Exploitation en espace noyau sous Linux 2.6

Stéphane DUVERGER

EADS
Suresnes, FRANCE

SSTIC 31 Mai 2007

Tour d'horizon du noyau

EADS

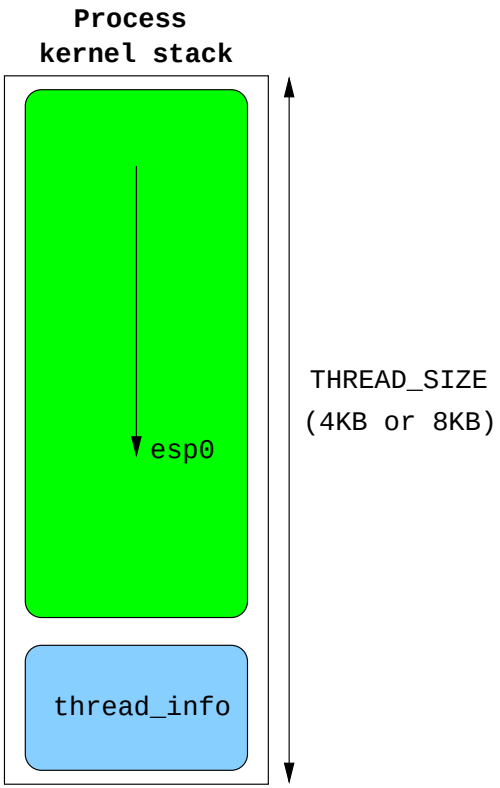
Exploitation de drivers Wifi

EADS

Première partie I

Tour d'horizon du noyau

Thread Info



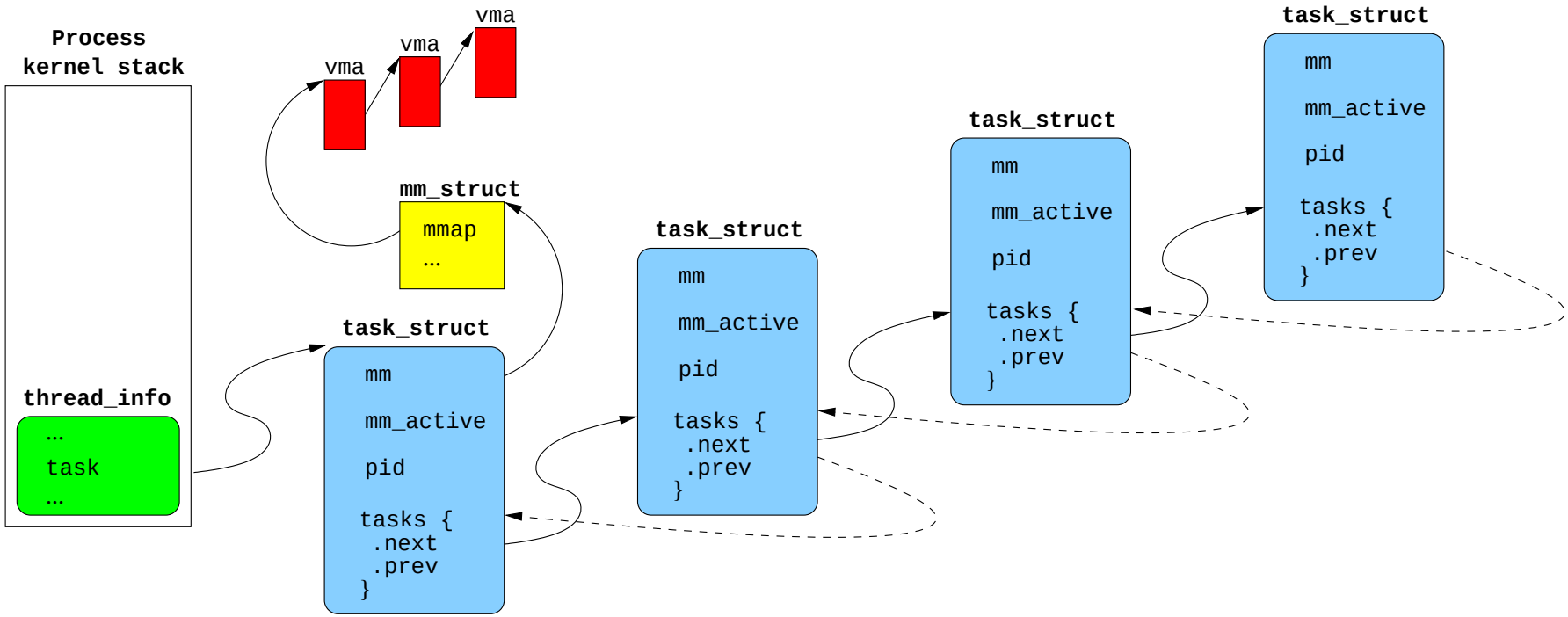
struct thread_info

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

facile à récupérer (4Ko)

```
mov   %esp, %eax
and   $0xffffe000, %eax
```

Vue générale de ces structures de données



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()
```

- définit réellement la tâche
- liste chaînée des tâches
- espace d'adressage de la tâche
- `thread_struct` :
  - liée à l'architecture
  - debug registers
  - `thread.esp0` : *saved context*

EADS

MM Struct

struct mm_struct

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

- représente l'espace d'adressage du processus
- liste de morceaux de cet espace : *vma*
- adresse du répertoire de pages (*page directory*)

VM Area Struct

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;
    ...
};
```

- une ou plusieurs pages contiguës de mémoire virtuelle
