Use After Free Exploitation Technique in Linux Kernel 7.0-rc7

& Linux Kernel 7.0 Mainline

Using the slab_sheaf Union State Confusion Technique

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

1. 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.

2. Anatomy of the Union in slab_sheaf

2.1 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  };
```

Source: *linux-7.0/mm/slub.c:404 — struct slab_sheaf declaration*

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

2.2 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.

2.3 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 */
};
```

Source: *linux-7.0-rc7 /include/linux/types.h — definition of callback_head / rcu_head and struct list_head*

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.

3. State Machine of a slab_sheaf

3.1 Possible States

A slab_sheaf can be in one of the following states at any given time:

State	Description	Union Used As	Code Section
PERCPU_MAIN	Main sheaf in slub_percpu_sheaves of a CPU	Not used	alloc_from_pcs, free_to_pcs (mm/slub.c:4672, 5764)
PERCPU_SPARE	Spare sheaf in slub_percpu_sheaves	Not used	__pcs_replace_*_main (mm/slub.c:4561)
PERCPU_RCU	Becomes rcu_free sheaf, batches kfree_rcu()	Not used	__kfree_rcu_sheaf (mm/slub.c:5862)
IN_BARN	Resides in barn (NUMA pool), linked via barn_list	barn_list (list_head)	barn_put_*, barn_get_* (mm/slub.c:3086-3240)
RCU_PENDING	call_rcu() issued, awaiting grace period	rcu_head (callback + linkage)	call_rcu, rcu_free_sheaf (mm/slub.c:5949, 5789)

3.2 State Transitions and Union Usage

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:

```
/* linux-7.0-rc7 /mm/slub.c:5829 — inside rcu_free_sheaf() */
if (data_race(barn->nr_full) < MAX_FULL_SHEAVES) {
   stat(s, BARN_PUT);
   barn_put_full_sheaf(barn, sheaf);
   return;
}

/* linux-7.0-rc7 /mm/slub.c:3130 — barn_put_full_sheaf() */
static void barn_put_full_sheaf(struct node_barn *barn, struct slab_sheaf *sheaf)
```

```
{
    unsigned long flags;
    spin_lock_irqsave(&barn->lock, flags);
    list_add(&sheaf->barn_list, &barn->sheaves_full);
    /* list_add writes to:
     * barn_list.next (+0x00) = barn->sheaves_full.next (old first element)
     * barn_list.prev (+0x08) = &barn->sheaves_full (list head)
     * Effect: rcu_head.func (+0x08) overwritten by barn_list.prev */
    barn->nr_full++;
    spin_unlock_irqrestore(&barn->lock, flags);
}
```

4. The Dangling State! 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?

4.1 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.

