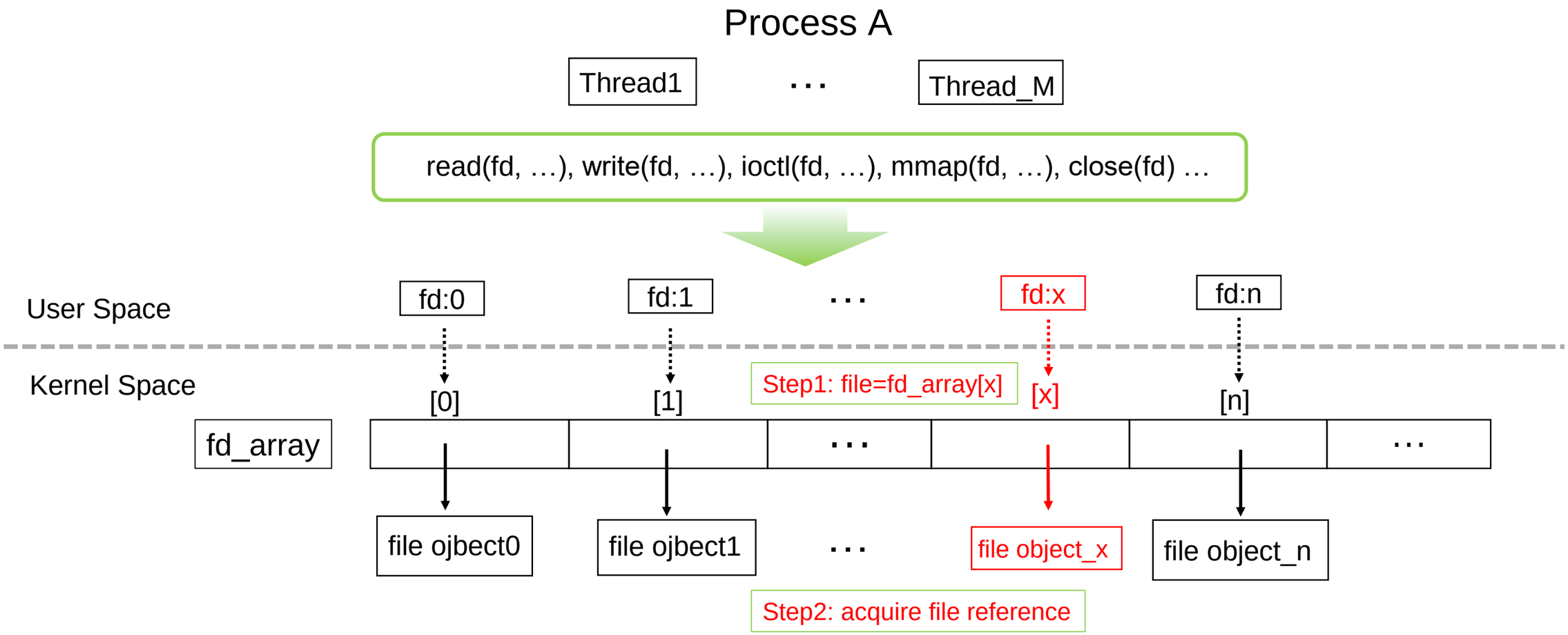
Background

Introduction to file descriptor—— fd export operation in kernel



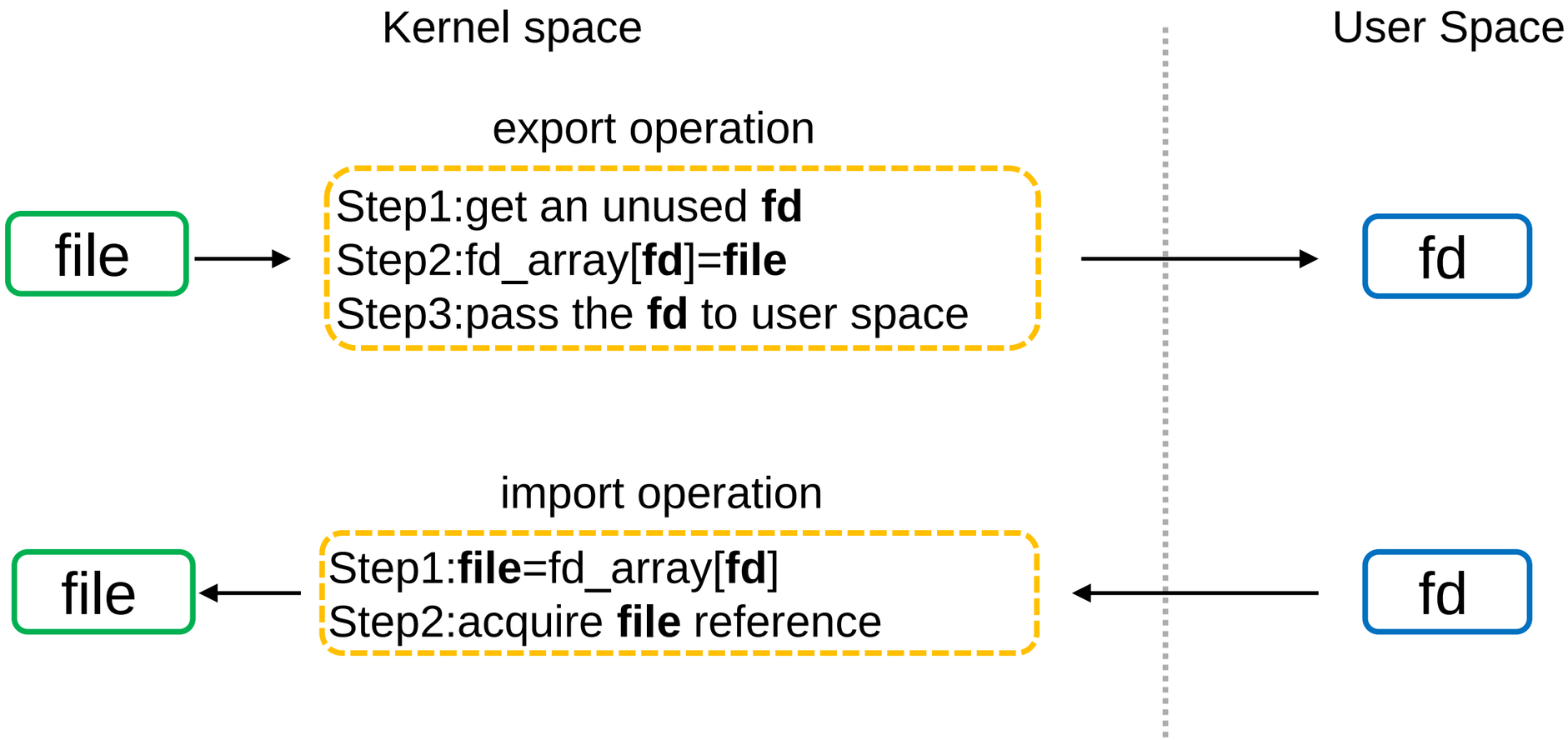
Background

Introduction to file descriptor—— fd import operation in kernel



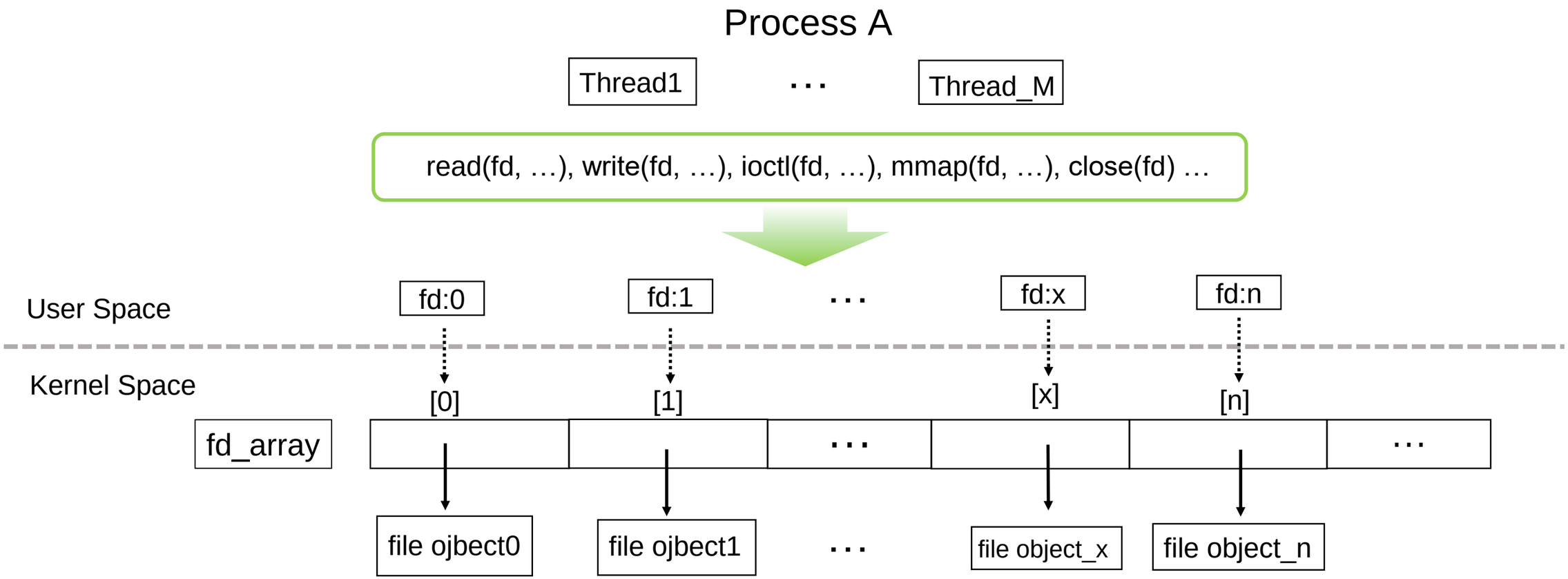
Background

Introduction to file descriptor——fd export operation and fd import operation



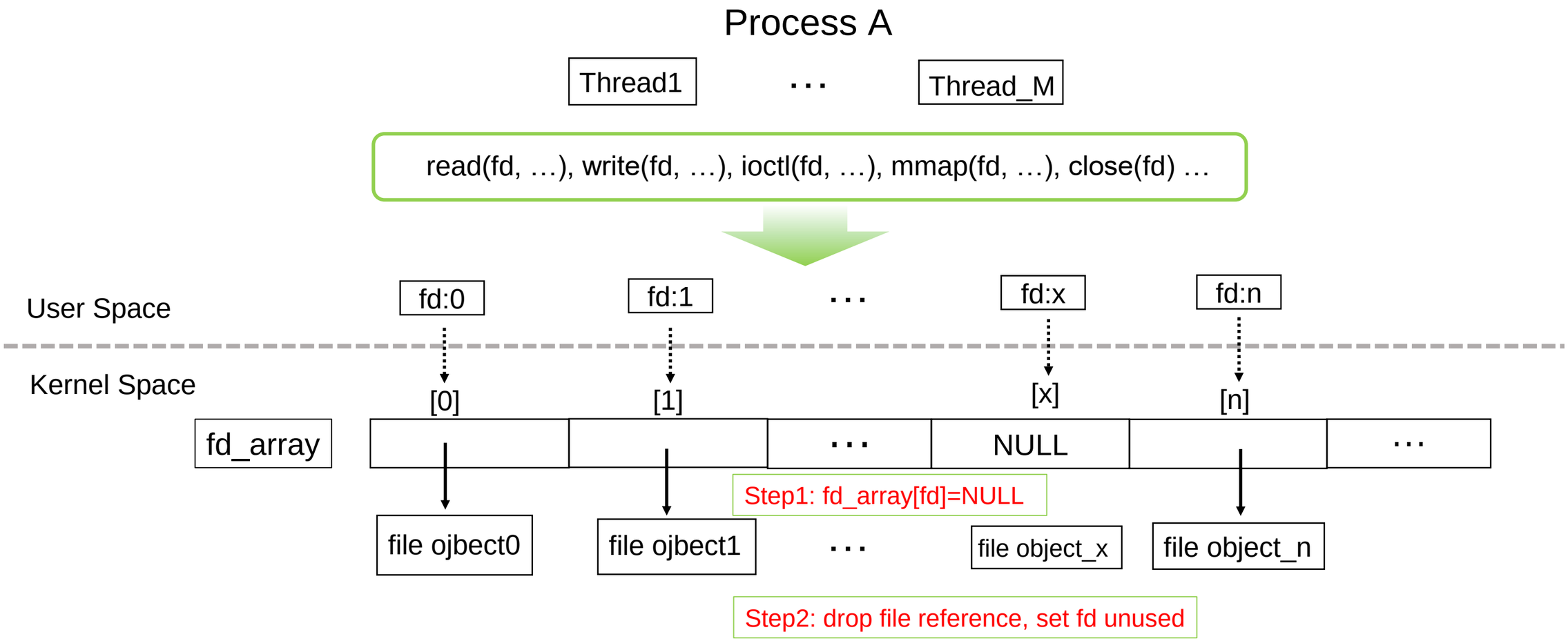
Background

Introduction to file descriptor—— User process close(fd)



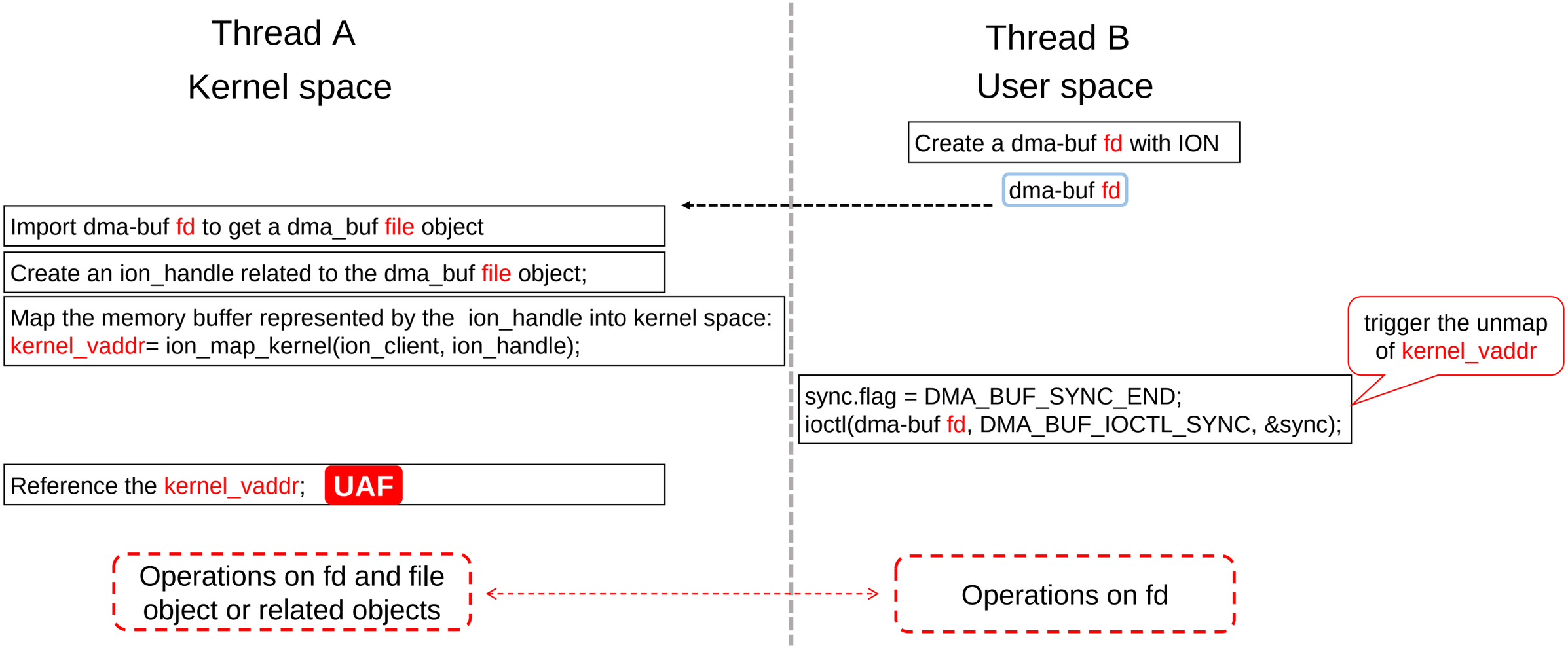
Background

Introduction to file descriptor—— User process close(fd)



Background

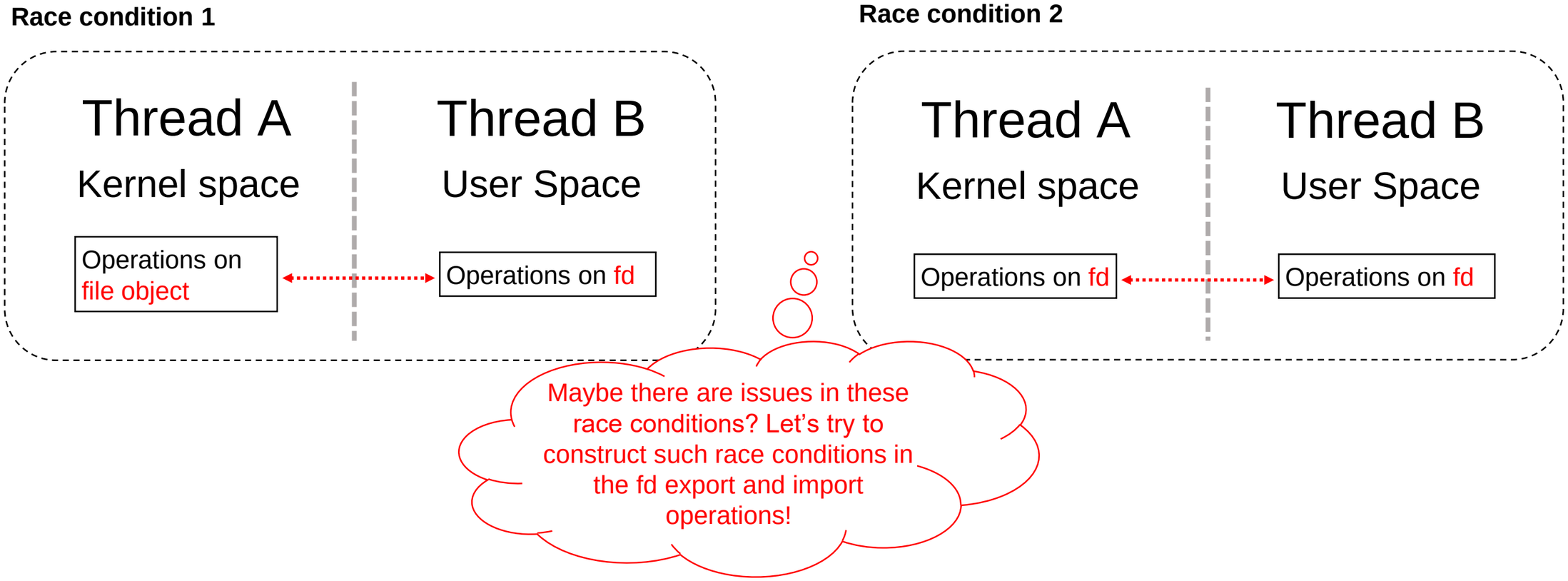
Why file descriptor——Inspired by CVE-2021-0929



