# Teknik Exploitasi Use After Free di Linux Kernel 7.0-rc7
## dengan Teknik slab_sheaf Union State Confusion

*by: Antonius*
Country: Indonesia
https://www.bluedragonsec.com – https://github.com/bluedragonsecurity

## 1. Latar Belakang

Linux kernel 7.0-rc7 menggunakan arsitektur SLUB Sheaves yang menggantikan struktur lama kmem_cache_cpu. Dalam arsitektur baru ini, setiap sheaf direpresentasikan oleh struct slab_sheaf yang dideklarasikan di mm/slub.c baris 404 di Linux kernel 7.0-rc7 .

Dari hasil analisa source code Linux kernel 7.0-rc7 , salah satu karakteristik desain yang paling menarik dari struct slab_sheaf adalah penggunaan union pada bagian awal struct. Union ini memungkinkan satu blok memori yang sama digunakan untuk tiga tujuan berbeda tergantung pada konteks. Teknik exploitasi Linux kernel use after free dengan slab_sheaf Union State Confusion berkaitan langsung dengan ambiguitas yang muncul dari penggunaan union ini.

## 2. Anatomi Union di slab_sheaf

### 2.1 Deklarasi Union (mm/slub.c:404)

Berikut adalah deklarasi lengkap bagian union dari struct slab_sheaf, diverifikasi langsung dari source tree linux-7.0-rc7 & linux-7.0 Mainline (keduanya berada pada baris yang sama) :

```
404  struct slab_sheaf {
405      union {
406          struct rcu_head rcu_head;
407          struct list_head barn_list;
408          /* only used for prefilled sheafs */
409          struct {
410              unsigned int capacity;
411              bool pfmemalloc;
412          };
413      };
414      struct kmem_cache *cache;
415      unsigned int size;
416      int node; /* only used for rcu_sheaf */
417      void *objects[];
418  };
```

**Sumber:** `linux-7.0/mm/slub.c:404` — *deklarasi struct slab_sheaf*

Union ini tidak memberi nama alias tambahan — ketiga varian langsung bisa diakses melalui nama field masing-masing (rcu_head, barn_list, capacity, pfmemalloc).

## 2.2 Layout Memori Aktual dari Union

```
┌──(robohax㉿robohax-20bws2ng00)-[~/Desktop/KERNEL/linux-7.0]
└─$ pahole -C slab_sheaf vmlinux
struct slab_sheaf {
        union {
                struct callback_head callback_head __attribute__((__aligned__(8))); /*     0    16 */
                struct list_head   barn_list;                /*     0    16 */
                struct {
                        unsigned int capacity;               /*     0     4 */
                        bool       pfmemalloc;               /*     4     1 */
                };                                           /*     0     8 */
        } __attribute__((__aligned__(8)));                   /*     0    16 */
        struct kmem_cache *        cache;                    /*    16     8 */
        unsigned int               size;                     /*    24     4 */
        int                        node;                     /*    28     4 */
        void *                     objects[];                /*    32     0 */

        /* size: 32, cachelines: 1, members: 5 */
        /* forced alignments: 1 */
        /* last cacheline: 32 bytes */
} __attribute__((__aligned__(8)));
```

```
┌──(robohax㉿robohax-20bws2ng00)-[~/Desktop/KERNEL/linux-7.0]
└─$ pahole -C list_head vmlinux
struct list_head {
        struct list_head *         next;                 /*     0     8 */
        struct list_head *         prev;                 /*     8     8 */

        /* size: 16, cachelines: 1, members: 2 */
        /* last cacheline: 16 bytes */
};

┌──(robohax㉿robohax-20bws2ng00)-[~/Desktop/KERNEL/linux-7.0]
└─$ 
```

Meskipun ketiga varian union dideklarasikan secara terpisah, dalam memori semuanya menempati rentang byte yang sama persis, dimulai dari offset +0x00 struct slab_sheaf.

| Offset | Ukuran | Nama Field (via rcu_head) | Nama Field (via barn_list) |
|---|---|---|---|
| `+0x00` | `8 byte` | `rcu_head.next` (pointer linkage internal RCU) | `barn_list.next` (pointer ke elemen berikutnya) |
| `+0x08` | `8 byte` | **`rcu_head.func` (pointer ke callback function)** | **`barn_list.prev` (pointer ke elemen sebelumnya)** |

Kolom pertama pada tabel di atas menunjukkan bahwa **`rcu_head.next`** dan **`barn_list.next`** berada di offset yang identik (+0x00). Demikian pula **`rcu_head.func`** dan **`barn_list.prev`** sama-sama di +0x08. Ini adalah konsekuensi langsung dari cara kerja union di C: semua anggota union berbagi alamat awal yang sama. Tidak ada pemisahan memori di antara ketiganya.