- `vm_start` $\leqslant$ *range* $<$ `vm_end`
- `vm_flags` : `VM_READ`, `VM_EXEC`, `VM_WRITE`, `VM_GROWSDOWN`, ...
- `vm_page_prot` : répercution des `vm_flags` sur les pte (*page table entries*)

Correspondance physique

- si adresse virtuelle $\geqslant$ `PAGE_OFFSET`
  alors adresse physique $=$ adresse virtuelle - `PAGE_OFFSET`

macros

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

- en mode protégé, la mémoire vidéo est accessible :
  - physiquement en `0xb8000`
  - virtuellement en `0xb8000 + PAGE_OFFSET = 0xc00b8000`
- pour charger l'espace d'adressage d'un autre processus :
  - `task->mm->pgd` $\Leftrightarrow$ adresse virtuelle du *page directory*
  - nous devons disposer de son adresse physique afin de recharger `cr3`

Kernel Control Path

kernel control path

succession d'opérations effectuées en mode noyau

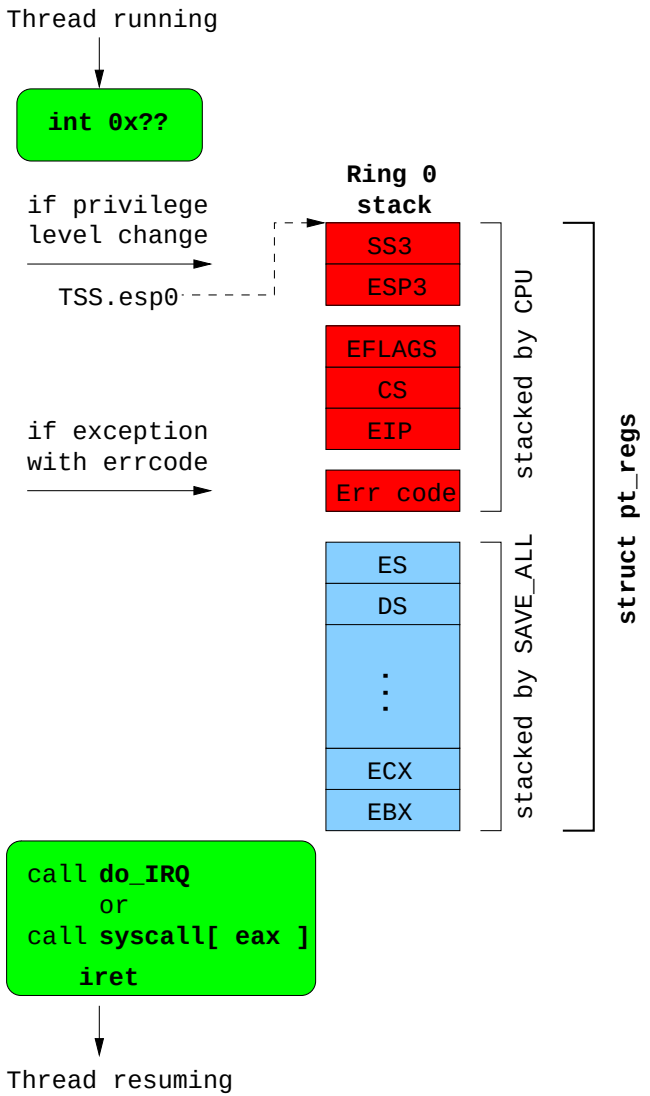
- survient suite à une interruption, une exception ou un appel système
- selon le *kernel control path*, le contexte noyau est différent :
  - *process context*
  - *interrupt context*
- selon le contexte : accès restreint aux services du noyau

