# Voyage au cœur de la mémoire

Damien AUMAITRE

Sogeti/ESEC
damien@security-labs.org

**Résumé** Dans cet article, nous allons parler des utilisations possibles de l'accès à la mémoire physique. Nous vous convions à un voyage au sein des structures composant le noyau de Windows. Nous verrons ensuite quelles sont les informations que nous pouvons extraire à partir de la mémoire physique. Nous finirons par quelques démonstrations d'utilisations offensives et défensives de ces informations.

## 1 Introduction

En 2005 Maximillian Dornseif [1] et en 2006 Adam Boileau [2] présentent tous les deux des travaux compromettant le système d'exploitation via le FireWire. Adam Boileau a démontré la possibilité d'exécuter du code arbitraire en utilisant uniquement l'accès à la mémoire physique. Le code de ses démonstrations n'a pas été diffusé publiquement, nous nous sommes donc intéressés à essayer de les reproduire.

Au fur et à mesure de nos expérimentations, nous avons créé un framework d'introspection de la mémoire. Nous allons utiliser les résultats issus du développement de ce framework pour détailler certaines structures internes composant le noyau d'un Windows XP SP2 32 bits.

Dans un premier temps, nous ferons un bref rappel sur les mécanismes mis en jeu lorsque nous accédons à la mémoire physique via le FireWire. Ensuite, nous décrirons comment retrouver l'espace d'adressage virtuel. Nous verrons comment nous pouvons retrouver les objets manipulés par le noyau au sein de la mémoire physique. Finalement, nous montrerons des utilisations issues de la connaissance du fonctionnement interne du noyau.

Nous allons utiliser tout au long de l'article un dump mémoire, téléchargeable sur le site de *pyflag* : http://www.pyflag.net/images/test_images/Memory/xp-laptop-2005-06-25.img.e01. Il nous servira pour nos exemples et le lecteur curieux y trouvera un terrain d'expérimentation.

L'image est compressée via le format EWF (Expert Witness Compression Format), une bibliothèque est disponible sur https://www.uitwisselplatform.nl/projects/libewf/ pour la décompresser.

## 2 Accès à la mémoire physique

Avant de pouvoir expérimenter avec la mémoire, nous avons besoin d'y accéder. Nous ne reviendrions pas en détail sur les méthodes permettant d'accéder à la mémoire physique (que ce soit par un dump ou en live). Nicolas Ruff dans [3] a très bien présenté ces différentes techniques ainsi que leurs avantages et inconvénients.

Nous allons par contre présenter plus en détail le fonctionnement du bus FireWire et son utilisation pour obtenir un accès en lecture/écriture à la mémoire physique.

### 2.1 Origine du FireWire

Apple a développé le FireWire à la fin des années 80. Le FireWire a été standardisé à l'IEEE en 1995 en tant que IEEE 1394.

En 2000, Sun, Apple, Compaq, Intel, Microsoft, National Semiconductor et Texas Instruments finissent d'écrire la spécification OHCI 1394 (Open Host Controller Interface) qui fournit une manière standardisée d'implémenter les contrôleurs FireWire autant du point de vue matériel qu'au niveau de l'interface de programmation.

### 2.2 Fonctionnement

Bien entendu le fonctionnement complet du bus FireWire ne sera pas abordé ici. La spécification OHCI [4] répondra à la plupart des questions du lecteur curieux. Nous allons simplement nous focaliser sur les mécanismes impliqués lorsque nous utilisons le FireWire pour lire la mémoire physique.

**DMA** Le DMA (Direct Memory Access) permet aux périphériques de communiquer directement entre eux sans passer par le processeur. Il est utilisé par exemple dans les contrôleurs de disques durs, les cartes graphiques, les cartes réseaux, les cartes sons, les périphériques USB et FireWire.

Sans le DMA, chaque octet échangé entre les périphériques doit passer par le processeur qui le recopie entre les périphériques. Avec le DMA, le processeur initie le transfert. Il reste donc disponible pour d'autres tâches. Lorsque le transfert est terminé, le processeur est averti par une interruption.

Il existe deux types de DMA. Le premier appelé «third party DMA» est historiquement utilisé sur le bus ISA. Le transfert de données est effectué par le contrôleur DMA de la carte mère. Le deuxième mode, appelé «bus mastering DMA», est utilisé sur les bus plus récents (comme le bus PCI). Dans ce cas, c'est

le périphérique qui prend le contrôle du bus et qui effectue le transfert.

En résumé, avec le bus mastering, un périphérique peut prendre le contrôle du bus de données sans que le processeur en soit averti et sans avoir besoin du processeur. De plus, ce périphérique a accès en lecture-écriture à toute la mémoire physique du système.

Comment ce mécanisme fonctionne-t-il ? La pièce maîtresse est le contrôleur OHCI (Open Host Controller Interface).

La configuration du contrôleur OHCI est réalisée par le «configuration ROM model» (ISO/IEC 13213). Cette ROM est aussi appelée CSR (Configuration and Status Registers).

Il existe beaucoup de registres de configuration que nous n'allons pas détailler. Arrêtons nous cependant sur les registres suivants : les registres «*Asynchronous Request Filter*» et «*Physical Request Filter*».

Ils sont constitués chacun de 4 mots de 32 bits (FilterHi(set), FilterHi(clear), FilterLo(set), FilterLo(clear)) indiquant les permissions allouées aux périphériques.

Lorsque le registre «Physical Request Filter» est à zéro, aucun accès à la mémoire physique n'est autorisé.

**Adressage** Le FireWire peut fonctionner avec une topologie en étoile ou en arbre. Il peut gérer jusqu'à 63 périphériques (ou nœuds) par bus.

Chaque périphérique a un identifiant unique appelé UUID. Les accès aux périphériques se font grâce à un numéro d'identifiant qui est assigné dynamiquement à chaque reconfiguration du bus. Les périphériques ont une adresse de 64 bits :

- 10 bits sont utilisés pour le Bus ID (pour un maximum de 1023 bus) ;
- 6 bits sont utilisés pour le Node ID (chaque bus peut avoir 63 nœuds) ;
- 48 bits sont utilisés pour l'adressage mémoire dans les noeuds.

La figure 1 présente l'espace mémoire vu par le node. Le registre `physicalUpperBound` est optionel et seulement implémenté sur certains contrôleurs particuliers. Il permet de modifier la borne haute de la mémoire physique accessible. S'il n'est pas implémenté alors la limite est de 4GB.

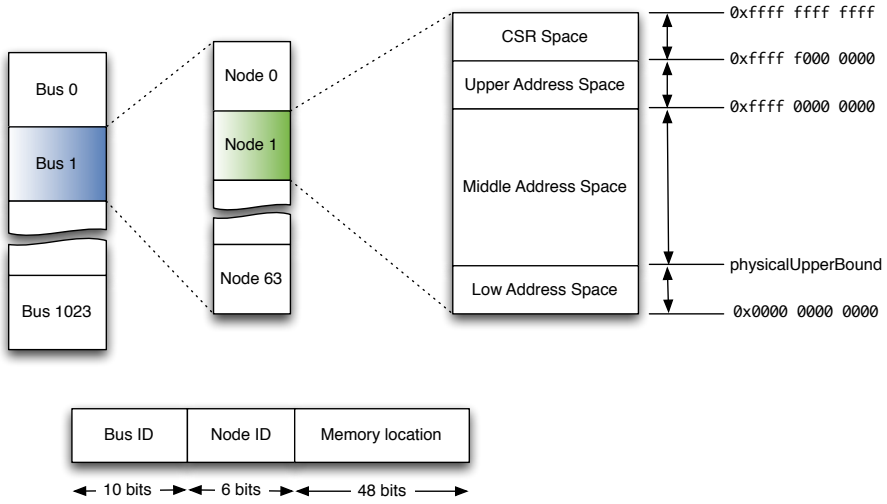


**Fig. 1.** FireWire

**Accès à la mémoire** L'utilisation du FireWire comme moyen d'accès à la mémoire n'est pas nouveau.

Maximillian Dornseif a présenté [1] lors de la conférence CanSecWest en 2005 une utilisation originale d'un iPod. Il a pu en utilisant un iPod, ayant comme système d'exploitation un noyau Linux, prendre le contrôle de machines tournant sous Linux ou sous Mac OS X.

Lors de la présentation (c'est-à-dire en 2005), Maximillian a donné une liste des systèmes affectés par l'attaque. OS X et FreeBSD supportent la lecture et l'écriture. Linux ne supporte que la lecture. Concernant les OS signés Microsoft, Windows 2000 plante et cela ne fonctionne pas sous Windows XP.

En 2006, Adam Boileau a présenté [2] à la conférence RuxCon un moyen d'utiliser le FireWire avec Windows XP.

Pourquoi ce qui ne fonctionnait pas en 2005 a fonctionné en 2006 ? Par défaut, le DMA et l'accès à la mémoire physique sont autorisés sur tous les OS exceptés ceux de Microsoft.

Adam Boileau a remarqué que lors du branchement de l'iPod (en fait de n'importe quel périphérique requérant le DMA), Windows positionne les registres cités précédemment à une valeur permettant l'utilisation du DMA. Il a donc remplacé l'identification du contrôleur OHCI de son portable par celle d'un iPod. Cette manipulation leurre le système d'exploitation qui autorise donc

l'utilisation du DMA.

À quoi ressemble cette identification et comment pouvons-nous faire pour la récupérer ? Dans la norme OHCI [4] il est écrit que la ROM est accessible à partir de l'adresse `0xffff f000 0000+0x400`. Sa taille maximale étant de `0x400`, il suffit de lire la mémoire du périphérique à partir de l'adresse `0xffff f000 0400` jusqu'à l'adresse `0xffff f000 0800`. Les CSR doivent être lus mot par mot sinon le périphérique refuse la lecture. Par exemple en branchant un disque dur FireWire, nous obtenons la ROM suivante :

```
00000000  04 04 00 29 31 33 39 34  00 ff 50 02 00 d0 4b 54  |...)1394..P...KT|
00000010  08 07 9a 99 00 06 77 2f  0c 00 83 c0 03 00 d0 4b  |......w/.......K|
00000020  81 00 00 14 04 00 91 1f  81 00 00 1a d1 00 00 01  |................|
00000030  00 0c 4b 98 12 00 60 9e  13 01 04 83 3c 00 01 02  |..K...‘.....<...|
00000040  54 00 c0 00 3a 00 3c 08  38 00 60 9e 39 01 04 d8  |T...:.<.8.‘.9...|
00000050  3b 00 00 00 3d 00 00 03  14 4e 00 00 17 00 00 01  |;...=....N......|
00000060  81 00 00 13 14 4e 00 01  17 00 00 00 81 00 00 1d  |.....N..........|
00000070  00 07 c2 03 00 00 00 00  00 00 00 00 4c 61 43 69  |............LaCi|
00000080  65 20 47 72 6f 75 70 20  53 41 00 00 00 00 00 00  |e Group SA......|
00000090  00 06 17 fa 00 00 00 00  00 00 00 00 4f 58 46 57  |............OXFW|
000000a0  39 31 31 2b 00 00 00 00  00 00 00 00 00 0c 81 61  |911+...........a|
000000b0  00 00 00 00 00 00 00 00  4c 61 43 69 65 20 48 61  |........LaCie Ha|
000000c0  72 64 20 44 72 69 76 65  20 46 69 72 65 57 69 72  |rd Drive FireWir|
000000d0  65 2b 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |e+..............|
000000e0  00 0c 81 61 00 00 00 00  00 00 00 00 4c 61 43 69  |...a........LaCi|
000000f0  65 20 48 61 72 64 20 44  72 69 76 65 20 46 69 72  |e Hard Drive Fir|
00000100  65 57 69 72 65 2b 00 00  00 00 00 00 00 00 00 00  |eWire+..........|
00000110  00 00 00 00 ff ff ff ff                           |........|
```

Qu'en est-il de Windows Vista ? Si les registres sont positionnés aux bonnes valeurs, les transferts DMA sont autorisés et nous pouvons lire la mémoire de la même manière que sous XP.

Ces registres sont remis à 0 lors du reset du bus FireWire. Il faut donc régulièrement répéter le processus. Le redémarrage du bus est automatique et inclut une reconfiguration du bus. Il intervient dès qu'un périphérique est connecté, déconnecté, suspendu ou redémarré.

Il est possible de positionner manuellement la valeur de ces registres. Étant donné que le contrôleur est la plupart du temps implémenté sur une carte PCI, les CSR sont mappés en mémoire sur le bus PCI. Par exemple sur notre système, les CSR sont mappés de l'adresse `0xf0500000` à l'addresse `0xf05007ff` (mémoire physique). Le registre «*Asynchronous Request Filter*» se situe à un offset de `0x100`, «*Physical Request Filter* à un offset de `0x110`.

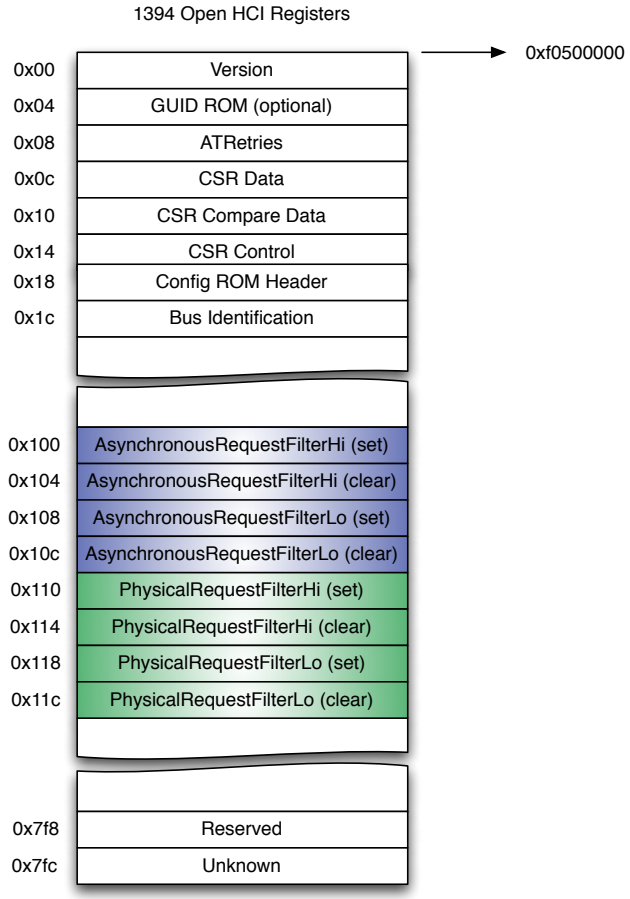


**Fig. 2.** Registres du contrôleur OHCI

Avec un débogueur (ou tout autre moyen permettant d'écrire dans la mémoire physique), il suffit d'écrire la valeur `0xffffffff` sur l'adresse mémoire `0xf0500000+0x110` et `0xf0500000+0x118` pour lire et écrire où nous voulons dans la mémoire.

Il y a 4 mots de 32 bits pour chacun de ces registres car il peut y avoir jusqu'à 63 nœuds par bus et chacun des bits autorise ou interdit un nœud. En positionnant tous les bits à 1, nous autorisons l'accès à la mémoire pour tous les nœuds.

## 3 Reconstruction de l'espace d'adressage

Une fois obtenu un accès à la mémoire physique, le plus difficile est de donner un sens à cette masse de données. En effet, la mémoire physique est un immense puzzle. Les données ne sont pas adjacentes dans la mémoire.

Il faut donc reconstituer la mémoire virtuelle. Le processeur et le système d'exploitation manipulent des adresses virtuelles et non des adresses physiques. De plus, tous les systèmes récents utilisent la pagination pour isoler les espaces d'adressage des différents processus. Suivant le mode de fonctionnement du processeur, le mécanisme pour traduire une adresse virtuelle en une adresse physique est légèrement différent.

### 3.1 Gestion de la mémoire par le processeur

Pour convertir une adresse virtuelle en une adresse physique, le processeur utilise des tables à plusieurs niveaux. Sur les processeurs récents, la pagination est contrôlée par 3 flags dans plusieurs registres de configuration (`CR`) du processeur.

Le premier flag, appelé PG, est contrôlé par la valeur du bit 31 du registre `cr0`. Dans notre cas, étant donné que nous voulons convertir une adresse virtuelle, ce bit est toujours à 1.

Le deuxième flag, appelé PSE (Page Size Extension), est contrôlé par le bit 4 du registre `cr4`. Lorsqu'il est actif, il est possible d'utiliser des pages de mémoire plus grandes (4 MB ou 2 MB au lieu de 4 KB).

Le dernier flag, appelé PAE (Physical Address Extension), est contrôlé par le bit 5 du registre `cr4`. Lorsque le processeur fonctionne en mode PAE, il permet d'adresser plus de 4 GB de mémoire. Ce mode est aussi utilisé par Windows dans le cadre de l'utilisation du DEP (Data Execution Prevention).

Il y a donc 4 possibilités de traduction d'une adresse virtuelle suivant la configuration du processeur :

- la page a une taille de 4KB et le PAE n'est pas activé ;
- la page a une taille de 4MB et le PAE n'est pas activé ;
- la page a une taille de 4KB et le PAE est actif ;
- la page a une taille de 2MB et la PAE est actif.

De plus, les grandes pages de mémoire peuvent coexister avec les pages de taille classique. Elles sont souvent utilisées pour les pages contenant le noyau car elles peuvent rester plus facilement dans le cache TLB [5] (Translation Lookaside Buffer).

Dans le cas le plus simple, c'est-à-dire lorsque le processeur utilise des pages de 4KB en mode non PAE, la traduction d'une adresse virtuelle en une adresse physique se fait en deux étapes, chacune utilisant une table de traduction. La première table est appelée le répertoire de pages (Page Directory) et la seconde la table de pages (Page Table). Une indirection à deux niveaux est utilisée pour réduire la mémoire utilisée pour stocker les tables.

Chaque processus possède un répertoire de pages. L'adresse physique du répertoire de pages en cours d'utilisation est stockée dans un registre du processeur appelé `cr3`. Les 10 premiers bits de l'adresse virtuelle déterminent l'index dans le répertoire de pages qui pointe sur la bonne table de page. Les 10 bits du milieu déterminent l'index dans la table de pages qui pointe sur la page physique contenant les données. Les 12 derniers bits représentent l'offset au sein de la page. La figure 3 illustre ce processus.

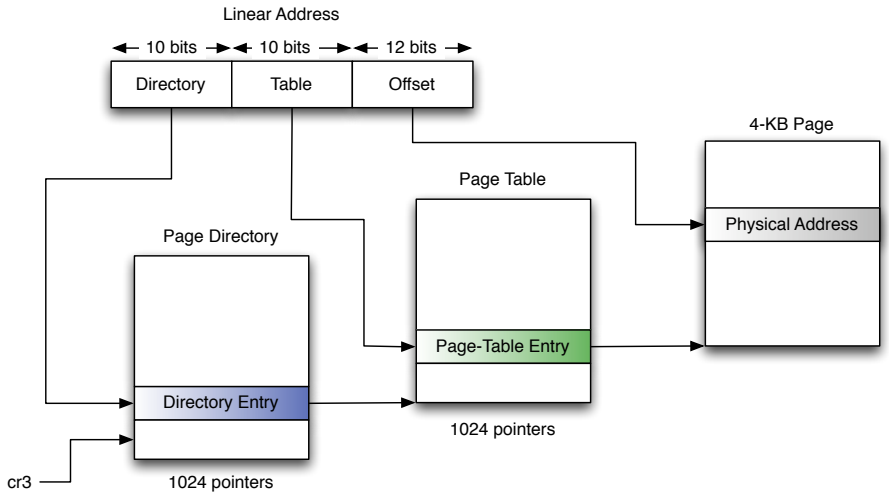


**Fig. 3.** Page de 4KB sans PAE

Lorsque le processeur utilise des pages de 4 MB, la traduction est encore plus simple car elle implique l'utilisation d'un seul répertoire (comme nous pouvons le voir sur la figure 4).