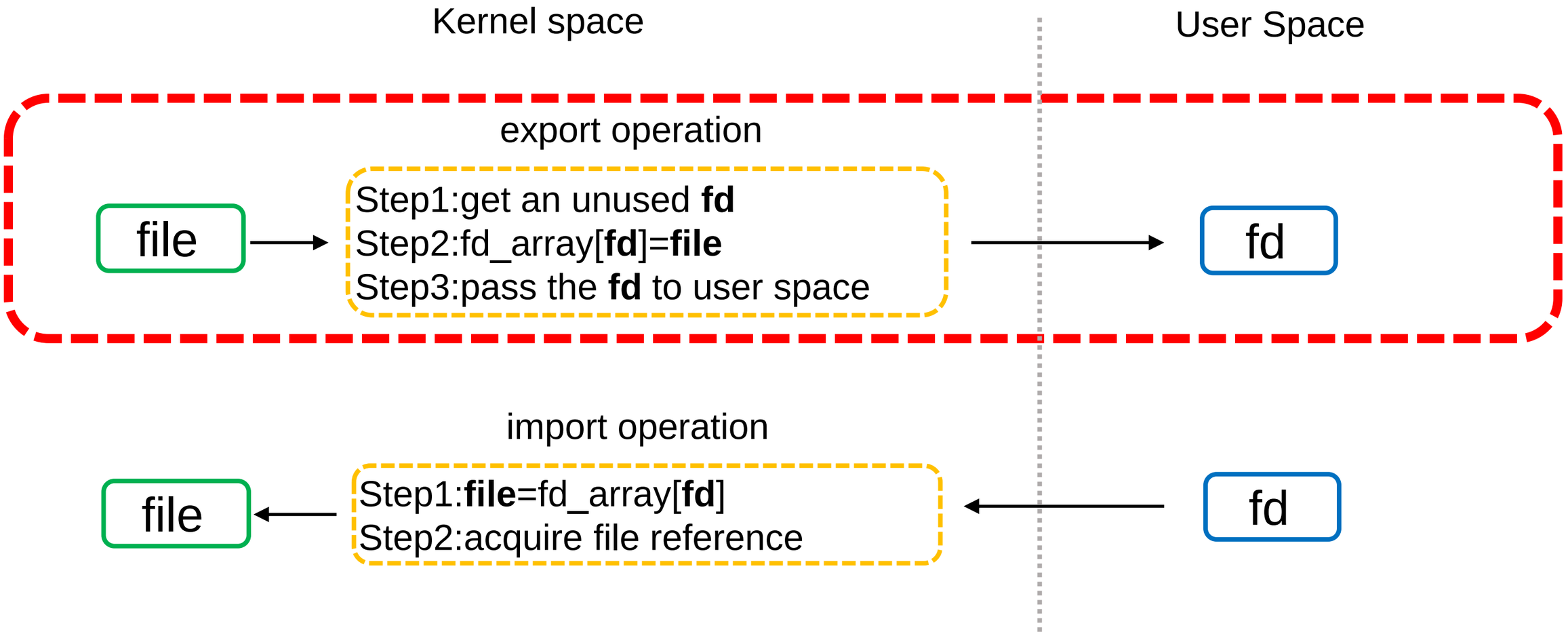
Background

Why file descriptor——Inspired by CVE-2021-0929

A file descriptor can be **shared** between kernel space and user space, race condition can happen between kernel and user operations：



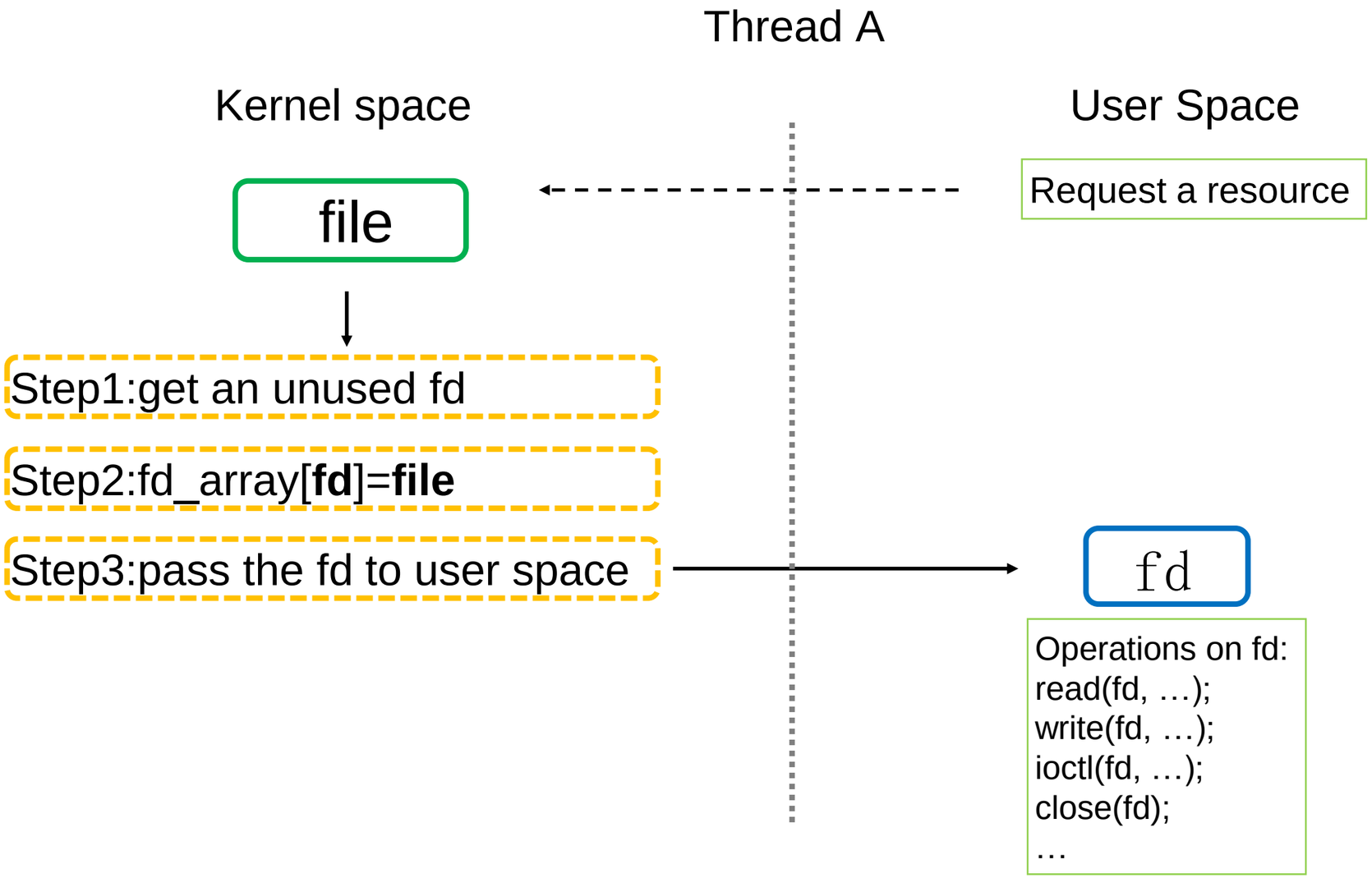
Diving into issues in the fd export operation



Diving into issues in the fd export operation

- Scenario of fd export operation
- UAF caused by race condition
- Find the issues
- Fixes

Scenario of fd export operation



Scenario of fd export operation

Example:

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        …
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

…

```
static long do_sys_openat2(int dfd, const char __user *filename,
                           struct open_how *how)
{
        …
        fd = get_unused_fd_flags(how->flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                …
                fd_install(fd, f);
                …
        }
        …
        return fd;
}
```

Step1:get an unused fd
get_unused_fd_flags()

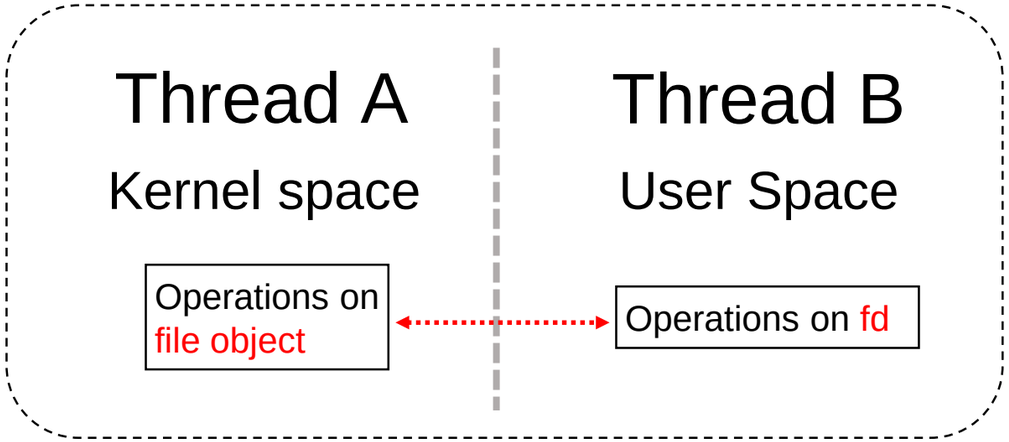
Step2.fd_array[**fd**]=**file:**
fd_install(fd, file)

Step3.pass the fd to user space:
fd as return value

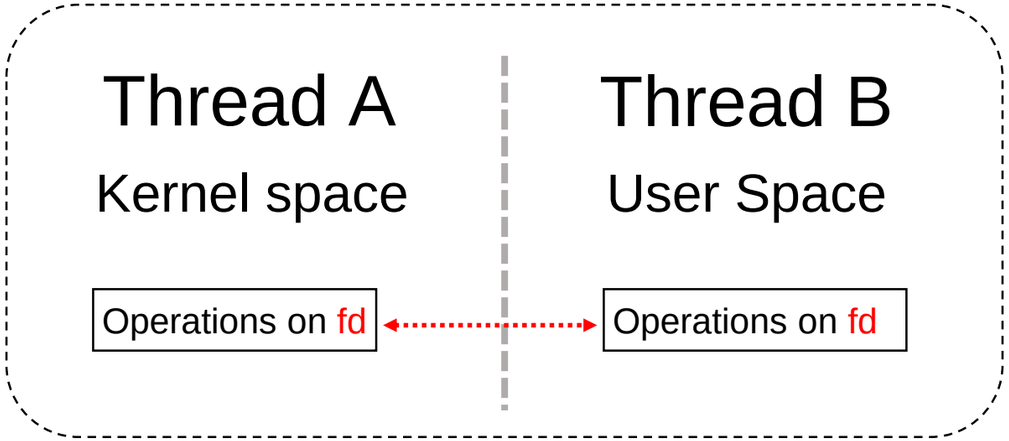
Scenario of fd export operation

But this regular fd export operation is executed sequentially, which is still far from the race conditions we want to see:

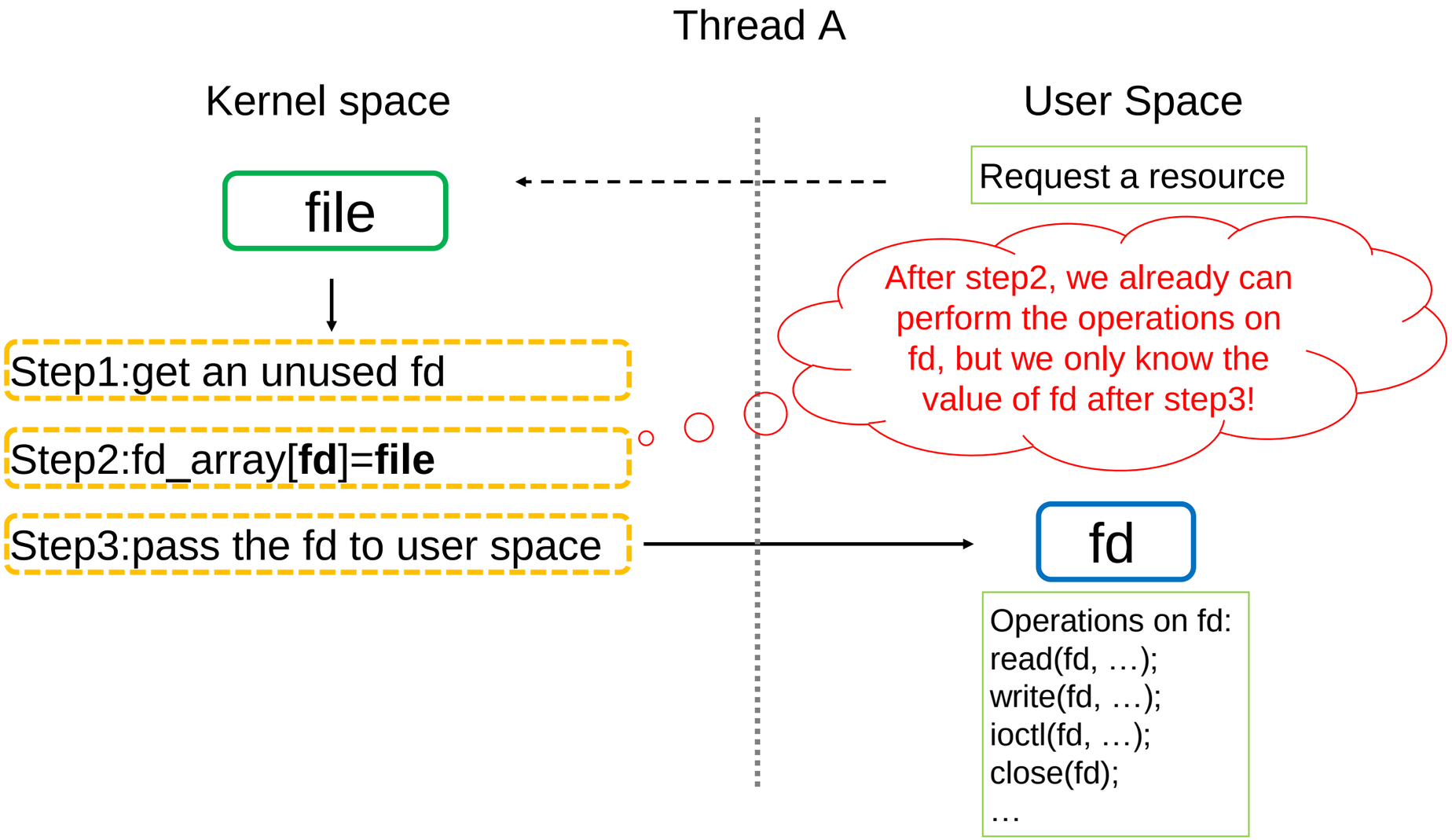
Race condition 1



Race condition 2



UAF caused by race condition



UAF caused by race condition

Hold on! Do we have to wait for fd to be passed from kernel to know the value of it ?

Fd is predictable:

- Assigned in ascending order

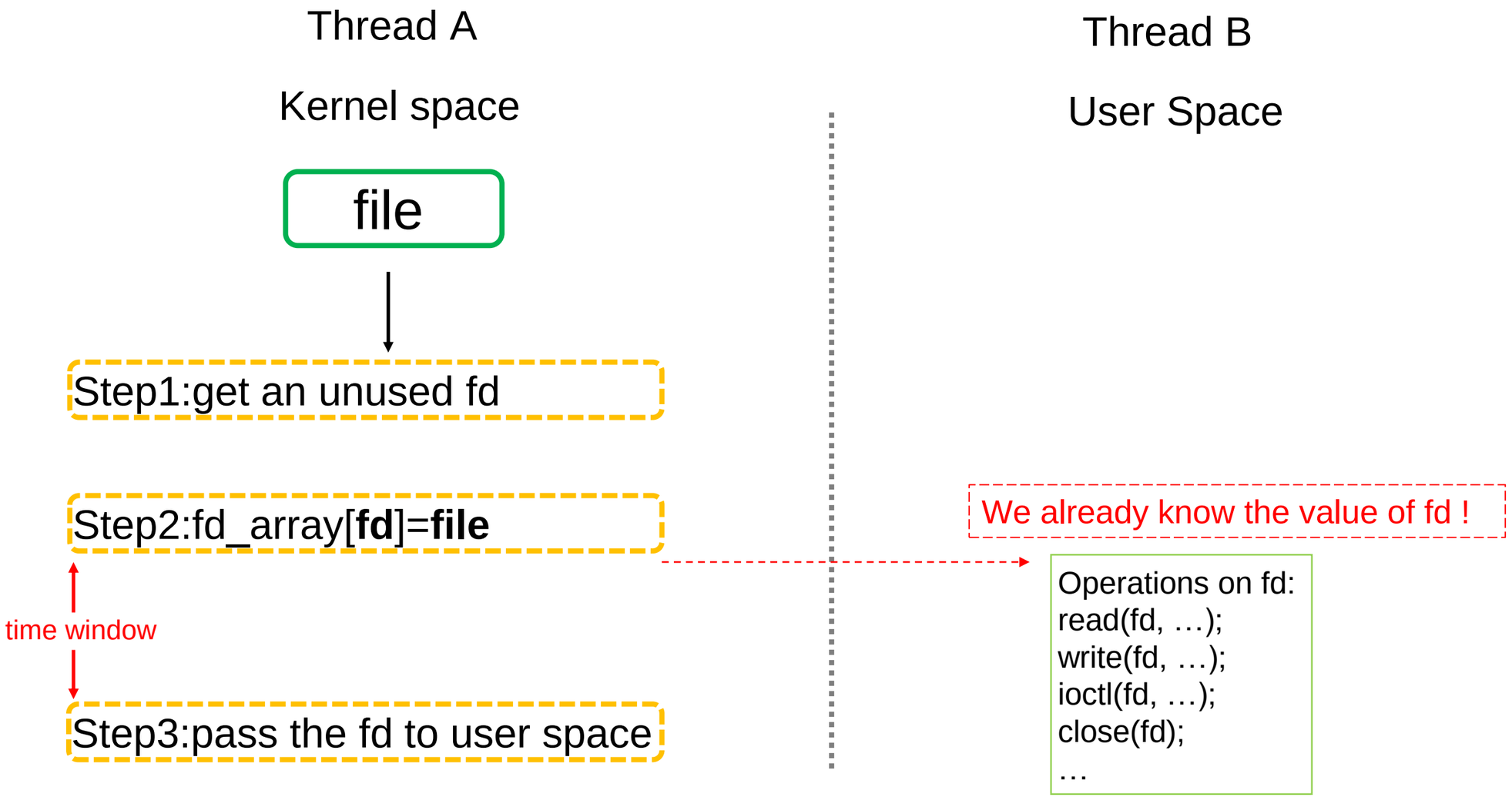
For a new process, fd 0，1，2 are usually occupied, 3 will be the next fd exported from kernel, and then 4, 5, 6……

- Reused after close(fd)

```
int fd = open(file_path, …);
close(fd);
int fd2 = open(file_path2,…);
```