## 2.3 Definisi Tipe Masing-Masing Field

Memahami tipe masing-masing field penting untuk mengerti makna overlap tersebut. Berikut adalah definisi aktual dari kedua struct yang terlibat:

```
/* linux-7.0-rc7 /include/linux/types.h */
/* rcu_head adalah typedef dari struct callback_head */
struct callback_head {
    struct callback_head *next;                    /* +0x00: pointer linkage
```

```
internal RCU */
    void (*func)(struct callback_head *head);     /* +0x08: pointer ke
callback function */
} __attribute__((aligned(sizeof(void *))));
#define rcu_head callback_head

struct list_head {
      struct list_head *next, *prev;
};
```

**Sumber:** `linux-7.0-rc7 /include/linux/types.h` — *definisi callback_head / rcu_head*

Perbedaan paling kritikal ada di offset **+0x08**: dalam konteks `rcu_head`, byte di posisi ini adalah **pointer ke fungsi callback**. Dalam konteks `barn_list`, byte yang sama adalah **pointer ke elemen sebelumnya** dalam doubly-linked list barn. Satu blok memori 8 byte yang sama diinterpretasikan sebagai dua hal berbeda tergantung dari mana kode membacanya.

Demikian pula di offset **+0x00**: dalam konteks `rcu_head` ini adalah pointer linkage internal yang digunakan oleh infrastruktur RCU untuk menghubungkan callback yang pending. Dalam konteks `barn_list` ini adalah `next` pointer dari doubly-linked list.

## 3. State Machine Sebuah slab_sheaf

Agar tidak bingung tentang union ini, kita perlu terlebih dahulu paham bagaimana siklus hidup (lifecycle) sebuah slab_sheaf berlangsung di kernel 7.0-rc7 .

### 3.1 State-State yang Mungkin

Sebuah slab_sheaf dapat berada dalam salah satu state berikut pada waktu tertentu:

| State | Deskripsi | Union Digunakan Sebagai | Bagian Kode |
|---|---|---|---|
| `PERCPU_MAIN` | Menjadi main sheaf di `slub_percpu_sheaves` suatu CPU | Tidak digunakan | `alloc_from_pcs`, `free_to_pcs` |
| `PERCPU_SPARE` | Menjadi spare sheaf di `slub_percpu_sheaves` | Tidak digunakan | `__pcs_replace_*_main` |
| `PERCPU_RCU` | Menjadi `rcu_free` sheaf, batch kfree_rcu() | Tidak digunakan | `__kfree_rcu_sheaf` |
| `IN_BARN` | Berada dalam barn (pool NUMA), linked via `barn_list` | `barn_list (list_head)` | `barn_put_*, barn_get_*` |
| `RCU_PENDING` | Sudah di-`call_rcu()`, menunggu grace period RCU selesai | `rcu_head` (callback + linkage) | `call_rcu, rcu_free_sheaf` |

### 3.2 Transisi State dan Penggunaan Union

Yang penting adalah transisi antara state **IN_BARN** dan state **RCU_PENDING**. Saat sheaf dipindahkan dari barn ke jalur RCU:

1. `list_del(&sheaf->barn_list)` dipanggil saat sheaf diambil dari barn (mm/slub.c:3103, 3155, 3192). Setelah ini, `barn_list.next (+0x00)` dan `barn_list.prev (+0x08)` berisi nilai dari saat sheaf ada di doubly-linked list barn, atau nilai `LIST_POISON` yang di-set oleh `list_del()`.
2. Sheaf dimasukkan ke `pcs->rcu_free`: union masih berisi nilai lama dari `barn_list`, belum di-clear.
3. Saat sheaf penuh: `call_rcu(&sheaf->rcu_head, rcu_free_sheaf)` dipanggil (mm/slub.c:5949). Pada titik ini, RCU infrastructure akan menulis nilai barunya sendiri ke `rcu_head.next (+0x00)` berupa pointer linkage internal RCU, dan ke **rcu_head.func (+0x08)** berupa pointer ke fungsi `rcu_free_sheaf` — menimpa nilai lama `barn_list.next` dan `barn_list.prev`.