Lorsque le mode PAE est activé, les choses se compliquent un peu. Il existe une table de plus (appelée Page Directory Pointer Table). De plus, les pointeurs ont maintenant une taille de 8 octets contrairement au mode sans PAE où les pointeurs font une taille de 4 octets. Mis à part ces différences, la traduction d'adresse est relativement similaire.

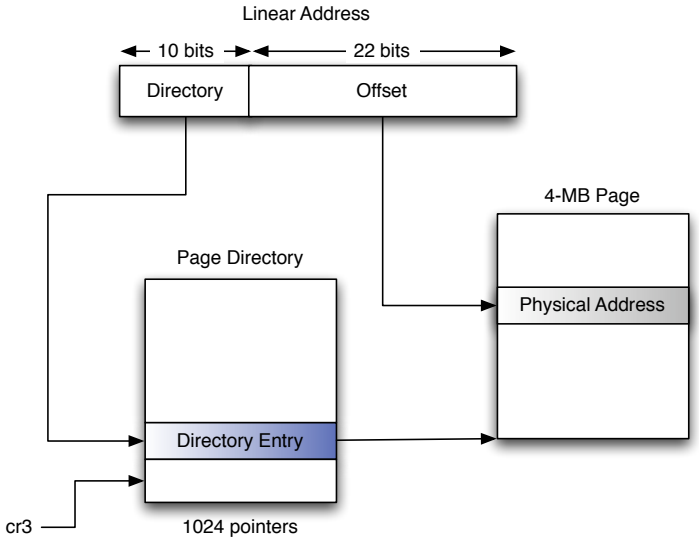


**Fig. 4.** Page de 4MB sans PAE

En mode PAE, les grandes pages ont une taille de 2 MB seulement. C'est dû au fait que l'offset est codé sur 21 bits au lieu de 22 auparavant. Cela est la seule différence.

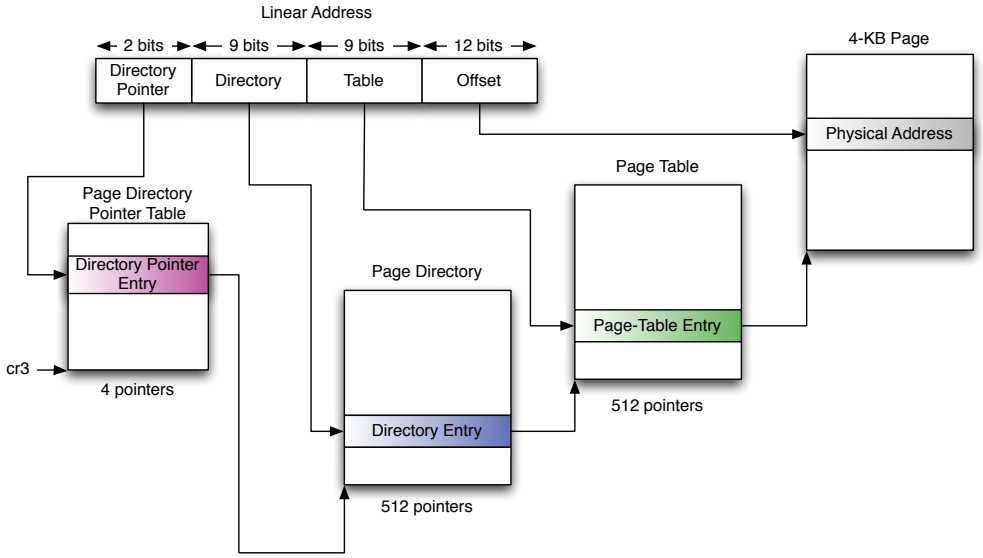


**Fig. 5.** Page de 4KB avec PAE

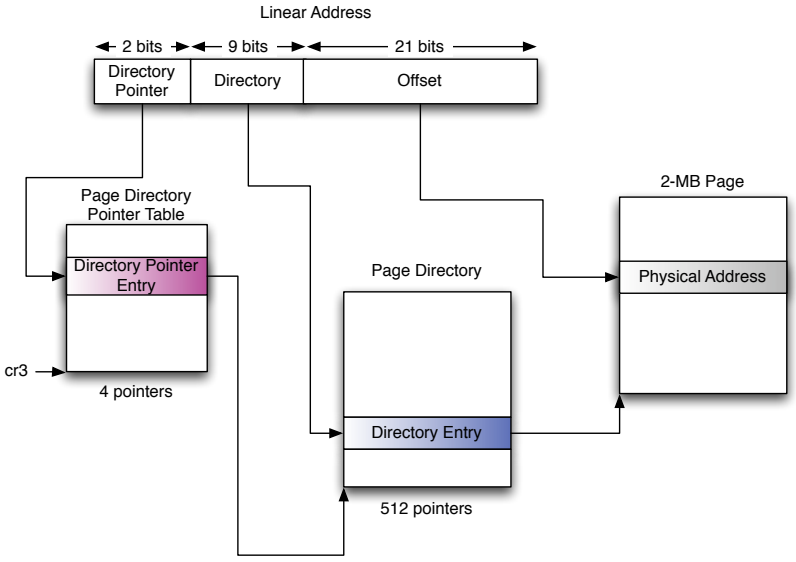


**Fig. 6.** Page de 2 MB avec PAE

Maintenant que nous savons convertir toutes les adresses virtuelles en adresses physiques, il nous manque quand même quelque chose ! En effet, il est nécessaire d'avoir la valeur du registre `cr3` pour effectuer la traduction. Comment pouvons-nous la retrouver ?

Andreas Schuster [6] a proposé une méthode consistant à parcourir la mémoire physique à la recherche d'une signature indiquant la présence d'une structure `_EPROCESS`. Cette structure est utilisée par le noyau pour représenter un processus. Elle contient en particulier une sauvegarde du registre `cr3` permettant de reconstituer l'espace virtuel du processus. Cependant, cette méthode est très dépendante de la version de Windows utilisée, car les offsets de la structure `_EPROCESS` ne sont pas les mêmes.

Sans rentrer dans les détails, cette méthode repose sur le fait qu'au début de la structure se trouvent des octets qui ont une valeur fixe pour chaque version de Windows. Il suffit ensuite de faire quelques tests pour valider si la structure potentielle a une chance raisonnable d'être une structure `_EPROCESS`.

Les lecteurs intéressés peuvent se reporter à son article qui explique en détail le mécanisme mis en jeu.

L'intérêt principal de la pagination est de proposer aux processus (et donc aux applications) une vue de la mémoire indépendante de la quantité de mémoire physique présente sur le système. Sur un processeur 32 bits, il y a donc virtuellement 4 GB de mémoire. Il parait donc évident que toutes les adresses virtuelles n'aboutissent pas à une adresse physique. Comment est-ce que le processeur sait si l'adresse demandée est valide ou non ? Il regarde le contenu des entrées des différents répertoires de pages. Regardons maintenant la structure de ces entrées.

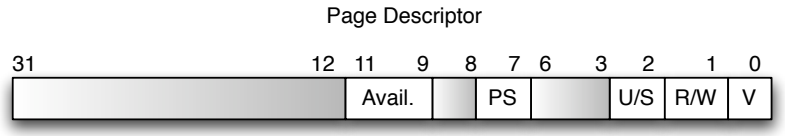


**Fig. 7.** Format des descripteurs de pages sans PAE

Les bits importants sont les suivants. Le bit 0, appelé `V` (Valid) indique si la page est valide (présente en mémoire). Le bit 1, appelé `R/W` (Read/Write) indique si la page est en lecture/écriture. Le bit 2, appelé `U/S` indique si la page est accessible en mode utilisateur ou en mode noyau. Le bit 7 indique si la

page est une grande page ou non. Les bits 9 à 11 sont utilisables par le système d'exploitation à sa discrétion. Nous verrons plus loin l'utilisation qui en est faite.

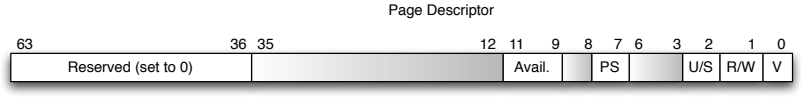


**Fig. 8.** Format des descripteurs de pages avec PAE

Lorsque le PAE est activé, nous voyons bien que les descripteurs de pages ont maintenant une taille de 8 octets.

Nous avons maintenant toutes les connaissances nécessaires pour traduire une adresse virtuelle en une adresse physique. Nous allons illustrer le mécanisme par un exemple.

Considérons que le registre `cr3` a la valeur `0x39000` et que le mode PAE est inactif. Nous désirons convertir l'adresse virtuelle `0x81ebc128`.

Tout d'abord, nous retrouvons l'entrée correspondante dans le *Page Directory*. Nous prenons les 10 bits de poids fort de l'adresse virtuelle, soit `0x207`. En les multipliant par 4 et en rajoutant `0x39000`, nous obtenons l'adresse physique du PDE (Page Directory Entry). À cette adresse, nous lisons la valeur `0x01c001e3`. Le bit `V` vaut 1, ce qui signifie que la page est présente en mémoire. Le bit `PS` vaut 1, il s'agit donc d'une grande page (4MB). Le bit `U/S` est à 0, cela signifie que la page est accessible uniquement en mode noyau. La *Page Base Adresse* est la valeur des 10 bits de poids fort du PDE, soit `0x1c00`. Il s'agit en fait du numéro de la page physique. En prenant les 22 bits de poids faible de l'adresse virtuelle, nous obtenons l'offset dans la page, soit `0x2bc128L`.

Cette page commence donc physiquement à l'adresse `0x1c00000` et regroupe les adresses virtuelles allant de `0x81c00000` à `0x81ffffff`. L'adresse physique correspondant à notre adresse virtuelle est l'adresse `0x1ebc128`.

Nous sommes donc capable maintenant de retrouver la structure représentant un processus, d'en déduire la valeur du *Page Directory* et donc de pouvoir reconstruire son espace d'adressage.

Lorsque nous parcourons les entrées des différents descripteurs de pages de chacun des processus, nous remarquons qu'il existe beaucoup de pages invalides. En examinant la mémoire occupée par le système, il existe des pages allouées que nous ne retrouvons pas.

Cela vient du fait que pour l'instant nous avons uniquement la vision du processeur. Pour pouvoir convertir plus d'adresses, il est nécessaire d'étudier le fonctionnement du gestionnaire de mémoire du noyau Windows.

### 3.2 Gestion de la mémoire par le noyau Windows

Lorsque nous cherchons à reconstruire la mémoire virtuelle à partir de la mémoire physique, il arrive assez souvent que la page physique où se trouve la donnée recherchée soit marquée comme *non valide*. Cela ne veut pas forcément dire que la page soit effectivement inaccessible (si elle est en swap par exemple).

Pour différentes raisons, le système d'exploitation peut mettre le bit `V` (Valid) à 0 le temps d'effectuer certaines tâches. Dans le cas de Windows, il existe plusieurs types de pages invalides. En copiant le comportement du gestionnaire de mémoire, il est possible de retrouver l'adresse physique de certaines pages marquées comme invalides.

Pour gérer les cas de mémoire partagée, le noyau utilise des PTE (Page Table Entry) particuliers appelés *Prototype PTE*. Ces PTE sont contenues dans une structure du noyau appelée `_SEGMENT`.

```
struct _SEGMENT, 14 elements, 0x40 bytes
  +0x000 ControlArea     : Ptr32 to struct _CONTROL_AREA, 13 elements, 0x30 bytes
   +0x004 TotalNumberOfPtes : Uint4B
   +0x008 NonExtendedPtes  : Uint4B
   +0x00c WritableUserReferences : Uint4B
   +0x010 SizeOfSegment    : Uint8B
   +0x018 SegmentPteTemplate : struct _MMPTE, 1 elements, 0x4 bytes
   +0x01c NumberOfCommittedPages : Uint4B
  +0x020 ExtendInfo      : Ptr32 to struct _MMEXTEND_INFO, 2 elements, 0x10 bytes
   +0x024 SystemImageBase  : Ptr32 to Void
   +0x028 BasedAddress     : Ptr32 to Void
   +0x02c u1               : union __unnamed, 2 elements, 0x4 bytes
   +0x030 u2               : union __unnamed, 2 elements, 0x4 bytes
   +0x034 PrototypePte     : Ptr32 to struct _MMPTE, 1 elements, 0x4 bytes
   +0x038 ThePtes          : [1] struct _MMPTE, 1 elements, 0x4 bytes
```

À partir du champ `ThePtes`, nous trouvons un tableau de structures `_MMPTE` (ie. une structure du noyau pour représenter les PTE). Le champ `TotalNumberOfPtes` nous donne la taille du tableau.

Windows se sert des bits 10 et 11 des PDE et PTE (dans les manuels Intel, ces bits sont réservés pour être utilisé par le système d'exploitation). Le bit 10 est appelé le bit `Prototype`, le bit 11 `Transition`. Suivant les valeurs de ces 2 bits, la traduction en adresse physique est différente.

**Premier cas** : le bit `P` et le bit `T` sont tous les deux à zéro. Si les bits 1 à 4 et 12 à 31 sont nuls alors la page est marquée comme `Demand Zero`. Cela signifie

que le système d'exploitation renverra une page remplie de zéro. Dans notre cas, nous pouvons considérer que la page lue ne contient que des zéros. Dans le cas où les bits ne sont pas nuls, la page est alors marquée comme `PageFile`. La page est donc en swap et il faut avoir accès au fichier de swap pour la retrouver.

**Deuxième cas** : le bit `P` est à 0 et le bit `T` est à 1. La page est dite en `Transition`. Cela signifie qu'elle a été modifiée en mémoire mais pas encore sur le disque. Elle peut cependant être toujours retrouvée par le mécanisme de traduction classique.

**Troisième cas** : le bit `P` du PTE est à 1. La page est marquée comme `Prototype`. Le bit `T` n'a pas de sens ici. Les PTE prototypes sont utilisés lorsque plusieurs processus utilisent la même page. Les processus pointent sur la page prototype qui, elle, pointe vers la vraie page. Du coup, le système d'exploitation ne doit modifier que la page prototype. C'est une sorte de lien symbolique de PTE. Cela évite de mettre à jour tous les processus chaque fois que la page est déplacée.

Pour plus d'informations, le lecteur intéressé pourra se reporter à l'article de Jesse D. Kornblum [7].

Kornblum propose une formule dans son article pour retrouver le PTE original à partir du PTE prototype : `0xe1000000 + (PrototypeIndex << 2)` où `PrototypeIndex` représente les bits 1 à 7 et les bits 11 à 31 du PTE (soit 28 bits).

Suivant la valeur des différents flags, les PTE prototypes ont plusieurs significations :

- si le bit 0 est à 1, alors la page est en mémoire et accessible classiquement ;
- si le bit `T` est à 1 et le bit `P` est à 0, alors elle est aussi accessible classiquement ;
- comme le cas précédent avec le bit `D`, alors elle est accessible ;
- Si tous les bits sont à 0 sauf les bits `PageFileNumber` et `PageFileOffset` alors la page est en swap ;
- Si le bit `P` est à 1, alors la page appartient à un fichier mappé en mémoire et est gérée par le «*cache manager*».

**Quatrième cas** : le PTE est entièrement nul. Dans ce cas, nous n'avons pas assez d'information et il nous faudra utiliser les VAD (Virtual Address Descriptor).

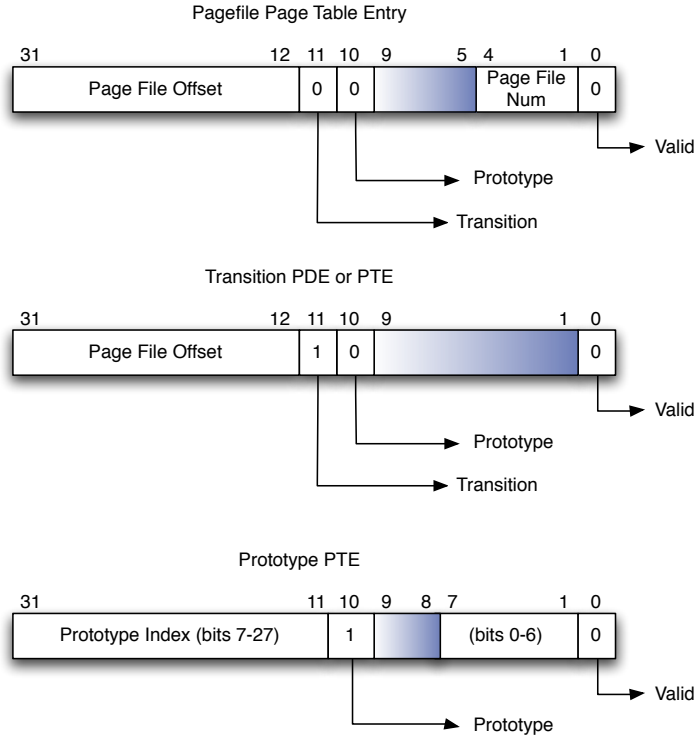


**Fig. 9.** Format des PTE invalides

Nous allons maintenant détailler le fonctionnement et la structure des VAD.

Pour éviter de charger en mémoire toutes les pages demandées par un processus, le *memory manager* utilise un algorithme permettant de charger en mémoire la page au moment où le thread utilise la mémoire précédemment allouée (*lazy evaluation*). La mémoire est allouée uniquement lorsqu'elle est utilisée.

Pour garder la trace des adresses virtuelles qui ont été allouées dans l'espace d'adressage du processus, le *memory manager* utilise une structure d'arbre binaire appelée `VAD`. Chaque processus possède un pointeur vers cet arbre décrivant l'espace d'adressage du processus. Les nœuds de l'arbre sont des structures `_MMVAD`[1] :

```
struct _MMVAD, 10 elements, 0x28 bytes
   +0x000 StartingVpn      : Uint4B
```

[1] En fait, il existe 3 types de structures pour représenter les nœuds : `_MMVAD`, `_MMVAD_SHORT` et `_MMVAD_LONG`. La seule manière de les différencier consiste à regarder le tag du pool contenant la structure.

```
   +0x004 EndingVpn        : Uint4B
   +0x008 Parent          : Ptr32 to struct _MMVAD, 10 elements, 0x28 bytes
   +0x00c LeftChild       : Ptr32 to struct _MMVAD, 10 elements, 0x28 bytes
   +0x010 RightChild      : Ptr32 to struct _MMVAD, 10 elements, 0x28 bytes
   +0x014 u                : union __unnamed, 2 elements, 0x4 bytes
   +0x018 ControlArea     : Ptr32 to struct _CONTROL_AREA, 13 elements, 0x30 bytes
   +0x01c FirstPrototypePte : Ptr32 to struct _MMPTE, 1 element, 0x4 bytes
   +0x020 LastContiguousPte : Ptr32 to struct _MMPTE, 1 element, 0x4 bytes
   +0x024 u2               : union __unnamed, 2 elements, 0x4 bytes
```

Les champs `StartingVpn` et `EndingVpn` représentent la zone d'adresses virtuelles qui est concernée par la structure. Leur valeur s'exprime en multiples de la taille d'une page (`0x1000` octets).