4.2 Window Between call_rcu and barn_list Usage

Consider the following flow in __kfree_rcu_sheaf() (mm/slub.c: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 */
}
```

4.3 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.

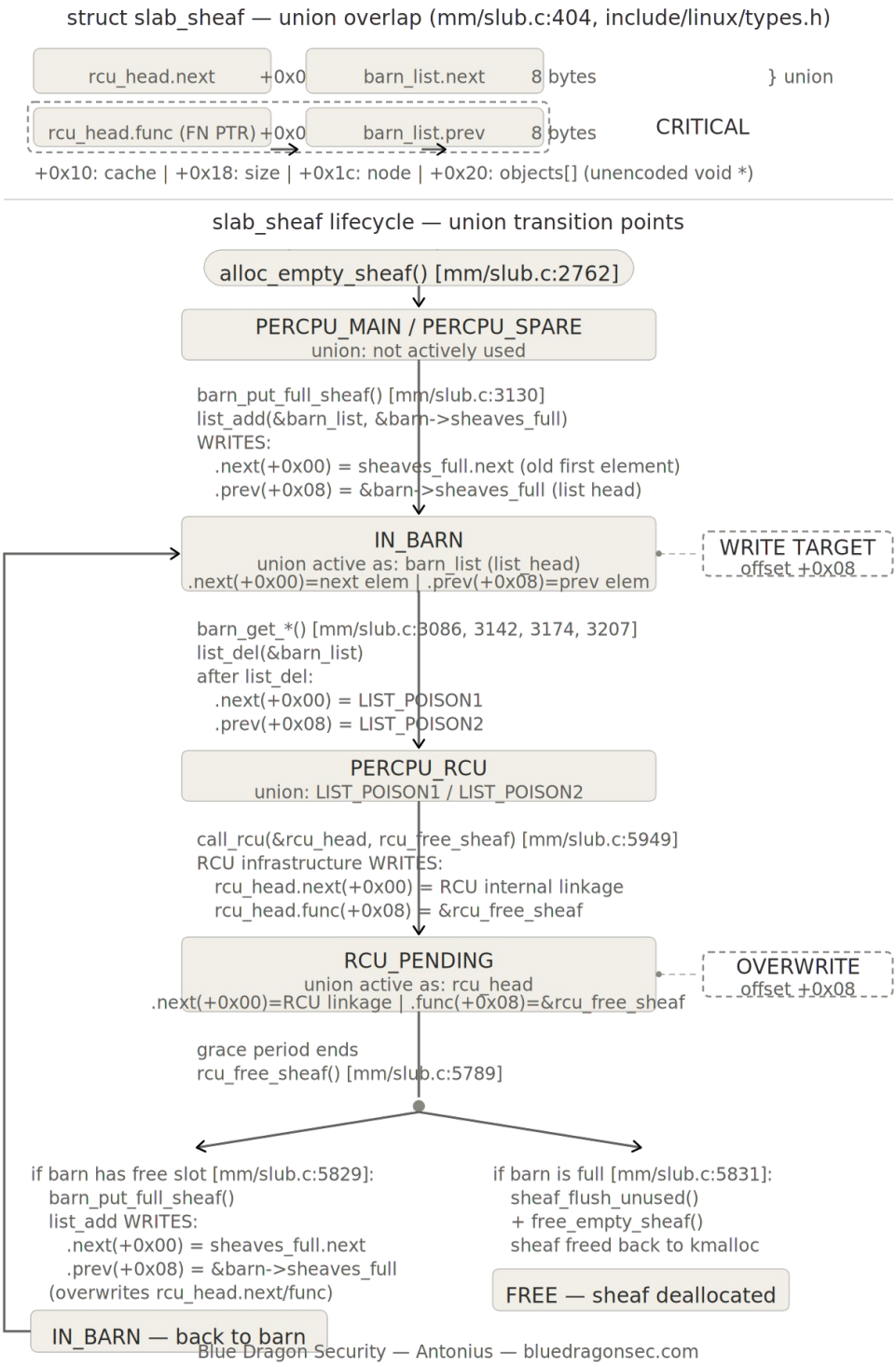
4.4 So What Does This Mean?

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.

5. Complete Lifecycle and Transition Points

To fully understand this exploitation technique, below is the lifecycle diagram of slab_sheaf with the relevant union transition points:



6. Required Conditions

This technique is not a vulnerability that can be directly exploited from scratch. It is a design property that amplifies impact if another vulnerability 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.

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

7. Mitigation & Defensive Considerations

7.1 Existing Approaches

Kernel 7.0-rc7 has several protection layers: KASAN for detecting access to freed memory, lockdep for lock usage detection, locking discipline (spin_lock_irqsave for barn, local_trylock for percpu), and kCFI/FineIBT for Control Flow Integrity validation of indirect function calls.

7.2 What's Missing

Existing mitigations do not explicitly mark slab_sheaf state or validate union interpretation. Stronger approaches would include adding an explicit state field (enum/bitfield) or separating the union into distinct non-overlapping fields.

8. 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.

Source Code References

All references verified from the linux-7.0-rc7 source tree

Path	Lines	Description
linux-7.0-rc7 /mm/slub.c	396-419	Declaration of node_barn, slab_sheaf, slub_percpu_sheaves
linux-7.0-rc7 /mm/slub.c	3086-3240	All barn operations: barn_get_*, barn_put_*
linux-7.0-rc7 /mm/slub.c	3130-3140	barn_put_full_sheaf() — list_add(&barn_list)
linux-7.0-rc7 /mm/slub.c	5789-5842	rcu_free_sheaf() — RCU callback + re-entry to barn
linux-7.0-rc7 /mm/slub.c	5862-5955	__kfree_rcu_sheaf() — free path via RCU
linux-7.0-rc7 /mm/slub.c	5949	call_rcu(&rcu_sheaf->rcu_head, rcu_free_sheaf)
linux-7.0-rc7 /include/linux/types.h	—	callback_head (rcu_head): next(+0x00), func(+0x08)
linux-7.0-rc7 /include/linux/types.h	—	struct list_head: next(+0x00), prev(+0x08)