Arah sebaliknya juga berlaku. Dalam rcu_free_sheaf() (mm/slub.c:5789), setelah RCU grace period selesai, jika barn masih memiliki slot, sheaf dikembalikan ke barn:

```
/* linux-7.0-rc7 /mm/slub.c:5829 — dalam rcu_free_sheaf(), setelah grace
period */
```

```
5828         */
5829
5830        if (data_race(barn->nr_full) < MAX_FULL_SHEAVES) {
5831            stat(s, BARN_PUT);
5832            barn_put_full_sheaf(barn, sheaf);
5833            return;
5834        }
```

```
/* linux-7.0-rc7 /mm/slub.c:3130 — barn_put_full_sheaf() */
```

```
3130  static void barn_put_full_sheaf(struct node_barn *barn, struct slab_sheaf *sheaf)
3131  {
3132      unsigned long flags;
3133
3134      spin_lock_irqsave(&barn->lock, flags);
3135
3136      list_add(&sheaf->barn_list, &barn->sheaves_full);
3137      barn->nr_full++;
3138
3139      spin_unlock_irqrestore(&barn->lock, flags);
3140  }
```

**Sumber:** `linux-7.0-rc7 /mm/slub.c:5789-5835` — *rcu_free_sheaf() — callback setelah RCU grace period*

**Sumber:** `linux-7.0-rc7 /mm/slub.c:3130-3140` — *barn_put_full_sheaf() — list_add ke sheaves_full*

Ini berarti satu slab_sheaf dapat berulang kali melewati siklus: **barn → rcu_free sheaf → call_rcu → grace period → barn lagi**. Setiap perpindahan melibatkan perubahan interpretasi union dari `barn_list` ke `rcu_head` dan sebaliknya.

## 4. State yang Gantung! Inti dari Teknik slab_sheaf Union State Confusion

Teknik ini berpusat pada satu pertanyaan: apakah ada momen di mana kernel membaca atau menulis union slab_sheaf dengan interpretasi yang tidak konsisten dengan state aktual sheaf?

### 4.1 Tidak Ada State Flag

Struct `slab_sheaf` tidak memiliki field eksplisit yang merekam state mana yang sedang aktif. Tidak ada `enum state`, tidak ada flag `BARN_OR_RCU`. Kode kernel menentukan interpretasi yang benar hanya dari konteks aliran program — misalnya, jika sheaf baru saja keluar dari barn, maka `barn_list` yang valid; jika sheaf baru saja di-`call_rcu()`, maka `rcu_head` yang valid.

Ini adalah pendekatan yang umum di kernel (yang dibuat dari bahasa C) dan tidak salah secara desain selama state machine dijalankan dengan benar. Tapi ini menciptakan dependensi pada validitas urutan operasi.

### 4.2 Window Antara call_rcu dan Penggunaan barn_list

Perhatikan alur berikut dalam `__kfree_rcu_sheaf()` (mm/slub.c:5862). Kita bisa melihat secara langsung dari source kernel bagaimana window terbentuk:

```
/* linux-7.0-rc7 /mm/slub.c:5862 — __kfree_rcu_sheaf() */
bool __kfree_rcu_sheaf(struct kmem_cache *s, void *obj)
{
    struct slub_percpu_sheaves *pcs;
    struct slab_sheaf *rcu_sheaf;

    /* ... local_trylock, get pcs ... */

    /* Langkah 1: rcu_free sheaf diambil dari spare atau barn */
    /* Pada titik ini union berisi nilai valid sebagai barn_list  */
    /* (jika diambil dari barn) atau nilai random (dari spare)    */

    /* linux-7.0-rc7 /mm/slub.c:5935 */
do_free:
    rcu_sheaf = pcs->rcu_free;

    /* Langkah 2: objek dikumpulkan ke objects[] */
    /* linux-7.0-rc7 /mm/slub.c:5935 */
    rcu_sheaf->objects[rcu_sheaf->size++] = obj;

    /* Langkah 3: saat sheaf penuh, call_rcu dipanggil */
    if (likely(rcu_sheaf->size < s->sheaf_capacity)) {
        rcu_sheaf = NULL;
    } else {
        pcs->rcu_free = NULL;
        rcu_sheaf->node = numa_mem_id();
    }

    /* linux-7.0-rc7 /mm/slub.c:5949 */
    if (rcu_sheaf)
        call_rcu(&rcu_sheaf->rcu_head, rcu_free_sheaf);
    /* call_rcu menulis:
     *   rcu_head.next (+0x00) = RCU linkage internal
     *   rcu_head.func (+0x08) = &rcu_free_sheaf
     * Menimpa barn_list.next (+0x00) dan barn_list.prev (+0x08) */

    local_unlock(&s->cpu_sheaves->lock);
    return true;
}
```

