Linux Kernel Use-After-Free Exploitation in Linux Kernel 7 Series Using slab_sheaf Union State Confusion

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

Forewords

It's been a long time since I last did Linux kernel exploit research; the last time was back in the Linux 2.6 era, right before I shifted my focus to kernel rootkits. However, in 2026, a fascinating development in Linux 7.0 brought me back to the field. It revolves around a fundamental shift in the kernel heap exploitation paradigm - moving toward a mindset of 'Hijacking the Allocator State.
I have conducted deep research into Linux 7.0-rc1 and even reverse-engineered the mechanics of these SLUB sheaves. This writring is based on Linux 7.0-rc7 source code, but can be applied to any Linux 7.* series.

Background

Linux kernel 7.0-rc7 & linux kernel 7.0 mainline uses the SLUB Sheaves architecture, replacing the legacy kmem_cache_cpu structure. In this new architecture, each sheaf is represented by struct slab_sheaf, declared at mm/slub.c line 404 in Linux kernel 7.0-rc7. From source code analysis of Linux kernel 7.0-rc7, one of the most interesting design characteristics of struct slab_sheaf is the use of a union at the beginning of the struct. This union allows the same block of memory to be used for three different purposes depending on context. The Linux kernel use-after-free exploitation technique using slab_sheaf Union State Confusion is directly related to the ambiguity that arises from the use of this union.

Anatomy of the Union in slab_sheaf

Union Declaration (mm/slub.c:404)

Below is the complete declaration of the union portion of struct slab_sheaf, verified directly from the linux-7.0-rc7 & linux-7.0 Mainline source tree (both at the same line number):

```
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  };
```

This union does not add alias names. All three variants are directly accessible through their respective field names (rcu_head, barn_list, capacity, pfmemalloc).

Actual Memory Layout of the 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 callback_head vmlinux
struct callback_head {
        struct callback_head *     next;                 /*     0     8 */
        void                       (*func)(struct callback_head *); /*     8     8 */

        /* size: 16, cachelines: 1, members: 2 */
        /* last cacheline: 16 bytes */
} __attribute__((__aligned__(8)));


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

```
┌──(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 */
};
```

Although the three union variants are declared separately, in memory they all occupy the exact same byte range, starting from offset +0x00 of struct slab_sheaf.

Offset	Size	Field Name (via rcu_head)	Field Name (via barn_list)
+0x00	8 bytes	rcu_head.next (RCU internal linkage pointer)	barn_list.next (pointer to next element)
+0x08	8 bytes	rcu_head.func (pointer to callback function)	barn_list.prev (pointer to previous element)

The first column shows that rcu_head.next and barn_list.next are at an identical offset (+0x00).

Likewise rcu_head.func and barn_list.prev are both at +0x08. This is a direct consequence of how unions work in C: all union members share the same starting address.

This is a direct consequence of how unions work in C: all members of a union share the same starting address. There is no memory separation between them.

Type Definitions of Each Field

Understanding the type of each field is important for grasping the meaning of this overlap.

Below are the actual definitions of the two structs involved:

```
/* linux-7.0-rc7 /include/linux/types.h */
/* rcu_head is a typedef of struct callback_head */
struct callback_head {
    struct callback_head *next;             /* +0x00: RCU internal linkage pointer */
    void (*func)(struct callback_head *head);  /* +0x08: pointer to callback function */
} __attribute__((aligned(sizeof(void *))));
#define rcu_head callback_head

/* linux-7.0-rc7 /include/linux/types.h */
struct list_head {
    struct list_head *next;    /* +0x00: pointer to next element */
    struct list_head *prev;    /* +0x08: pointer to previous element */
};
```