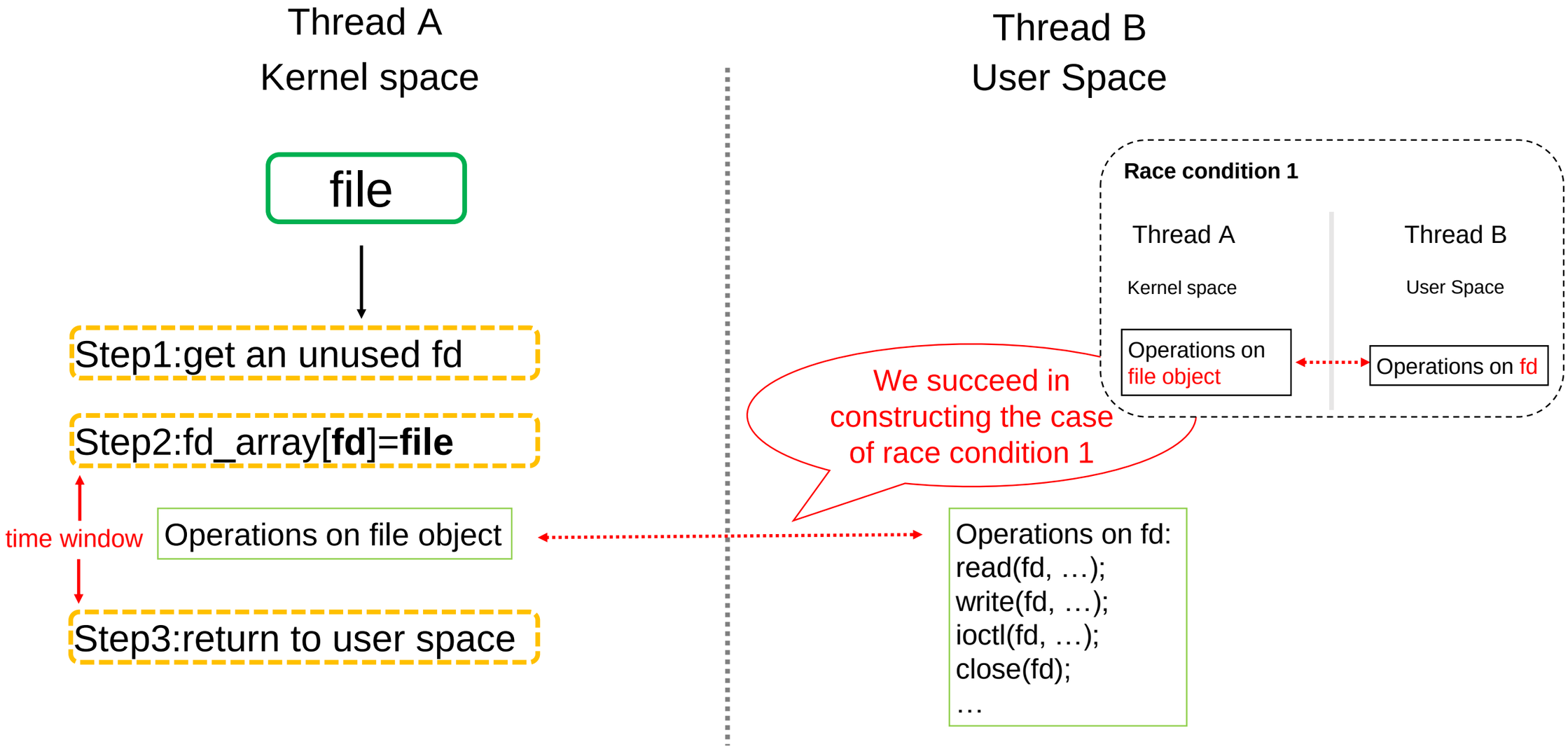
fd2=fd

UAF caused by race condition



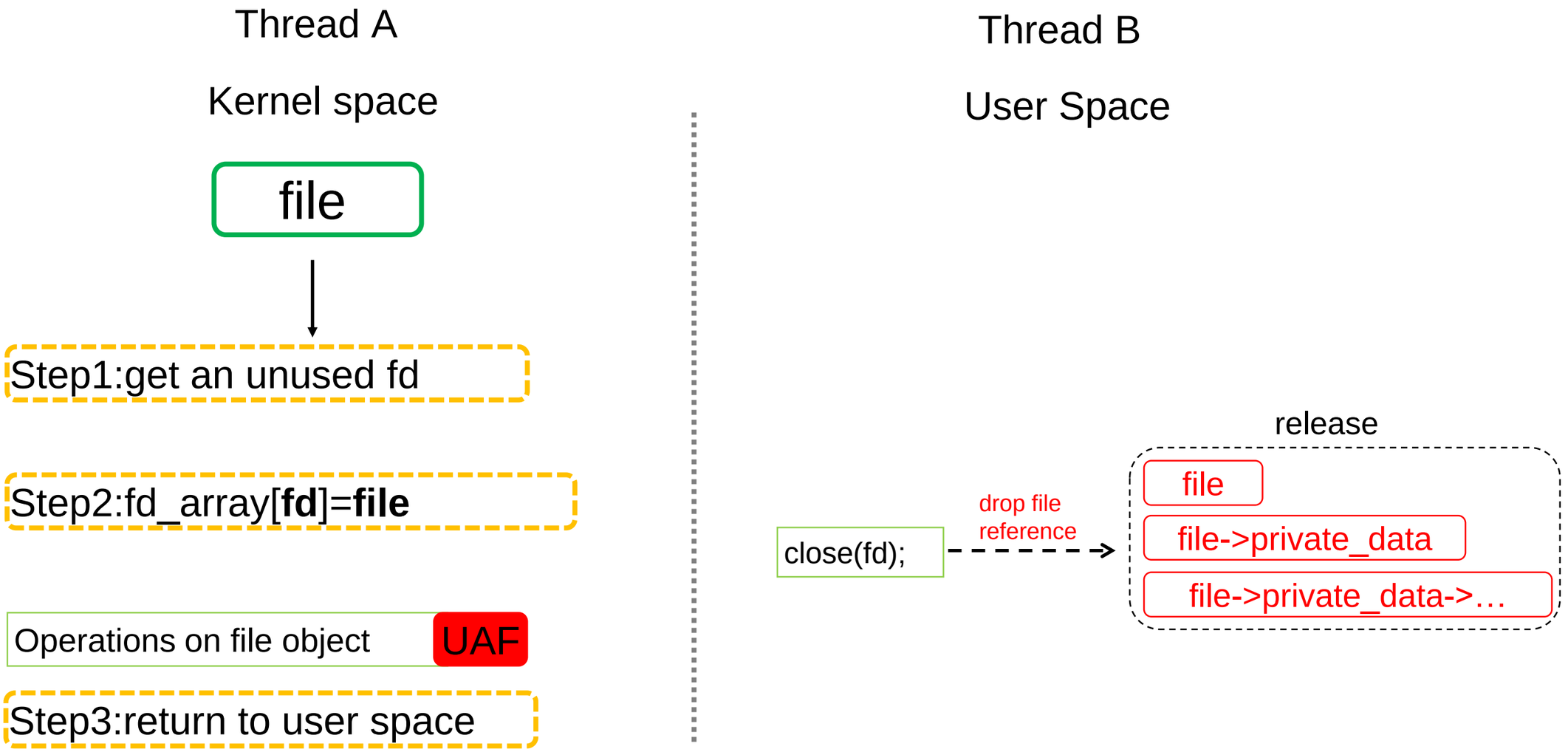
UAF caused by race condition

More assumption:



UAF caused by race condition

A potential UAF scenario:



UAF caused by race condition

Looking for all kinds of kernel APIs which perform the “step2”: Step2:fd_array[**fd**]=**file**

- fd_install(fd, file)

- anon_inode_getfd()
- dma_buf_fd()
- sync_fence_install()
- ion_share_dma_buf_fd()
- …

They all wrap fd_install(fd, file)

UAF caused by race condition

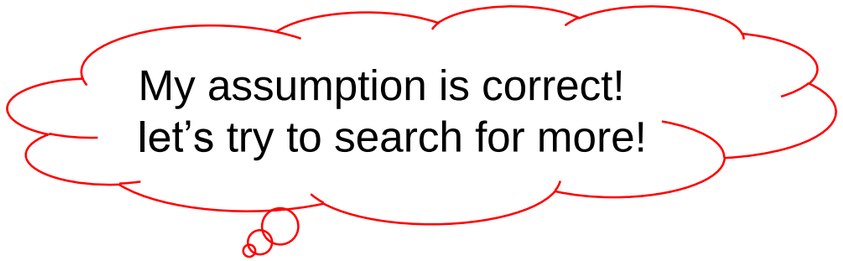
Try to search for the bug pattern: “reference file or related objects after the step2”

From Vendor Q:

```
static int get_fd(uint32_t handle, int64_t *fd)
{
        int unused_fd = -1, ret = -1;
        struct file *f = NULL;
        struct context *cxt = NULL;
        …
        cxt = kzalloc(sizeof(*cxt), GFP_KERNEL);
        …
        unused_fd = get_unused_fd_flags(O_RDWR);
        …
        f = anon_inode_getfile(INVOKE_FILE, &invoke_fops, cxt, O_RDWR);
        …
        *fd = unused_fd;
        fd_install(*fd, f);
        ((struct context *)(f->private_data))->handle = handle;
        return 0;
…
}
```

From Vendor M:

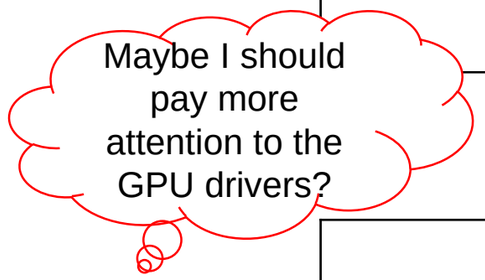
```
int ged_ge_alloc(int region_num, uint32_t *region_sizes)
{
        unsigned long flags;
        int i;
        struct GEEntry *entry =
                (struct GEEntry *)kmem_cache_zalloc(gPoolCache, …);
        …
        entry->alloc_fd = get_unused_fd_flags(O_CLOEXEC);
        …
        entry->file = anon_inode_getfile("gralloc_extra",
                &GEEntry_fops, entry, 0);
        …
        fd_install(entry->alloc_fd, entry->file);

        return entry->alloc_fd;
        …
}
```



UAF caused by race condition

I found since the end of 2021:

From	CVE-id/issue	fd exported by function	Feature
Vendor M	CVE-2022-21771	fd_install()	GPU related driver
	CVE-2022-21773	dma_buf_fd()	dma-buf related
	Duplicated issue#1	dma_buf_fd()	dma-buf related
Vendor Q	CVE-2022-33225	fd_install()	
Vendor S	Issue#1	fd_install()	sync_file related
	Issue#2	dma_buf_fd()	dma-buf related
Linux Mainstream	Issue#1	anon_inode_getfd()	Amd GPU driver
	Issue#2	dma_buf_fd()	dma-buf related



ARM Mali GPU driver	CVE-2022-28349	anon_inode_getfd()	can be triggered from untrusted apps
	CVE-2022-28350	fd_install()	sync_file related, can be triggered from untrusted apps

UAF caused by race condition

CVE-2022-28349—— A Nday in ARM Mali GPU driver

```
int kbase_vinstr_hwcnt_reader_setup(
        struct kbase_vinstr_context *vctx,
        struct kbase_ioctl_hwcnt_reader_setup *setup)
{
        int errcode;
        int fd;
        struct kbase_vinstr_client *vcli = NULL;
        …
        errcode = kbasep_vinstr_client_create(vctx, setup, &vcli);
        …
        errcode = anon_inode_getfd(
                "[mali_vinstr_desc]",
                &vinstr_client_fops,
                vcli,
                O_RDONLY | O_CLOEXEC);
        …
        fd = errcode;
        …
        list_add(&vcli->node, &vctx->clients);
        …
}
```

Affect:

- Midgard GPU Kernel Driver: All versions from r28p0 – r29p0
- Bifrost GPU Kernel Driver: All versions from r17p0 – r23p0
- Valhall GPU Kernel Driver: All versions from r19p0 – r23p0

Android 10 devices of some vendors are affected !

UAF caused by race condition

CVE-2022-28350—— A 0day in ARM Mali GPU driver

```
static int kbase_kcpu_fence_signal_prepare(…)
{
        struct sync_file *sync_file;
        int ret = 0;
        int fd;
        …
        sync_file = sync_file_create(fence_out);
        …
        fd = get_unused_fd_flags(O_CLOEXEC);
        …
        fd_install(fd, sync_file->file);
        …
        if (copy_to_user(u64_to_user_ptr(fence_info->fence), &fence,
                         sizeof(fence))) {
                ret = -EFAULT;
                goto fd_flags_fail;
        }
        return 0;
fd_flags_fail:
        fput(sync_file->file);
        …
        return ret;
}
```

Affect:

Valhall GPU Kernel Driver: All versions from r29p0 – r36p0

Android 12 devices of some vendors are affected !

UAF caused by race condition

Exploit of CVE-2022-28350

- A known exploit method

Given by Mathias Krause from grsecurity for a similar vulnerability CVE-2022-22942:

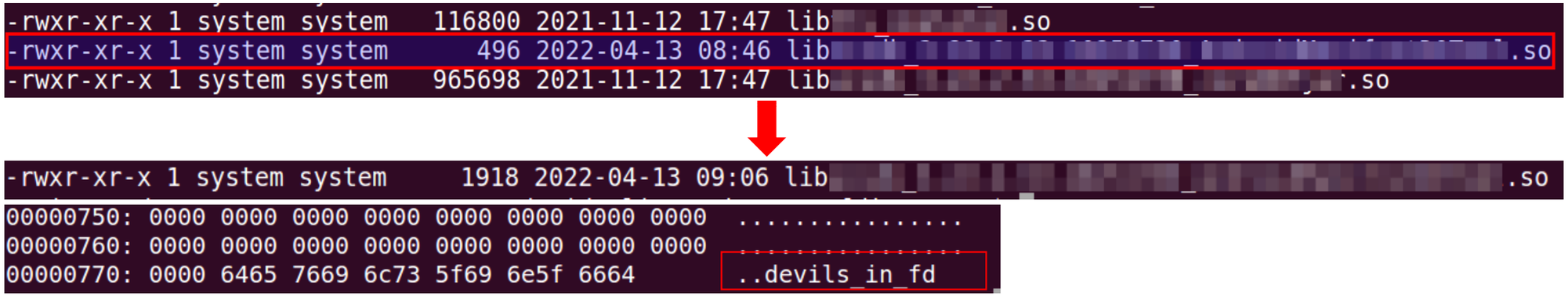
- No need for KASLR、SMEP/SMAP 、 KCFI bypass
- Read/write privileged files from unprivileged processes

The method won't work on Android because of SELinux ☹

- **My new exploit method**

- Bypass SELinux and work on the affected Android 12 devices
- Write privileged files from untrusted apps

(Details are put in the supplement part of the slides)



Find the issues

Check if the file or related objects are referenced after these functions:

- fd_install(fd, file)

- anon_inode_getfd()
- dma_buf_fd()
- sync_fence_install()
- ion_share_dma_buf_fd()
- …

They all wrap fd_install(fd, file)

Fixes

- Don’t reference the file or related objects after step2 of fd export operation in kernel until return to user space

√:

```
static long do_sys_openat2(int dfd, const char __user *filename,
                           struct open_how *how)
{
        struct open_flags op;
        int fd = build_open_flags(how, &op);
        …
        fd = get_unused_fd_flags(how->flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        …
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}
```

return to user space directly

Fixes

- Reference the file object or related objects with lock protection, and share the lock in file_release of fd:

```
int fd_export_func(…) {
        mutex_lock(g_lock);
        fd_install(file, fd);
        Reference file or related objects;
        mutex_unlock(g_lock);
        return fd;
}
```

```
int file_release(…) {
        …
        mutex_lock(g_lock);
        …
        mutex_unlock(g_lock);
        …
}
```

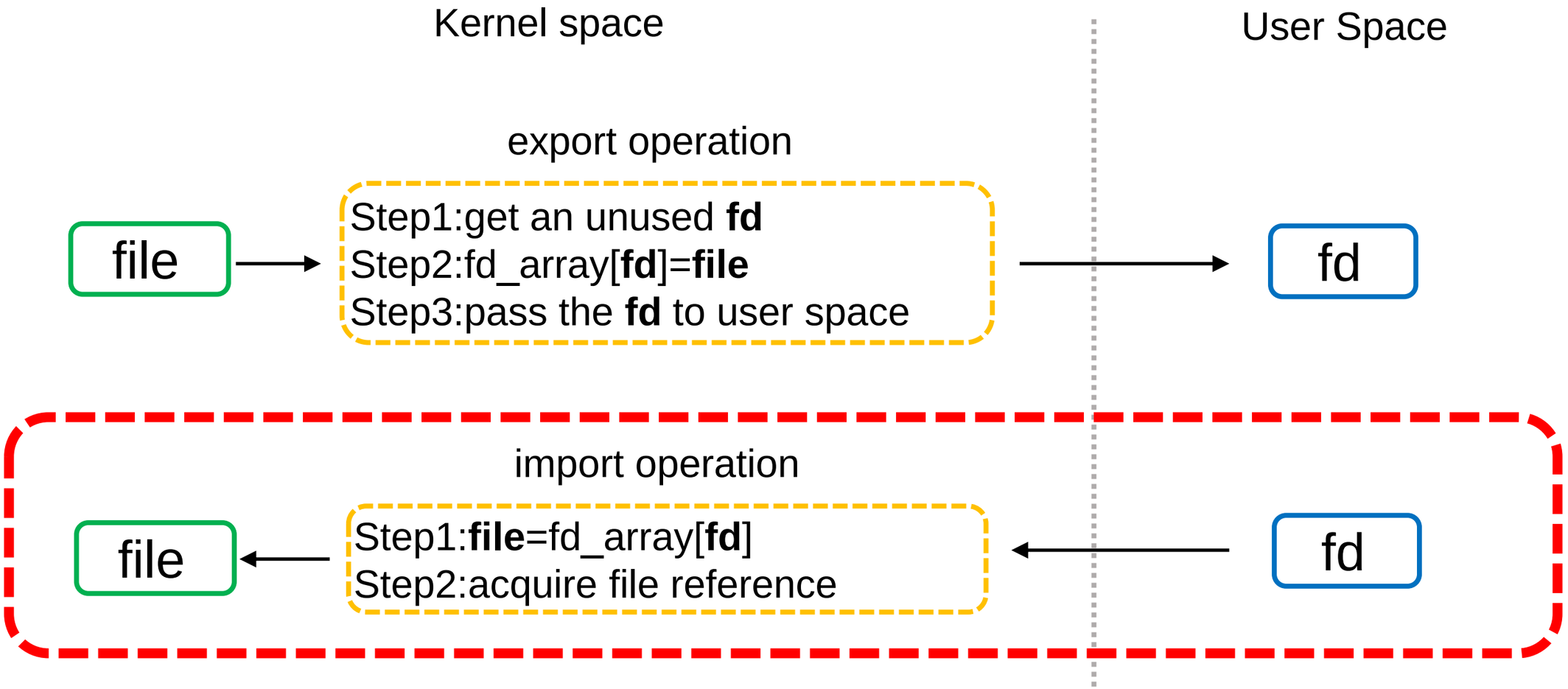
close(fd)

√: (From vendor S)

```
void hpa_trace_add_task(void)
{
        struct hpa_trace_task *task;
        …
        mutex_lock(&hpa_trace_lock);
        …
        task = kzalloc(sizeof(*task), GFP_KERNEL);
        …
        fd = get_unused_fd_flags(O_RDONLY | O_CLOEXEC);
        …
        task->file = anon_inode_getfile(name, &hpa_trace_task_fops, task, O_RDWR);
        …
        fd_install(fd, task->file);
        list_add_tail(&task->node, &hpa_task_list);
        mutex_unlock(&hpa_trace_lock);
        …
}
```

```
static int hpa_trace_task_release(struct inode *inode, struct file *file)
{
        struct hpa_trace_task *task = file->private_data;
        …
        mutex_lock(&hpa_trace_lock);
        list_del(&task->node);
        mutex_unlock(&hpa_trace_lock);
        kfree(task);
        return 0;
}
```

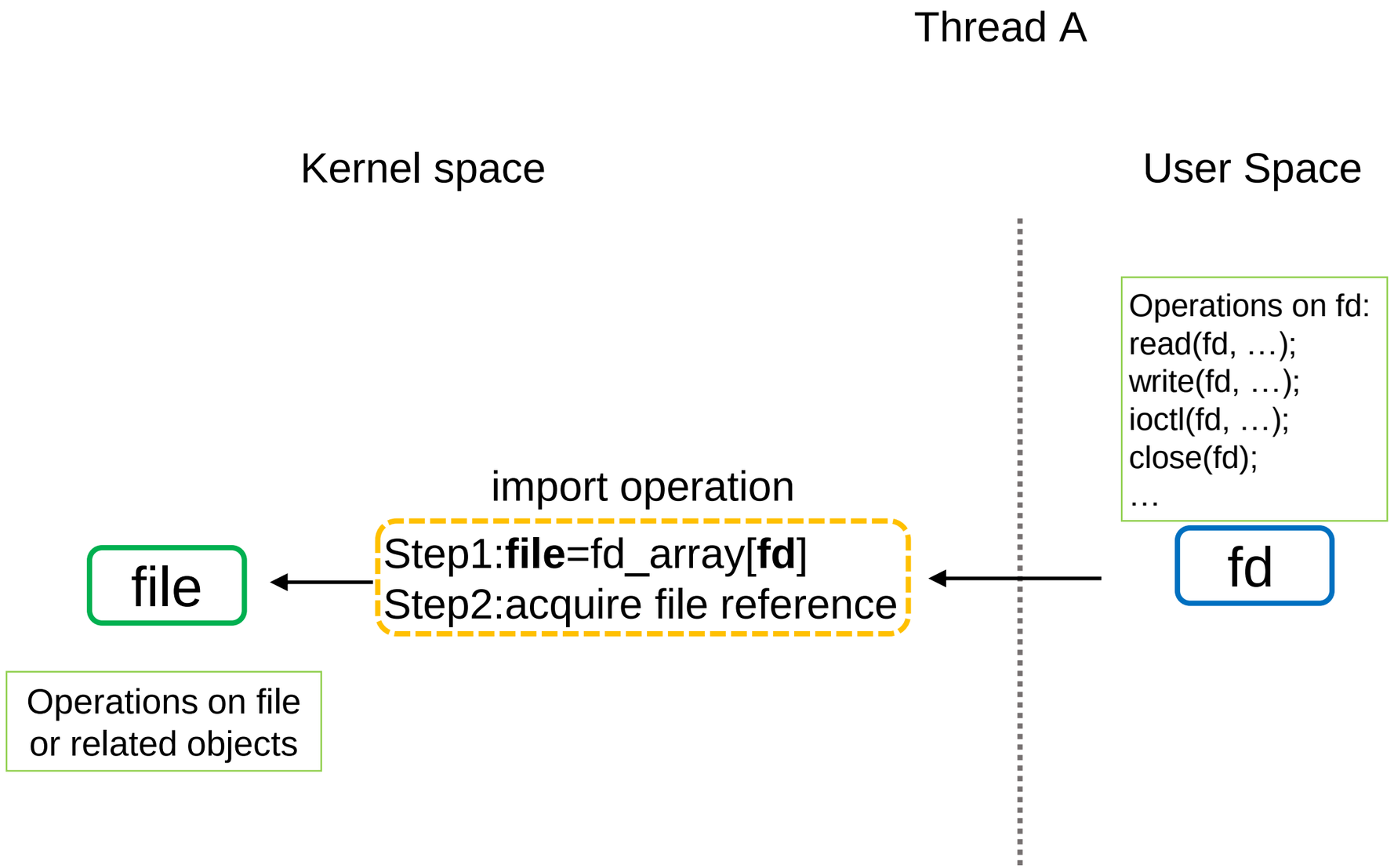
Diving into issues in the fd import operation