**Sumber:** `linux-7.0-rc7 /mm/slub.c:5862–5955` — *__kfree_rcu_sheaf() — jalur free via RCU*

**Sumber:** `linux-7.0-rc7 /mm/slub.c:5949` — *call_rcu(&rcu_sheaf->rcu_head, rcu_free_sheaf)*

Antara Langkah 1 dan Langkah 3, ada jendela di mana union berisi nilai lama yang mungkin berasal dari konteks barn. Jika pada jendela ini ada kode lain yang membaca union dengan interpretasi `rcu_head`, nilai yang dibaca adalah `barn_list.next` (di +0x00) dan `barn_list.prev` (di +0x08) — bukan nilai `rcu_head` yang valid.

### 4.3 Skenario Re-entry ke Barn Setelah RCU

Skenario yang lebih menarik secara teoritis terjadi dalam `rcu_free_sheaf()` (mm/slub.c:5789). Setelah RCU grace period selesai dan callback dipanggil:

```
/* linux-7.0-rc7 /mm/slub.c:5789 — rcu_free_sheaf() */
static void rcu_free_sheaf(struct rcu_head *head)
{
    struct slab_sheaf *sheaf;
    struct node_barn  *barn = NULL;
```

```
    struct kmem_cache *s;

    /* container_of mendapatkan sheaf dari pointer rcu_head */
    sheaf = container_of(head, struct slab_sheaf, rcu_head);
    /* union sekarang berisi nilai rcu_head (dari RCU infrastructure):
     *   rcu_head.next (+0x00) = RCU linkage
     *   rcu_head.func (+0x08) = &rcu_free_sheaf  */

    s = sheaf->cache;

    if (__rcu_free_sheaf_prepare(s, sheaf))
        goto flush;

    n    = get_node(s, sheaf->node);
    barn = n->barn;

    if (unlikely(sheaf->size == 0))
        goto empty;

    /* KONDISI: jika barn masih ada slot */
    /* linux-7.0-rc7 /mm/slub.c:5829 */
    if (data_race(barn->nr_full) < MAX_FULL_SHEAVES) {
        stat(s, BARN_PUT);
        barn_put_full_sheaf(barn, sheaf);
        /* barn_put_full_sheaf memanggil list_add(&sheaf->barn_list, ...) */
        /* list_add MENULIS KE:
         *   barn_list.next (+0x00) = &barn->sheaves_full (timpa
rcu_head.next)
         *   barn_list.prev (+0x08) = &barn->sheaves_full (timpa
rcu_head.func)
         */
        return;
    }
flush:
    sheaf_flush_unused(s, sheaf);
    /* ... */
}
```

**Sumber:** `linux-7.0-rc7 /mm/slub.c:5789–5842` — *rcu_free_sheaf() — callback RCU + siklus kembali ke barn*

Saat `barn_put_full_sheaf()` memanggil `list_add(&sheaf->barn_list, ...)`, operasi ini menulis ke `barn_list.next (+0x00)` dan `barn_list.prev (+0x08)`. Nilai yang ditulis adalah pointer ke `list_head` di dalam barn. Karena `barn_list` dan `rcu_head` berbagi offset yang sama, efek visualnya adalah nilai **rcu_head.func (+0x08)** tertimpa oleh `barn_list.prev`, dan nilai **rcu_head.next (+0x00)** tertimpa oleh `barn_list.next`.