Confusion

contexte noyau $\neq$ contexte sauvé (ensemble des registres)

Kernel Control Path

sauvegarde du contexte à l'entrée d'un Kernel Control Path



- le cpu peut passer de *ring 3* à *ring 0*
- auquel cas il charge ss0 et esp0 (pile noyau du processus)
- le noyau sauve tous les registres dans cette pile == *saved context*
- le traitement de l'appel système ou de l'interruption peut commencer

Process context

process context

Concerne la majorité des opérations noyau effectuées avec la pile noyau d'un processus

- le traitement d'un appel système s'effectue en *process context*
- le noyau n'est quasiment soumis à *aucune* contrainte
- en particulier, on peut : `schedule()`, `sleep()`, ...
- la vie du shellcode est plus belle en *process context*

Interrupt context
Traitement d'une interruption

- en *interrupt context* :
  - doit être rapide
  - contraintes fortes (*locking*, services noyau, ...)
  - `schedule()` == *BUG : scheduling while atomic*
- découpé en 2 phases :
  - le *Top-half* :
    - lire un *buffer*, acquitter l'interruption et rendre la main
    - généralement ininterruptible
    - noyaux 2.6 et piles de 4Ko ⇒ une pile d'interruption par processeur
    - systématiqument en *interrupt context* (*hardirq context*)
  - le *Bottom-half* :
    - interruptible
    - exécution retardée, différents types
    - selon le type, on peut se trouver en *process context*
    - plus volumineux, plus succeptible de contenir des failles

EADS

Interrupt context

Les différents *Bottom-halves*

- *SoftIRQs* :
  - optimisés, nombre fixe restreint
  - utilisés en cas de contraintes de temps fortes
  - exécution programmée par le *handler* d'interruption
- *TaskLets* :
  - reposent sur des *softIRQs* dédiés
  - ordonnancement également explicite via `tasklet_schedule()`

$\Longrightarrow$ ils s'exécutent en *interrupt context* !

- *WorkQueues* :
  - *WorkQueue* par défaut traitée par *events/cpu*
  - succession d'appels de fonctions en *process context*
  - besoin d'enregister une `struct execute_work`