Les propriétés de la zone de mémoire sont stockées dans une structure appelée `_MMVAD_FLAGS` (il s'agit d'un champ `u` de la structure précédente) :

```
struct _MMVAD_FLAGS, 10 elements, 0x4 bytes
   +0x000 CommitCharge     : Bitfield Pos 0, 19 Bits
   +0x000 PhysicalMapping  : Bitfield Pos 19, 1 Bit
   +0x000 ImageMap         : Bitfield Pos 20, 1 Bit
   +0x000 UserPhysicalPages : Bitfield Pos 21, 1 Bit
   +0x000 NoChange         : Bitfield Pos 22, 1 Bit
   +0x000 WriteWatch       : Bitfield Pos 23, 1 Bit
   +0x000 Protection       : Bitfield Pos 24, 5 Bits
   +0x000 LargePages       : Bitfield Pos 29, 1 Bit
   +0x000 MemCommit        : Bitfield Pos 30, 1 Bit
   +0x000 PrivateMemory    : Bitfield Pos 31, 1 Bit
```

Pour les structures différentes de `_MMVAD_SHORT`, il existe d'autres flags contenus dans le champ `u2` :

```
struct _MMVAD_FLAGS2, 9 elements, 0x4 bytes
   +0x000 FileOffset       : Bitfield Pos 0, 24 Bits
   +0x000 SecNoChange      : Bitfield Pos 24, 1 Bit
   +0x000 OneSecured       : Bitfield Pos 25, 1 Bit
   +0x000 MultipleSecured  : Bitfield Pos 26, 1 Bit
   +0x000 ReadOnly         : Bitfield Pos 27, 1 Bit
   +0x000 LongVad          : Bitfield Pos 28, 1 Bit
   +0x000 ExtendableFile   : Bitfield Pos 29, 1 Bit
   +0x000 Inherit          : Bitfield Pos 30, 1 Bit
   +0x000 CopyOnWrite      : Bitfield Pos 31, 1 Bit
```

Un processus s'alloue de la mémoire pour 2 raisons. La première est évidente, c'est pour son usage propre ; il s'agit alors de mémoire dite *Private*. La seconde se produit lorsque le processus a besoin de mapper une partie d'un fichier en mémoire, ce que le noyau appelle une *section*. Dans ce cas, la mémoire est dite *Shared*.

Lorsqu'il s'agit d'un fichier mappé en mémoire, il est possible de récupérer d'autres informations. Le champ `ControlArea` pointe sur une structure `_CONTROL_AREA`.

```
struct _CONTROL_AREA, 13 elements, 0x30 bytes
   +0x000 Segment        : Ptr32 to struct _SEGMENT, 14 elements, 0x40 bytes
    +0x004 DereferenceList  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x00c NumberOfSectionReferences : Uint4B
    +0x010 NumberOfPfnReferences : Uint4B
    +0x014 NumberOfMappedViews : Uint4B
    +0x018 NumberOfSubsections : Uint2B
    +0x01a FlushInProgressCount : Uint2B
    +0x01c NumberOfUserReferences : Uint4B
    +0x020 u                : union __unnamed, 2 elements, 0x4 bytes
   +0x024 FilePointer     : Ptr32 to struct _FILE_OBJECT, 27 elements, 0x70 bytes
   +0x028 WaitingForDeletion : Ptr32 to struct _EVENT_COUNTER, 3 elements, 0x18 bytes
    +0x02c ModifiedWriteCount : Uint2B
    +0x02e NumberOfSystemCacheViews : Uint2B
```

Cette structure contient un pointeur vers une structure `_SEGMENT`. Nous avons déjà parlé de cette structure. Il s'agit de celle qui contient les PTE prototypes dédiés au fichier partagé. Lorsque nous tombons sur un PTE rempli de zéros (ie. invalide et dans un état inconnu), nous pouvons retrouver suffisamment d'informations dans cette structure. Les VAD décrivent les adresses qui ont été réservées (par forcément committées). Si l'adresse est valide, nous pouvons avoir une chance de retrouver l'information voulue.

La structure `_CONTROL_AREA` contient aussi un pointeur vers un structure `_FILE_OBJECT`. Cette structure est utilisée pour représenter les fichiers.

```
struct _FILE_OBJECT, 27 elements, 0x70 bytes
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject    : Ptr32 to struct _DEVICE_OBJECT, 25 elements, 0xb8 bytes
   +0x008 Vpb              : Ptr32 to struct _VPB, 9 elements, 0x58 bytes
   +0x00c FsContext        : Ptr32 to Void
   +0x010 FsContext2       : Ptr32 to Void
  +0x014 SectionObjectPointer : Ptr32 to struct _SECTION_OBJECT_POINTERS, 3 elements, 0xc bytes
   +0x018 PrivateCacheMap  : Ptr32 to Void
   +0x01c FinalStatus      : Int4B
  +0x020 RelatedFileObject : Ptr32 to struct _FILE_OBJECT, 27 elements, 0x70 bytes
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B
```

```
   +0x030 FileName         : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x038 CurrentByteOffset : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x040 Waiters          : Uint4B
   +0x044 Busy             : Uint4B
   +0x048 LastLock         : Ptr32 to Void
   +0x04c Lock             : struct _KEVENT, 1 elements, 0x10 bytes
   +0x05c Event            : struct _KEVENT, 1 elements, 0x10 bytes
  +0x06c CompletionContext : Ptr32 to struct _IO_COMPLETION_CONTEXT, 2 elements, 0x8 bytes
```

Nous obtenons par exemple le nom du fichier en regardant la valeur du champ `FileName`.

Une figure vaut mieux qu'un long discours. Nous allons illustrer avec la figure 10 les structures impliquées dans le mapping de l'exécutable `lsass.exe`.

Grâce à la structure `_EPROCESS`, nous retrouvons l'adresse du nœud racine de l'arbre `VAD`. Nous savons que l'exécutable est mappé à partir de l'adresse virtuelle `0x1000000`. Nous parcourons l'arbre des VAD jusqu'à trouver le VAD responsable de cette plage d'adresses. En suivant les structures `_CONTROL_AREA` et `_FILE_OBJECT`, nous vérifions bien qu'il s'agit du fichier `lsass.exe`.

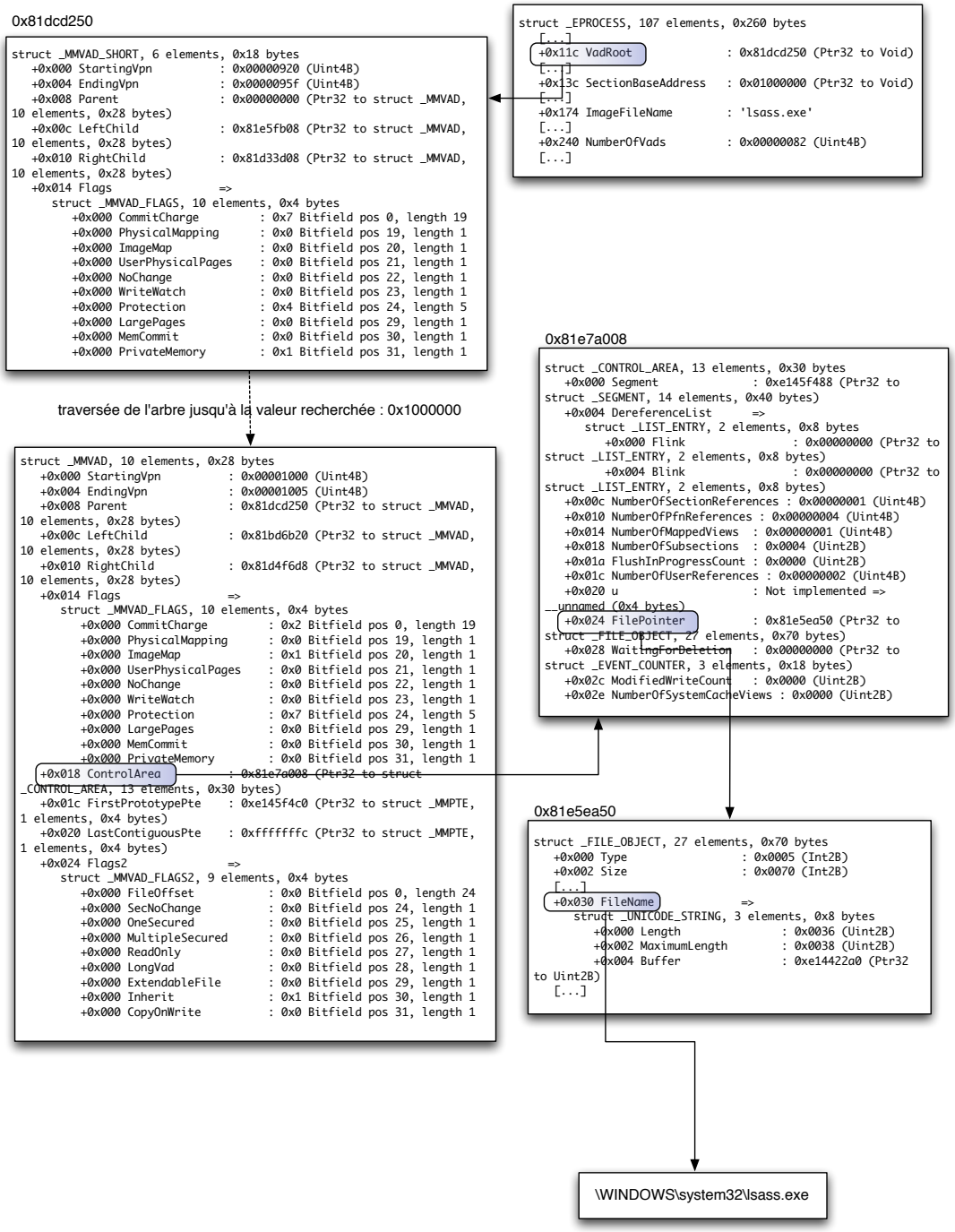


**Fig. 10.** VAD lsass.exe

## 4 Applications

Nous avons vu comment accéder à la mémoire. Nous avons vu comment donner du sens à cette masse de données. Maintenant que nous sommes en mesure de reconstituer l'adressage virtuel, nous allons vous convier à un petit voyage au sein du noyau de Windows. Nous n'avons pas la place, ni la prétention, de décrire exhaustivement les structures internes du noyau. Nous allons présenter celles qui nous ont semblées les plus pertinentes. Pour ceux qui veulent plus d'informations sur le fonctionnement du noyau, il existe le très bon livre de Mark E. Russinovich et David A. Solomon [8].

Nous verrons dans un premier temps comment retrouver des tables très importantes pour le processeur et pour le noyau. Nous pouvons nous en servir comme détecteur de rootkits primaire en vérifiant par exemple que les pointeurs de fonctions sont cohérents. Ensuite nous montrerons comment réaliser des clones d'applications célèbres du monde Windows. Nous verrons par exemple une manière d'implémenter *WinObj*, *regedit* ou encore *ProcessExplorer* en n'interprétant que la mémoire. Nous finirons par une brève discussion sur la possibilité d'exécuter du code sur la machine cible. Cela sous-entend bien sûr que nous avons un accès en lecture-écriture à la mémoire (via le FireWire par exemple).

### 4.1 À la recherche des tables du processeur

En premier lieu, nous allons étudier certaines structures du noyau spécifiques au processeur. Nous allons pouvoir récupérer par exemple la GDT et l'IDT ainsi que des pointeurs sur des structures clés du noyau.

Le PCR (Processor Control Region) et le PCRB (Processor Control Block) sont utilisés par le noyau et par HAL pour obtenir des informations spécifiques au matériel et à l'architecture. Ces structures contiennent des données sur l'état de chaque processeur du système. Avec WinDbg, il suffit d'utiliser la commande `!pcr` pour observer le contenu du PCR et la commande `!prcb` pour celui du PRCB.

Le noyau utilise la structure `_KPCR` pour stocker le PCR.

```
struct _KPCR, 27 elements, 0xd70 bytes
   +0x000 NtTib            : struct _NT_TIB, 8 elements, 0x1c bytes
   +0x01c SelfPcr         : Ptr32 to struct _KPCR, 27 elements, 0xd70 bytes
   +0x020 Prcb           : Ptr32 to struct _KPRCB, 91 elements, 0xc50 bytes
   +0x024 Irql             : UChar
   +0x028 IRR              : Uint4B
   +0x02c IrrActive        : Uint4B
   +0x030 IDR              : Uint4B
   +0x034 KdVersionBlock   : Ptr32 to Void
   +0x038 IDT            : Ptr32 to struct _KIDTENTRY, 4 elements, 0x8 bytes
   +0x03c GDT            : Ptr32 to struct _KGDTENTRY, 3 elements, 0x8 bytes
   +0x040 TSS             : Ptr32 to struct _KTSS, 35 elements, 0x20ac bytes
   +0x044 MajorVersion     : Uint2B
   +0x046 MinorVersion     : Uint2B
   +0x048 SetMember        : Uint4B
   +0x04c StallScaleFactor : Uint4B
   +0x050 DebugActive      : UChar
   +0x051 Number           : UChar
   +0x052 Spare0           : UChar
   +0x053 SecondLevelCacheAssociativity : UChar
   +0x054 VdmAlert         : Uint4B
   +0x058 KernelReserved   : [14] Uint4B
   +0x090 SecondLevelCacheSize : Uint4B
   +0x094 HalReserved      : [16] Uint4B
   +0x0d4 InterruptMode    : Uint4B
   +0x0d8 Spare1           : UChar
   +0x0dc KernelReserved2  : [17] Uint4B
   +0x120 PrcbData         : struct _KPRCB, 91 elements, 0xc50 bytes
```

Cette structure a plusieurs champs utiles pour l'analyse de la mémoire. En particulier, les adresses de l'IDT (Interrupt Descriptor Table) et de la GDT (Global Descriptor Table) y figurent ainsi qu'un pointeur vers une zone de données utilisée par le débogueur : `KdVersionBlock`. Nous verrons un peu plus loin l'intérêt de cette zone. Le PRCB y figure aussi sous la forme d'une structure `_KPRCB`.

```
struct _KPRCB, 91 elements, 0xc50 bytes
   ...
  +0x004 CurrentThread    : Ptr32 to struct _KTHREAD, 73 elements, 0x1c0 bytes
  +0x008 NextThread       : Ptr32 to struct _KTHREAD, 73 elements, 0x1c0 bytes
  +0x00c IdleThread       : Ptr32 to struct _KTHREAD, 73 elements, 0x1c0 bytes
   ...
  +0x01c ProcessorState   : struct _KPROCESSOR_STATE, 2 elements, 0x320 bytes
   ...
```

Le PRCB est une extension du PCR. En particulier, il comprend une structure appelée `KPROCESSOR_STATE` qui stocke tous les registres du processeur. Deux registres qui nous intéressent y figurent : le registre `idtr` et le registre `gdtr`.

```
struct _KPROCESSOR_STATE, 2 elements, 0x320 bytes
   +0x000 ContextFrame     : struct _CONTEXT, 25 elements, 0x2cc bytes
  +0x2cc SpecialRegisters : struct _KSPECIAL_REGISTERS, 15 elements, 0x54 bytes

struct _KSPECIAL_REGISTERS, 15 elements, 0x54 bytes
   +0x000 Cr0              : Uint4B
   +0x004 Cr2              : Uint4B
   +0x008 Cr3              : Uint4B
   +0x00c Cr4              : Uint4B
   +0x010 KernelDr0        : Uint4B
   +0x014 KernelDr1        : Uint4B
   +0x018 KernelDr2        : Uint4B
   +0x01c KernelDr3        : Uint4B
   +0x020 KernelDr6        : Uint4B
   +0x024 KernelDr7        : Uint4B
   +0x028 Gdtr             : struct _DESCRIPTOR, 3 elements, 0x8 bytes
   +0x030 Idtr             : struct _DESCRIPTOR, 3 elements, 0x8 bytes
   +0x038 Tr               : Uint2B
   +0x03a Ldtr             : Uint2B
   +0x03c Reserved         : [6] Uint4B
```

Ces registres nous fournissent l'adresse et le nombre d'entrées de la GDT et de l'IDT.

Lorsque nous avons parlé du PCR, nous avons mentionné la présence d'un pointeur appelé `KdVersionBlock`. Cette structure est décrite dans deux articles sur http://rootkit.com. Le premier, écrit par Edgar Barbosa [9], comporte des imprécisions qui sont corrigées dans le deuxième article, écrit par Alex Ionescu [10].

`KdVersionBlock` n'est pas documenté directement par WinDbg. Cependant grâce à l'article d'Alex Ionescu, nous déduisons qu'il s'agit d'une structure `DBGKD_GET_VERSION64`.

```
struct _DBGKD_GET_VERSION64, 13 elements, 0x28 bytes
   +0x000 MajorVersion     : 0xf
   +0x002 MinorVersion     : 0xa28
```

```
+0x004 ProtocolVersion  : 6
+0x006 Flags            : 2
+0x008 MachineType      : 0x14c
+0x00a MaxPacketType    : 0xc ''
+0x00b MaxStateChange   : 0x3 ''
+0x00c MaxManipulate    : 0x2d '-'
+0x00d Simulation       : 0 ''
+0x00e Unused           : [1] 0
+0x010 KernBase         : 0xffffffff`804d7000
+0x018 PsLoadedModuleList : 0xffffffff`8055a620
+0x020 DebuggerDataList : 0xffffffff`80691e74
```

Des informations très intéressantes sont présentes dans cette structure. Tout d'abord, nous trouvons l'adresse de chargement du noyau (`KernBase`) et un pointeur (`PsLoadedModuleList`) sur une liste doublement chaînée contenant les modules chargés en mémoire. Nous expliquerons plus loin comment naviguer au sein de cette liste.