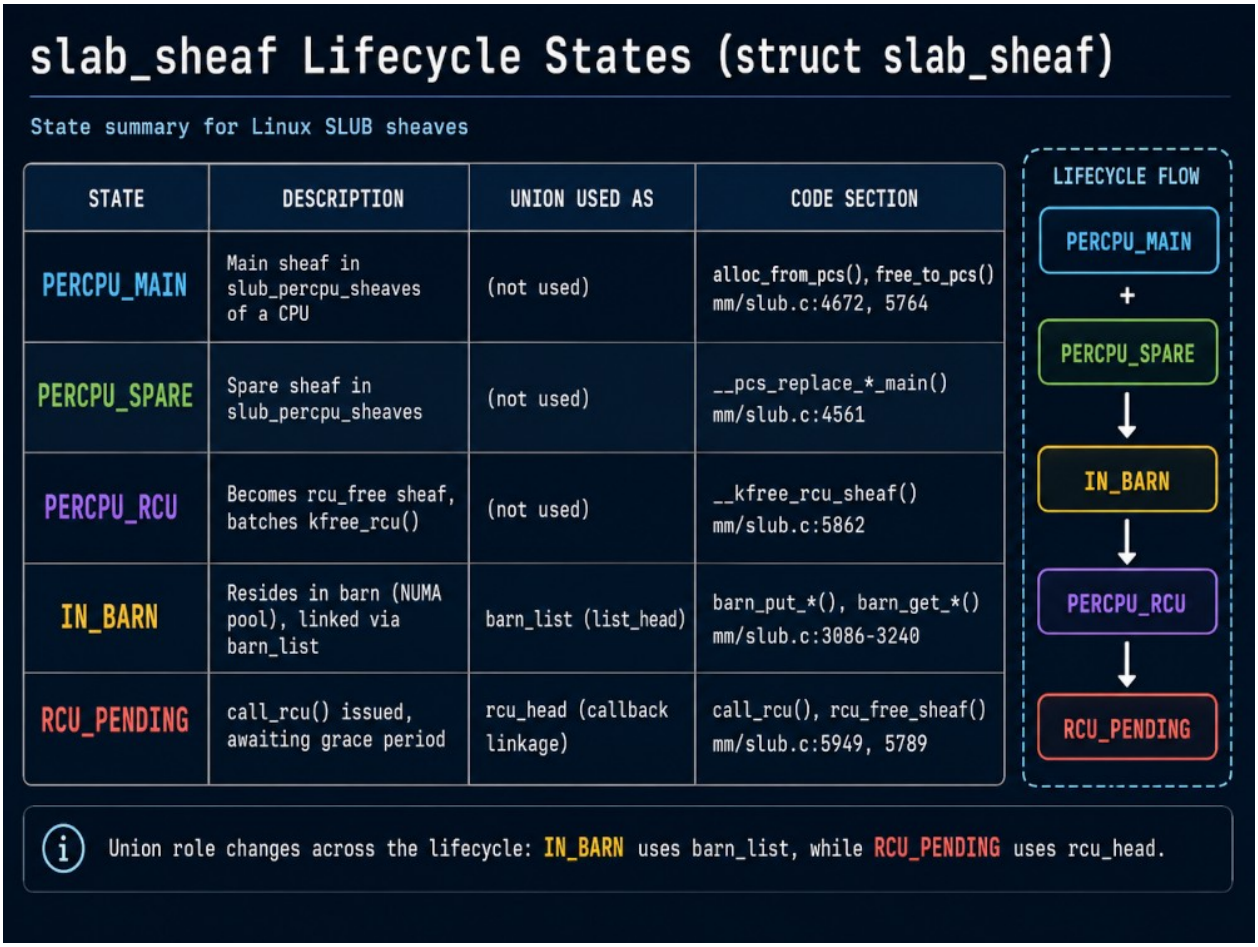
The most critical difference is at offset +0x08: in the rcu_head context, the byte at this position is a pointer to a callback function. In the barn_list context, the same byte is a pointer to the previous element in the barn's doubly-linked list. A single 8-byte block of memory is interpreted
as two different things depending on which code path reads it.

State Machine of a slab_sheaf

To avoid confusion about this union, we first need to understand the life cycle of a slab_sheaf in kernel 7.0-rc7.

Possible States

At any given time, a slab_sheaf may be in one of the following states:



State Transitions and Union Use

The critical transitions are between the IN_BARN and RCU_PENDING states. When a sheaf is moved from the barn to the RCU path:

1. list_del(&sheaf->barn_list) is called when the sheaf is removed from the barn (mm/slub.c:3103, 3155, 3192). After this, barn_list.next (+0x00) and barn_list.prev (+0x08) contain LIST_POISON1 and LIST_POISON2 respectively, set unconditionally by list_del().
2. The sheaf is placed into pcs->rcu_free: the union still contains the LIST_POISON values from list_del().
3. When the sheaf is full: call_rcu(&sheaf->rcu_head, rcu_free_sheaf) is called (mm/slub.c:5949). The RCU infrastructure writes rcu_head.next (+0x00) as RCU internal linkage, and rcu_head.func (+0x08) as &rcu_free_sheaf - overwriting the previous values.

The reverse direction also applies. In rcu_free_sheaf() (mm/slub.c:5789), after the RCU grace period completes, if the barn has free slots, the sheaf is returned to the barn:

```
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       }
```

```
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  }
```

The Hanging State: The Core of the slab_sheaf Union State Confusion Technique

This technique centers on one question: is there a moment where the kernel reads or writes the slab_sheaf union with an interpretation inconsistent with the sheaf's actual state?