Interrupt context

Évasion de l'*interrupt context*

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
```

- initialise et enregistre un futur appel de fonction
- shellcode doit retrouver ce service par recherche de motif :

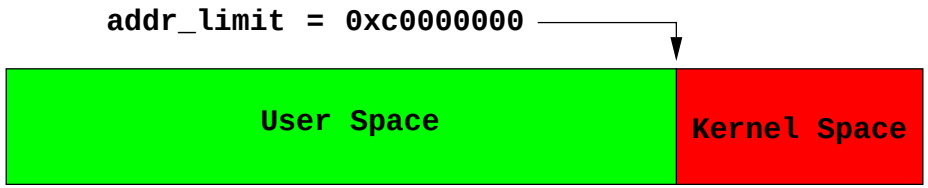
```
call    *%ecx
xor     %eax, %eax
```

- on doit disposer d'une zone mémoire fiable pour stocker notre `struct execute_work`
- code exécuté doit rendre la main à *events*

EADS

Appels système
Limite de l'espace d'adressage

- invoqués par une interruption (`int 0x80`) $\Rightarrow$ indépendant de toute adresse
- le noyau vérifie que leurs paramètres se trouvent sous la limite d'adressage
- sinon il serait possible d'écraser/lire la mémoire noyau :



écraser la mémoire noyau

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

lire la mémoire noyau

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

- cas général, pour une tâche ring 3 :

$$@\ param < GET\_FS() = thread\_info.addr\_limit < 3Go$$

- appel système depuis ring 0 :

$$SET\_FS(4Go) \Longleftrightarrow thread\_info.addr\_limit = 4Go$$

Deuxième partie II

Exploitation de drivers Wifi

Contraintes

- ce que l'on souhaite : injection/modification distante
- nous devons disposer de zones mémoires :
  - fiables et facilement retrouvables
  - non modifiées durant l'intervalle injection/exécution
  - tout particulièrement en *interrupt context*
- par chance :
  - taille espace noyau $>$ taille espace utilisateur
  - accès à la mémoire physique
  - zones initialisées uniquement au boot

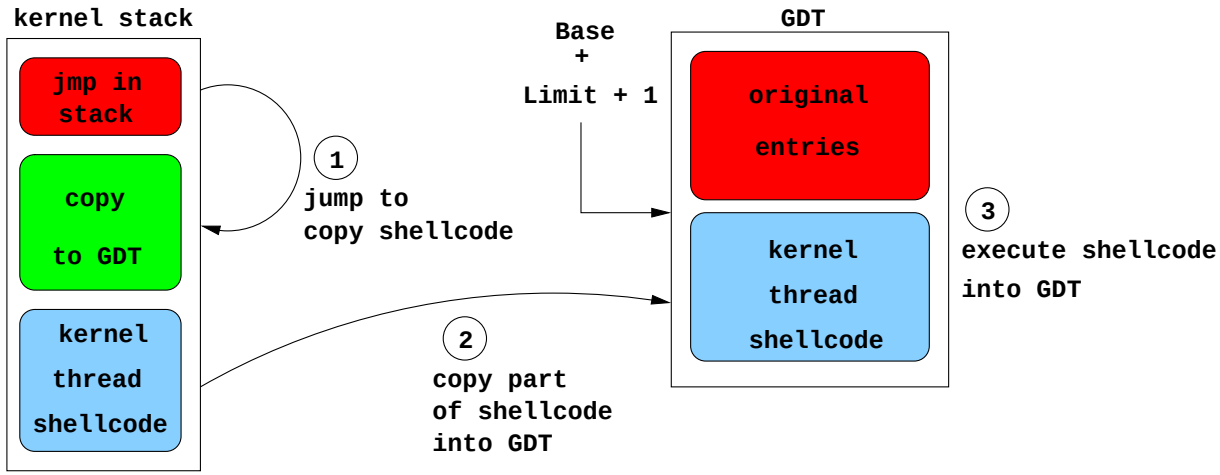
EADS

Contenu de la GDT d'un noyau 2.6.20