Mais ce qui est le plus utile est le champ `DebuggerDataList`. Il s'agit d'un pointeur sur une structure non documentée par WinDbg. Cependant elle est documentée dans le fichier `wdbgexts.h` situé dans le répertoire d'include de WinDbg. Il s'agit d'une structure `KDDEBUGGER_DATA64` (voir listing 1.1). Nous avons juste mis les premières lignes du fichier :

```
typedef struct _KDDEBUGGER_DATA64 {
    DBGKD_DEBUG_DATA_HEADER64 Header;
    //
    // Base address of kernel image
    //
    ULONG64   KernBase;
    //
    // DbgBreakPointWithStatus is a function which takes an argument
    // and hits a breakpoint.  This field contains the address of the
    // breakpoint instruction.  When the debugger sees a breakpoint
    // at this address, it may retrieve the argument from the first
    // argument register, or on x86 the eax register.
    //
    ULONG64   BreakpointWithStatus;       // address of breakpoint
    //
    // Address of the saved context record during a bugcheck
    //
    // N.B. This is an automatic in KeBugcheckEx's frame, and
    // is only valid after a bugcheck.
    //
    ULONG64   SavedContext;
    //
    // help for walking stacks with user callbacks:
    //
    // The address of the thread structure is provided in the
    // WAIT_STATE_CHANGE packet.  This is the offset from the base of
    // the thread structure to the pointer to the kernel stack frame
    // for the currently active usermode callback.
    //
    USHORT  ThCallbackStack;            // offset in thread data
    //
    // these values are offsets into that frame:
    //
```

```
    USHORT  NextCallback;               // saved pointer to next callback
        frame
    USHORT  FramePointer;               // saved frame pointer
    //
    // pad to a quad boundary
    //
    USHORT  PaeEnabled:1;
    //
    // Address of the kernel callout routine.
    //
    ULONG64   KiCallUserMode;             // kernel routine
    //
    // Address of the usermode entry point for callbacks.
    //
    ULONG64   KeUserCallbackDispatcher;   // address in ntdll
    //
    // Addresses of various kernel data structures and lists
    // that are of interest to the kernel debugger.
    //
    ULONG64   PsLoadedModuleList;
    ULONG64   PsActiveProcessHead;
    ULONG64   PspCidTable;

    [...]

    } KDDEBUGGER_DATA64, *PKDDEBUGGER_DATA64;
```

**Listing 1.1.** le début de la structure KDDEBUGGER_DATA64

Le premier élément est une structure DBGKD_DEBUG_DATA_HEADER64 (voir le listing 1.2). KDDEBUGGER_DATA64 contient les adresses d'une grande quantité de variables non-exportées du noyau. Par exemple, nous pouvons citer :

- PsActiveProcessHead qui pointe sur le premier processus ;
- PspCidTable qui permet d'itérer sur les processus et les threads ;
- ObpRootDirectoryObject qui pointe sur l'objet racine du gestionnaire d'objets (nous verrons plus loin comment nous servir de cela).

```
//
// This structure is used by the debugger for all targets
// It is the same size as DBGKD_DATA_HEADER on all systems
//
typedef struct _DBGKD_DEBUG_DATA_HEADER64 {
    //
    // Link to other blocks
    //
    LIST_ENTRY64 List;
    //
    // This is a unique tag to identify the owner of the block.
    // If your component only uses one pool tag, use it for this, too.
    //
    ULONG           OwnerTag;
    //
    // This must be initialized to the size of the data block,
    // including this structure.
    //
    ULONG           Size;
} DBGKD_DEBUG_DATA_HEADER64, *PDBGKD_DEBUG_DATA_HEADER64;
```

**Listing 1.2.** la structure DBGKD_DEBUG_DATA_HEADER64

La structure KUSER_SHARED_DATA est mappée en 0xffdf0000 en espace noyau et en 0x7ffe0000 en espace utilisateur. Elle sert de moyen de communication

entre l'espace noyau et l'espace utilisateur. Elle contient plusieurs informations utiles sur la configuration du système.

```
struct _KUSER_SHARED_DATA, 39 elements, 0x338 bytes
   +0x000 TickCountLow     : Uint4B
   +0x004 TickCountMultiplier : Uint4B
   +0x008 InterruptTime    : struct _KSYSTEM_TIME, 3 elements, 0xc bytes
   +0x014 SystemTime       : struct _KSYSTEM_TIME, 3 elements, 0xc bytes
   +0x020 TimeZoneBias     : struct _KSYSTEM_TIME, 3 elements, 0xc bytes
   +0x02c ImageNumberLow   : Uint2B
   +0x02e ImageNumberHigh  : Uint2B
   +0x030 NtSystemRoot     : [260] Uint2B
   +0x238 MaxStackTraceDepth : Uint4B
   +0x23c CryptoExponent   : Uint4B
   +0x240 TimeZoneId       : Uint4B
   +0x244 Reserved2        : [8] Uint4B
   +0x264 NtProductType    : Enum _NT_PRODUCT_TYPE,  3 total enums
   +0x268 ProductTypeIsValid : UChar
   +0x26c NtMajorVersion   : Uint4B
   +0x270 NtMinorVersion   : Uint4B
   +0x274 ProcessorFeatures : [64] UChar
   +0x2b4 Reserved1        : Uint4B
   +0x2b8 Reserved3        : Uint4B
   +0x2bc TimeSlip         : Uint4B
  +0x2c0 AlternativeArchitecture : Enum _ALTERNATIVE_ARCHITECTURE_TYPE,  3 total enums
  +0x2c8 SystemExpirationDate : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x2d0 SuiteMask        : Uint4B
   +0x2d4 KdDebuggerEnabled : UChar
   +0x2d5 NXSupportPolicy  : UChar
   +0x2d8 ActiveConsoleId  : Uint4B
   +0x2dc DismountCount    : Uint4B
   +0x2e0 ComPlusPackage   : Uint4B
   +0x2e4 LastSystemRITEventTickCount : Uint4B
   +0x2e8 NumberOfPhysicalPages : Uint4B
   +0x2ec SafeBootMode     : UChar
   +0x2f0 TraceLogging     : Uint4B
   +0x2f8 TestRetInstruction : Uint8B
   +0x300 SystemCall       : Uint4B
   +0x304 SystemCallReturn : Uint4B
   +0x308 SystemCallPad    : [3] Uint8B
   +0x320 TickCount        : struct _KSYSTEM_TIME, 3 elements, 0xc bytes
   +0x320 TickCountQuad    : Uint8B
   +0x330 Cookie           : Uint4B
```

Tout ça est bien joli mais comment pouvons-nous retrouver ces structures dans la mémoire physique ? Il existe plusieurs moyens. En fait, il y a un PCR (et un KPRCB) par processeur. Sous Windows XP, la structure `_KPCR` correspondant au premier processeur est toujours mappée sur l'adresse virtuelle `0xffdff000`. En revanche, ceci n'est pas vrai sous Vista. Nous allons utiliser la

forme particulière de la structure pour la retrouver grâce à une signature accompagnée de quelques tests.

À l'offset `0x1c` (champ `SelfPcr`) du `_KPCR`, il y a un pointeur sur l'adresse virtuelle de la structure. De plus, à l'offset `0x20` (champ `Prcb`), il y a un pointeur sur la structure `_KPRCB` qui se situe à un offset de `0x120` par rapport au début.

```
struct _KPCR, 27 elements, 0xd70 bytes
[...]
   +0x01c SelfPcr          : Ptr32 to struct _KPCR, 27 elements, 0xd70 bytes
   +0x020 Prcb             : Ptr32 to struct _KPRCB, 91 elements, 0xc50 bytes
[...]
   +0x120 PrcbData          : struct _KPRCB, 91 elements, 0xc50 bytes
```

Il suffit donc de parser chaque page de la mémoire physique avec l'algorithme suivant :

- soit `pos` la position courante ;
- si la différence entre le pointeur à la position `pos+0x20` moins le pointeur à la position `pos+0x1c` est `0x120` et l'adresse physique du pointeur à la position `pos+0x1c` est égal à `pos` alors `pos` pointe sur le début du `_KPCR` ;
- sinon recommencer en itérant la position courante.

Sur le dump mémoire, nous trouvons que le PCR se situe à l'adresse physique `0x40000` et qu'il a pour adresse virtuelle `0xffdff000`.

Nous obtenons par la même occasion d'autres informations intéressantes : le KdVersionBlock a pour adresse virtuelle `0x8054c038`, l'IDT `0x8003f400` et la GDT `0x8003f000`.

La structure `_KPRCB` regorge d'informations utiles. Nous trouvons dans le désordre :

- la valeur des registres `Gdtr` et `Idtr` qui vont nous donner toutes les informations nécessaires pour récupérer l'IDT et la GDT comme nous le verrons par la suite ;
- un pointeur sur la structure `_KTHREAD` du thread courant (`0x81ec2778`) et un sur le thread `Idle` (`0x80558c20`).

La structure `KdVersionBlock` donne d'autres informations. Elle nous permet de trouver l'adresse du début du noyau (`0x804d7000`) ainsi qu'un pointeur sur la liste doublement chaînée (`PsLoadedModuleList`) des modules chargés (`0x8055a420`) et un pointeur sur des données de debug (`0x80691b74`).

Les données de debug sont contenues dans une liste chainée. Il suffit de parcourir cette liste jusqu'à trouver que le tag dans le header soit `KGDB`.

Nous déduisons de cette structure les adresses de certaines variables non-exportées importantes du noyau. Nous allons nous servir de `PsLoadedModuleList` (`0x823fc3b0`) et de `ObpRootDirectoryObject` (`0xe10001e8`) dans la suite.

Mais avant d'en arriver là, regardons comment obtenir l'IDT et la GDT.

Le processeur utilise deux tables très importantes : l'IDT (Interrupt Descriptor Table) et la GDT (Global Descriptor Table). La première est utilisée pour gérer les interruptions et la deuxième pour gérer la mémoire. Dans cette partie, nous allons présenter leurs structures ainsi que la méthode pour les récupérer.

Dans la partie précédente, nous avons pu retrouver la valeur des registres `gdtr` et `idtr` grâce au KPRCB. Chacun d'eux est stocké dans une structure `_DESCRIPTOR`.

```
struct _DESCRIPTOR, 3 elements, 0x8 bytes
         +0x000 Pad    : Uint2B
         +0x002 Limit  : Uint2B
         +0x004 Base   : Uint4B
```

Le champ `Base` donne l'adresse virtuelle de la table tandis que le champ `Limit` donne l'offset de la fin de la table.

Chacune des entrées a une taille de 8 octets. Nous en déduisons la taille de l'IDT dans notre exemple.

```
[...]
+0x030 Idtr                 =>
      struct _DESCRIPTOR, 3 elements, 0x8 bytes
         +0x000 Pad                  : 0x0000 (Uint2B)
         +0x002 Limit                : 0x07ff (Uint2B)
         +0x004 Base                 : 0x8003f400 (Uint4B)
[...]
```

$(0x7ff+1)/8 = 256$, il y a donc 256 entrées dans l'IDT. Pour avoir le format exact, il suffit de se reporter aux manuels Intel.

Pour la GDT, le principe est exactement le même.

```
[...]
+0x028 Gdtr                 =>
      struct _DESCRIPTOR, 3 elements, 0x8 bytes
         +0x000 Pad                  : 0x0000 (Uint2B)
         +0x002 Limit                : 0x03ff (Uint2B)
         +0x004 Base                 : 0x8003f000 (Uint4B)
[...]
```

Et nous en déduisons que la GDT possède 128 entrées.

### 4.2 Appels systèmes

Auparavant, le noyau Windows utilisait l'interruption `0x2e` pour gérer les appels systèmes. L'entrée de l'IDT pointait sur la routine responsable des appels systèmes. Le numéro de l'appel système était stocké dans le registre `eax`. Le registre `ebx` pointait sur la liste des paramètres à passer à l'appel système.

Avec l'arrivée des processeurs Pentium II, le noyau utilise maintenant l'instruction `SYSENTER` pour sa gestion des appels systèmes. L'adresse du gestionnaire est stockée au démarrage du système dans un registre particulier utilisé par cette instruction.

Les registres utilisés sont appelés `MSR` (Model Specific Registers) et ceux qui nous intéressent sont situés aux offsets `0x174`, `0x175` et `0x176`. Le premier registre, appelé `SYSENTER_CS_MSR`, stocke le sélecteur du segment où se situe le gestionnaire. Le deuxième registre, appelé `SYSENTER_ESP_MSR`, contient la valeur du registre `esp` qui sera chargée après l'instruction tandis que le dernier registre, `SYSENTER_EIP_MSR`, contient l'adresse du gestionnaire.

En utilisant WinDbg, nous nous assurons de la véracité de ces propos :

```
lkd> rdmsr 174
msr[174] = 00000000‘00000008
lkd> rdmsr 175
msr[175] = 00000000‘f8978000
lkd> rdmsr 176
msr[176] = 00000000‘804de6f0
```

L'adresse contenue dans le registre `SYSENTER_EIP_MSR` est bien celle du gestionnaire d'appels systèmes :

```
lkd> u 804de6f0
nt!KiFastCallEntry:
804de6f0 b923000000      mov     ecx,23h
804de6f5 6a30            push    30h
804de6f7 0fa1            pop     fs
804de6f9 8ed9            mov     ds,cx
804de6fb 8ec1            mov     es,cx
804de6fd 8b0d40f0dfff    mov     ecx,dword ptr ds:[0FFDFF040h]
804de703 8b6104          mov     esp,dword ptr [ecx+4]
804de706 6a23            push    23h
```

Lorsque l'instruction `SYSTENTER` est exécutée, l'exécution passe en mode noyau et le gestionnaire est appelé. Le numéro de l'appel système est stocké dans le registre `eax`, la liste des arguments dans le registre `edx`. Pour revenir en mode utilisateur, le gestionnaire utilise l'instruction `SYSEXIT` sauf lorsque le processeur est en single-step, il utilise alors l'instruction `IRETD`. Sur les processeurs AMD supérieurs aux K6, Windows utilise les instructions `SYSCALL` et `SYSRET`

qui sont similaires aux instructions Intel.

Le code pour exécuter un appel système suit toujours le même schéma. Par exemple, lorsque nous utilisons `CreateFile`, les instructions suivantes sont exécutées :

```
ntdll!NtCreateFile:
7c91d682 b825000000      mov     eax,25h
7c91d687 ba0003fe7f     mov    edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c91d68c ff12            call    dword ptr [edx]
7c91d68e c22c00          ret     2Ch
```

Nous remarquons le numéro de l'appel système (`0x25`) et l'adresse du gestionnaire des appels systèmes (`0x7ffe0300`).

```
kd> dd 7ffe0300 L1
7ffe0300  7c91eb8b
```

Nous avons déjà rencontré cette adresse lorsque nous avons examiné la zone `KUserSharedData`.

```
ntdll!_KUSER_SHARED_DATA
   ...
   +0x300 SystemCall       : 0x7c91eb8b
   ...
```

Le gestionnaire a le code suivant :

```
ntdll!KiFastSystemCall:
7c91eb8b 8bd4            mov     edx,esp
7c91eb8d 0f34            sysenter
```

Nous retrouvons donc bien notre instruction `SYSTENTER`. La table des appels systèmes est similaire à l'IDT, il s'agit d'une table où chaque entrée pointe sur la fonction responsable de l'appel système.

Chaque thread contient un pointeur sur la table des appels systèmes. Ce pointeur, appelé `ServiceTable`, se situe dans la structure `KTHREAD`.

```
struct _KTHREAD, 73 elements, 0x1c0 bytes
   ...
   +0x0e0 ServiceTable     : 0x80559640
   ...
```

Le noyau utilise 2 tables de descripteurs (`SERVICE_DESCRIPTOR_TABLE`) pour les appels systèmes. Chaque table de descripteurs peut contenir jusqu'à 4 tables de services (`SYSTEM_SERVICE_TABLE`).

La première table de descripteurs, `KeServiceDescriptorTable`, contient les appels systèmes implémentés dans `ntoskrnl.exe`.

La deuxième table, `KeServiceDescriptorTableShadow`, contient les appels systèmes du noyau plus les services GDI et USER implémentés dans `win32k.sys`.

Chaque table est stockée dans une structure non documentée que nous pouvons représenter comme ceci :

```
typedef struct _SYSTEM_SERVICE_TABLE
{
    PNTPROC ServiceTable;            // array of entry points
    PDWORD CounterTable;             //  array of usage counters
    DWORD     ServiceLimit ;         //  number of table entries
    PBYTE     ArgumentTable;         //  array of byte counts
)
SYSTEM_SERVICE_TABLE
```

Toujours avec WinDbg, nous pouvons regarder les tables des appels systèmes. Nous commençons par la table `KeServiceDescriptorTable` :

```
kd> dds KeServiceDescriptorTable L4
80559680  804e26a8 nt!KiServiceTable
80559684  00000000
80559688  0000011c
8055968c  80512eb8 nt!KiArgumentTable
```

Nous voyons la table contenant les adresses des appels systèmes ainsi que le tableau contenant le nombre d'arguments de chaque appel système. Nous regardons le début de la table des appels systèmes :

```
kd> dds KiServiceTable L4
804e26a8  8057f302 nt!NtAcceptConnectPort
804e26ac  80578b8c nt!NtAccessCheck
804e26b0  8058a7ae nt!NtAccessCheckAndAuditAlarm
804e26b4  8058f7e4 nt!NtAccessCheckByType
[...]
```

Nous faisons la même chose pour la table `KeServiceDescriptorTableShadow` :

```
kd> dds KeServiceDescriptorTableShadow L8
80559640  804e26a8 nt!KiServiceTable
80559644  00000000
80559648  0000011c
8055964c  80512eb8 nt!KiArgumentTable
80559650  bf999280 win32k!W32pServiceTable
80559654  00000000
80559658  0000029b
8055965c  bf999f90 win32k!W32pArgumentTable
```

Et nous en profitons pour regarder les appels systèmes contenus dans `win32k.sys` :

```
kd> dds win32k!W32pServiceTable L4
bf999280  bf935662 win32k!NtGdiAbortDoc
bf999284  bf947213 win32k!NtGdiAbortPath
bf999288  bf87a92d win32k!NtGdiAddFontResourceW
bf99928c  bf93eddc win32k!NtGdiAddRemoteFontToDC
[...]
```

### 4.3 Modules

Après avoir vu comment obtenir la GDT, l'IDT et la SSDT, nous allons nous intéresser maintenant à la liste des modules chargés en mémoire.

Dans la partie précédente, nous avons trouvé que le noyau maintient une variable globale non exportée (`PsLoadedModuleList`) pointant sur la liste des modules (*drivers*) chargés. En fait, la valeur de ce pointeur est l'adresse d'une structure `_LDR_DATA_TABLE_ENTRY` :

```
struct _LDR_DATA_TABLE_ENTRY, 18 elements, 0x50 bytes
   +0x000 InLoadOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x008 InMemoryOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
  +0x010 InInitializationOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x018 DllBase          : Ptr32 to Void
   +0x01c EntryPoint       : Ptr32 to Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x02c BaseDllName      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x03c SectionPointer   : Ptr32 to Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 to Void
   +0x048 EntryPointActivationContext : Ptr32 to Void
   +0x04c PatchInformation : Ptr32 to Void
```

Au début de la structure se situent 3 listes doublements chaînées. Chacune de ces listes contient la liste des modules, la seule différence se faisant sur l'ordre de parcours. Chaque élément de la liste est aussi une structure `_LDR_DATA_TABLE_ENTRY`. Nous obtenons d'autres informations utiles comme le point d'entrée, le nom du module etc.

Dans notre exemple, nous obtenons la liste suivante (abrégée pour des raisons de taille) :

- `\WINDOWS\system32\ntoskrnl.exe` : `0x804d7000` ;
- `\WINDOWS\system32\hal.dll` : `0x806ec000` :
- `\WINDOWS\system32\KDCOM.DLL` : `0xf8a51000` :

...

- `\SystemRoot\System32\DRIVERS\USBSTOR.SYS` : `0xf8881000` ;
- `\SystemRoot\system32\drivers\kmixer.sys` : 0xf56ed000.

Le noyau utilise d'autres structures pour représenter les *drivers* et les *devices*. Ces structures sont utiles car elles contiennent des pointeurs de fonctions

souvent hookées par les rootkits.