There Is No State Flag

struct slab_sheaf has no explicit field recording which state is currently active. There is no enum state, no BARN_OR_RCU flag. The kernel determines the correct interpretation solely from program flow context. This is common in C based kernel code and is not inherently flawed, but it
creates a dependency on the validity of operation ordering.

The Window Between call_rcu and barn_list Use

Consider the following flow in __kfree_rcu_sheaf() (mm/slub.c around line 5862):

```
/* linux-7.0-rc7 /mm/slub.c:5862 — __kfree_rcu_sheaf() */
bool __kfree_rcu_sheaf(struct kmem_cache *s, void *obj)
{
   /* Step 1: rcu_free sheaf obtained from spare or barn */
do_free:
   rcu_sheaf = pcs->rcu_free;

   /* Step 2: objects are collected into objects[] */
   rcu_sheaf->objects[rcu_sheaf->size++] = obj;
```

```
    /* Step 3: when sheaf is full, call_rcu is invoked */
    if (likely(rcu_sheaf->size < s->sheaf_capacity)) {
        rcu_sheaf = NULL;
    } else {
        pcs->rcu_free = NULL;
        rcu_sheaf->node = numa_mem_id();
    }
    if (rcu_sheaf)
        call_rcu(&rcu_sheaf->rcu_head, rcu_free_sheaf);
    /* call_rcu writes:
     * rcu_head.next (+0x00) = RCU internal linkage
     * rcu_head.func (+0x08) = &rcu_free_sheaf */
}
```

Re-entry to Barn After RCU Scenario

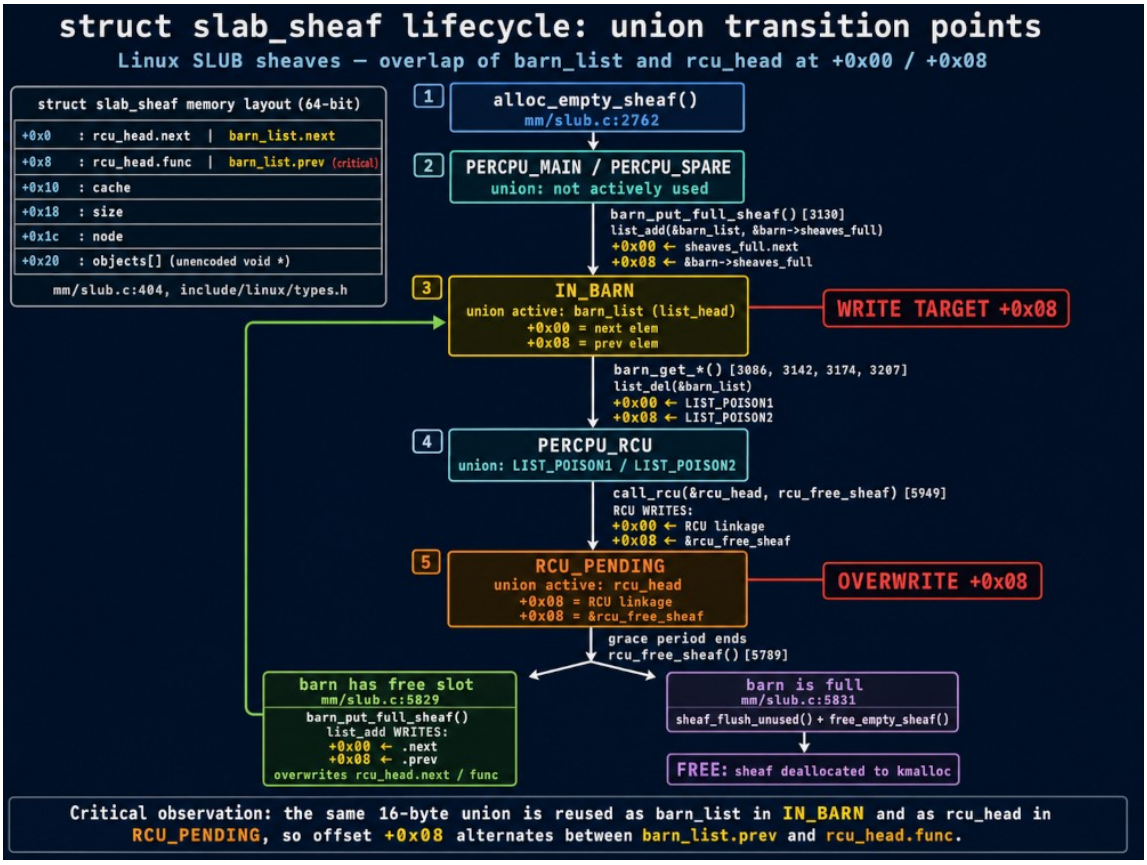
After the RCU grace period completes and the callback is invoked in rcu_free_sheaf() (mm/slub.c:5789), if the barn has free slots, barn_put_full_sheaf() calls list_add(&sheaf->barn_list, ...). This writes to barn_list.next (+0x00) and barn_list.prev (+0x08), overwriting rcu_head.next and rcu_head.func respectively.