```
+ 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      --
```

Infection de la GDT



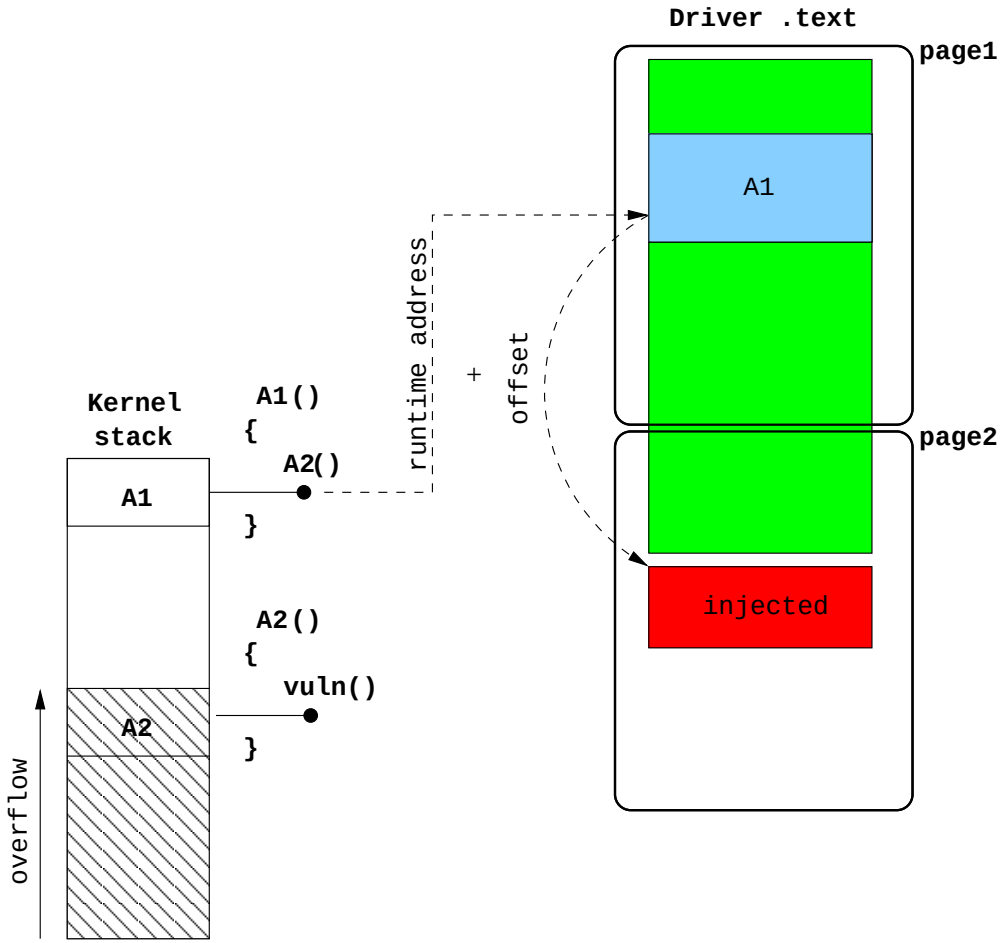
- endroit de prédilection pour l'injection
- relativement vide :
  - 32 descripteurs de 8 octets utilisés sur 8192 possibles
  - 8160*8 octets libres
- adresse facilement calculable :

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

Infection du module exploité

problème

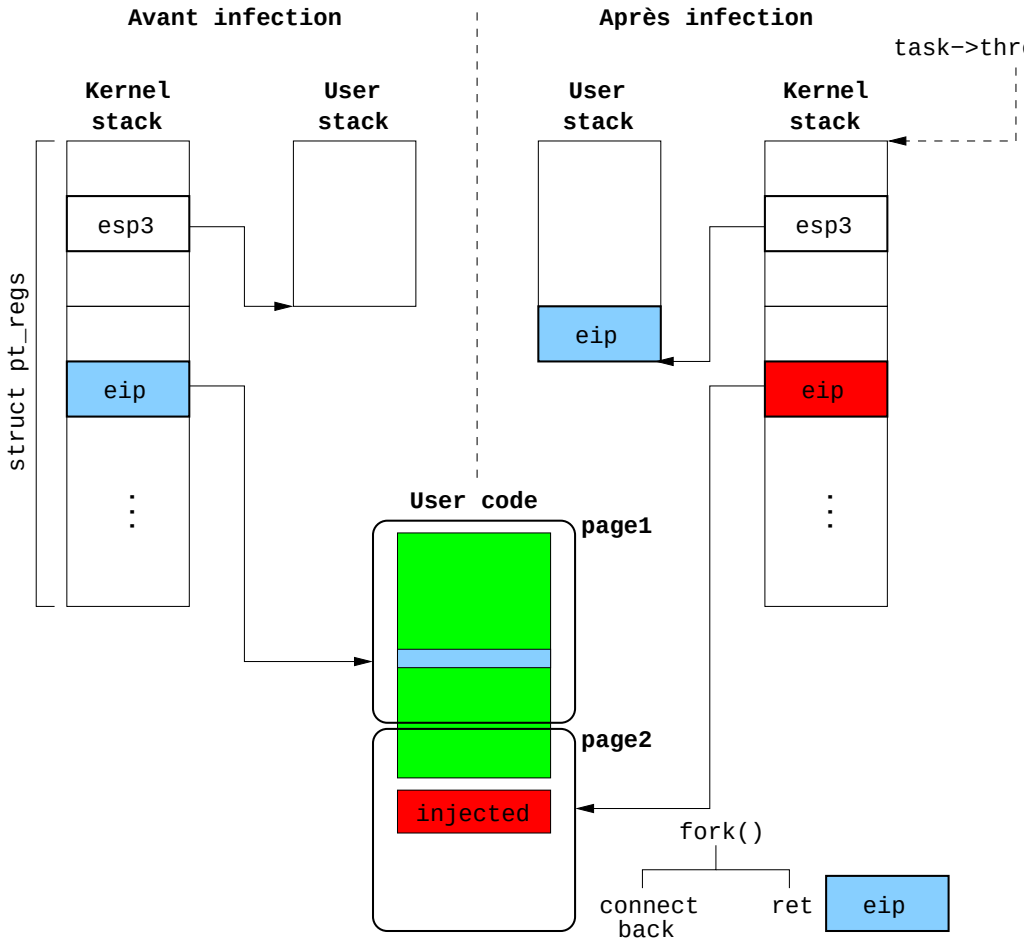
relocalisation dynamique des modules $\simeq$ espace randomisé



contournement

- allocation de pages mémoires $\gg$ taille réelle du code du module
- tirer partie des registres et zones pointées par ces registres
- instructions de sauts par registre (ie `jmp %esp`)
- récupérer l'adresse d'un $n^{\text{ème}}$ appelant
- la combiner avec un *offset* entre cet appelant et la fin de la zone de code

Infection de processus utilisateurs : init