Diving into issues in the fd import operation

- Scenario of fd import operation
- Fd type confusion caused by race condition
- Find the issues
- Fixes

Scenario of fd import operation



Scenario of fd import operation

Example:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
        return ksys_write(fd, buf, count);
}
```

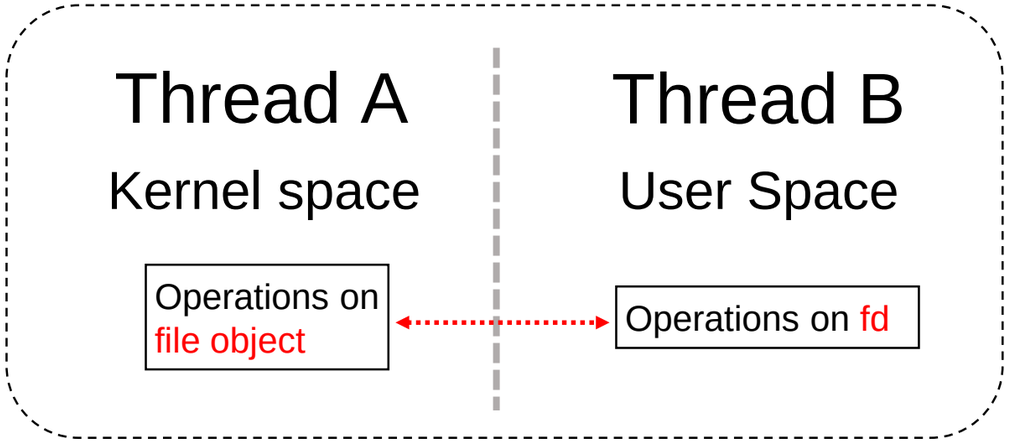
Step1:**file**=fd_array[**fd**]
Step2:acquire file reference

```
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
        struct fd f = fdget_pos(fd);
        …
        if (f.file) {
                …
                ret = vfs_write(f.file, buf, count, ppos);
                …
                fdput_pos(f);
        }
…
}
```

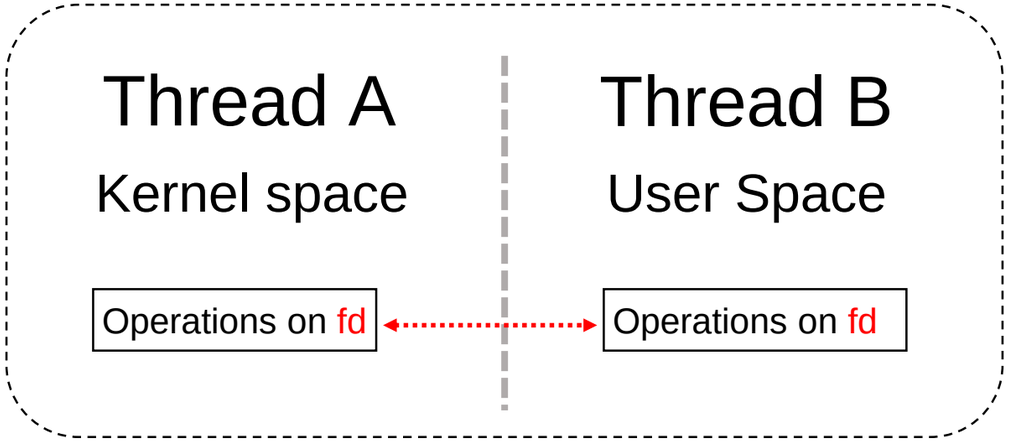
Scenario of fd import operation

But this regular fd import operation is executed sequentially, which is still far from the race conditions we want to see:

Race condition 1



Race condition 2



Searching for all kinds of scenarios of fd import operation in kernel…

Fd type confusion caused by race condition

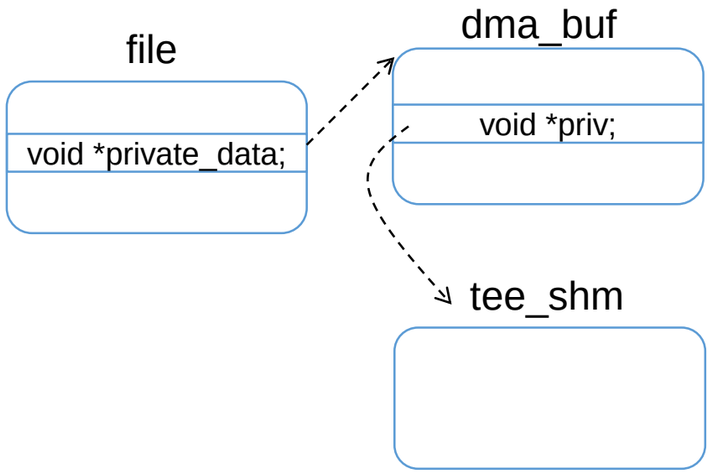
Special case1: CVE-2022-21772

```
TEEC_Result TEEC_RegisterSharedMemory(struct TEEC_Context *ctx,
                                      struct TEEC_SharedMemory *shm)
{
        int fd;
        size_t s;
        struct dma_buf *dma_buf;
        struct tee_shm *tee_shm;
        …
        fd = teec_shm_alloc(ctx->fd, s, &shm->id);
        …
        dma_buf = dma_buf_get(fd);
        close(fd);
        …
        tee_shm = dma_buf->priv;

        …
        shm->shadow_buffer = tee_shm->kaddr;
        …
        return TEEC_SUCCESS;
}
```

create a specific dma-buf fd

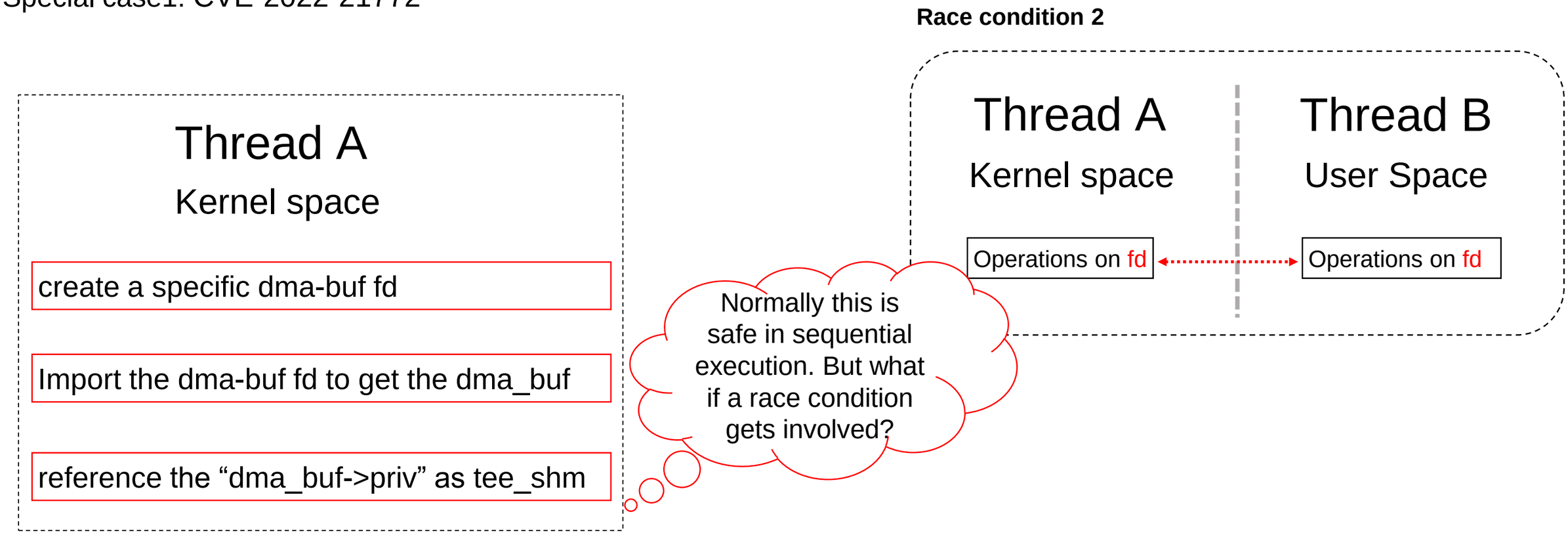
import the dma-buf fd to get the dma_buf

reference the “dma_buf->priv” as tee_shm



Fd type confusion caused by race condition

Special case1: CVE-2022-21772



Fd type confusion caused by race condition

Special case1: CVE-2022-21772

Thread A

Kernel space

create a specific dma-buf fd:
fd = teec_shm_alloc(ctx->fd, s, &shm->id);

Import the dma-buf fd to get the **dma_buf**:
dma_buf = dma_buf_get(fd);

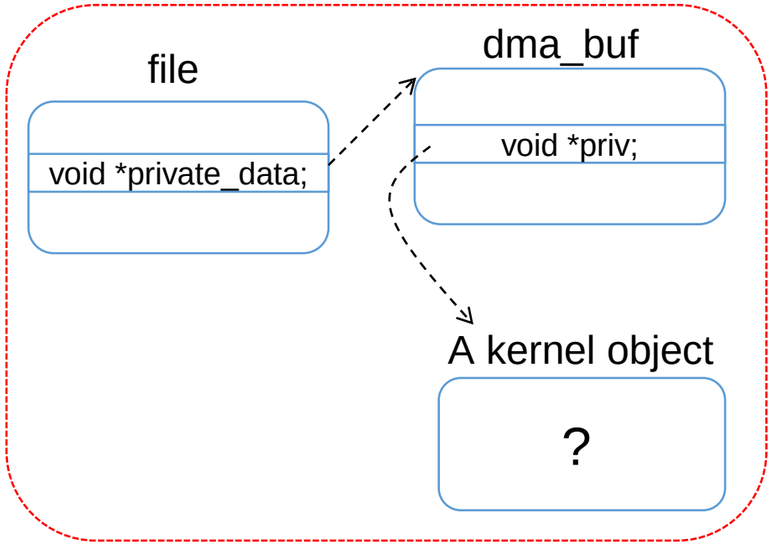
reference the "**dma_buf-**>priv" as tee_shm:
tee_shm = **dma_buf-**>priv;

fd type confusion happens!

Thread B

User Space

Recreate the fd:
close(fd);
fd = create_a_diff_dma_buf_fd();



Fd type confusion caused by race condition

Special case2:

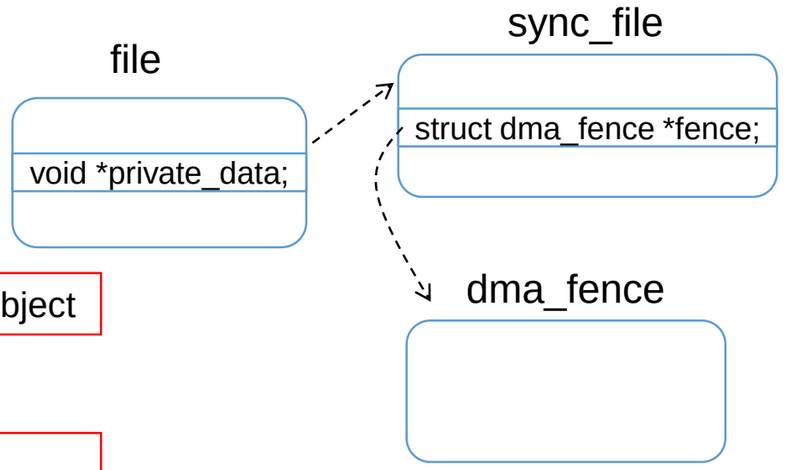
```
struct sync_file*internal_sync_fence_fdget(int fd)
{
        struct file *file;
        struct dma_fence *fence = sync_file_get_fence(fd);

        /* Verify whether the fd is a valid sync file. */
        if (unlikely(!fence))
                return NULL;

        dma_fence_put(fence);

        file = fget(fd);
        return file->private_data;
}
```

Import fd to get dma_fence object

Check the dma_fence object

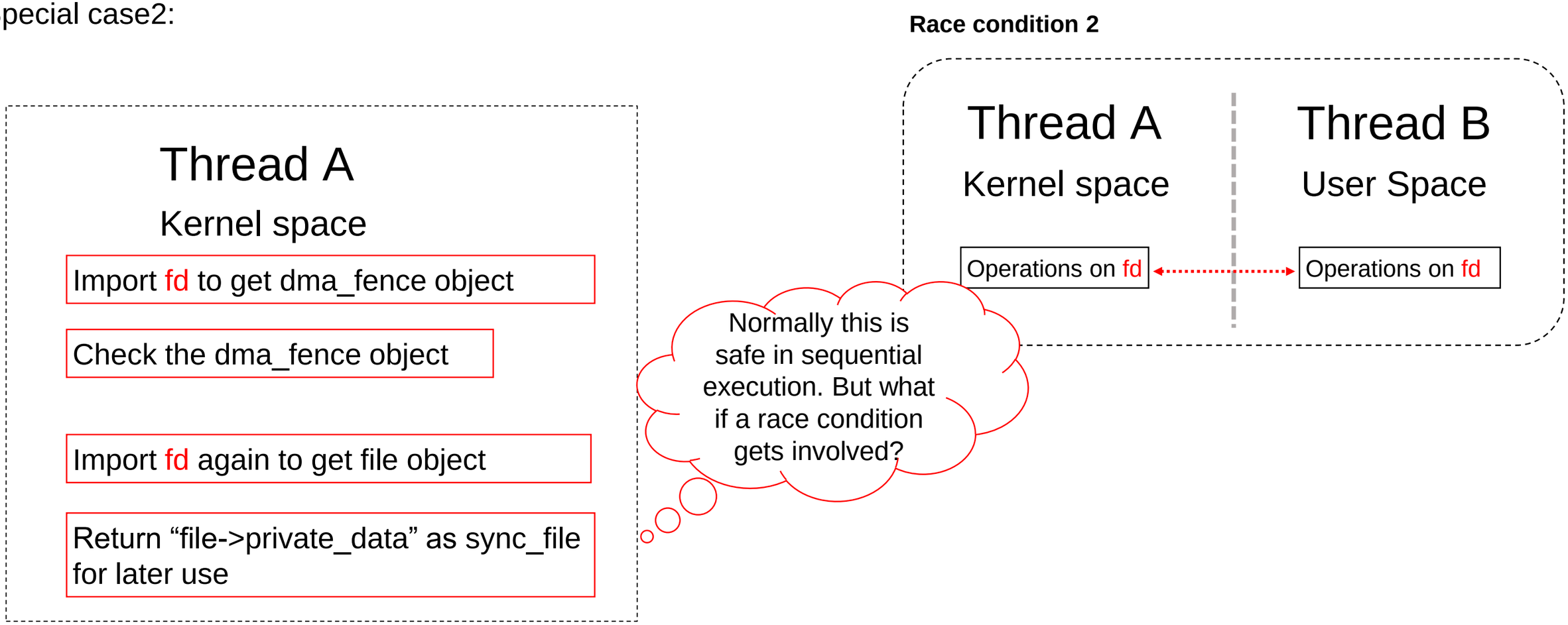
Import fd again to get file object

Return “file->private_data” as sync_file for later use



Fd type confusion caused by race condition

Special case2:



Fd type confusion caused by race condition

Special case2:

Thread A
Kernel space

Import fd to get dma_fence object:
struct dma_fence *fence = sync_file_get_fence(fd);

Check the dma_fence object

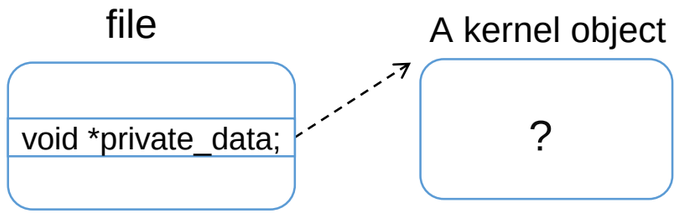
Import fd to get file object:
file = fget(fd);

Return “file->private_data” as sync_file for later use

fd type confusion happens!

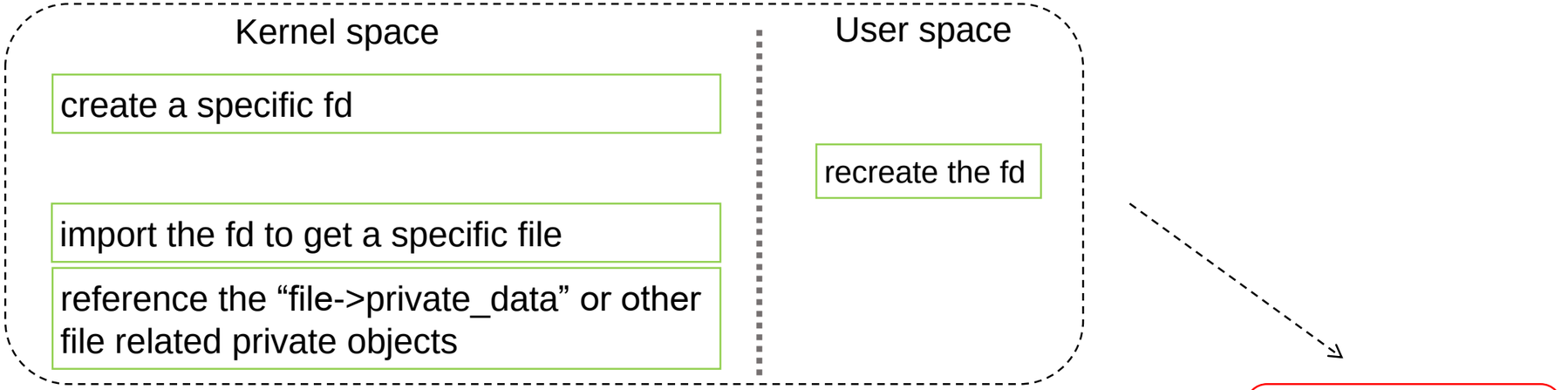
Thread B
User Space

Recreate the fd:
close(fd);
fd = open();

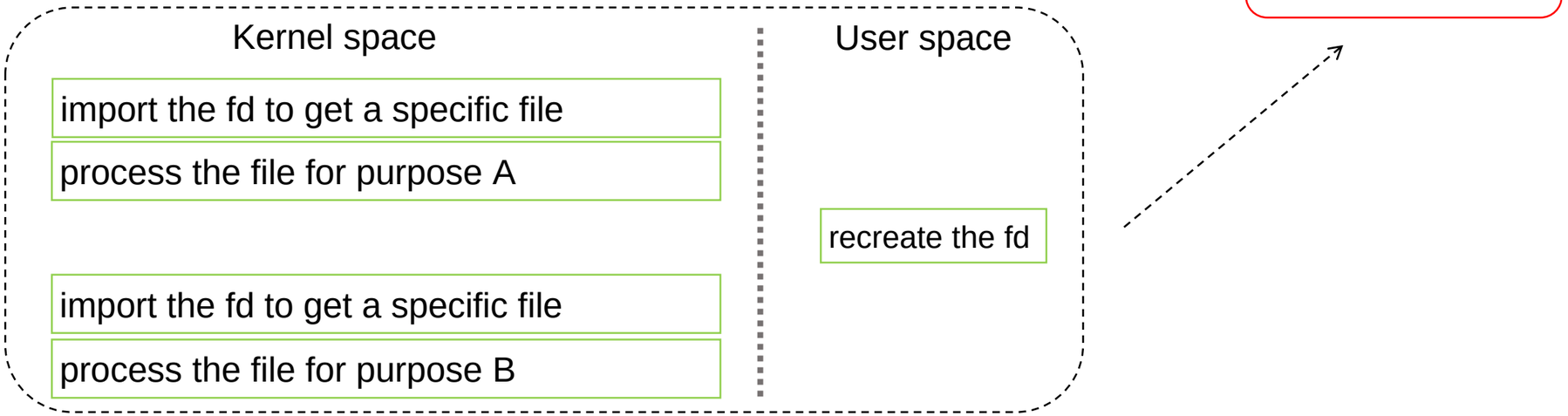


Fd type confusion caused by race condition

- Case1: fd time-of-create time-of-import



- Case2: fd double import



fd type confusion might happen!

Find the issues

There are still two questions that need to be answered:

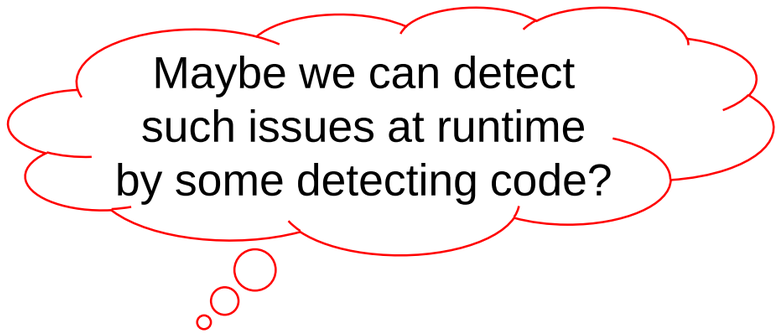
- Are there more issues like these?
- How to find these issues more effectively?

The difficulty of fuzzing the fd type confusion caused by race condition:

- The buggy code is lurking in kernel, the user process can barely notice it!
- The race window can be tiny!

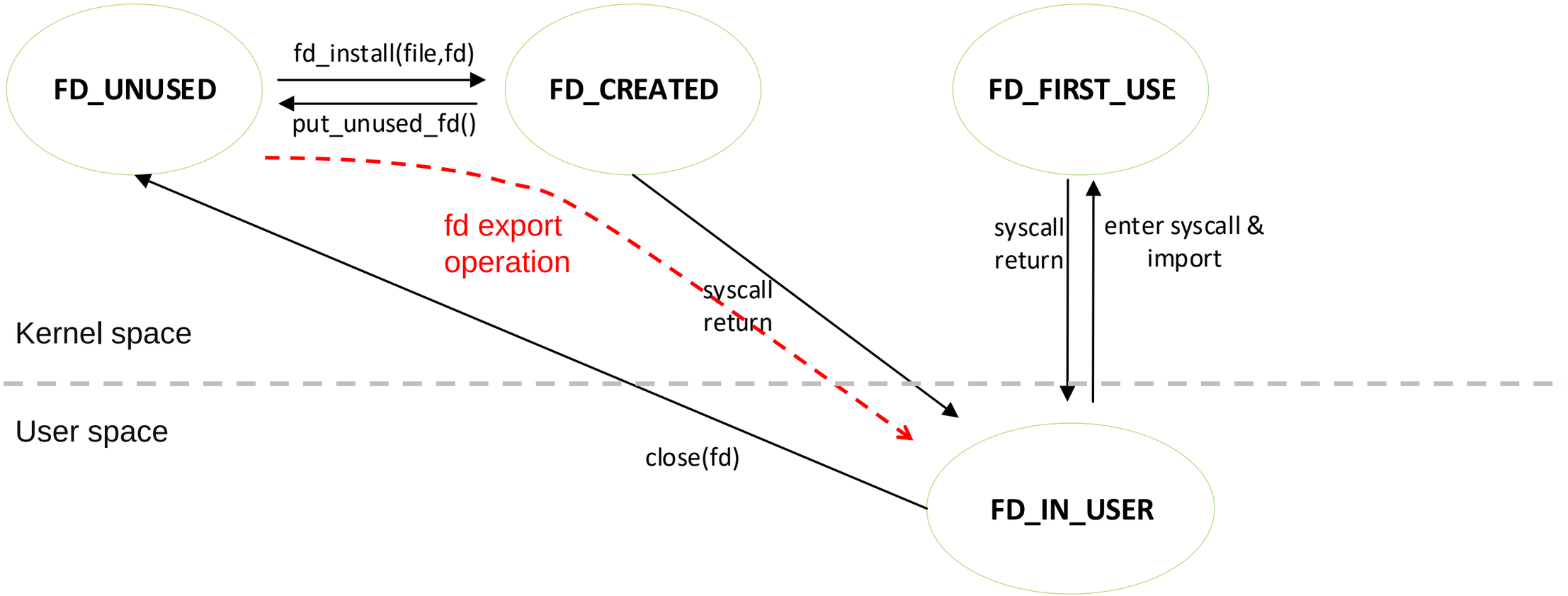
CVE-2022-21772