So What Does This Mean?

If there is a write primitive that can modify the slab_sheaf union while in the RCU_PENDING state, the value written to offset +0x08 will be interpreted by the RCU infrastructure as a pointer to a callback function. Symmetrically, modification in the IN_BARN state to offset +0x08 will carry over as rcu_head.func when the sheaf transitions to the RCU path.

Modification at offset +0x08 is the most interesting point for exploitation because it touches rcu_head.func, a function pointer called by the RCU mechanism after the grace period.

To understand this exploitation technique more completely, below is a full slab_sheaf life cycle diagram with the relevant union transition points:


struct slab_sheaf lifecycle: union transition points
Linux SLUB sheaves — overlap of barn_list and rcu_head at +0x00 / +0x08
struct slab_sheaf memory layout (64-bit)
+0x0 : rcu_head.next | barn_list.next
+0x8 : rcu_head.func | barn_list.prev (critical)
+0x10 : cache
+0x18 : size
+0x1c : node
+0x20 : objects[] (unencoded void *)
mm/slub.c:404, include/linux/types.h
1
alloc_empty_sheaf()
mm/slub.c:2762
2
PERCPU_MAIN / PERCPU_SPARE
union: not actively used
barn_put_full_sheaf() [3130]
list_add(&barn_list, &barn->sheaves_full)
+0x00 ← sheaves_full.next
+0x08 ← &barn->sheaves_full
3
IN_BARN
union active: barn_list (list_head)
+0x00 = next elem
+0x08 = prev elem
WRITE TARGET +0x08
barn_get_*() [3086, 3142, 3174, 3207]
list_del(&barn_list)
+0x00 ← LIST_POISON1
+0x08 ← LIST_POISON2
4
PERCPU_RCU
union: LIST_POISON1 / LIST_POISON2
call_rcu(&rcu_head, rcu_free_sheaf) [5949]
RCU WRITES:
+0x00 ← RCU linkage
+0x08 ← &rcu_free_sheaf
5
RCU_PENDING
union active: rcu_head
+0x08 = RCU linkage
+0x08 = &rcu_free_sheaf
OVERWRITE +0x08
grace period ends
rcu_free_sheaf() [5789]
barn has free slot
mm/slub.c:5829
barn_put_full_sheaf()
list_add WRITES:
+0x00 ← .next
+0x08 ← .prev
overwrites rcu_head.next / func
barn is full
mm/slub.c:5831
sheaf_flush_unused() + free_empty_sheaf()
FREE: sheaf deallocated to kmalloc
Critical observation: the same 16-byte union is reused as barn_list in IN_BARN and as rcu_head in RCU_PENDING, so offset +0x08 alternates between barn_list.prev and rcu_head.func.

Required Conditions

It is important to stress that this technique is not a vulnerability that can be exploited directly from scratch. It is a design property that amplifies the impact when some other bug already exists.

Initial access primitive: there must be another vulnerability (UAF, buffer overflow, race condition) that provides ability to read or write memory overlapping with a slab_sheaf.

Correct timing: (a) For poisoning via IN_BARN: write to offset +0x08 while sheaf is in barn. (b) For direct overwrite: write to offset +0x08 after call_rcu() but before grace period completes (window of hundreds of microseconds to milliseconds).

Sheaf location knowledge: the attacker must know the target slab_sheaf address, generally requiring a separate information leak or converting a UAF write to provide information leak (heap leaks).

In other terms: this technique is an amplifier, not an initial vector.

In other words, this technique is an amplifier, not an initial vector. It turns a limited memory access primitive into something potentially more significant because the manipulated value carries special meaning for kernel infrastructure (the RCU machinery or barn list management).

Conclusion

The slab_sheaf Union State Confusion technique centers on one architectural fact: struct slab_sheaf uses a union to overlay rcu_head and barn_list at the same memory offsets, without an explicit state marker.

The critical point is offset +0x08, where barn_list.prev and rcu_head.func overwrite each other. In the RCU context, this is a function pointer called after the grace period. In the barn context, it is a list pointer. This technique can be adapted for linux kernel 7.0 mainline.

The core idea is simple: one write, two interpretations. The attacker writes as ordinary data, the kernel reads it as a function address and jumps to it. That is “state confusion" confusion between two contexts that share the same memory.