modus operandi

- recherche facile par `pid`
- charger son espace d'adressage (`cr3`)
- modifier `eip` du *saved context* avec l'adresse du code injecté
- insérer dans la pile ring 3 d'`init` l'eip du *saved context*
- injecter un shellcode dans la section de code, effectuant un `fork()` :
  - fils : *connect back*
  - père : `ret`

Contexte d'exploitation

Scapy Packet

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

kernel control path

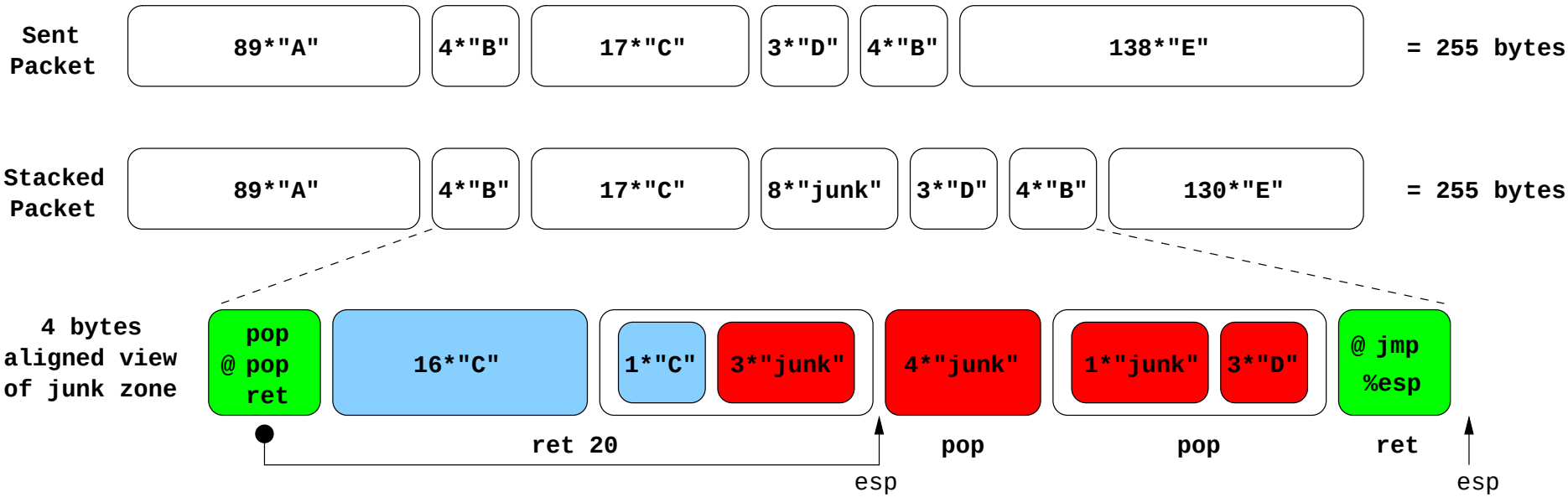
```
 1 common_interrupt()
 2 do_IRQ()
 3 irq_exit()
 4 do_softirq()
 5 __do_softirq()
 6 tasklet_action()
 7 ndis_irq_handler()
 8 ... some driver functions called
 9 vulnerable function()
10 ssid_copy()
```

épilogue de la fonction

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

- stack overflow au niveau du SSID de paquets *Probe Response*