```
…
fd = teec_shm_alloc(ctx->fd, s, &shm->id);
…
dma_buf = dma_buf_get(fd);
close(fd);
…
```



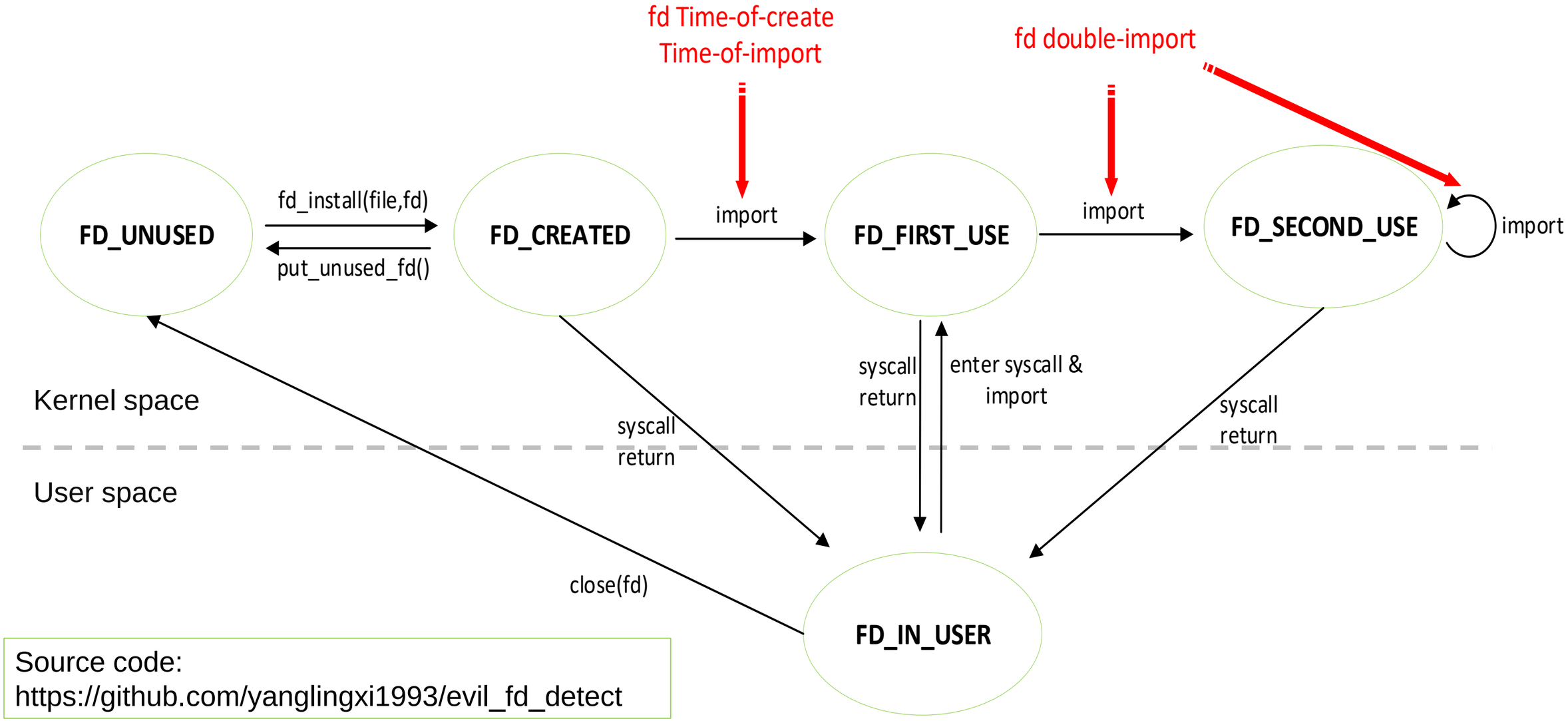
Find the issues

Regular lifecycle of an fd:



Find the issues

Detecting the potential issues:



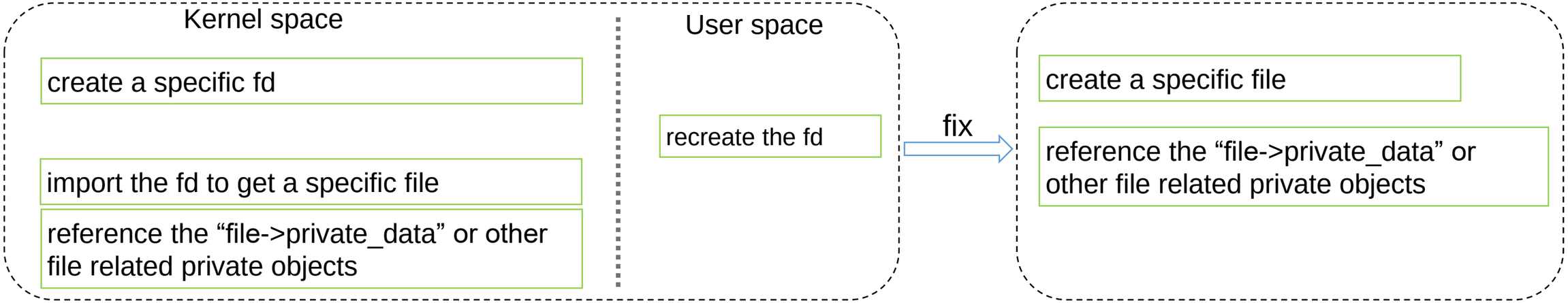
Source code:
https://github.com/yanglingxi1993/evil_fd_detect

Bug hunting result

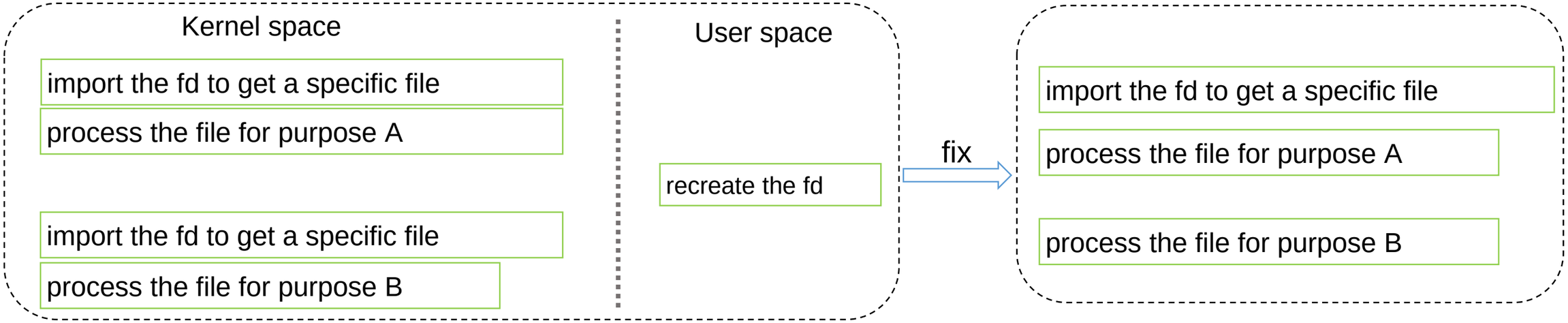
type	From	CVE-id/issue	Found by
fd time-of-create time-of-import	Vendor M	CVE-2022-21772	code auditing
		Issue#1	detect tool
		Issue#2	detect tool
	Vendor S	Issue#1	code auditing
	Vendor Q	Issue#1	detect tool
fd double import	Vendor M	CVE-2022-20082	code auditing
		Issue#1	detect tool
		Issue#2	detect tool
	Vendor Q	Issue#1	code auditing
		Issue#2	code auditing
		Issue#3	code auditing

Fixes

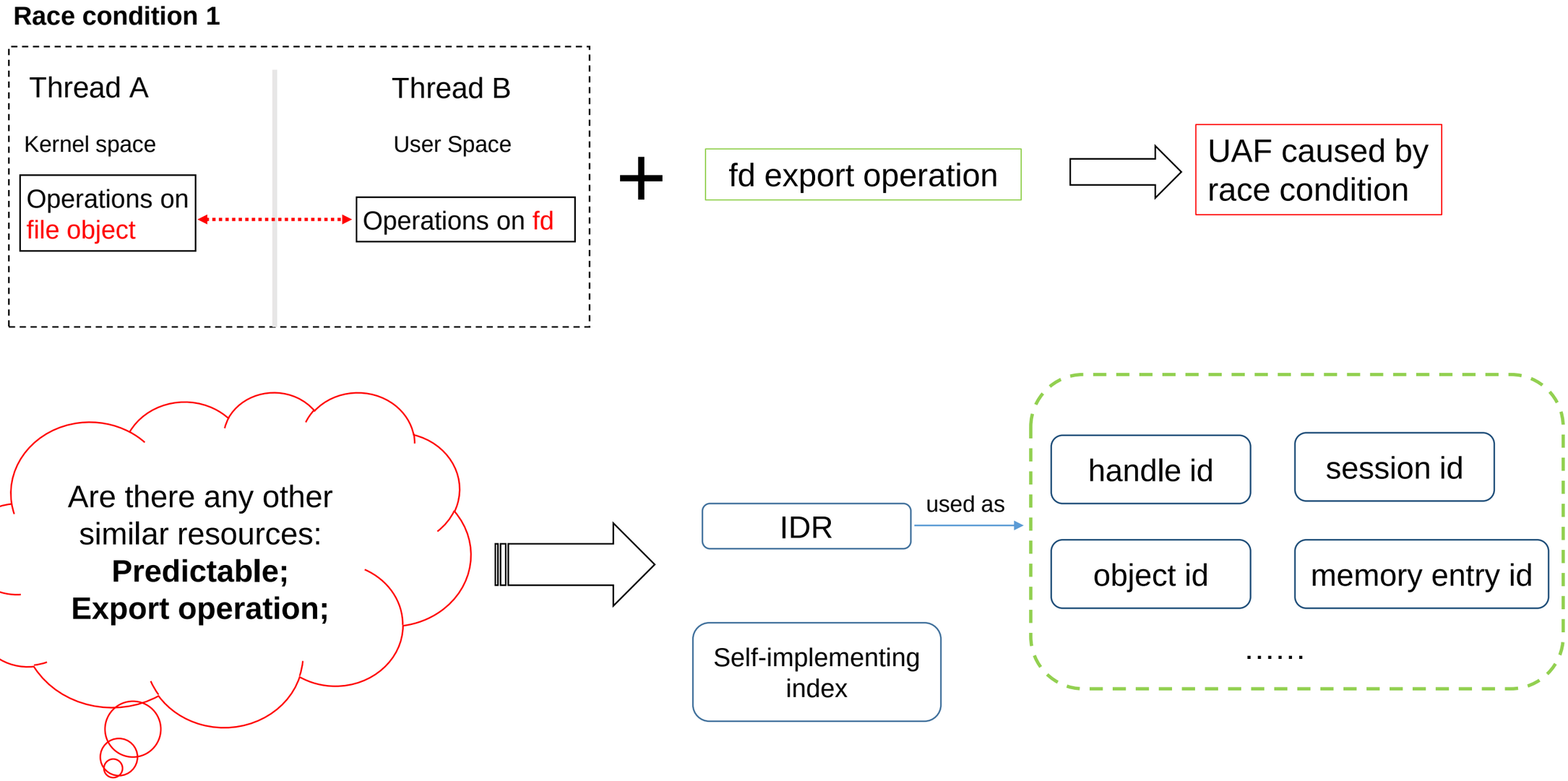
- Case1: fd time-of-create time-of-import



- Case2: fd double import

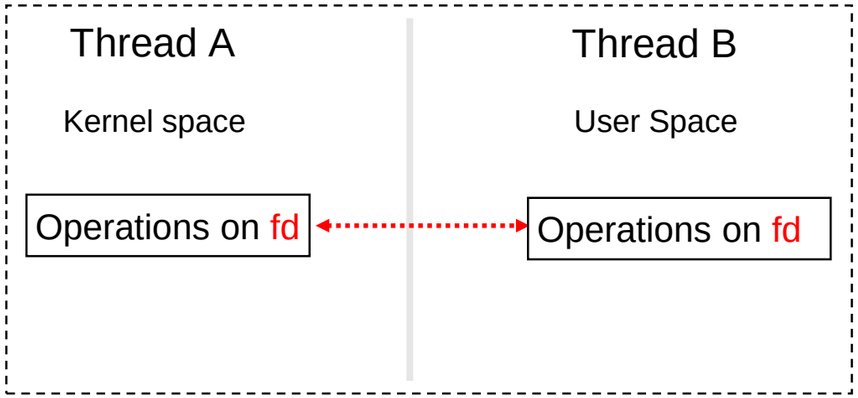


Conclusion & Future work



Conclusion & Future work

Race condition 2



\+ fd import operation ⇨ fd type confusion caused by race condition

Are there any other similar resources: **import operation;**

⇨

- IDR
- pid → task_struct
- user address → vma
- …

Acknowledge

Thanks to *某因幡*, Ye Zhang, Chenfu Bao, Shufan Yang, Lin Wu, Yakun Zhang, Zheng Huang, Tim Xia

Supplement

- Exploit of CVE-2022-28350
- Small race windows can be exploitable!
  - UAF caused by race condition in fd export operation
  - Fd type confusion caused by race condition in fd import operation

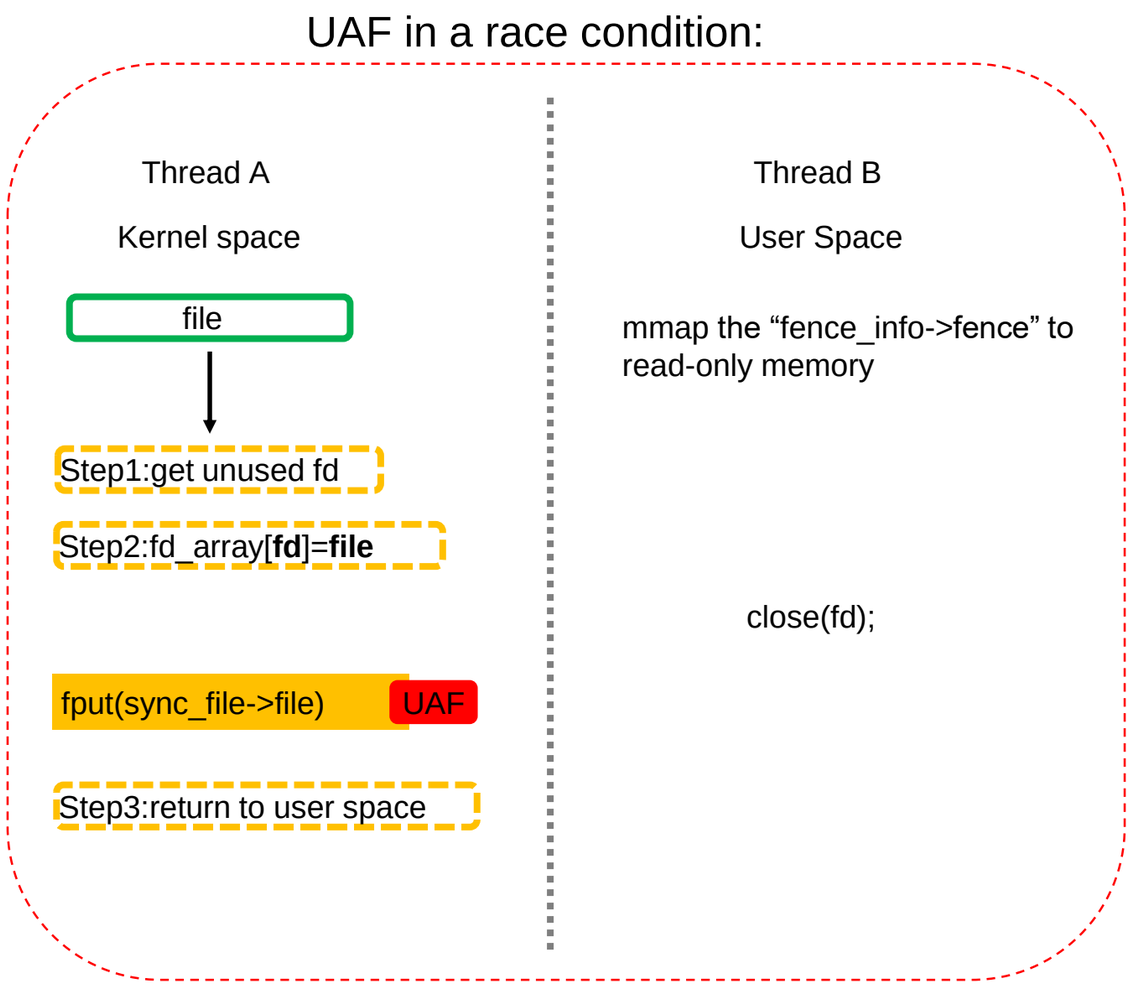
Supplement

- Exploit of CVE-2022-28350
- Small race windows can be exploitable!
  - UAF caused by race condition in fd export operation
  - Fd type confusion caused by race condition in fd import operation

Exploit of CVE-2022-28350

What will CVE-2022-28350 lead to?

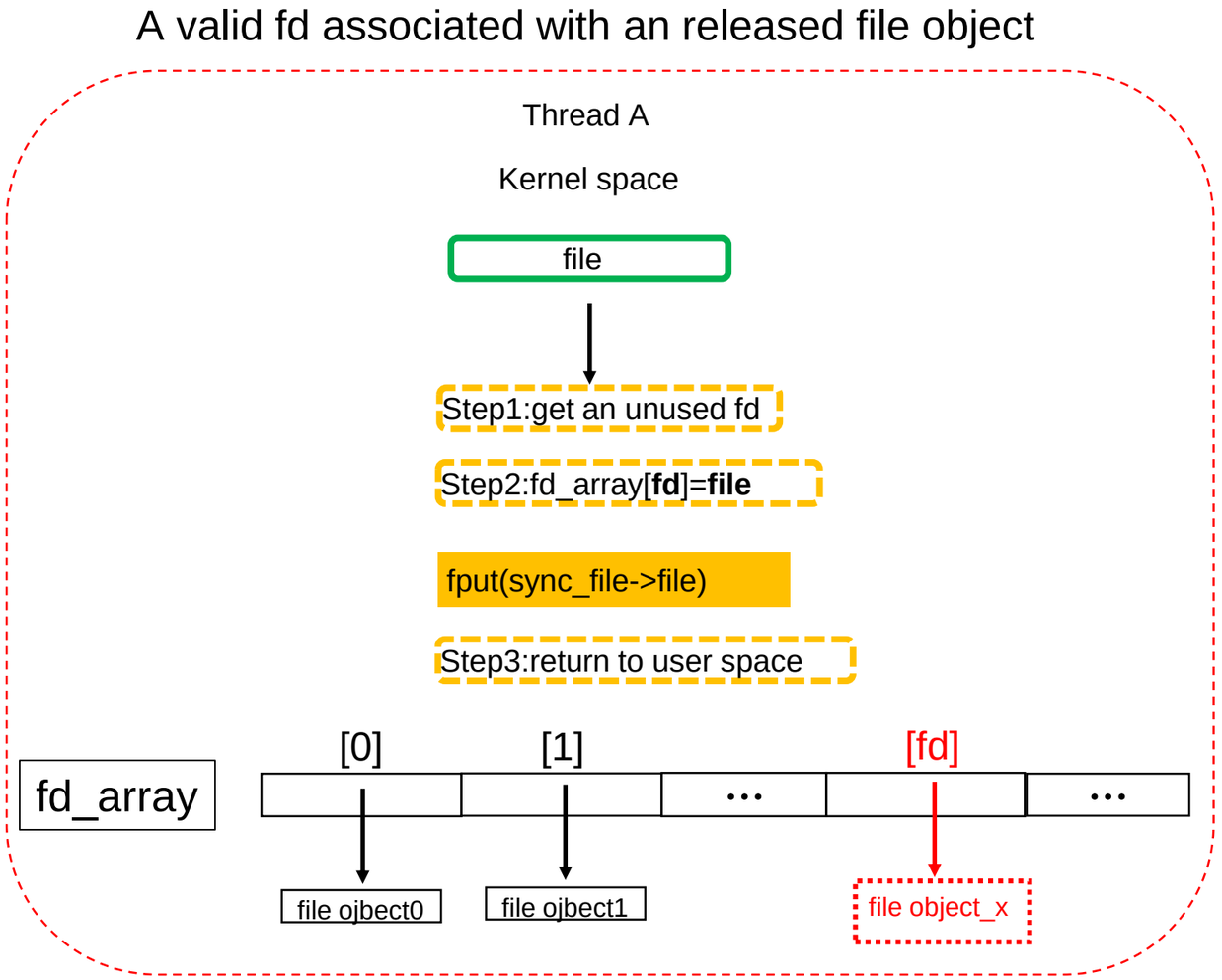
```
static int kbase_kcpu_fence_signal_prepare(…)
{
        …
        struct sync_file *sync_file;
        int ret = 0;
        int fd;