### 4.4 Jadi Apa Artinya ?

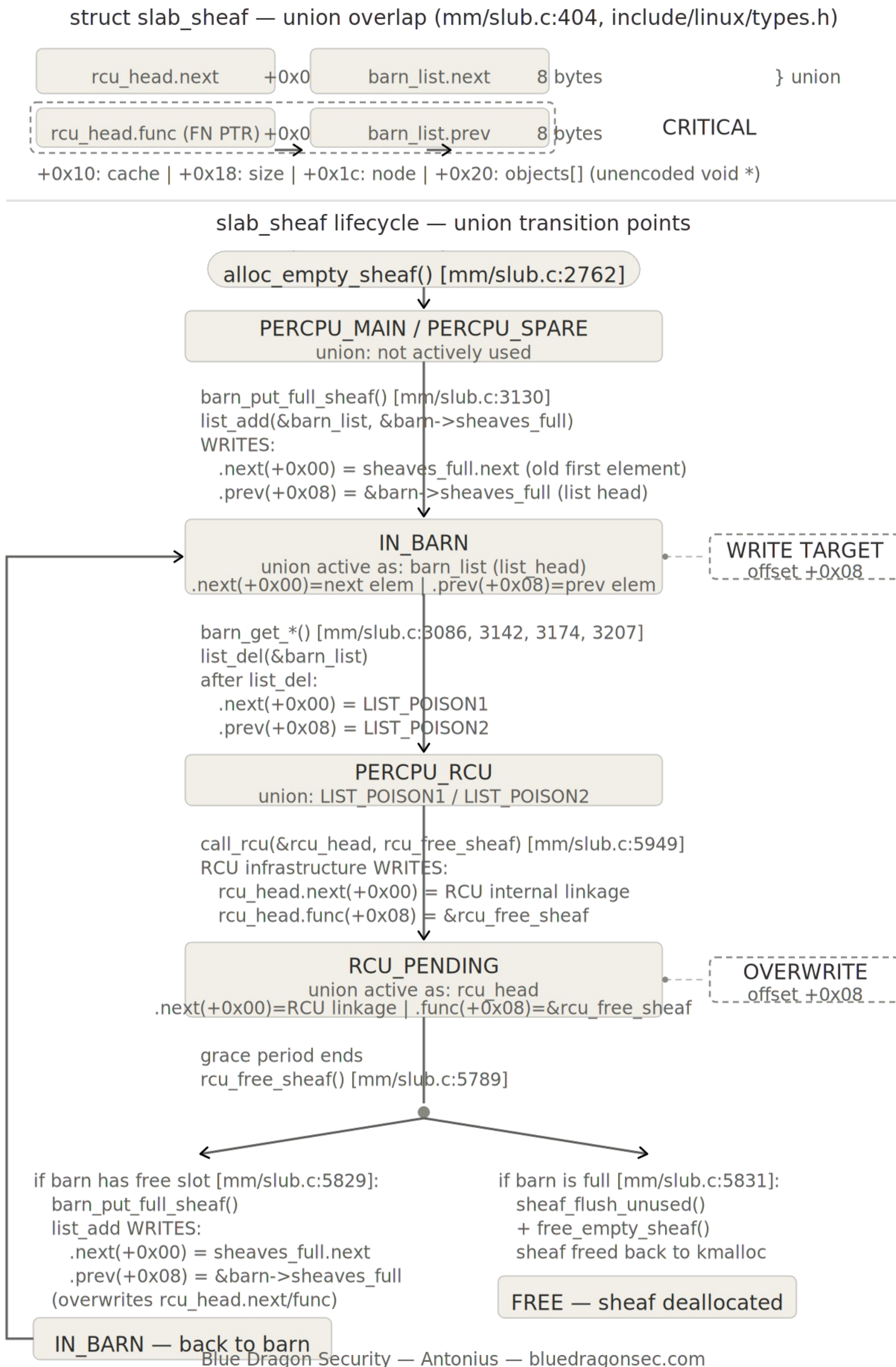
Secara teori, ambiguitas ini menjadi perhatian dalam konteks berikut:

- Jika ada mekanisme yang memungkinkan pembacaan `slab_sheaf` pada saat yang salah — misalnya melalui kerentanan information leak atau race condition yang memberikan jendela observasi ke isi sheaf — pembaca tersebut mungkin menginterpretasikan nilai union secara berbeda dari yang dimaksud kernel.
- Jika ada primitif tulis yang dapat memodifikasi union `slab_sheaf` saat sheaf dalam state **RCU_PENDING**, nilai yang ditulis ke **offset +0x08** akan diinterpretasikan oleh RCU infrastructure sebagai **pointer ke callback function**. Ini karena dalam konteks `rcu_head`, offset +0x08 adalah `rcu_head.func`.
- Secara simetris: modifikasi di state **IN_BARN** ke **offset +0x08** akan diinterpretasikan oleh kode barn sebagai `barn_list.prev`, yang merupakan pointer ke elemen sebelumnya dalam doubly-linked list. Ketika sheaf kemudian dipindahkan ke RCU path, nilai `barn_list.prev` yang telah dimodifikasi tersebut akan terbaca sebagai `rcu_head.func` oleh RCU infrastructure.

Perbedaan offset yang kritikal ini perlu dipahami dengan benar. Modifikasi di **offset +0x08** adalah titik paling menarik untuk eksploitasi karena menyentuh `rcu_head.func` yang merupakan function pointer yang dipanggil oleh mekanisme RCU setelah grace period. Modifikasi di **offset +0x00** menyentuh `rcu_head.next` yang merupakan pointer linkage internal RCU — potensinya lebih ke arah info-leak atau korrupsi struktur RCU.

## 5. Siklus Hidup Lengkap dan Titik Transisi

Untuk memahami teknik exploitasi ini secara menyeluruh, berikut adalah diagram siklus hidup slab_sheaf dengan titik-titik transisi union yang relevan:



# 6. Kondisi yang Diperlukan

Teknik ini bukan kerentanan yang dapat langsung dieksploitasi dari scratch. Ini adalah properti desain yang memperbesar dampak jika sudah ada vulner lain.

**Kondisi yang diperlukan:**

- **Primitif akses awal:** harus ada kerentanan lain — seperti UAF, buffer overflow, atau race condition yang memberikan kemampuan membaca atau menulis ke memori yang overlap dengan sebuah `slab_sheaf`.
- **Timing yang tepat:** akses tersebut harus terjadi saat sheaf berada dalam transisi antara state IN_BARN dan RCU_PENDING, atau saat `rcu_free_sheaf()` sedang mengembalikan sheaf ke barn. Secara lebih spesifik:
    a. Untuk poison via state IN_BARN: tulis ke offset +0x08 (barn_list.prev) saat sheaf masih di barn. Nilai ini akan dibawa ke rcu_head.func ketika sheaf memasuki RCU path berikutnya.
    b. Untuk overwrite langsung: tulis ke offset +0x08 (rcu_head.func) setelah call_rcu() menuliskan rcu_free_sheaf di sana, namun sebelum RCU grace period selesai dan callback dipanggil. Grace period biasanya berlangsung ratusan microseconds hingga beberapa milliseconds — window yang realistis.
- **Pengetahuan lokasi sheaf:** penyerang harus mengetahui alamat `slab_sheaf` target di memori. Ini umumnya membutuhkan information leak tersendiri.

Dalam istilah lain: teknik ini adalah amplifier, bukan initial vector. Ia mengubah primitif akses memori terbatas menjadi sesuatu yang berpotensi lebih signifikan karena nilai yang dimanipulasi punya makna khusus bagi infrastruktur kernel (RCU machinery atau barn list management).

# 7. Mitigasi & Pertimbangan Defensif

## 7.1 Pendekatan Yang Sudah Ada ?

Kernel 7.0-rc7 memiliki beberapa lapisan perlindungan yang membatasi dampak praktis dari ambiguitas ini:

- **KASAN (Kernel Address Sanitizer):** pada build debug, KASAN mendeteksi akses ke memori yang belum diinisialisasi atau sudah di-free. Ini membantu mendeteksi kondisi transisi yang salah saat pengujian.
- **lockdep:** framework deteksi penggunaan lock yang salah. Beberapa transisi state melibatkan lock yang dilindungi lockdep, sehingga akses di luar konteks lock yang benar akan terdeteksi.
- **Locking discipline:** setiap barn operation dilindungi `spin_lock_irqsave` (mm/slub.c:3132, 3157), dan setiap percpu operation dilindungi `local_trylock` (mm/slub.c:4561 dst). Ini membatasi window di mana state transisi dapat dieksploitasi.
- **kCFI / FineIBT:** kernel 7.0-rc7 menggunakan Control Flow Integrity untuk memvalidasi indirect function calls. Karena `rcu_head.func (+0x08)` adalah target indirect call oleh RCU infrastructure, overwrite dengan sembarang alamat tidak langsung menghasilkan eksekusi — nilai yang dituliskan harus punya CFI hash yang valid.