Le noyau utilise une structure appelée `_DRIVER_OBJECT` pour représenter ces drivers.

```
struct _DRIVER_OBJECT, 15 elements, 0xa8 bytes
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 to struct _DEVICE_OBJECT, 25 elements, 0xb8 bytes
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 to Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 to Void
   +0x018 DriverExtension  : Ptr32 to struct _DRIVER_EXTENSION, 6 elements, 0x1c bytes
   +0x01c DriverName       : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x024 HardwareDatabase : Ptr32 to struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x028 FastIoDispatch   : Ptr32 to struct _FAST_IO_DISPATCH, 28 elements, 0x70 bytes
   +0x02c DriverInit       : Ptr32 to     long
   +0x030 DriverStartIo    : Ptr32 to     void
   +0x034 DriverUnload     : Ptr32 to     void
   +0x038 MajorFunction    : [28] Ptr32 to     long
```

À la fin de cette structure se situe un tableau de pointeurs de fonctions (champ `MajorFunction`). Pour quiconque ayant déjà programmé des drivers sous Windows, ces fonctions correspondent aux `IRP_MJ_*` (IRP = I/O Request Packet). Toujours dans notre exemple, nous obtenons pour le driver `usbuhci` les fonctions suivantes :

```
Driver: \Driver\usbuhci (0xf88a1000)
Devices:
0x81ead480 0x81ea4028
Dispatch routines
  [0x00] IRP_MJ_CREATE                      0xf827609a
  [0x01] IRP_MJ_CREATE_NAMED_PIPE           0x805031be
  [0x02] IRP_MJ_CLOSE                       0xf827609a
  [0x03] IRP_MJ_READ                        0x805031be
  [0x04] IRP_MJ_WRITE                       0x805031be
  [0x05] IRP_MJ_QUERY_INFORMATION           0x805031be
  [0x06] IRP_MJ_SET_INFORMATION             0x805031be
  [0x07] IRP_MJ_QUERY_EA                    0x805031be
  [0x08] IRP_MJ_SET_EA                      0x805031be
  [0x09] IRP_MJ_FLUSH_BUFFERS               0x805031be
  [0x0a] IRP_MJ_QUERY_VOLUME_INFORMATION    0x805031be
  [0x0b] IRP_MJ_SET_VOLUME_INFORMATION      0x805031be
  [0x0c] IRP_MJ_DIRECTORY_CONTROL           0x805031be
  [0x0d] IRP_MJ_FILE_SYSTEM_CONTROL         0x805031be
  [0x0e] IRP_MJ_DEVICE_CONTROL              0xf827609a
  [0x0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     0xf827609a
  [0x10] IRP_MJ_SHUTDOWN                    0x805031be
  [0x11] IRP_MJ_LOCK_CONTROL                0x805031be
```

```
  [0x12] IRP_MJ_CLEANUP                   0x805031be
  [0x13] IRP_MJ_CREATE_MAILSLOT           0x805031be
  [0x14] IRP_MJ_QUERY_SECURITY            0x805031be
  [0x15] IRP_MJ_SET_SECURITY              0x805031be
  [0x16] IRP_MJ_POWER                     0xf827609a
  [0x17] IRP_MJ_SYSTEM_CONTROL            0xf827609a
  [0x18] IRP_MJ_DEVICE_CHANGE             0x805031be
  [0x19] IRP_MJ_QUERY_QUOTA               0x805031be
  [0x1a] IRP_MJ_SET_QUOTA                 0x805031be
  [0x1b] IRP_MJ_PNP                       0xf827609a
```

Beaucoup de ces pointeurs de fonctions ont comme valeur `0x805031be`, il s'agit simplement des fonctions qui ne sont pas implémentées par le driver.

D'autres pointeurs de fonctions se trouvent dans la structure `_FAST_IO_DISPATCH` :

```
struct _FAST_IO_DISPATCH, 28 elements, 0x70 bytes
   +0x000 SizeOfFastIoDispatch : Uint4B
   +0x004 FastIoCheckIfPossible : Ptr32 to     unsigned char
   +0x008 FastIoRead       : Ptr32 to     unsigned char
   +0x00c FastIoWrite      : Ptr32 to     unsigned char
   +0x010 FastIoQueryBasicInfo : Ptr32 to     unsigned char
   +0x014 FastIoQueryStandardInfo : Ptr32 to     unsigned char
   +0x018 FastIoLock       : Ptr32 to     unsigned char
   +0x01c FastIoUnlockSingle : Ptr32 to     unsigned char
   +0x020 FastIoUnlockAll  : Ptr32 to     unsigned char
   +0x024 FastIoUnlockAllByKey : Ptr32 to     unsigned char
   +0x028 FastIoDeviceControl : Ptr32 to     unsigned char
   +0x02c AcquireFileForNtCreateSection : Ptr32 to     void
   +0x030 ReleaseFileForNtCreateSection : Ptr32 to     void
   +0x034 FastIoDetachDevice : Ptr32 to     void
   +0x038 FastIoQueryNetworkOpenInfo : Ptr32 to     unsigned char
   +0x03c AcquireForModWrite : Ptr32 to     long
   +0x040 MdlRead          : Ptr32 to     unsigned char
   +0x044 MdlReadComplete  : Ptr32 to     unsigned char
   +0x048 PrepareMdlWrite  : Ptr32 to     unsigned char
   +0x04c MdlWriteComplete : Ptr32 to     unsigned char
   +0x050 FastIoReadCompressed : Ptr32 to     unsigned char
   +0x054 FastIoWriteCompressed : Ptr32 to     unsigned char
   +0x058 MdlReadCompleteCompressed : Ptr32 to     unsigned char
   +0x05c MdlWriteCompleteCompressed : Ptr32 to     unsigned char
   +0x060 FastIoQueryOpen  : Ptr32 to     unsigned char
   +0x064 ReleaseForModWrite : Ptr32 to     long
   +0x068 AcquireForCcFlush : Ptr32 to     long
   +0x06c ReleaseForCcFlush : Ptr32 to     long
```

La question intéressante maintenant est la suivante : comment trouvons-nous ces structures ?

Dans la partie suivante, nous allons voir le fonctionnement de *l'object manager*. Dans le répertoire `\Driver` se trouve l'ensemble des structures `_DRIVER_OBJECT`. Il suffit d'énumérer ce répertoire pour avoir la liste des drivers ainsi que les adresses de leurs pointeurs de fonctions.

Pour retrouver le module à partir du driver, il suffit de suivre le champ `DriverSection` de la structure `_DRIVER_OBJECT`. Il pointe vers une structure `_LDR_DATA_TABLE_ENTRY`.

### 4.4 WinObj 101

Dans cette partie, nous allons décrire les structures internes utilisées par le gestionnaire d'objets et ainsi poser les bases de la réalisation d'un clone de l'utilitaire *WinObj* de SysInternals [11].

Le noyau Windows utilise un modèle objet pour gérer ses structures internes, ceci étant réalisé par une partie de l'exécutif appelé «object manager».

Nous n'allons pas décrire en détail le but ni l'implémentation de l'object manager mais simplement expliquer comment nous pouvons retrouver ces structures en mémoire et comment elles interagissent entre elles.

La structure principale s'appelle `_OBJECT_HEADER`, et a la forme suivante :

```
struct _OBJECT_HEADER, 12 elements, 0x20 bytes
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 to Void
  +0x008 Type          : Ptr32 to struct _OBJECT_TYPE, 12 elements, 0x190 bytes
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
   +0x00f Flags            : UChar
  +0x010 ObjectCreateInfo : Ptr32 to struct _OBJECT_CREATE_INFORMATION, 10 elements, 0x30 bytes
   +0x010 QuotaBlockCharged : Ptr32 to Void
   +0x014 SecurityDescriptor : Ptr32 to Void
   +0x018 Body             : struct _QUAD, 1 elements, 0x8 bytes
```

Il y a plusieurs champs intéressants. Les champs `PointerCount` et `HandleCount` parlent d'eux-mêmes. Au contraire les champs `NameInfoOffset`, `HandleInfoOffset` et `QuotaInfoOffset` sont moins parlants. Le premier, `NameInfoOffset` est égal à la valeur (en octets) qu'il faut soustraire à l'adresse de la structure. A cette adresse, nous trouvons une structure appelée `_OBJECT_HEADER_NAME_INFO` qui a la forme suivante :

```
struct _OBJECT_HEADER_NAME_INFO, 3 elements, 0x10 bytes
  +0x000 Directory      : Ptr32 to struct _OBJECT_DIRECTORY, 6 elements, 0xa4 bytes
   +0x004 Name            : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x00c QueryReferences  : Uint4B
```

Le champ `Directory` est un pointeur sur une structure appelée `_OBJECT_DIRECTORY` qui est en fait un conteneur d'objets. L'object manager range les objets de manière hiérarchique. Nous verrons plus loin le principe de rangement. Nous retrouvons le nom de l'objet grâce au champ `Name`. Si le champ `Directory` est nul alors il s'agit de l'objet racine. L'adresse de l'objet racine peut être trouvée grâce au champ `ObpRootDirectoryObject` de la structure `DebugData`.

Revenons à la structure `_OBJECT_HEADER`. Un autre champ intéressant est le champ `Type` qui pointe sur une structure `_OBJECT_TYPE` :

```
struct _OBJECT_TYPE, 12 elements, 0x190 bytes
   +0x000 Mutex            : struct _ERESOURCE, 13 elements, 0x38 bytes
   +0x038 TypeList         : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x040 Name             : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x048 DefaultObject    : Ptr32 to Void
   +0x04c Index            : Uint4B
   +0x050 TotalNumberOfObjects : Uint4B
   +0x054 TotalNumberOfHandles : Uint4B
   +0x058 HighWaterNumberOfObjects : Uint4B
   +0x05c HighWaterNumberOfHandles : Uint4B
   +0x060 TypeInfo       : struct _OBJECT_TYPE_INITIALIZER, 20 elements, 0x4c bytes
   +0x0ac Key              : Uint4B
   +0x0b0 ObjectLocks      : [4] struct _ERESOURCE, 13 elements, 0x38 bytes
```

En dehors de champs explicites (`Name` par exemple), elle contient un pointeur vers une structure `_OBJECT_TYPE_INITIALIZER` qui est très intéressante pour un attaquant comme nous allons le voir par la suite.

```
struct _OBJECT_TYPE_INITIALIZER, 20 elements, 0x4c bytes
   +0x000 Length           : Uint2B
   +0x002 UseDefaultObject : UChar
   +0x003 CaseInsensitive  : UChar
   +0x004 InvalidAttributes : Uint4B
   +0x008 GenericMapping   : struct _GENERIC_MAPPING, 4 elements, 0x10 bytes
   +0x018 ValidAccessMask  : Uint4B
   +0x01c SecurityRequired : UChar
   +0x01d MaintainHandleCount : UChar
   +0x01e MaintainTypeList : UChar
   +0x020 PoolType         : Enum _POOL_TYPE,  15 total enums
   +0x024 DefaultPagedPoolCharge : Uint4B
   +0x028 DefaultNonPagedPoolCharge : Uint4B
   +0x02c DumpProcedure    : Ptr32 to     void
   +0x030 OpenProcedure    : Ptr32 to     long
   +0x034 CloseProcedure   : Ptr32 to     void
   +0x038 DeleteProcedure  : Ptr32 to     void
   +0x03c ParseProcedure   : Ptr32 to     long
   +0x040 SecurityProcedure : Ptr32 to     long
   +0x044 QueryNameProcedure : Ptr32 to     long
   +0x048 OkayToCloseProcedure : Ptr32 to     unsigned char
```

En effet, à la fin de la structure nous trouvons des pointeurs de fonctions (`DumpProcedure`, `OpenProcedure`, etc.) qui sont appelées par l'*object manager*. En hookant ces pointeurs, il est donc possible de surveiller par exemple l'ouverture de processus, de threads ou de n'importe quel type d'objet du noyau.

Voyons maintenant comment les objets sont organisés hiérarchiquement. Ils sont contenus dans des structures appelées `_OBJECT_DIRECTORY`.

```
struct _OBJECT_DIRECTORY, 6 elements, 0xa4 bytes
  +0x000 HashBuckets     : [37] Ptr32 to struct _OBJECT_DIRECTORY_ENTRY, 2 elements, 0x8 bytes
   +0x094 Lock             : struct _EX_PUSH_LOCK, 5 elements, 0x4 bytes
  +0x098 DeviceMap       : Ptr32 to struct _DEVICE_MAP, 5 elements, 0x30 bytes
   +0x09c SessionId        : Uint4B
   +0x0a0 Reserved         : Uint2B
   +0x0a2 SymbolicLinkUsageCount : Uint2B
```

Les objets sont stockés dans une table de hachage (pointée par le champ `HashBuckets`). Pour gérer les collisions, chaque structure `_OBJECT_DIRECTORY_ENTRY` possède une liste doublement chainée des éléments possédant le même hash.

```
struct _OBJECT_DIRECTORY_ENTRY, 2 elements, 0x8 bytes
  +0x000 ChainLink       : Ptr32 to struct _OBJECT_DIRECTORY_ENTRY, 2 elements, 0x8 bytes
   +0x004 Object           : Ptr32 to Void
```

On retrouve logiquement l'objet dans le champ `Object`.

Revenons à notre `_OBJECT_HEADER`. Juste après celui-ci se trouve le corps de l'objet proprement dit (soit à un offset de `0x18`). Suivant le type de l'objet, le corps va être par exemple une structure `_EPROCESS` pour un processus, `_ETHREAD` pour un thread, `_OBJECT_DIRECTORY` pour un répertoire etc.

Nous avons représenté sur la figure 11 ces différentes interactions en prenant comme exemple le répertoire `WindowStations`.

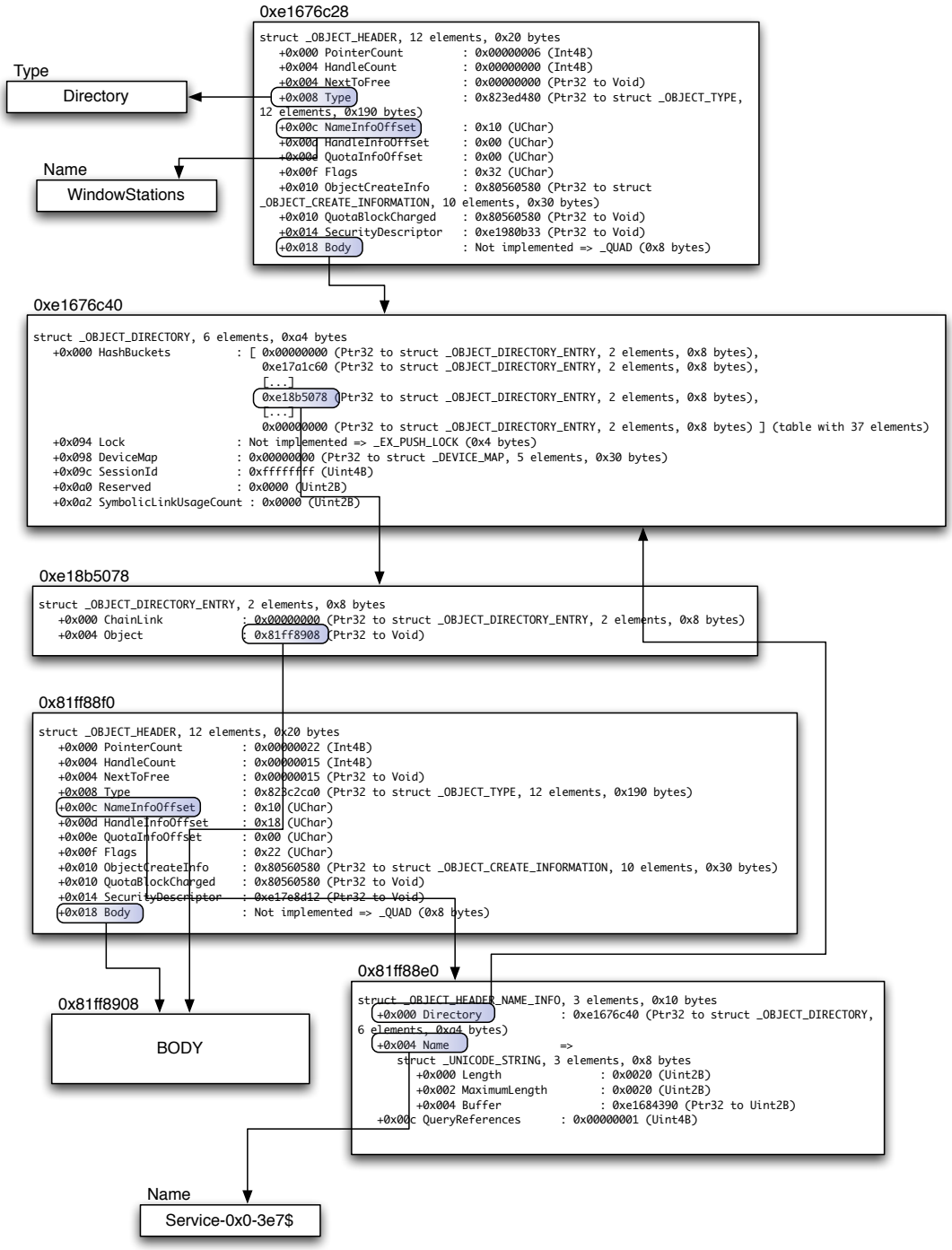


**Fig. 11.** Liens entre les objets du répertoire `WindowStations`

### 4.5 Regedit 101

Après avoir vu l'*object manager*, nous allons examiner les structures mises en jeu par le *configuration manager* de façon à poser les bases de la réalisation d'un clone de `regedit`.

Précédemment nous avons vu que tous les objets du noyau sont représentés par des structures bien précises ; les clés de la base de registre aussi. Dans le gestionnaire d'objets, le type est `Key`. Il s'agit d'une structure `_CM_KEY_BODY`.

```
struct _CM_KEY_BODY, 7 elements, 0x44 bytes
   +0x000 Type             : Uint4B
   +0x004 KeyControlBlock  : Ptr32 to struct _CM_KEY_CONTROL_BLOCK, 25 elements, 0x48 bytes
   +0x008 NotifyBlock      : Ptr32 to struct _CM_NOTIFY_BLOCK, 8 elements, 0x2c bytes
   +0x00c ProcessID        : Ptr32 to Void
   +0x010 Callers          : Uint4B
   +0x014 CallerAddress    : [10] Ptr32 to Void
   +0x03c KeyBodyList      : struct _LIST_ENTRY, 2 elements, 0x8 bytes
```

Nous retrouvons dans celle-ci un pointeur sur une structure `_CM_KEY_CONTROL_BLOCK` :

```
struct _CM_KEY_CONTROL_BLOCK, 25 elements, 0x48 bytes
   +0x000 RefCount         : Uint2B
   +0x002 Flags            : Uint2B
   +0x004 ExtFlags         : Bitfield Pos 0, 8 Bits
   +0x004 PrivateAlloc     : Bitfield Pos 8, 1 Bit
   +0x004 Delete           : Bitfield Pos 9, 1 Bit
   +0x004 DelayedCloseIndex : Bitfield Pos 10, 12 Bits
   +0x004 TotalLevels      : Bitfield Pos 22, 10 Bits
   +0x008 KeyHash          : struct _CM_KEY_HASH, 4 elements, 0x10 bytes
   +0x008 ConvKey          : Uint4B
   +0x00c NextHash         : Ptr32 to struct _CM_KEY_HASH, 4 elements, 0x10 bytes
   +0x010 KeyHive          : Ptr32 to struct _HHIVE, 24 elements, 0x210 bytes
   +0x014 KeyCell          : Uint4B
   +0x018 ParentKcb        : Ptr32 to struct _CM_KEY_CONTROL_BLOCK, 25 elements, 0x48 bytes
   +0x01c NameBlock        : Ptr32 to struct _CM_NAME_CONTROL_BLOCK, 7 elements, 0x10 bytes
   +0x020 CachedSecurity   : Ptr32 to struct _CM_KEY_SECURITY_CACHE, 5 elements, 0x28 bytes
   +0x024 ValueCache       : struct _CACHED_CHILD_LIST, 3 elements, 0x8 bytes
   +0x02c IndexHint        : Ptr32 to struct _CM_INDEX_HINT_BLOCK, 2 elements, 0x8 bytes
   +0x02c HashKey          : Uint4B
   +0x02c SubKeyCount      : Uint4B
   +0x030 KeyBodyListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x030 FreeListEntry    : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x038 KcbLastWriteTime : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x040 KcbMaxNameLen    : Uint2B
   +0x042 KcbMaxValueNameLen : Uint2B
   +0x044 KcbMaxValueDataLen : Uint4B
```

À partir de cette structure, nous retrouvons plusieurs pointeurs intéressants. Le premier est sur une structure `_HHIVE` qui représente la ruche de la base de registre.

```
struct _HHIVE, 24 elements, 0x210 bytes
   +0x000 Signature        : Uint4B
   +0x004 GetCellRoutine   : Ptr32 to     _CELL_DATA*
   +0x008 ReleaseCellRoutine : Ptr32 to     void
   +0x00c Allocate         : Ptr32 to     void*
   +0x010 Free             : Ptr32 to     void
   +0x014 FileSetSize      : Ptr32 to     unsigned char
   +0x018 FileWrite        : Ptr32 to     unsigned char
   +0x01c FileRead         : Ptr32 to     unsigned char
   +0x020 FileFlush        : Ptr32 to     unsigned char
  +0x024 BaseBlock       : Ptr32 to struct _HBASE_BLOCK, 17 elements, 0x1000 bytes
   [...]
   +0x041 ReadOnly         : UChar
   +0x042 Log              : UChar
   +0x044 HiveFlags        : Uint4B
   +0x048 LogSize          : Uint4B
   +0x04c RefreshCount     : Uint4B
   +0x050 StorageTypeCount : Uint4B
   +0x054 Version          : Uint4B
   +0x058 Storage          : [2] struct _DUAL, 7 elements, 0xdc bytes
```

Il y a au début de cette structure plusieurs pointeurs de fonctions qui peuvent être très utiles si nous désirons poser des hooks assez furtifs. Le champ `_HBASE_BLOCK` pointe sur le premier block de la ruche.

```
struct _HBASE_BLOCK, 17 elements, 0x1000 bytes
   +0x000 Signature        : Uint4B
   +0x004 Sequence1        : Uint4B
   +0x008 Sequence2        : Uint4B
   +0x00c TimeStamp        : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x014 Major            : Uint4B
   +0x018 Minor            : Uint4B
   +0x01c Type             : Uint4B
   +0x020 Format           : Uint4B
   +0x024 RootCell         : Uint4B
   +0x028 Length           : Uint4B
   +0x02c Cluster          : Uint4B
   +0x030 FileName         : [64] UChar
   +0x070 Reserved1        : [99] Uint4B
   +0x1fc CheckSum         : Uint4B
   +0x200 Reserved2        : [894] Uint4B
   +0xff8 BootType         : Uint4B
   +0xffc BootRecover      : Uint4B
```

Nous y retrouvons par exemple le nom de la ruche (quand il existe) ainsi que la cellule racine.