        …
        sync_file = sync_file_create(fence_out);
        …
        fd = get_unused_fd_flags(O_CLOEXEC);
        …
        fd_install(fd, sync_file->file);
        …
        if (copy_to_user(u64_to_user_ptr(fence_info->fence), &fence,
                         sizeof(fence))) {
                ret = -EFAULT;
                goto fd_flags_fail;
        }
        return 0;

fd_flags_fail:
        fput(sync_file->file);
        …
        return ret;
}
```



Exploit of CVE-2022-28350

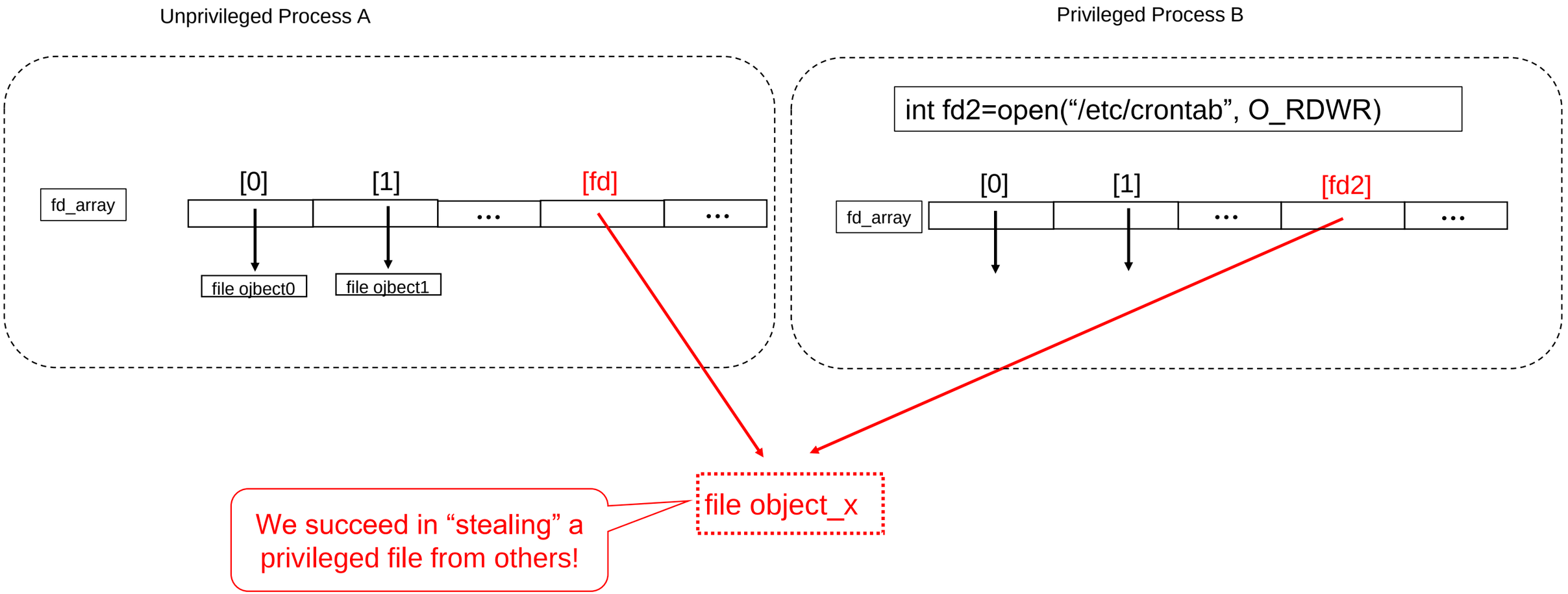
But the CVE-2022-28350 can do more:

```
static int kbase_kcpu_fence_signal_prepare(…)
{
        …
        struct sync_file *sync_file;
        int ret = 0;
        int fd;

        …
        sync_file = sync_file_create(fence_out);
        …
        fd = get_unused_fd_flags(O_CLOEXEC);
        …
        fd_install(fd, sync_file->file);
        …
        if (copy_to_user(u64_to_user_ptr(fence_info->fence), &fence,
                         sizeof(fence))) {
                ret = -EFAULT;
                goto fd_flags_fail;
        }
        return 0;

fd_flags_fail:
        fput(sync_file->file);
        …
        return ret;
}
```



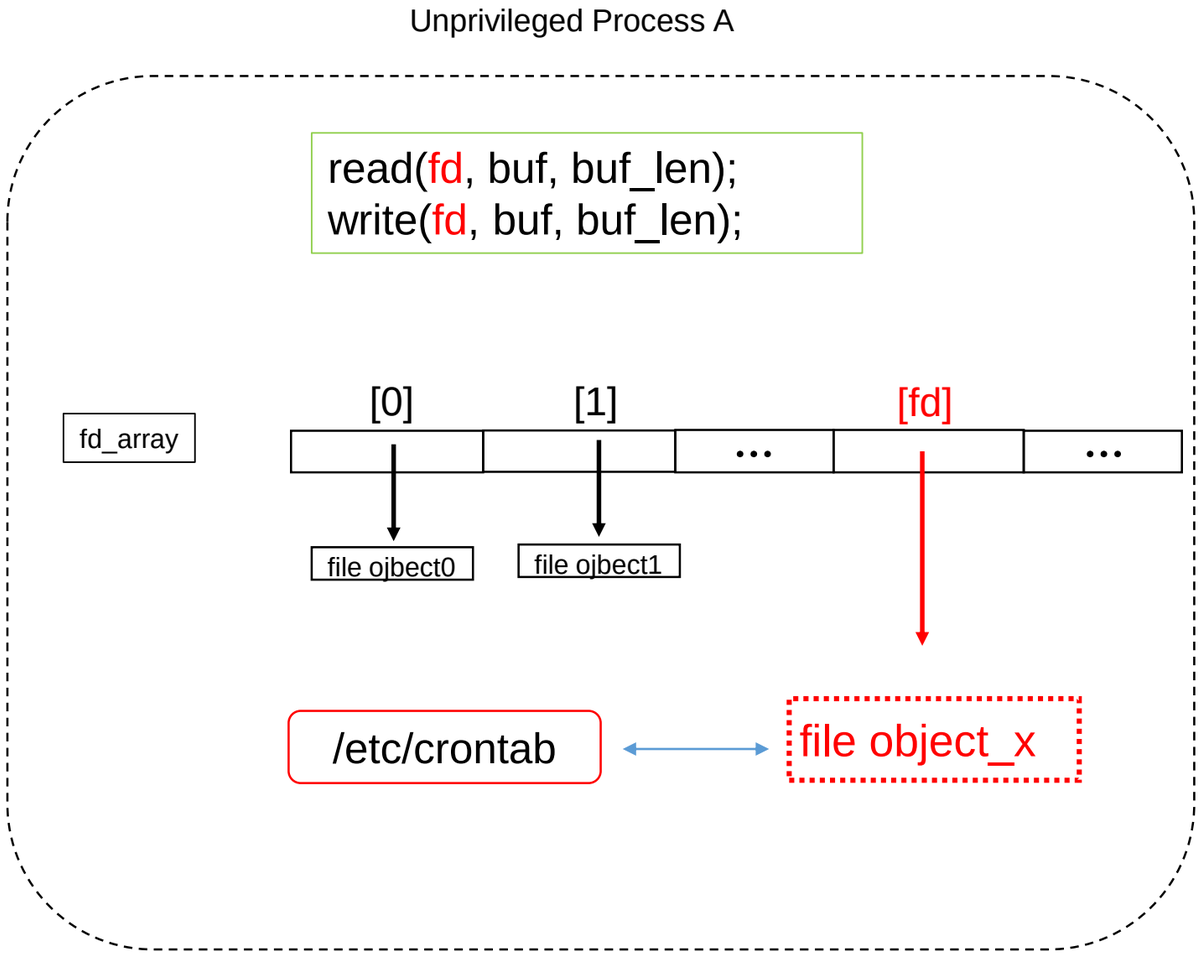
Exploit of CVE-2022-28350

So what if the released file object get reused by some other privileged processes when opening a privileged file?



Exploit of CVE-2022-28350

If the SELinux is disabled, the unprivileged process will have the ability to read/write the “stolen” privileged file:



Is it strange that we can bypass the DAC of privileged file to perform the read/write operation?

-rw-r--r-- 1 root root 722 4月 6 2016 /etc/crontab

The answer is:
The DAC is only checked in open(). There are no DAC checks in read() and write() ☺

Exploit of CVE-2022-28350

The exploitation method of “stealing” privileged file from others has been mentioned by Mathias Krause here ,
but this won’t work on Android.

On Android, the unprivileged process cannot read/write the “stolen”privileged file because of SELinux ☹

read(fd, buf, buf_len);

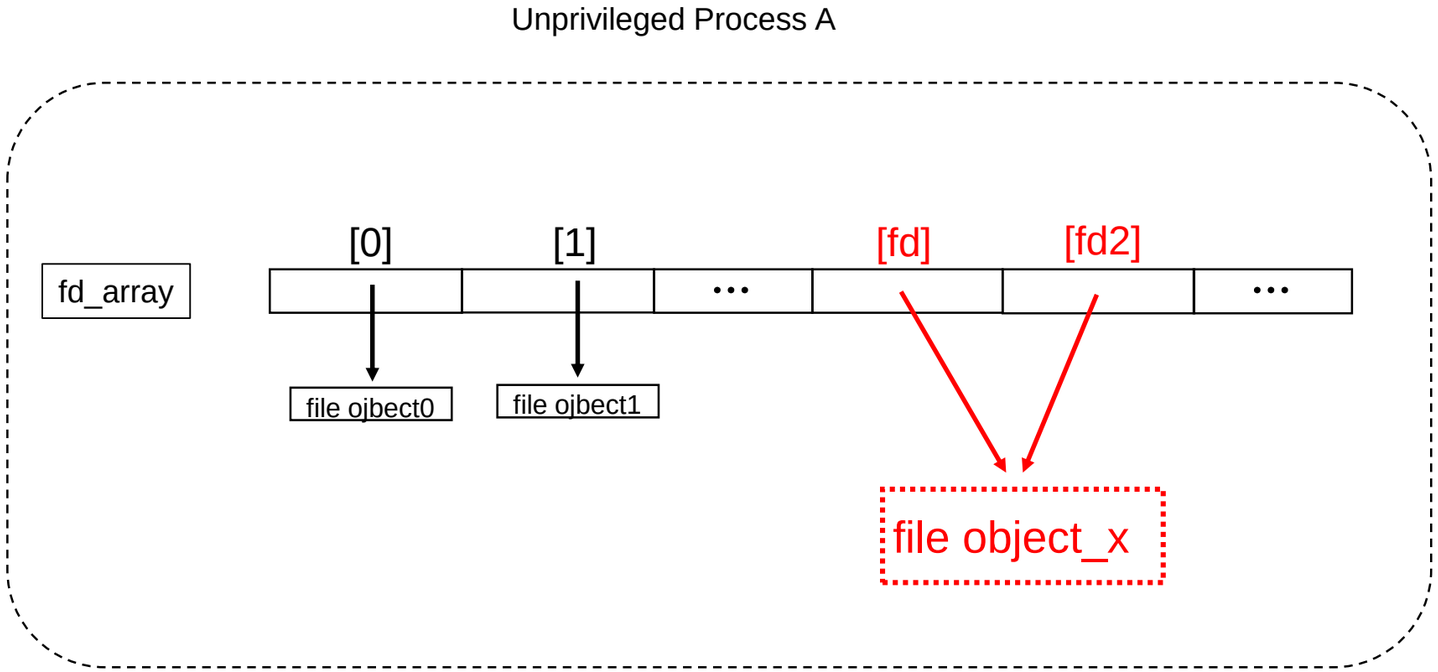
write(fd, buf, buf_len);

```
int rw_verify_area(int read_write, struct file *file, const loff_t *ppos,
size_t count)
{
        …
        return security_file_permission(file,
                        read_write == READ ? MAY_READ : MAY_WRITE);
}
```

Exploit of CVE-2022-28350

Let’s find some other way out!

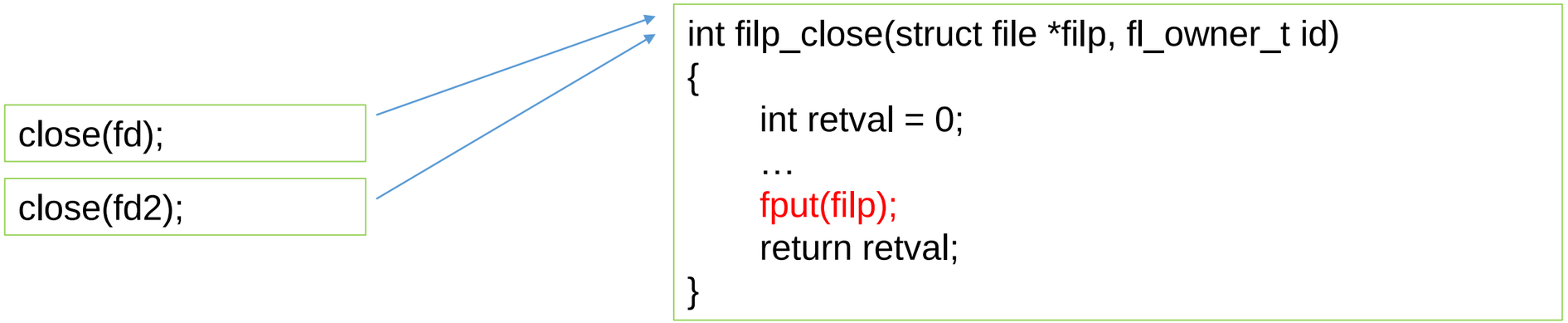
What if the released file object gets reused in the same process?



Two different fds are associated with a same file object! But the refcount of the file object is still 1

Exploit of CVE-2022-28350

What happens if we close both fd and fd2?

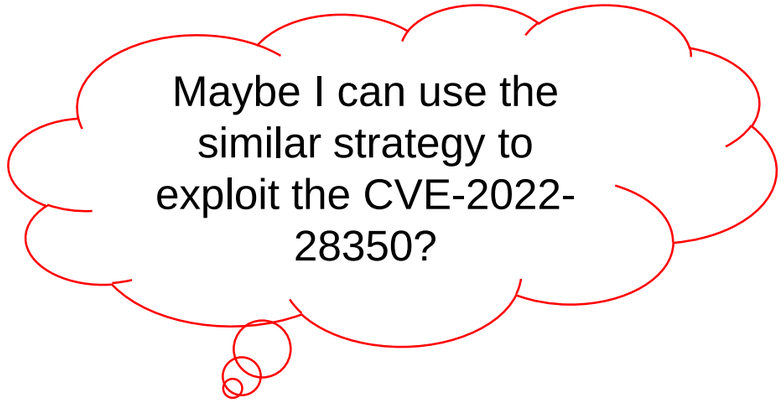


A double-fput() vulnerability has been constructed!!!

Exploit of CVE-2022-28350

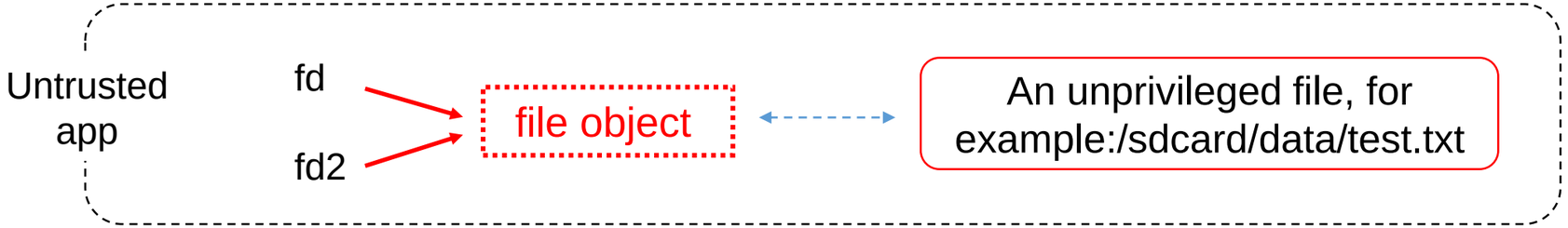
What can we do with a double-fput() vulnerability?

Jann Horn from Google Project Zero has given an answer to this question here, he showed how to write a privileged file from a unprivileged process with a double-fput() vulnerability！



My exploit for CVE-2022-28350

Step1: Construct the scene with CVE-2022-28350



My exploit for CVE-2022-28350

Step2: try to write the privileged file in a race condition

Thread A

write(fd, evil_content, len);

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t
count,…)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_WRITE))
                return -EBADF;
        …
        ret = rw_verify_area(WRITE, file, pos, count);
        …









        if (file->f_op->write)
                ret = file->f_op->write(file, buf, count, pos);
        …
        return ret;
}
```

write mode check

SELinux check

Succeed in writing the privileged file!