- driver *closed source*, débogage ring 0 nécessaire
- fonction vulnérable :
  - lancée par `tasklet_action()` : *interrupt context*
  - remonte `esp` de 20 octets en sortant
  - insert 8 octets dans le paquet copié en pile
- shellcode va nécessiter plus de place

État de la pile noyau : return from vuln()



- sur 255 octets envoyés, 244 sont utilisables
- le `ret 20` place `esp` dans les 8 octets insérés (*junk zone*)
- exécution du shellcode en 2 étapes :
  - `pop;pop;ret` pour sauter la *junk zone*
  - `jmp %esp` classique

EADS

Rendre la main au driver

kernel control path

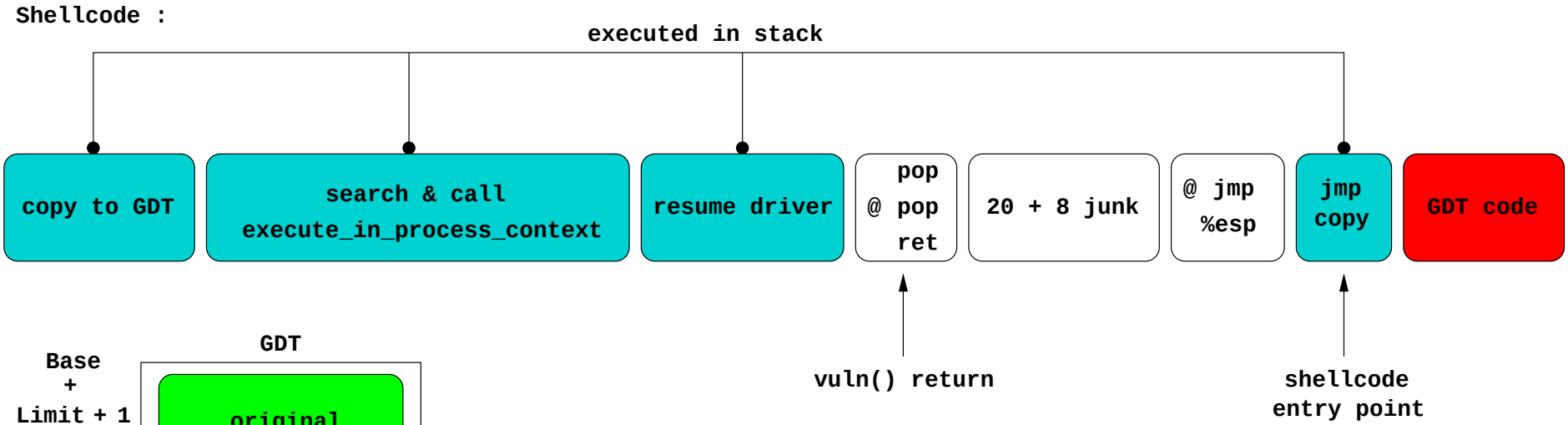
```
 1 common_interrupt()
 2 do_IRQ()
 3 irq_exit()
 4 do_softirq()
 5 __do_softirq()
 6 tasklet_action()
 7 ndis_irq_handler()
 8 ... some driver functions called
 9 vulnerable function()
10 ssid_copy()
```

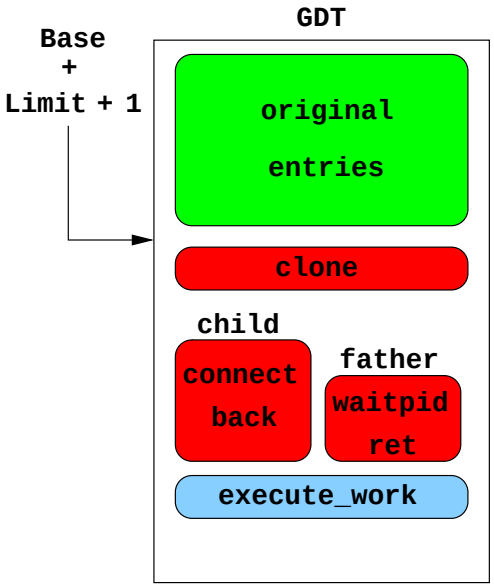
tasklet_action() epilogue