Si nous revenons sur notre structure `_CM_KEY_BODY`, nous trouvons grâce au champ `NameBlock` une structure contenant le nom de notre clé.

```
struct _CM_NAME_CONTROL_BLOCK, 7 elements, 0x10 bytes
   +0x000 Compressed       : UChar
   +0x002 RefCount         : Uint2B
   +0x004 NameHash         : struct _CM_NAME_HASH, 4 elements, 0xc bytes
   +0x004 ConvKey          : Uint4B
   +0x008 NextHash       : Ptr32 to struct _CM_KEY_HASH, 4 elements, 0x10 bytes
   +0x00c NameLength       : Uint2B
   +0x00e Name             : [1] Uint2B
```

Avec les champs `NameLength` et `Name`, nous retrouvons sans aucun problème son nom.

Comment sont organisées les clés au sein de la ruche lorsque celle-ci est dans la mémoire ?

En effet, les index des cellules ne sont pas directement des adresses. Comment faire pour retrouver ces adresses ?

Le noyau (plus précisément le «configuration manager») utilise un principe similaire à celui utilisé pour convertir les adresses virtuelles en adresses physiques. L'index est découpé en 4 parties :

- Les 12 premiers bits correspondent à un offset ;
- les 9 suivants correspondent à un index dans une table appelée `_HMAP_TABLE` ;
- les 10 bits suivants correspondent à un index dans une autre table appelée `_HMAP_DIRECTORY` ;
- le bit de poids fort indique dans quel *Storage* se situe la clé.

Les adresses des tables `_HMAP_DIRECTORY` se situent dans la structure `_DUAL` que nous trouvons dans le champ `Storage` de la ruche.

```
struct _DUAL, 7 elements, 0xdc bytes
   +0x000 Length           : Uint4B
   +0x004 Map            : Ptr32 to struct _HMAP_DIRECTORY, 1 elements, 0x1000 bytes
   +0x008 SmallDir        : Ptr32 to struct _HMAP_TABLE, 1 elements, 0x2000 bytes
   +0x00c Guard            : Uint4B
   +0x010 FreeDisplay      : [24] struct _RTL_BITMAP, 2 elements, 0x8 bytes
   +0x0d0 FreeSummary      : Uint4B
   +0x0d4 FreeBins         : struct _LIST_ENTRY, 2 elements, 0x8 bytes
```

Avec le champ `Map` et l'index dans la structure `_HMAP_DIRECTORY`, nous trouvons un pointeur sur une structure `_HMAP_TABLE`. Celui-ci avec l'index dans la table nous donne finalement un pointeur sur une structure `_HMAP_ENTRY`. Celle-ci a la forme suivante :

```
struct _HMAP_ENTRY, 4 elements, 0x10 bytes
   +0x000 BlockAddress     : Uint4B
   +0x004 BinAddress       : Uint4B
```

```
   +0x008 CmView         : Ptr32 to struct _CM_VIEW_OF_FILE, 7 elements, 0x24 bytes
   +0x00c MemAlloc          : Uint4B
```

Le champ qui nous intéresse est `BlockAddress`. En ajoutant sa valeur à celle de l'offset calculé à partir de l'index recherché, nous obtenons enfin l'adresse de la clé de registre, plus exactement d'une structure représentant une clé de registre.

Il y a en fait 2 structures `_DUAL`. Le *configuration manager* utilise 2 types de stockage : le *Stable Storage* et le *Volatile Storage*. Le second est utilisé pour les clés qui n'existent qu'en mémoire. Pour savoir quelle est la table à utiliser, il suffit de regarder le bit de poids fort de l'index.

Les clés de registres peuvent être de 5 types. Le premier type est la structure `_CM_KEY_NODE` :

```
struct _CM_KEY_NODE, 19 elements, 0x50 bytes
   +0x000 Signature        : Uint2B
   +0x002 Flags            : Uint2B
   +0x004 LastWriteTime    : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x00c Spare            : Uint4B
   +0x010 Parent           : Uint4B
   +0x014 SubKeyCounts     : [2] Uint4B
   +0x01c SubKeyLists      : [2] Uint4B
   +0x024 ValueList        : struct _CHILD_LIST, 2 elements, 0x8 bytes
  +0x01c ChildHiveReference : struct _CM_KEY_REFERENCE, 2 elements, 0x8 bytes
   +0x02c Security         : Uint4B
   +0x030 Class            : Uint4B
   +0x034 MaxNameLen       : Uint4B
   +0x038 MaxClassLen      : Uint4B
   +0x03c MaxValueNameLen  : Uint4B
   +0x040 MaxValueDataLen  : Uint4B
   +0x044 WorkVar          : Uint4B
   +0x048 NameLength       : Uint2B
   +0x04a ClassLength      : Uint2B
   +0x04c Name             : [1] Uint2B
```

La signature est *kn* pour «key node». Il s'agit des clés qui contiennent d'autres clés.

Le deuxième type a pour signature *lk*. Il s'agit des clés représentant un lien vers une autre clé. Dans ce cas, nous retrouvons toutes les informations nécessaires dans le champ `ChildHiveReference`.

```
struct _CM_KEY_REFERENCE, 2 elements, 0x8 bytes
   +0x000 KeyCell          : Uint4B
   +0x004 KeyHive        : Ptr32 to struct _HHIVE, 24 elements, 0x210 bytes
```

En troisième, nous trouvons les clés représentant les sous-clés. La signature est *lf* ou *lh*. Il existe deux signatures car il y a deux types de listes : avec ou sans checksum [12]. Il s'agit d'une structure `_CM_KEY_INDEX` :

```
struct _CM_KEY_INDEX, 3 elements, 0x8 bytes
   +0x000 Signature        : Uint2B
   +0x002 Count            : Uint2B
   +0x004 List             : [1] Uint4B
```

Il suffit d'itérer sur le champ `List` pour obtenir des index qui donneront des clés.

En quatrième, nous trouvons les clés représentant des couples clés-valeurs. La signature est *kv* et la structure est `_CM_KEY_VALUE`.

```
struct _CM_KEY_VALUE, 8 elements, 0x18 bytes
   +0x000 Signature        : Uint2B
   +0x002 NameLength       : Uint2B
   +0x004 DataLength       : Uint4B
   +0x008 Data             : Uint4B
   +0x00c Type             : Uint4B
   +0x010 Flags            : Uint2B
   +0x012 Spare            : Uint2B
   +0x014 Name             : [1] Uint2B
```

Si la valeur de `NameLength` vaut 0, il s'agit d'une clé anonyme (appelée *(par défaut)* sous *regedit*). Ensuite il faut regarder le bit de poids fort du champ `DataLength`. Si le bit est à 1, il suffit de lire les données directement à partir des champs `Data` et `DataLength`. Si le bit est à 0, nous trouvons alors dans le champ `Data` un index qui, une fois converti, donne une adresse à partir de laquelle nous pouvons lire les données.

Le dernier type a pour signature *sk*. Il s'agit d'une structure `_CM_KEY_SECURITY`.

```
struct _CM_KEY_SECURITY, 7 elements, 0x28 bytes
   +0x000 Signature        : Uint2B
   +0x002 Reserved         : Uint2B
   +0x004 Flink            : Uint4B
   +0x008 Blink            : Uint4B
   +0x00c ReferenceCount   : Uint4B
   +0x010 DescriptorLength : Uint4B
  +0x014 Descriptor      : struct _SECURITY_DESCRIPTOR_RELATIVE, 7 elements, 0x14 bytes
```

Elle représente les permissions d'accès aux clés et appartient à une liste doublement chaînée.

Afin d'illustrer ces relations, nous allons examiner un cas concret. Toujours dans le cas de `lsass.exe`, nous observons qu'un de ses handles est la clé `LSA` dans la ruche `SYSTEM`. La figure 12 montre les relations permettant d'obtenir ces renseignements.

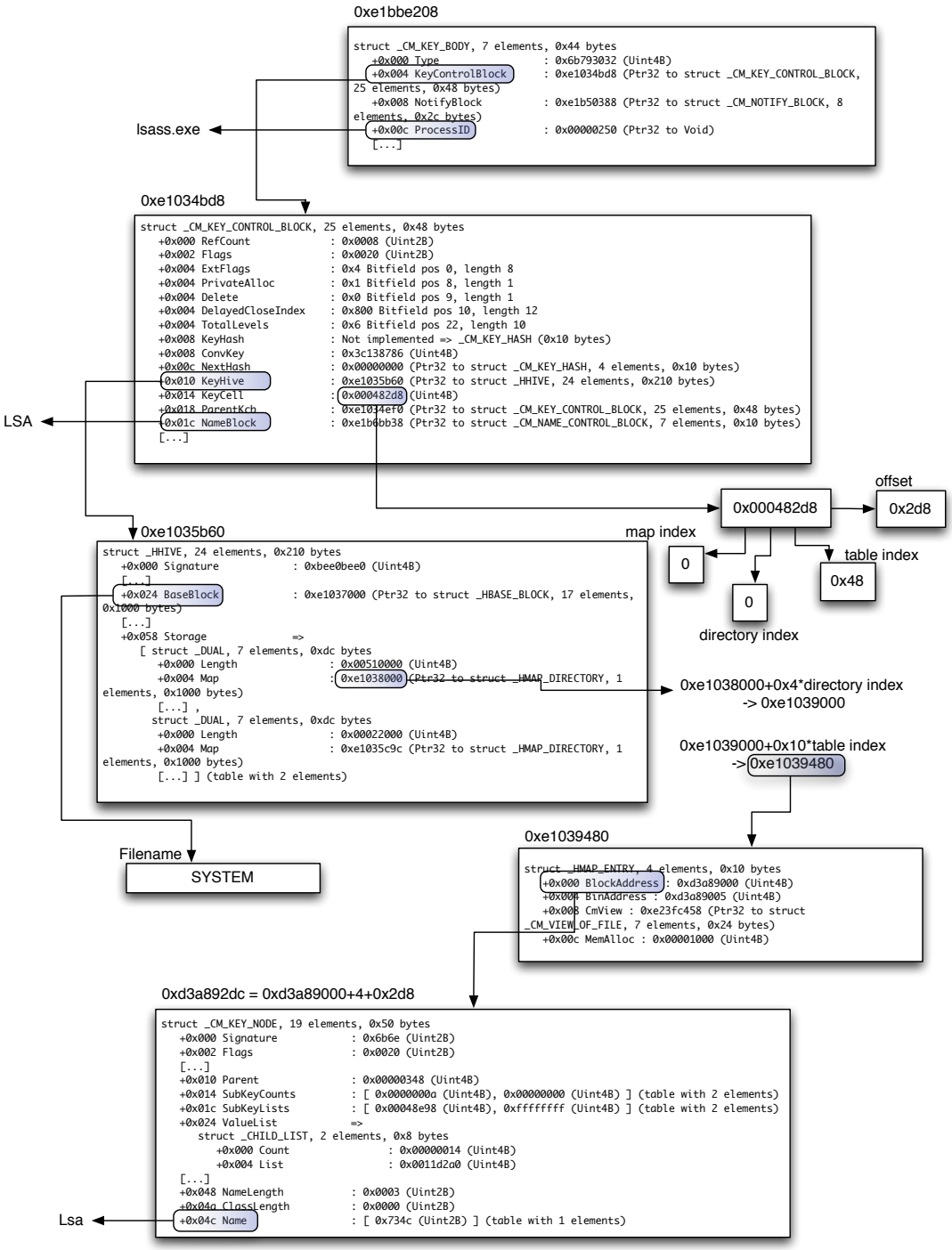


**Fig. 12.** Liens entre les structures d'une clé de registre

### 4.6 ProcessExplorer 101

Après `WinObj` et `regedit`, nous nous attaquons à un autre clone d'une application très utile de Sysinternals [11] : `ProcessExplorer`.

Dans cette partie, nous allons expliquer quelles sont les structures utilisées par le noyau pour gérer les processus et quelles sont les informations qu'il est possible de récupérer. Nous verrons comment obtenir la liste de tous les processus, comment obtenir les différents handles, comment naviguer dans les threads et finalement comment dumper un processus.

**Parcourir la liste des processus** Le noyau Windows utilise très souvent des listes doublement chaînées pour stocker ses structures. Les éléments des listes sont des structures `LIST_ENTRY`.

```
struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x000 Flink          : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x004 Blink          : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
```

Lorsque nous naviguons dans ces listes, il faut juste penser que l'élément pointé est une autre structure `LIST_ENTRY`. Donc pour retrouver la structure qui nous intéresse, il faut enlever un offset dépendant de la structure en question. Prenons par exemple le cas où nous voulons avoir la liste de tous les processus du système. Nous récupérons l'adresse de la première entrée de la liste doublement chaînée des processus :

```
kd> ?poi(PsActiveProcessHead)
Evaluate expression: -2109962056 = 823c88b8
```

Ce n'est pas une structure `EPROCESS` qui se trouve à l'adresse `0x823c88b8`, mais une structure `LIST_ENTRY`. Pour retrouver le processus il faut retrancher un offset valant `0x88` (il s'agit de l'offset du champ `ActiveProcessLinks` de la structure `EPROCESS`). Nous vérifions que nous avons bien le premier processus créé (c'est-à-dire le processus `System`) :

```
kd> dt -v nt!_EPROCESS poi(PsActiveProcessHead)-0x88
struct _EPROCESS, 107 elements, 0x260 bytes
   ...
   +0x084 UniqueProcessId  : 0x00000004
   +0x088 ActiveProcessLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
 [ 0x82078908 - 0x805606d8 ]
   ...
   +0x174 ImageFileName    : [16]  "System"
   ...
```

**Handles ouverts** Abordons maintenant les handles. Ceux-ci représentent basiquement un accès sur un objet, c'est-à-dire des pointeurs indirects sur des ressources du noyau.

Chaque processus possède une table de handle (le champ `ObjectTable`). Elle est représentée par une structure `_HANDLE_TABLE`.

```
struct _HANDLE_TABLE, 14 elements, 0x44 bytes
   +0x000 TableCode        : Uint4B
   +0x004 QuotaProcess     : Ptr32 to struct _EPROCESS, 107 elements, 0x260 bytes
   +0x008 UniqueProcessId  : Ptr32 to Void
   +0x00c HandleTableLock  : [4] struct _EX_PUSH_LOCK, 5 elements, 0x4 bytes
   +0x01c HandleTableList  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   [...]
   +0x03c HandleCount      : Int4B
   +0x040 Flags            : Uint4B
   +0x040 StrictFIFO       : Bitfield Pos 0, 1 Bit
```

Suivant le nombre de handles, le champ `TableCode` peut pointer soit sur un tableau de structures `_HANDLE_TABLE_ENTRY` soit sur un tableau de pointeurs, voire, si le nombre de handles est suffisamment important, sur un tableau de pointeurs de tableaux de pointeurs.

Sur une architecture standard, les pages ont une taille de 4096 octets. Sachant que la structure `_HANDLE_ENTRY` a une taille de `0x10` octets, il peut y avoir jusqu'à 512 handles par page. En fait, il y en a 511 car la première entrée est réservée par le système.

Comme une page mémoire peut contenir jusqu'à 1024 pointeurs, nous en déduisons donc :

- si le nombre de handles est inférieur à 511, il n'y a pas de table de pointeurs ;
- si le nombre de handles est compris entre 512 et 523264 (511*1024), il y a une table de pointeurs ;
- si le nombre de handles est supérieur à 523264, il y a plusieurs tables de pointeurs.

La structure `_HANDLE_TABLE_ENTRY` a la forme suivante :

```
struct _HANDLE_TABLE_ENTRY, 8 elements, 0x8 bytes
   +0x000 Object           : Ptr32 to Void
   +0x000 ObAttributes     : Uint4B
   +0x000 InfoTable        : Ptr32 to struct _HANDLE_TABLE_ENTRY_INFO, 1 elements, 0x4 bytes
   +0x000 Value            : Uint4B
   +0x004 GrantedAccess    : Uint4B
   +0x004 GrantedAccessIndex : Uint2B
```

```
+0x006 CreatorBackTraceIndex : Uint2B
+0x004 NextFreeTableEntry : Int4B
```

Etant donné que les allocations mémoires se font avec une granularité de 8 octets, le noyau utilise les 3 bits de poids faible du champ `Object` pour stocker certains attributs du handle (`Audit on close`, `Inheritable` et `Protect from close`). Il utilise aussi le bit de poids fort pour locker le handle si besoin est. Pour retrouver l'adresse de l'objet, il suffit donc de faire `(Object |0x80000000) & 0xffffffff8`.