## 7.2 Yang Belum Ada ?

Mitigasi yang ada tidak secara eksplisit menandai state `slab_sheaf` atau memvalidasi interpretasi union. Pendekatan yang lebih kuat secara teoritis adalah:

- Menambahkan state field eksplisit ke struct `slab_sheaf` (enum atau bitfield) yang diupdate atomis saat transisi. Ini memungkinkan validasi runtime bahwa union diakses dengan interpretasi yang benar.
- Memisahkan union menjadi field terpisah dengan masa hidup (lifetime) yang jelas dan tidak overlap. Meski ini meningkatkan ukuran struct, ia menghilangkan ambiguitas sepenuhnya.

# 8. Kesimpulan

Teknik **slab_sheaf Union State Confusion** berpusat pada satu fakta arsitektur dari kernel 7.0-rc7 : struct `slab_sheaf` menggunakan union untuk menumpuk `rcu_head` dan `barn_list` di offset memori yang sama, tanpa penanda state eksplisit yang mencegah akses dengan interpretasi yang salah.

Titik kritis yang paling menarik secara eksploitasi adalah **offset +0x08**, di mana `barn_list.prev` dan `rcu_head.func` saling menimpa tergantung konteks. Dalam konteks RCU, offset ini adalah function pointer yang dipanggil oleh infrastruktur RCU setelah grace period selesai (`mm/slub.c:5949`). Dalam konteks barn, offset yang sama adalah pointer ke elemen sebelumnya dalam doubly-linked list yang dikelola oleh `barn_put_full_sheaf()` (mm/slub.c:3130) dan `list_del()` (mm/slub.c:3103 dst).

Secara teoritis, ini menciptakan kondisi di mana nilai yang bermakna dalam satu konteks — pointer callback RCU di **+0x08** atau pointer doubly-linked list barn — dapat dibaca atau ditulis melalui konteks yang berbeda jika ada bug lain yang memungkinkan akses ke isi `slab_sheaf`.

Teknik ini bisa diadaptasi untuk linux kernel 7.0 mainline !

**Intinya sederhana !**

Di kernel 7, struct `slab_sheaf` menggunakan **union,** artinya **satu blok memori 8 byte yang sama** dipakai untuk dua hal yang berbeda tergantung situasi. Khususnya di offset +0x08:

Saat sheaf ada di **barn** (pool penyimpanan), kernel membaca 8 byte itu sebagai `barn_list.prev` — cuma pointer biasa ke elemen sebelumnya di linked list. Tidak berbahaya.

Saat sheaf masuk jalur **RCU** (mekanisme free tertunda), kernel membaca **8 byte yang sama persis** sebagai `rcu_head.func` — yaitu **alamat fungsi yang akan dipanggil kernel**. Ini sangat berbahaya karena kernel akan melompat ke alamat apapun yang tertulis di sana.

Yang kritis: **tidak ada penanda** di dalam struct yang memberitahu kernel "ini sedang dipakai sebagai barn_list" atau "ini sedang dipakai sebagai rcu_head". Kernel hanya mengandalkan urutan kode yang benar.

Jadi kalau attacker punya bug UAF yang memungkinkan menulis ke memori sebuah `slab_sheaf`, dia bisa:

**Tulis nilai terkontrol ke offset +0x08 saat sheaf di barn** → kernel mengira itu cuma `barn_list.prev` (pointer list biasa) → sheaf pindah ke jalur RCU → kernel membaca 8 byte yang sama sebagai `rcu_head.func` → **kernel memanggil alamat yang ditulis attacker** → eksekusi kode di kernel context.

Satu tulisan, dua interpretasi. Attacker menulis sebagai data biasa, kernel membacanya sebagai alamat fungsi lalu melompat ke sana. Itulah "state confusion", kebingungan antara dua konteks yang berbagi memori yang sama !