```
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
```

- beaucoup de *stack frames* écrasées
- contraint de forcer un retour de `tasklet_action()` dans `__do_softirq()`
- aligner %esp puis effectuer 3 `pop` et un `ret`

Infection de la GDT





- Dans la pile :
  - copie du *connect back* shellcode dans la GDT
  - préparation d'une `struct execute_work`
  - rend la main au driver
- Dans la GDT :
  - fils : *connect back*
  - père : `wait()` car *event/x* ne se termine pas

Infection d'init

- exécution dans la pile uniquement
- aucun appel système utilisé
- mode opératoire :
  - recherche d'`init` : `current_thread_info()->task->pid == 1`
  - chargement de `cr3` : `task->mm->pgd - PAGE_OFFSET`
  - supprime le bit `Write Protect` de `cr0`
  - ajoute l'`eip` du *saved context* dans la pile ring 3 :
    - `task->thread.esp0 - sizeof(ptregs)` == *saved context*
    - dans ce contexte on récupère `esp3`
  - récupère l'adresse de fin de la *vma* de .text - XXX octets
  - injecte le shellcode ring 3 à cette adresse
  - remplace `eip` du *saved context* par cette adresse
  - recharge `cr3` et `cr0` originaux
  - rend la main au driver

Conclusion

- démystification de l'exploitation de *stack overflow* en espace noyau sous Linux
- contourner contraintes
- tirer profit des commodités fournies aux développeurs noyau
- champ de l'exploitation en espace noyau :
  - pas totalement couvert ... loin de là
  - *bugs* fonctionnels et *race conditions* : `lost vma`

Questions ?

EADS