Nous prenons toujours en exemple notre processus `lsass.exe`. Toutes ces interactions sont résumées sur la figure suivante :

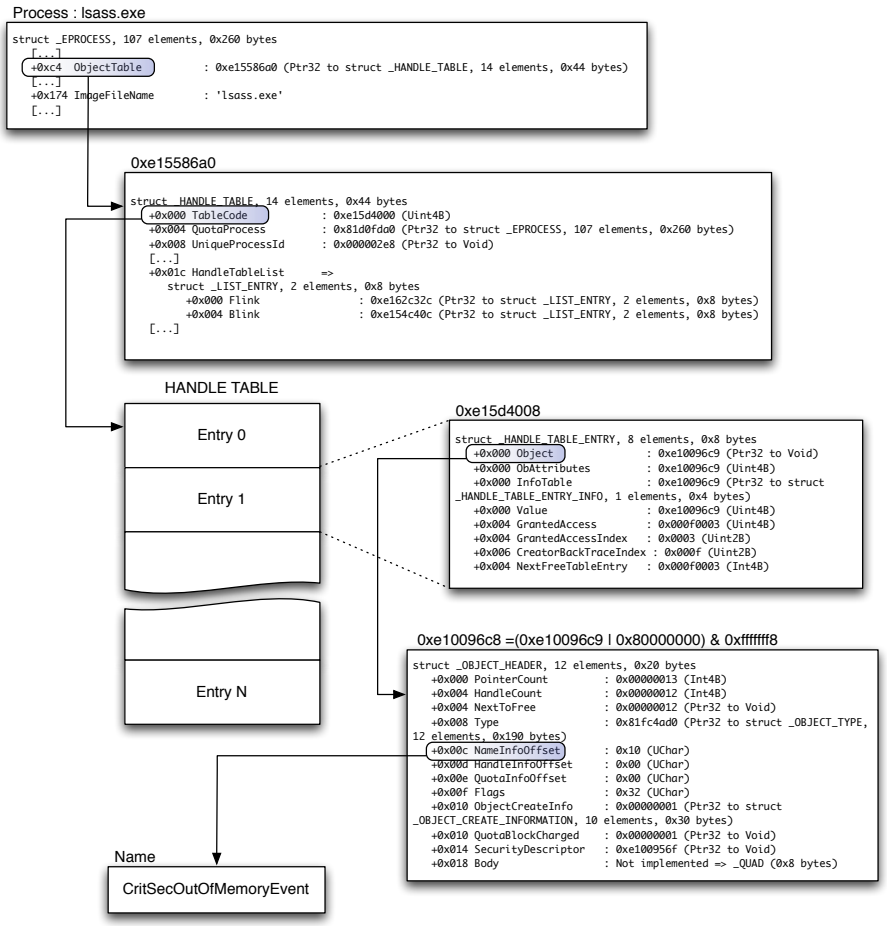


**Fig. 13.** Premier handle de `lsass.exe`

**Threads** Pour gérer les threads, le noyau utilise 2 structures : `_KTHREAD` pour ce qui a trait à la partie noyau et `_ETHREAD` pour la partie utilisateur.

```
struct _ETHREAD, 54 elements, 0x258 bytes
   +0x000 Tcb              : struct _KTHREAD, 73 elements, 0x1c0 bytes
   +0x1c0 CreateTime       : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   [...]
   +0x1ec Cid              : struct _CLIENT_ID, 2 elements, 0x8 bytes
   [...]
   +0x210 IrpList          : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x218 TopLevelIrp      : Uint4B
  +0x21c DeviceToVerify   : Ptr32 to struct _DEVICE_OBJECT, 25 elements, 0xb8 bytes
  +0x220 ThreadsProcess   : Ptr32 to struct _EPROCESS, 107 elements, 0x260 bytes
   +0x224 StartAddress     : Ptr32 to Void
   +0x228 Win32StartAddress : Ptr32 to Void
   [...]
   +0x22c ThreadListEntry  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   [...]
   +0x248 CrossThreadFlags : Uint4B
   +0x248 Terminated       : Bitfield Pos 0, 1 Bit
   +0x248 DeadThread       : Bitfield Pos 1, 1 Bit
   +0x248 HideFromDebugger : Bitfield Pos 2, 1 Bit
   +0x248 ActiveImpersonationInfo : Bitfield Pos 3, 1 Bit
   +0x248 SystemThread     : Bitfield Pos 4, 1 Bit
   +0x248 HardErrorsAreDisabled : Bitfield Pos 5, 1 Bit
   +0x248 BreakOnTermination : Bitfield Pos 6, 1 Bit
   +0x248 SkipCreationMsg  : Bitfield Pos 7, 1 Bit
   +0x248 SkipTerminationMsg : Bitfield Pos 8, 1 Bit
   +0x24c SameThreadPassiveFlags : Uint4B
   +0x24c ActiveExWorker   : Bitfield Pos 0, 1 Bit
   +0x24c ExWorkerCanWaitUser : Bitfield Pos 1, 1 Bit
   +0x24c MemoryMaker      : Bitfield Pos 2, 1 Bit
   [...]
```

La structure `_KTHREAD` se situe dans le champ `Tcb`.

```
struct _KTHREAD, 73 elements, 0x1c0 bytes
  +0x000 Header         : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
   +0x010 MutantListHead   : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x018 InitialStack     : Ptr32 to Void
   +0x01c StackLimit       : Ptr32 to Void
   +0x020 Teb              : Ptr32 to Void
   +0x024 TlsArray         : Ptr32 to Void
   +0x028 KernelStack      : Ptr32 to Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   [...]
   +0x0e0 ServiceTable     : Ptr32 to Void
   [...]
```

```
   +0x12c CallbackStack    : Ptr32 to Void
   +0x130 Win32Thread      : Ptr32 to Void
  +0x134 TrapFrame       : Ptr32 to struct _KTRAP_FRAME, 35 elements, 0x8c bytes
  +0x138 ApcStatePointer  : [2] Ptr32 to struct _KAPC_STATE, 5 elements, 0x18 bytes
   [...]
   +0x168 StackBase        : Ptr32 to Void
   +0x16c SuspendApc       : struct _KAPC, 14 elements, 0x30 bytes
   +0x19c SuspendSemaphore : struct _KSEMAPHORE, 2 elements, 0x14 bytes
   +0x1b0 ThreadListEntry  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x1b8 FreezeCount      : Char
   +0x1b9 SuspendCount     : Char
   +0x1ba IdealProcessor   : UChar
   +0x1bb DisableBoost     : UChar
```

Tous les threads d'un processus sont reliés entre eux par une liste doublement chaînée appelée `ThreadListEntry`. Nous retrouvons un pointeur vers la SSDT grâce au champ `ServiceTable`. Nous retrouvons également des pointeurs vers la pile utilisateur et la pile noyau ainsi que l'adresse de début du thread (champ `StartAddress`).

**Bibliothèques chargées en mémoire** Si nous voulons pousser notre clone de *ProcessExplorer* jusqu'au bout, il faut aussi que nous retrouvions les bibliothèques chargées dans l'espace d'adressage du processus.

Pour cela, nous allons d'abord regarder la structure gérant le PEB (Process Environment Block).

```
struct _PEB, 65 elements, 0x210 bytes
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 to Void
   +0x008 ImageBaseAddress : Ptr32 to Void
  +0x00c Ldr            : Ptr32 to struct _PEB_LDR_DATA, 7 elements, 0x28 bytes
  +0x010 ProcessParameters : Ptr32 to struct _RTL_USER_PROCESS_PARAMETERS, 28 elements, 0x290 bytes
   +0x014 SubSystemData    : Ptr32 to Void
   +0x018 ProcessHeap      : Ptr32 to Void
   [...]
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
  +0x070 CriticalSectionTimeout : union _LARGE_INTEGER, 4 elements, 0x8 bytes
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 to Ptr32 to Void
```

```
   [.. .]
   +0x0a4 OSMajorVersion   : Uint4B
   +0x0a8 OSMinorVersion   : Uint4B
   +0x0ac OSBuildNumber    : Uint2B
   +0x0ae OSCSDVersion     : Uint2B
   +0x0b0 OSPlatformId     : Uint4B
   +0x0b4 ImageSubsystem   : Uint4B
   +0x0b8 ImageSubsystemMajorVersion : Uint4B
   +0x0bc ImageSubsystemMinorVersion : Uint4B
   +0x0c0 ImageProcessAffinityMask : Uint4B
   [...]
   +0x1f0 CSDVersion       : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x1f8 ActivationContextData : Ptr32 to Void
   +0x1fc ProcessAssemblyStorageMap : Ptr32 to Void
   +0x200 SystemDefaultActivationContextData : Ptr32 to Void
   +0x204 SystemAssemblyStorageMap : Ptr32 to Void
   +0x208 MinimumStackCommit : Uint4B
```

Nous retrouvons dans cette structure par exemple le nombre de heaps (champ `NumberOfHeaps`), un pointeur sur un tableau contenant les heaps du processus (champ `ProcessHeaps`), l'adresse de chargement de l'exécutable (champ `ImageBaseAddress`) et un pointeur vers une structure représentant le travail effectué par le loader (champ `Ldr`).

Cette structure possède le format suivant :

```
struct _PEB_LDR_DATA, 7 elements, 0x28 bytes
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 to Void
  +0x00c InLoadOrderModuleList : struct _LIST_ENTRY, 2 elements, 0x8 bytes
  +0x014 InMemoryOrderModuleList : struct _LIST_ENTRY, 2 elements, 0x8 bytes
  +0x01c InInitializationOrderModuleList : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x024 EntryInProgress  : Ptr32 to Void
```

Chacune des listes chaînées (`InLoadOrderModuleList`, `InMemoryOrderModuleList` et `InInitializationOrderModuleList`) contient la liste des bibliothèques chargées dans la mémoire du processus. Le fonctionnement est identique à celui des modules vu précédemment.

La structure concernée est une structure `_LDR_DATA_TABLE_ENTRY`.

```
struct _LDR_DATA_TABLE_ENTRY, 18 elements, 0x50 bytes
   +0x000 InLoadOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x008 InMemoryOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
  +0x010 InInitializationOrderLinks : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x018 DllBase          : Ptr32 to Void
   +0x01c EntryPoint       : Ptr32 to Void
   +0x020 SizeOfImage      : Uint4B
```

```
   +0x024 FullDllName      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x02c BaseDllName      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x03c SectionPointer   : Ptr32 to Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 to Void
   +0x048 EntryPointActivationContext : Ptr32 to Void
   +0x04c PatchInformation : Ptr32 to Void
```

Il suffit d'itérer sur une des listes chaînées pour avoir la liste de toutes les bibliothèques chargées en mémoire.

Toujours en prenant comme exemple le processus `lsass.exe`, nous obtenons la liste suivante :

```
C:\WINDOWS\system32\lsass.exe: 0x01000000 (0x6000),
C:\WINDOWS\system32\ntdll.dll: 0x7c900000 (0xb0000),
C:\WINDOWS\system32\kernel32.dll: 0x7c800000 (0xf4000),
C:\WINDOWS\system32\ADVAPI32.dll: 0x77dd0000 (0x9b000),
C:\WINDOWS\system32\RPCRT4.dll: 0x77e70000 (0x91000),
C:\WINDOWS\system32\LSASRV.dll: 0x75730000 (0xb4000),
C:\WINDOWS\system32\MPR.dll: 0x71b20000 (0x12000),
C:\WINDOWS\system32\USER32.dll: 0x77d40000 (0x90000),
C:\WINDOWS\system32\GDI32.dll: 0x77f10000 (0x46000),
C:\WINDOWS\system32\MSASN1.dll: 0x77b20000 (0x12000),
C:\WINDOWS\system32\msvcrt.dll: 0x77c10000 (0x58000),
C:\WINDOWS\system32\NETAPI32.dll: 0x5b860000 (0x54000),
C:\WINDOWS\system32\NTDSAPI.dll: 0x767a0000 (0x13000),
C:\WINDOWS\system32\DNSAPI.dll: 0x76f20000 (0x27000),
C:\WINDOWS\system32\WS2_32.dll: 0x71ab0000 (0x17000),
C:\WINDOWS\system32\WS2HELP.dll: 0x71aa0000 (0x8000),
C:\WINDOWS\system32\WLDAP32.dll: 0x76f60000 (0x2c000),
C:\WINDOWS\system32\Secur32.dll: 0x77fe0000 (0x11000),
C:\WINDOWS\system32\SAMLIB.dll: 0x71bf0000 (0x13000),
C:\WINDOWS\system32\SAMSRV.dll: 0x74440000 (0x6a000),
C:\WINDOWS\system32\cryptdll.dll: 0x76790000 (0xc000),
C:\WINDOWS\system32\ShimEng.dll: 0x5cb70000 (0x26000),
C:\WINDOWS\AppPatch\AcGenral.DLL: 0x6f880000 (0x1ca000),
C:\WINDOWS\system32\WINMM.dll: 0x76b40000 (0x2d000),
C:\WINDOWS\system32\ole32.dll: 0x774e0000 (0x13d000),
C:\WINDOWS\system32\OLEAUT32.dll: 0x77120000 (0x8c000),
C:\WINDOWS\system32\MSACM32.dll: 0x77be0000 (0x15000),
C:\WINDOWS\system32\VERSION.dll: 0x77c00000 (0x8000),
C:\WINDOWS\system32\SHELL32.dll: 0x7c9c0000 (0x814000),
C:\WINDOWS\system32\SHLWAPI.dll: 0x77f60000 (0x76000),
C:\WINDOWS\system32\USERENV.dll: 0x769c0000 (0xb3000),
```

```
C:\WINDOWS\system32\UxTheme.dll: 0x5ad70000 (0x38000),
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2180_x-ww_a84f1f
C:\WINDOWS\system32\comctl32.dll: 0x5d090000 (0x97000),
C:\WINDOWS\system32\msprivs.dll: 0x20000000 (0xe000),
C:\WINDOWS\system32\kerberos.dll: 0x71cf0000 (0x4b000),
C:\WINDOWS\system32\msv1_0.dll: 0x77c70000 (0x23000),
C:\WINDOWS\system32\iphlpapi.dll: 0x76d60000 (0x19000),
C:\WINDOWS\system32\netlogon.dll: 0x744b0000 (0x65000),
C:\WINDOWS\system32\w32time.dll: 0x767c0000 (0x2c000),
C:\WINDOWS\system32\MSVCP60.dll: 0x76080000 (0x65000),
C:\WINDOWS\system32\schannel.dll: 0x767f0000 (0x27000),
C:\WINDOWS\system32\CRYPT32.dll: 0x77a80000 (0x94000),
C:\WINDOWS\system32\wdigest.dll: 0x74380000 (0xf000),
C:\WINDOWS\system32\rsaenh.dll: 0x0ffd0000 (0x28000),
C:\WINDOWS\system32\scecli.dll: 0x74410000 (0x2e000),
C:\WINDOWS\system32\SETUPAPI.dll: 0x77920000 (0xf3000),
C:\WINDOWS\system32\ipsecsvc.dll: 0x743e0000 (0x2f000),
C:\WINDOWS\system32\AUTHZ.dll: 0x776c0000 (0x11000),
C:\WINDOWS\system32\oakley.DLL: 0x75d90000 (0xce000),
C:\WINDOWS\system32\WINIPSEC.DLL: 0x74370000 (0xb000),
C:\WINDOWS\system32\pstorsvc.dll: 0x743a0000 (0xb000),
C:\WINDOWS\system32\psbase.dll: 0x743c0000 (0x1b000),
C:\Program Files\Google\Google Desktop Search\GoogleDesktopNetwork1.dll: 0x43000000 (0x5000),
C:\WINDOWS\system32\mswsock.dll: 0x71a50000 (0x3f000),
C:\WINDOWS\system32\hnetcfg.dll: 0x662b0000 (0x58000),
C:\WINDOWS\System32\wshtcpip.dll: 0x71a90000 (0x8000),
C:\WINDOWS\system32\dssenh.dll: 0x68100000 (0x24000)
```

La première valeur est l'adresse de chargement et la seconde la taille.

Une autre structure dans le PEB présente un intérêt : il s'agit de la structure `_RTL_USER_PROCESS_PARAMETERS`. Elle permet d'obtenir entre autre les variables d'environnement du processus. Pour cela, il suffit de lire à l'adresse désignée par le pointeur `Environment` une taille valant le champ `Length`.

```
struct _RTL_USER_PROCESS_PARAMETERS, 28 elements, 0x290 bytes
   +0x000 MaximumLength    : Uint4B
   +0x004 Length           : Uint4B
   +0x008 Flags            : Uint4B
   +0x00c DebugFlags       : Uint4B
   +0x010 ConsoleHandle    : Ptr32 to Void
   +0x014 ConsoleFlags     : Uint4B
   +0x018 StandardInput    : Ptr32 to Void
   +0x01c StandardOutput   : Ptr32 to Void
   +0x020 StandardError    : Ptr32 to Void
   +0x024 CurrentDirectory : struct _CURDIR, 2 elements, 0xc bytes
   +0x030 DllPath          : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x038 ImagePathName    : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x040 CommandLine      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x048 Environment      : Ptr32 to Void
```

```
   +0x04c StartingX        : Uint4B
   +0x050 StartingY        : Uint4B
   +0x054 CountX           : Uint4B
   +0x058 CountY           : Uint4B
   +0x05c CountCharsX      : Uint4B
   +0x060 CountCharsY      : Uint4B
   +0x064 FillAttribute    : Uint4B
   +0x068 WindowFlags      : Uint4B
   +0x06c ShowWindowFlags  : Uint4B
   +0x070 WindowTitle      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x078 DesktopInfo      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x080 ShellInfo        : struct _UNICODE_STRING, 3 elements, 0x8 bytes
   +0x088 RuntimeData      : struct _UNICODE_STRING, 3 elements, 0x8 bytes
  +0x090 CurrentDirectores : [32] struct _RTL_DRIVE_LETTER_CURDIR, 4 elements, 0x10 bytes
```

Nous retrouvons aussi d'autres informations utiles comme le nom de la fenêtre (champ `WindowTitle`), la ligne de commande (champ `CommandLine`), etc.

Pour `lsass.exe`, nous obtenons le résultat suivant :

```
ALLUSERSPROFILE=C:\\Documents and Settings\\All Users
CommonProgramFiles=C:\\Program Files\\Common Files
COMPUTERNAME=SPLATITUDE
ComSpec=C:\\WINDOWS\\system32\\cmd.exe
FP_NO_HOST_CHECK=NO
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Path=C:\\WINDOWS\\System32
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 8 Stepping 1, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0801
ProgramFiles=C:\\Program Files
SystemDrive=C:
SystemRoot=C:\\WINDOWS
TEMP=C:\\WINDOWS\\TEMP
TMP=C:\\WINDOWS\\TEMP
USERPROFILE=C:\\WINDOWS\\system32\\config\\systemprofile
windir=C:\\WINDOWS
```

**Dumper un processus** Lorsque nous analysons un dump mémoire (par exemple dans un cadre forensic), il peut être intéressant de pouvoir dumper un processus pour réaliser une analyse ultérieure.

Nous avons vu précédemment que dans le PEB, nous obtenons l'adresse de chargement de l'exécutable avec le champ `ImageBaseAddress`.

La méthode à suivre pour dumper l'exécutable est la suivante. Tout d'abord, nous lisons une page mémoire à partir de l'adresse indiquée par `ImageBaseAddress`. Sur cette page se trouve le header du fichier PE. Grâce à celui-ci, nous obtenons les adresses et la taille des différentes sections. Il suffit ensuite de lire page par page et de recopier les données aux offsets indiqués dans le PE pour obtenir un PE que nous pouvons charger dans un désassembleur pour une analyse future.

Nous allons illustrer cette démarche par un exemple. Nous allons dumper le processus `lsass.exe`.