Thread B

close(fd);close(fd2);
open(privileged_file_path, O_RDONLY);

release the file object

reuse the file object

file object ⟷ The privileged file

My exploit for CVE-2022-28350

The tiny race window is still a challenge:

Thread A

write(fd, evil_content, len);

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t
count,…)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_WRITE))
                return -EBADF;
        …
        ret = rw_verify_area(WRITE, file, pos, count);
        …

        if (file->f_op->write)
                ret = file->f_op->write(file, buf, count, pos);
        …
        return ret;
}
```

write mode check

SELinux check

race window

Succeed in writing the privileged file!

Thread B

```
close(fd);close(fd2);
open(privileged_file_path, O_RDONLY);
```

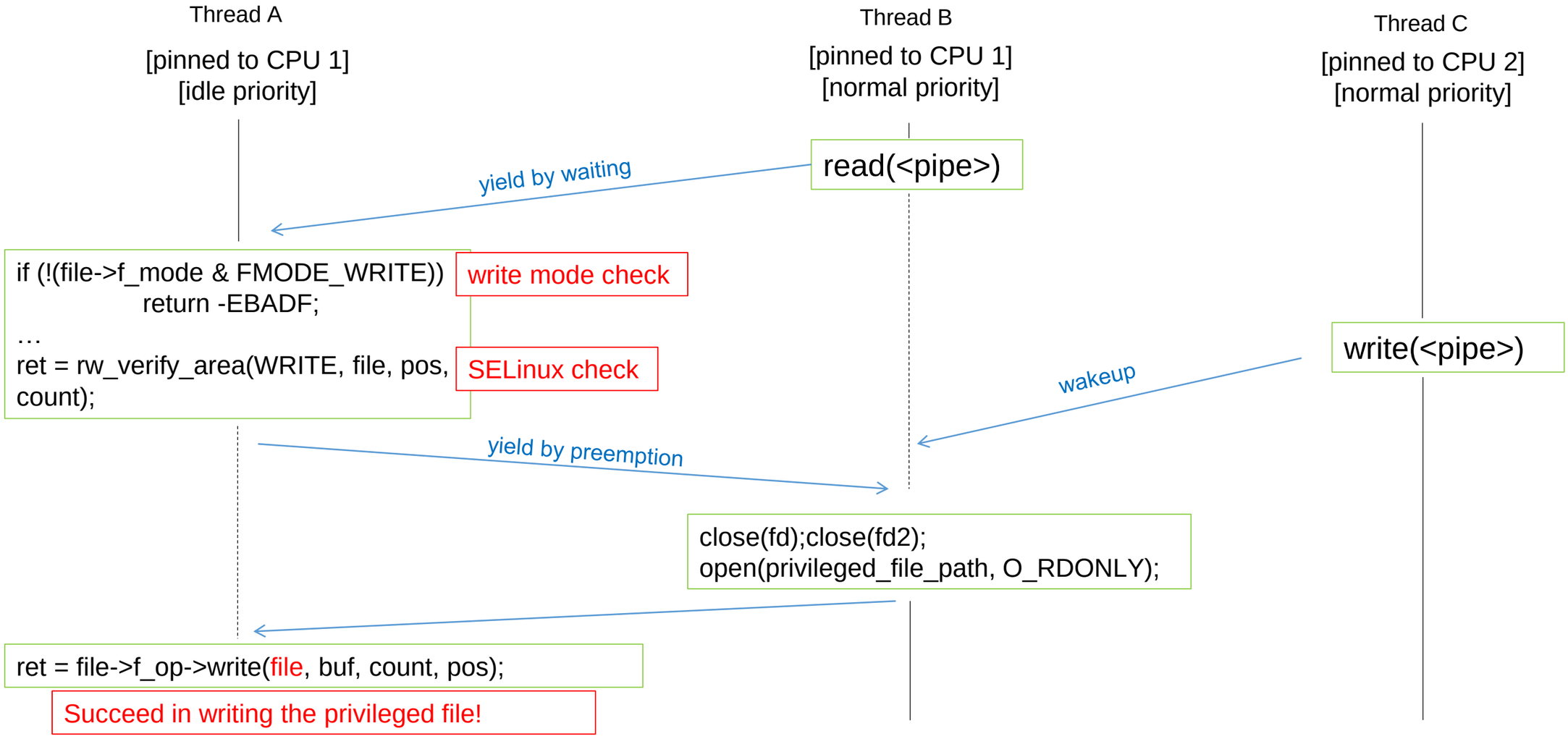
release the file object

reuse the file object

file object The privileged file

My exploit for CVE-2022-28350

Try to widen the race window with the method given by Jann Horn:



My exploit for CVE-2022-28350

The exploit will succeed in a big chance ☺ :

Tested on an affected Android 12 device

```
-rwxr-xr-x 1 system system   116800 2021-11-12 17:47 lib[redacted].so
-rwxr-xr-x 1 system system      496 2022-04-13 08:46 lib[redacted].so
-rwxr-xr-x 1 system system   965698 2021-11-12 17:47 lib[redacted].so
```

Attack from an untrusted app

```
-rwxr-xr-x 1 system system     1918 2022-04-13 09:06 lib[redacted].so
```

```
00000750: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000760: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000770: 0000 6465 7669 6c73 5f69 6e5f 6664       ..devils_in_fd
```

Supplement

- Exploit of CVE-2022-28350
- Small race windows can be exploitable!
  - UAF caused by race condition in fd export operation
  - Fd type confusion caused by race condition in fd import operation

UAF caused by race condition in fd export operation

A typical issue with a tiny race window:

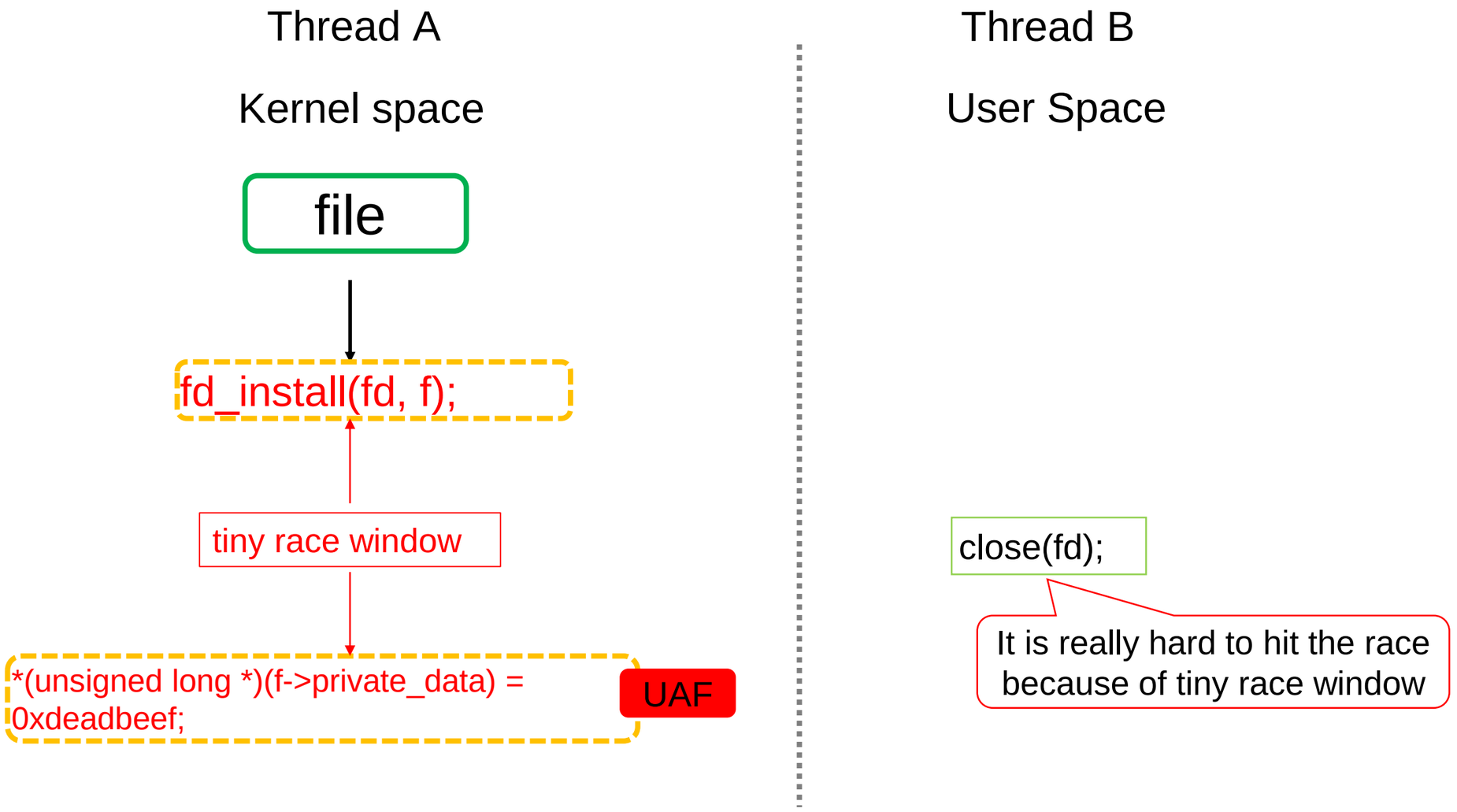
```
static long dev_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg)
{
        switch(cmd) {
                case UAF_TEST:
                {
                        int fd;
                struct file *f;
                void *cxt = kzalloc(128, GFP_KERNEL);
                …
                fd = get_unused_fd_flags(O_RDWR);
                …
                f = anon_inode_getfile("DEMO", &demo_fops, cxt,
O_RDWR);
                …
                fd_install(fd, f);
                *(unsigned long *)(f->private_data) = 0xdeadbeef;
                return put_user(fd, (int __user *)arg);
                }
…
```

Very tiny race windows!!!

```
static int demo_release(struct inode *nodp, struct file *filp)
{
        kfree(filp->private_data);
        return 0;
}

static const struct file_operations demo_fops = {
        .owner          = THIS_MODULE,
        .open           = demo_open,
        .release        = demo_release
};
```

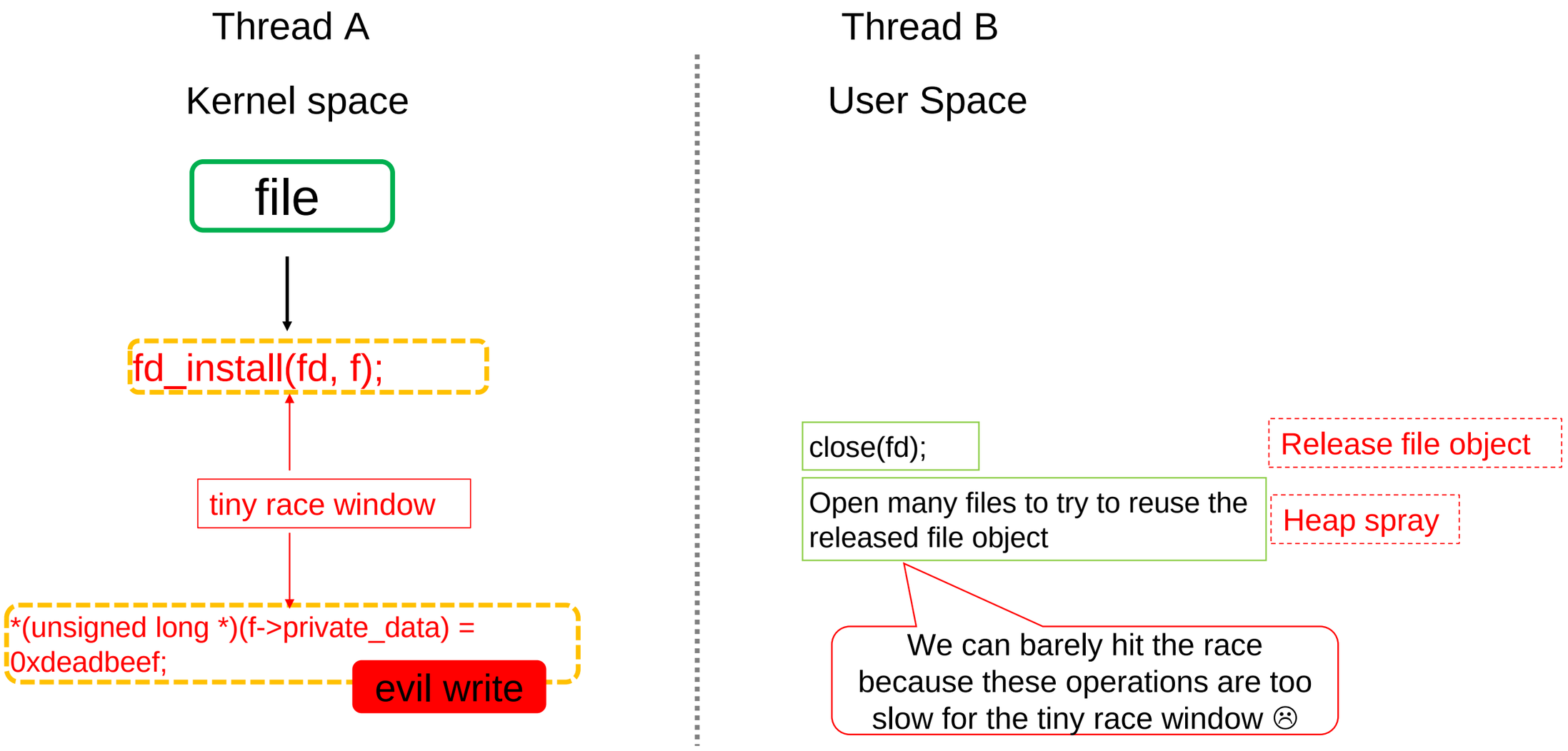
UAF caused by race condition in fd export operation

Try to trigger the UAF:



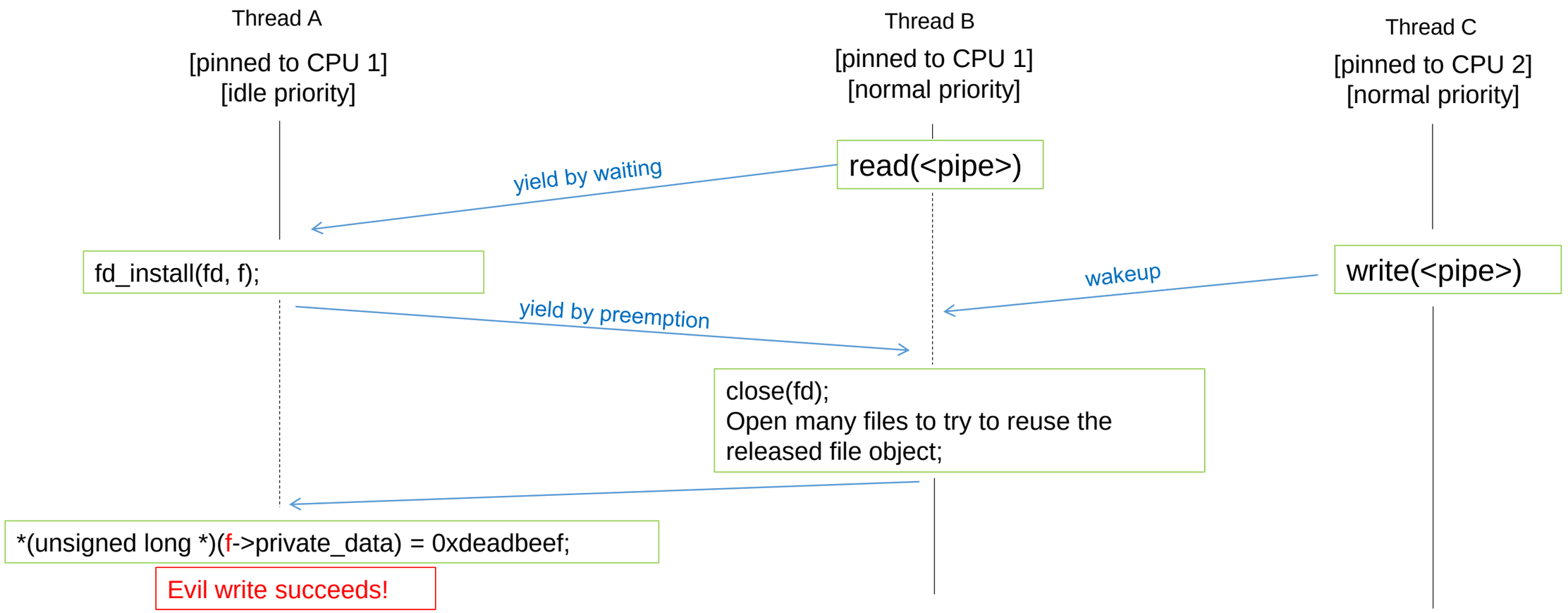
UAF caused by race condition in fd export operation

If we want to exploit the issue:



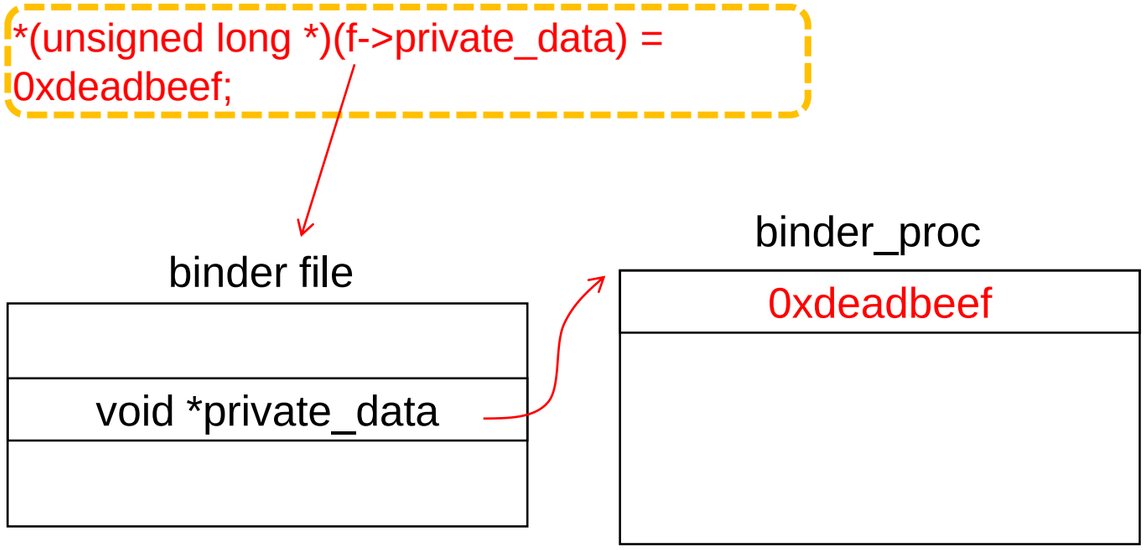
UAF caused by race condition in fd export operation

Try to widen the race window with the method given by Jann Horn:



UAF caused by race condition in fd export operation

We have a big chance to hit the race and turn the issue to a memory corruption:




```
[  T289] Unable to handle kernel paging request at virtual address 00000000deadbef7
[  T289] Mem abort info:
[  T289]   ESR = 0x96000045
[  T289]   EC = 0x25: DABT (current EL), IL = 32 bits
[  T289]   SET = 0, FnV = 0
[  T289]   EA = 0, S1PTW = 0
[  T289] Data abort info:
[  T289]   ISV = 0, ISS = 0x00000045
[  T289]   CM = 0, WnR = 1
[  T289] user pgtable: 4k pages, 39-bit VAs, pgdp=000000095c9d3000
[  T289] [00000000deadbef7] pgd=0000000000000000, p4d=0000000000000000, pud=0000000000000000
```

```
[  T289] Hardware name: Raven DVT (DT)
[  T289] Workqueue: events binder_deferred_func.cfi_jt
[  T289] pstate: 20c00005 (nzCv daif +PAN +UAO -TCO BTYPE=--)
[  T289] pc : binder_deferred_release+0x74/0x103c
[  T289] lr : binder_deferred_func+0x2bc/0x454
[  T289] sp : ffffffc014593c80
[  T289] x29: ffffffc014593cc0 x28: ffffff88e364b450
[  T289] x27: ffffffd27e88f230 x26: ffffff88e364b400
[  T289] x25: 0000000000000003 x24: ffffffd27e88f000
[  T289] x23: ffffff88e364b400 x22: ffffffd27e6e9000
[  T289] x21: ffffff8810b04a00 x20: ffffff8813254160
[  T289] x19: ffffff8810b04a00 x18: ffffffc012ffb028
[  T289] x17: 0000000000000002 x16: 000000000000000b
[  T289] x15: 0000000000000000 x14: 0000000000000018
[  T289] x13: ffffffd27bfb3a00 x12: 00000000000002d4c
[  T289] x11: 0000000080200000 x10: ffffffff236d9620
[  T289] x9 : ffffff88e6963400 x8 : 00000000deadbeef
[  T289] x7 : 0000000000000001
[  T289] x6 : 0000000000000000
[  T289] x5 : 0000000080200000 x4 : ffffffff236d9620
[  T289] x3 : 0000000080200000 x2 : ffffff8810b04a00
[  T289] x1 : 0000000000000000 x0 : 0000000000000000
[  T289] Call trace:
[  T289]  binder_deferred_release+0x74/0x103c
[  T289]  binder_deferred_func+0x2bc/0x454
[  T289]  process_one_work+0x248/0x820
[  T289]  worker_thread+0x438/0xbd8
[  T289]  kthread+0x150/0x200
[  T289]  ret_from_fork+0x10/0x30
```

Supplement

- Exploit of CVE-2022-28350
- Small race windows can be exploitable!
  - UAF caused by race condition in fd export operation
  - Fd type confusion caused by race condition in fd import operation

Fd type confusion caused by race condition in fd import operation

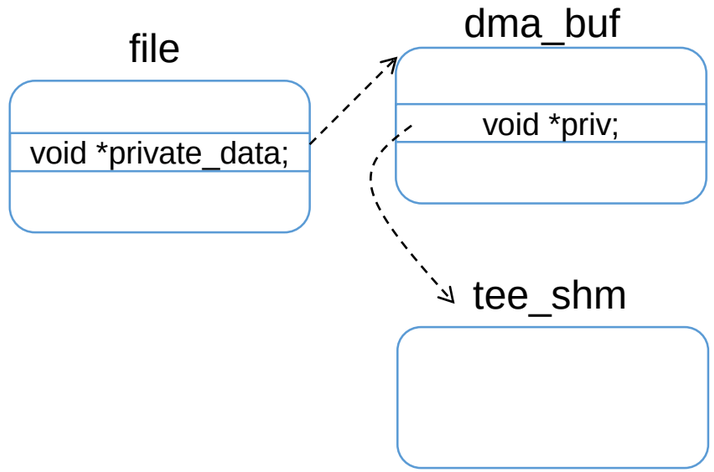
CVE-2022-21772

```
TEEC_Result TEEC_RegisterSharedMemory(struct TEEC_Context *ctx,
                                      struct TEEC_SharedMemory *shm)
{
        int fd;
        size_t s;
        struct dma_buf *dma_buf;
        struct tee_shm *tee_shm;
        …
        fd = teec_shm_alloc(ctx->fd, s, &shm->id);
        …
        dma_buf = dma_buf_get(fd);
        …
        tee_shm = dma_buf->priv;

        …
        shm->shadow_buffer = tee_shm->kaddr;
        …
        return TEEC_SUCCESS;
}
```

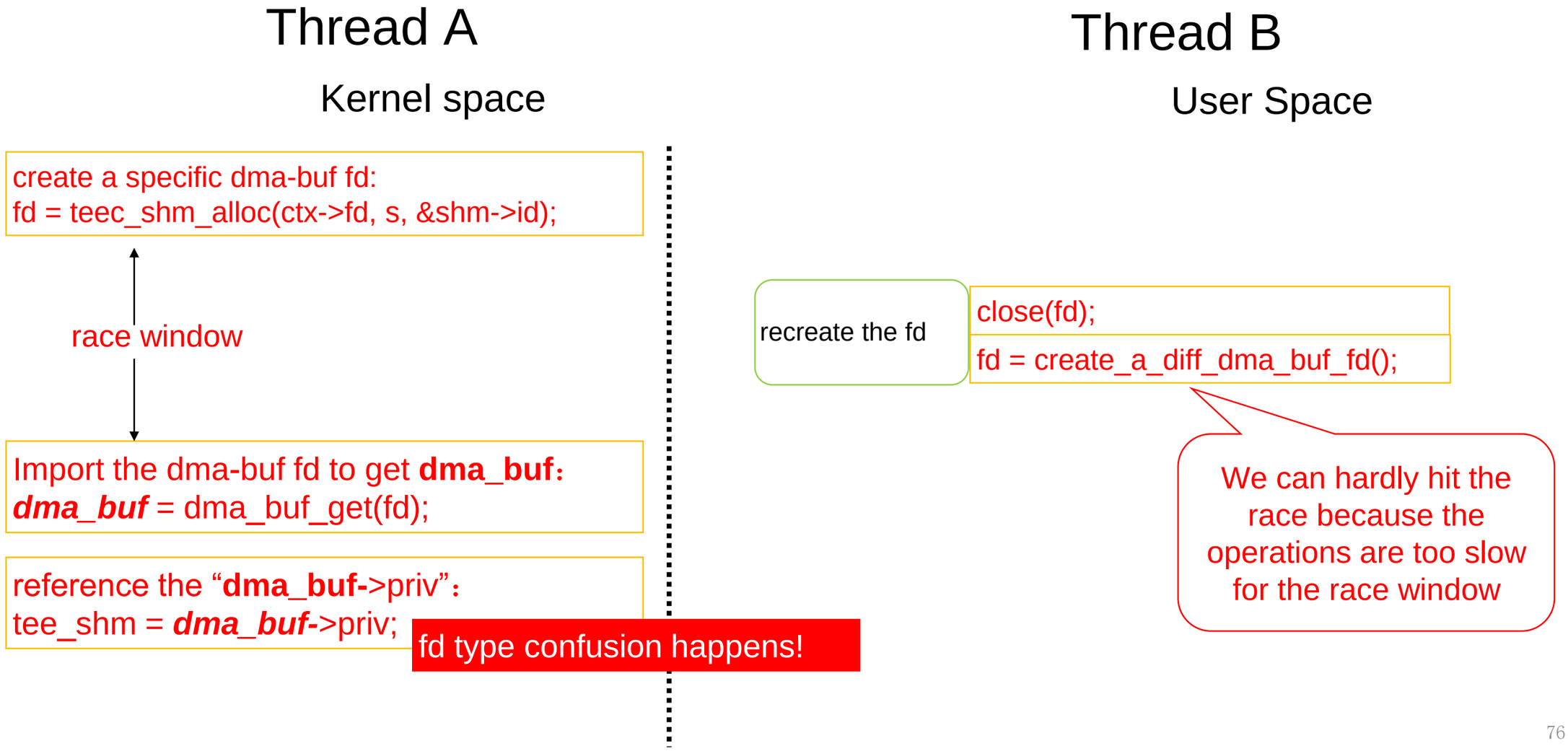
create a specific dma-buf fd

import the dma-buf fd to get dma_buf

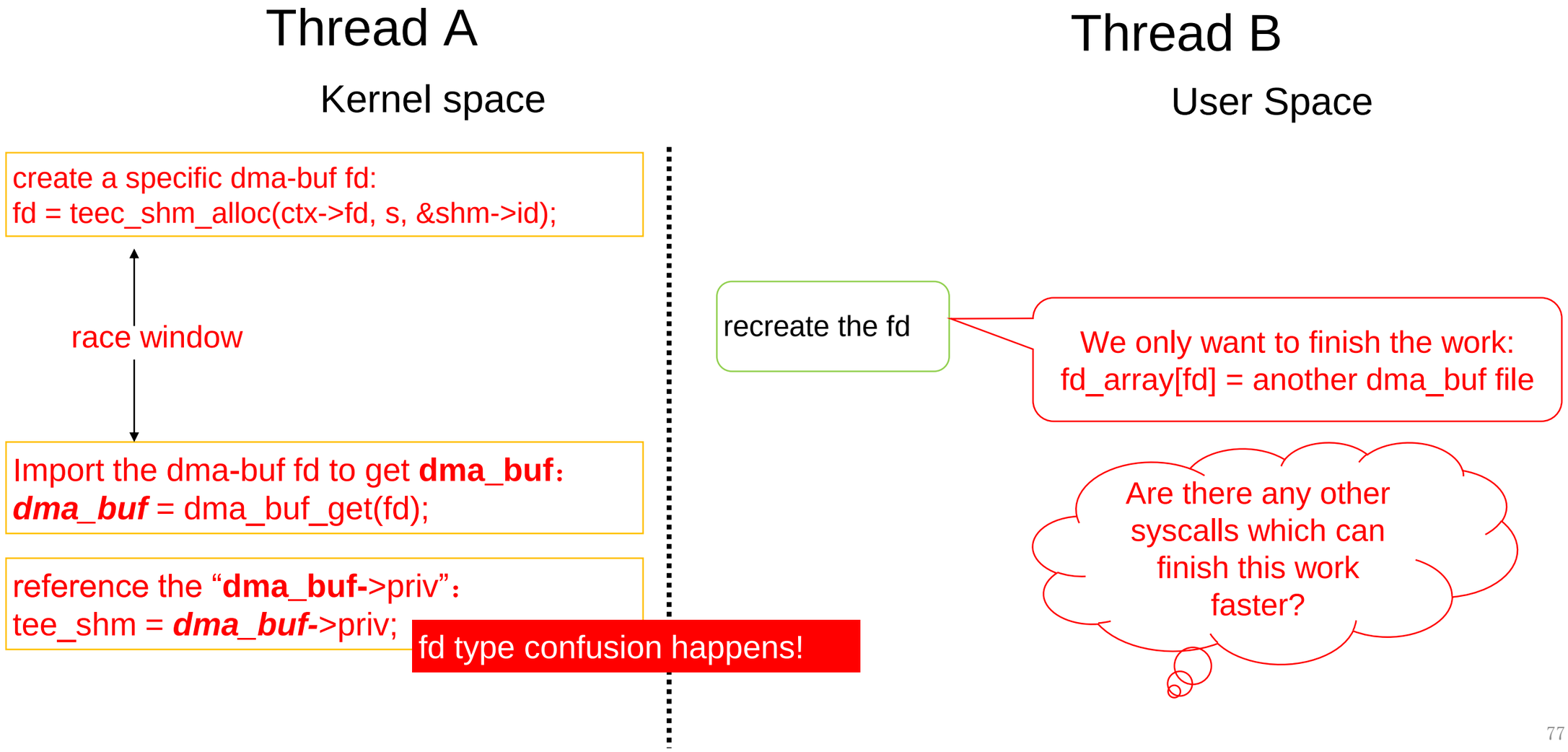
reference the “dma_buf->priv” as tee_shm



Fd type confusion caused by race condition in fd import operation



Fd type confusion caused by race condition in fd import operation



Fd type confusion caused by race condition in fd import operation

Syscall:dup2(int oldfd, int newfd)

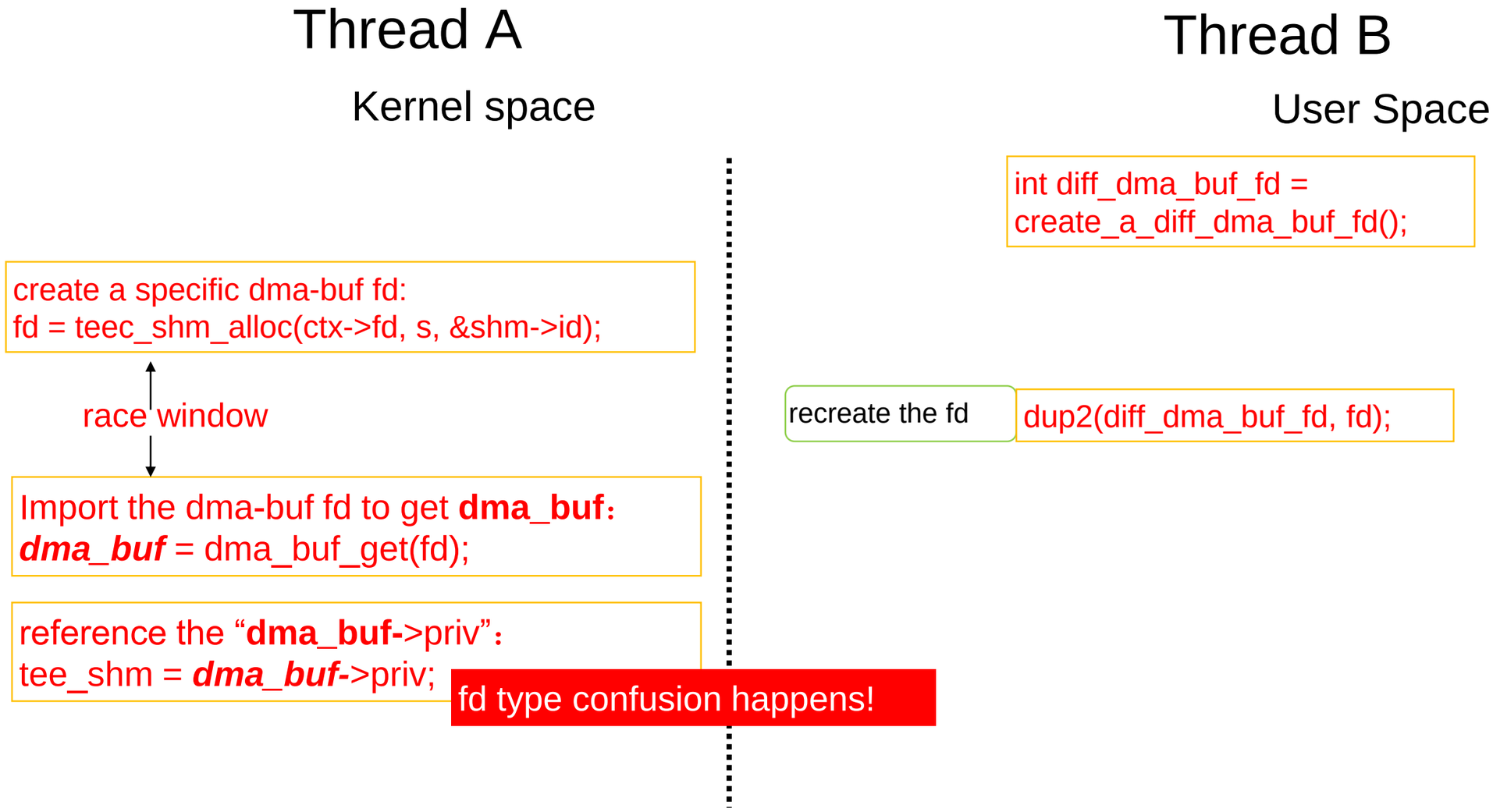
```
static int do_dup2(struct files_struct *files,
        struct file *file, unsigned fd, unsigned flags)
__releases(&files->file_lock)
{
        …
        rcu_assign_pointer(fdt->fd[fd], file);
        …
        if (tofree)
                filp_close(tofree, files);

        return fd;
…
}
```

fd_array[fd] = file

release the old file

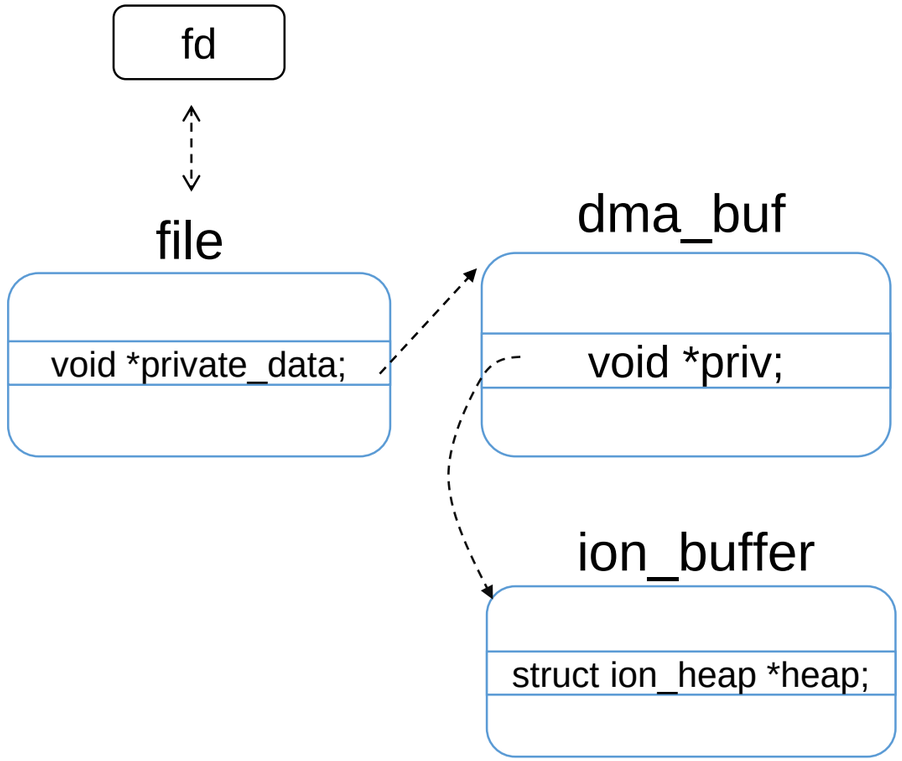
dup2() can finish the “fd_array[fd] = another dma_buf file” much faster !

Fd type confusion caused by race condition in fd import operation



Fd type confusion caused by race condition in fd import operation

We have a big chance to hit the race and turn the issue to a memory corruption:




```
void ion_buffer_destroy(struct ion_buffer *buffer)
{
        …
        buffer->heap->ops->free(buffer);
        vfree(buffer->pages);
        kfree(buffer);
}
```

memory corruption

```
-(0)[14974:poc][<ffffff87ab0832d0>] el1_da+0x24/0x3c
-(0)[14974:poc][<ffffff87abcadf38>] ion_buffer_destroy+0x120/0x154
-(0)[14974:poc][<ffffff87abcb13bc>] ion_dma_buf_release+0x6c/0xc4
-(0)[14974:poc][<ffffff87aba0703c>] dma_buf_release+0x58/0x158
-(0)[14974:poc][<ffffff87ab26c858>] __fput+0xbc/0x1b8
-(0)[14974:poc][<ffffff87ab26c748>] ____fput+0xc/0x14
-(0)[14974:poc][<ffffff87ab0d7208>] task_work_run+0x34/0xa8
-(0)[14974:poc][<ffffff87ab08afe0>] do_notify_resume+0x70/0x155c
-(0)[14974:poc]Exception stack(0xffffff802857bec0 to 0xffffff802857c000)
```

Thank you!