Le PEB nous informe que l'exécutable est chargé à l'adresse `0x1000000`. En chargeant cette page dans un éditeur de PE (dans notre cas le module `pefile` [13] d'Ero Carrera), nous obtenons les informations suivantes en lisant le contenu du header `IMAGE_OPTIONAL_HEADER`.

```
# pe.OPTIONAL_HEADER.dump()
=> ['[IMAGE_OPTIONAL_HEADER]',
 'Magic:                        0x10B     ',
 'MajorLinkerVersion:           0x7       ',
 'MinorLinkerVersion:           0xA       ',
 'SizeOfCode:                   0x1200    ',
 'SizeOfInitializedData:        0x1E00    ',
 'SizeOfUninitializedData:      0x0       ',
 'AddressOfEntryPoint:          0x14BD    ',
 'BaseOfCode:                   0x1000    ',
 'BaseOfData:                   0x3000    ',
 'ImageBase:                    0x1000000 ',
 'SectionAlignment:             0x1000    ',
 'FileAlignment:                0x200     ',
 'MajorOperatingSystemVersion:  0x5       ',
 'MinorOperatingSystemVersion:  0x1       ',
 'MajorImageVersion:            0x5       ',
 'MinorImageVersion:            0x1       ',
 'MajorSubsystemVersion:        0x4       ',
 'MinorSubsystemVersion:        0x0       ',
 'Reserved1:                    0x0       ',
 'SizeOfImage:                  0x6000    ',
 'SizeOfHeaders:                0x400     ',
 'CheckSum:                     0x120C5   ',
 'Subsystem:                    0x2       ',
 'DllCharacteristics:           0x8000    ',
 'SizeOfStackReserve:           0x40000   ',
 'SizeOfStackCommit:            0x6000    ',
 'SizeOfHeapReserve:            0x100000  ',
 'SizeOfHeapCommit:             0x1000    ',
 'LoaderFlags:                  0x0       ',
 'NumberOfRvaAndSizes:          0x10      ']
```

Nous obtenons que la taille de l'image fait `0x6000` octets. Ensuite, en examinant les différentes sections, nous avons les informations suivantes :

```
# for i in pe.sections: i.dump()
=> ['[IMAGE_SECTION_HEADER]',
 'Name:                          .text',
 'Misc:                          0x10D0    ',
 'Misc_PhysicalAddress:          0x10D0    ',
 'Misc_VirtualSize:              0x10D0    ',
 'VirtualAddress:                0x1000    ',
 'SizeOfRawData:                 0x1200    ',
 'PointerToRawData:              0x400     ',
 'PointerToRelocations:          0x0       ',
 'PointerToLinenumbers:          0x0       ',
 'NumberOfRelocations:           0x0       ',
 'NumberOfLinenumbers:           0x0       ',
 'Characteristics:               0x60000020']
['[IMAGE_SECTION_HEADER]',
 'Name:                          .data',
 'Misc:                          0x6C      ',
 'Misc_PhysicalAddress:          0x6C      ',
 'Misc_VirtualSize:              0x6C      ',
 'VirtualAddress:                0x3000    ',
 'SizeOfRawData:                 0x200     ',
 'PointerToRawData:              0x1600    ',
 'PointerToRelocations:          0x0       ',
 'PointerToLinenumbers:          0x0       ',
 'NumberOfRelocations:           0x0       ',
 'NumberOfLinenumbers:           0x0       ',
 'Characteristics:               0xC0000040']
['[IMAGE_SECTION_HEADER]',
 'Name:                          .rsrc',
 'Misc:                          0x1B40    ',
 'Misc_PhysicalAddress:          0x1B40    ',
 'Misc_VirtualSize:              0x1B40    ',
 'VirtualAddress:                0x4000    ',
 'SizeOfRawData:                 0x1C00    ',
 'PointerToRawData:              0x1800    ',
 'PointerToRelocations:          0x0       ',
 'PointerToLinenumbers:          0x0       ',
 'NumberOfRelocations:           0x0       ',
 'NumberOfLinenumbers:           0x0       ',
 'Characteristics:               0x40000040']
```

Il suffit de faire pour chaque section la démarche suivante : aller à l'offset `PointerToRawData` dans le fichier, lire à l'adresse `ImageBaseAddress+VirtualAddress` `SizeOfRawData` octets et les écrire dans le fichier. Nous obtenons au final un fichier d'une taille de `0x5C00` octets.

### 4.7 Injection et exécution de code arbitraire

Toutes les manipulations faites précédemment ne faisaient que lire et interpréter les différentes structures composant le noyau. Dans l'optique où nous disposons d'un accès en lecture/écriture à la mémoire (via le FireWire par exemple), nous pouvons envisager de modifier directement ces structures. Nous pouvons aussi penser à injecter et exécuter du code.

Nous pouvons envisager plusieurs types d'utilisations d'un accès à la mémoire physique. Du côté offensif, nous pouvons penser à un scénario d'accès physique à un poste en fonctionnement. Du patch de `SeAccessCheck` au déverrouillage de la station en passant par le lancement d'un shell admin, les possibilités sont nombreuses et variées.

Nous avons décidé de présenter différentes techniques permettant, sous réserve d'avoir un accès à la mémoire physique, d'exécuter du code soit en mode utilisateur soit en mode noyau.

Avant de présenter ces techniques, nous allons présenter brièvement une méthode connue d'élévation de privilèges.

**Élévation de privilèges** Pour commencer, voici un exemple de manipulation des objets du noyau. Nous allons élever localement les privilèges d'un processus. Ceux-ci sont stockés dans la structure `_EPROCESS` sous la forme d'un pointeur appelé `Token` situé à l'offset `0xc8`.

```
[.. .]
+0x0c8 Token            : struct _EX_FAST_REF, 3 elements, 0x4 bytes
      +0x000 Object           : 0xe10d6036
      +0x000 RefCnt           : Bitfield 0y110
      +0x000 Value            : 0xe10d6036
[.. .]
```

Ce *token* est un objet enveloppant les descripteurs de sécurité du processus. Il correspond à une structure `_TOKEN` décrivant les privilèges de l'objet (dans notre cas celui du processus). Pour obtenir l'adresse de la structure, il ne faut pas tenir des 3 bits de poids faible.

```
struct _TOKEN, 30 elements, 0xa8 bytes
   +0x000 TokenSource      : struct _TOKEN_SOURCE, 2 elements, 0x10 bytes
   +0x010 TokenId          : struct _LUID, 2 elements, 0x8 bytes
   +0x018 AuthenticationId : struct _LUID, 2 elements, 0x8 bytes
   +0x020 ParentTokenId    : struct _LUID, 2 elements, 0x8 bytes
   +0x028 ExpirationTime   : union _LARGE_INTEGER, 4 elements, 0x8 bytes
  +0x030 TokenLock       : Ptr32 to struct _ERESOURCE, 13 elements, 0x38 bytes
   +0x038 AuditPolicy     : struct _SEP_AUDIT_POLICY, 3 elements, 0x8 bytes
```

```
   +0x040 ModifiedId       : struct _LUID, 2 elements, 0x8 bytes
   +0x048 SessionId        : Uint4B
   +0x04c UserAndGroupCount : Uint4B
   +0x050 RestrictedSidCount : Uint4B
   +0x054 PrivilegeCount   : Uint4B
   +0x058 VariableLength   : Uint4B
   +0x05c DynamicCharged   : Uint4B
   +0x060 DynamicAvailable : Uint4B
   +0x064 DefaultOwnerIndex : Uint4B
  +0x068 UserAndGroups    : Ptr32 to struct _SID_AND_ATTRIBUTES, 2 elements, 0x8 bytes
  +0x06c RestrictedSids   : Ptr32 to struct _SID_AND_ATTRIBUTES, 2 elements, 0x8 bytes
   +0x070 PrimaryGroup     : Ptr32 to Void
  +0x074 Privileges      : Ptr32 to struct _LUID_AND_ATTRIBUTES, 2 elements, 0xc bytes
   +0x078 DynamicPart      : Ptr32 to Uint4B
   +0x07c DefaultDacl      : Ptr32 to struct _ACL, 5 elements, 0x8 bytes
   +0x080 TokenType        : Enum _TOKEN_TYPE,  2 total enums
  +0x084 ImpersonationLevel : Enum _SECURITY_IMPERSONATION_LEVEL,  4 total enums
   +0x088 TokenFlags       : Uint4B
   +0x08c TokenInUse       : UChar
  +0x090 ProxyData       : Ptr32 to struct _SECURITY_TOKEN_PROXY_DATA, 5 elements, 0x18 bytes
  +0x094 AuditData       : Ptr32 to struct _SECURITY_TOKEN_AUDIT_DATA, 3 elements, 0xc bytes
   +0x098 OriginatingLogonSession : struct _LUID, 2 elements, 0x8 bytes
   +0x0a0 VariablePart     : Uint4B
```

Nous imaginons le scénario suivant : en simple utilisateur, nous lançons un terminal de commandes (`cmd.exe`). Ensuite nous retrouvons la structure `EPROCESS` correspondant au processus `cmd.exe`. Avec WinDbg, nous examinons les privilèges actuels de notre terminal.

```
lkd> !token e10d6030
_TOKEN e10d6030
TS Session ID: 0
User: S-1-5-21-515967899-1078145449-839522115-1003
[...]
Privs:
 00 0x000000017 SeChangeNotifyPrivilege         Attributes - Enabled Default
 01 0x000000013 SeShutdownPrivilege                Attributes -
 02 0x000000019 SeUndockPrivilege                  Attributes -
 03 0x00000001e SeCreateGlobalPrivilege         Attributes - Enabled Default
[.. .]
```

L'étape suivante consiste à retrouver un autre processus ayant des droits élevés (par exemple `SYSTEM`).

Il suffit ensuite de prendre l'adresse du `Token` du processus privilégié et de l'écrire à la place de l'adresse du token du processus `cmd.exe`.

Celui-ci a alors les droits `SYSTEM`. Nous vérifions cela en examinant les privilèges de notre `cmd.exe`.

```
lkd> !token e1502bb8
_TOKEN e1502bb8
TS Session ID: 0
User: S-1-5-18
[.. .]
Privs:
 00 0x000000007 SeTcbPrivilege                 Attributes - Enabled Default
 01 0x000000002 SeCreateTokenPrivilege             Attributes -
 02 0x000000009 SeTakeOwnershipPrivilege           Attributes -
 03 0x00000000f SeCreatePagefilePrivilege       Attributes - Enabled Default
 04 0x000000004 SeLockMemoryPrivilege           Attributes - Enabled Default
 05 0x000000003 SeAssignPrimaryTokenPrivilege      Attributes -
 06 0x000000005 SeIncreaseQuotaPrivilege           Attributes -
 07 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes - Enabled Default
 08 0x000000010 SeCreatePermanentPrivilege       Attributes - Enabled Default
 09 0x000000014 SeDebugPrivilege                Attributes - Enabled Default
 10 0x000000015 SeAuditPrivilege                Attributes - Enabled Default
 11 0x000000008 SeSecurityPrivilege                Attributes - Enabled
 12 0x000000016 SeSystemEnvironmentPrivilege       Attributes -
 13 0x000000017 SeChangeNotifyPrivilege         Attributes - Enabled Default
 14 0x000000011 SeBackupPrivilege                  Attributes -
 15 0x000000012 SeRestorePrivilege                 Attributes -
 16 0x000000013 SeShutdownPrivilege                Attributes -
 17 0x00000000a SeLoadDriverPrivilege              Attributes - Enabled
 18 0x00000000d SeProfileSingleProcessPrivilege   Attributes - Enabled Default
 19 0x00000000c SeSystemtimePrivilege              Attributes -
 20 0x000000019 SeUndockPrivilege                  Attributes - Enabled
 21 0x00000001c SeManageVolumePrivilege            Attributes -
 22 0x00000001d SeImpersonatePrivilege          Attributes - Enabled Default
 23 0x00000001e SeCreateGlobalPrivilege         Attributes - Enabled Default
[...]
```

Les privilèges ont considérablement augmenté !

**Exécution de code** Le problème principal lorsque nous voulons exécuter du code vient de la nature même de l'accès que nous possédons. En effet nous pouvons lire et écrire arbitrairement dans la mémoire. C'est très utile pour modifier les structures internes mais comment faire pour prendre le contrôle du flux d'exécution ?

La solution passe par le détournement de pointeurs de fonctions. Si, par exemple, nous changeons l'adresse d'un gestionnaire d'interruption dans l'IDT par notre propre code, nous prendrons la main à chaque fois que cette interruption sera appelée.

Il se pose alors plusieurs problèmes :

– où stocker notre code ?

– quel pointeur allons-nous remplacer ?
– qu'allons-nous mettre dans notre code ?

Dans une optique où nous voudrions installer un morceau de code résident (du moins jusqu'à l'extinction de la machine), nous voulons aussi pouvoir communiquer avec lui. Comment faire ?

Une première idée serait de bâtir un protocole reposant sur les spécificités de l'accès à la mémoire. Par exemple dans le cas du FireWire, cela consisterait à utiliser ses fonctionnalités intrinsèques. Cela pose des difficultés. Tout d'abord, cela implique d'écrire un morceau de code pour l'OS cible et un pour l'OS attaquant. Étant donné la forte imbrication du code dans l'OS, il est clair que le code ne sera pas portable. En résumé, nous avons un bout de code responsable de la gestion de la communication pour chaque version d'OS.

Une deuxième idée consisterait plutôt à fonder le protocole directement sur la manipulation de la mémoire. Nous pouvons imaginer par exemple une structure de données permettant le dialogue entre le code s'exécutant sur la cible et l'attaquant.

Après cette parenthèse, revenons à notre problème principal. Première question : où stocker notre code ? Il existe une zone de mémoire partagée appelée `_KUSER_SHARED_DATA`, nous en avons déjà parlé au début de l'article. Cette zone a la particularité d'être mappée à une adresse fixe et d'être accessible à la fois en mode utilisateur et en mode noyau. Le début de la page est occupé par cette structure mais le reste de la page est disponible. La partie occupée a une taille de `0x300` octets. En prenant une marge de `0x100` octets, cela nous laisse environ `0xc00` octets.

En mode utilisateur, l'adresse est `0x7ffe0000`, en mode noyau c'est `0xffdf0000`. Il faut aussi penser aux permissions d'accès à cette page mémoire. Suivant la configuration du poste, elle peut être en lecture seule et non-exécutable. Comme le processeur appelle le gestionnaire de fautes de page uniquement s'il a rencontré une erreur, il est possible de modifier directement les PDE et PTE pour lever ces interdictions.

Dans le cas où le code est plus grand que la mémoire disponible, il faut passer par une étape supplémentaire et faire un bootstrap. Le code que nous copions réservera une zone de mémoire et renverra un pointeur vers celle-ci. Nous pouvons aussi imaginer parcourir la liste de toutes les pages disponibles et mettre de côté des pages pour notre utilisation personnelle.

Maintenant que nous savons où stocker notre code, nous allons examiner quels pointeurs nous pouvons écraser. Tout dépend du contexte dans lequel nous voulons exécuter notre code. En mode utilisateur, il existe un point d'entrée accédé

par tous les processus utilisant des appels systèmes. Son adresse est stockée dans la structure `_KUSER_SHARED_DATA` dans le champ `SystemCall`.

```
struct _KUSER_SHARED_DATA, 39 elements, 0x338 bytes
    [...]
   +0x300 SystemCall       : Uint4B
   +0x304 SystemCallReturn : Uint4B
   +0x308 SystemCallPad    : [3] Uint8B
   +0x320 TickCount        : struct _KSYSTEM_TIME, 3 elements, 0xc bytes
   +0x320 TickCountQuad    : Uint8B
   +0x330 Cookie           : Uint4B
```

Ensuite, il faut déterminer dans quel processus nous voulons que notre code s'exécute. Par exemple avec le bout de code assembleur suivant, nous exécutons le code uniquement si le PID du processus est le même que le dword écrit à l'adresse `0x7ffe03fc`.

```
entrypoint:
    push eax
    mov eax, dword [fs:0x18]
    mov eax, dword [ds:eax+0x20]
    cmp eax, dword [ds:0x7ffe03fc]
    jnz exit
    pushad
    mov eax, 0x7ffe0500
    call eax
    popad

exit:
    pop eax
    jmp dword [ds:0x7ffe03f8]
```

À l'adresse `0x7ffe0500` se situe le début du code à exécuter si le PID est le bon. À l'adresse `0x7ffe03f8` se situe l'adresse du handler d'origine.

Comme le code peut être écrit n'importe où, il doit être capable de retrouver tout seul les adresses des fonctions dont il a besoin. Pour cela, les techniques présentées dans [14,15,16] sont très utiles.

Pour exécuter du code en mode noyau, il y a énormément de possibilités. Il est possible de modifier directement l'IDT ou la SSDT, ou les pointeurs que nous retrouvons dans les structures utilisées par l'*object manager*.

En résumé, les possibilités sont infinies et limitées uniquement par l'imagination de l'attaquant.

## 5 Conclusion

Au travers de ce document, nous avons effectué un court voyage au sein de la mémoire de Windows. Nous avons d'abord rappelé comment utiliser le FireWire pour accéder à la mémoire. Ensuite, nous avons détaillé les différentes manières de traduire une adresse virtuelle en une adresse physique, d'abord du point de vue du processeur puis du point de vue du noyau. Finalement, nous avons utilisé tout cela pour expliquer les structures mises en jeu par certaines parties du noyau, en particulier l'*object manager* et le *configuration manager*. Nous avons brièvement montré comment injecter et exécuter du code via la mémoire.

Toutefois, nous n'avons exploré qu'une petite partie de la mémoire et de nombreuses structures restent à étudier. Nous pouvons par exemple citer celles mises en jeu par les jetons de sécurité ou encore celles utilisées par le gestionnaire de cache. Pour des raisons de performances, le noyau utilise un *cache manager* responsable du chargement en mémoire des fichiers. Ainsi, des données apparemment inaccessibles pourront peut-être être retrouvées en interprétant ses structures internes. En approfondissant la possibilité d'injection et d'exécution de code, nous pouvons envisager la réalisation d'un débogueur furtif entièrement via le FireWire. Il permettra, par exemple, l'étude de malwares particulièrement protégés.

## Références


1. Dornseif, M. : All your memory are belong to us. http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf (2005)
2. Boileau, A. : Hit by a bus : Physical access attacks with firewire. http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf (2006)
3. Ruff, N. : Autopsie d'une intrusion «tout en mémoire» sous windows. http://actes.sstic.org/SSTIC07/Forensics_Memoire_Windows/ (2007)
4. Intel : 1394 open host controller interface specification. http://developer.intel.com/technology/1394/download/ohci_11.htm
5. Wikipedia : Translation lookaside buffer. http://en.wikipedia.org/wiki/Translation_Lookaside_Buffer
6. Schuster, A. : Searching for processes and threads in microsoft windows memory dumps. http://dfrws.org/2006/proceedings/2-Schuster-pres.pdf (2006)
7. Kornblum, J.D. : Using every part of the buffalo in windows memory analysis. http://jessekornblum.com/research/papers/buffalo.pdf
8. Russinovich, M.E., Solomon, D.A. : Microsoft windows internals, 4th edition
9. Barbosa, E. : Find some non-exported kernel variables in windows xp. http://www.rootkit.com/newsread.php?newsid=101
10. ionescu007 : Getting kernel variables from kdversionblock, part2. http://www.rootkit.com/newsread.php?newsid=153
11. Russinovich, M., Cogswell, B. : Windows sysinternals. http://technet.microsoft.com/en-us/sysinternals/default.aspx
12. clark@hushmail.com : Security accounts manager. http://www.beginningtoseethelight.org/ntsecurity/index.php (2005)
13. Carrera, E. : pefile. http://code.google.com/p/pefile/
14. Jack, B. : Remote windows kernel exploitation step into the ring 0. http://research.eeye.com/html/papers/download/StepIntoTheRing.pdf (2005)
15. bugcheck, skape : Kernel-mode payloads on windows. http://www.uninformed.org/?v=3&a=4&t=pdf (2005)
16. skape, Skywing : A catalog of windows local kernel-mode backdoor techniques. http://www.uninformed.org/?v=8&a=2&t=pdf (2007)