Virtdbg : Un débogueur noyau utilisant la technologie de virtualisation matérielle VT-x

Damien Aumaitre et Christophe Devine
damien.aumaitre(@)sogeti.com
christophe.devine(@)sogeti.com

Sogeti/ESEC

Résumé L'observateur perturbe la mesure : cette règle s'applique aussi dans le monde de l'analyse dynamique de programmes, où le fait d'attacher un débogueur peut entraîner des effets de bords modifiant le comportement du système étudié. Ainsi, plusieurs codes malveillants tentent de détecter la présence d'un débogueur ou d'un environnement de virtualisation pour altérer leurs actions en conséquence.

Le projet virtdbg a pour but de créer un débogueur fondé sur un hyperviseur minimaliste. Il est injecté directement dans la mémoire du système cible en utilisant un transfert DMA et offre un ensemble de primitives simples permettant de contrôler la cible (pose de points d'arrêt, lecture et écriture de la mémoire, gestion des exceptions, etc.) Ces primitives s'appuient sur les fonctions de virtualisation matérielle des processeurs Intel et AMD récents. Le système est très peu altéré et le contrôle est total.

Les communications entre l'hyperviseur et le débogueur reposent sur le protocole gdbserver, implémenté au-dessus d'un médium physique choisi pour minimiser l'impact sur le système à étudier (USB ou Ethernet). Ce protocole a par ailleurs comme avantage d'être compatible avec différents débogueurs comme gdb, IDA Pro, GenDbg, etc.

1 Introduction

1.1 Revue de l'existant

Il existe un grand nombre de débogueurs ciblant différents systèmes d'exploitation, en particulier Windows. Sans prétendre à l'exhaustivité, voici une brève liste des plus connus.

Débogueurs distants Largement utilisé dans le monde du logiciel libre, gdb [1] présente l'avantage d'être versatile : il permet de

déboguer aussi bien le noyau de Linux que celui de Mac OS X. Gdb implémente le protocole de communication du même nom et communique avec un serveur pour déboguer à distance, que ce soit depuis un port série ou une connexion tcp/udp. Son principal désavantage est une interface utilisateur peu conviviale au premier abord.

GenDbg [2] est, comme son nom l'indique, un débogueur générique. De même que gdb et IDA Pro [3], il permet de déboguer un système d'exploitation en s'interfaçant avec un serveur gdb comme kgdb [4] pour le noyau Linux, ou l'équivalent Mac OS X [5].

Finalement, WinDbg [6] est destiné à déboguer les applications ou noyaux de la famille de systèmes d'exploitation Windows. Disposant de nombreuses extensions qui font parfois défaut à gdb et d'une aide intégrée, sa ligne de commande est considérée par l'auteur comme plus pratique d'utilisation – il souffre cependant de l'absence d'interface graphique. WinDbg utilise un protocole de communication propriétaire et non documenté, implémenté dans le noyau au dessus de différents médiums (série, câble de débogage USB, FireWire et plus récemment Ethernet).

Débogueurs locaux SoftICE [7] est l'ancêtre des débogueurs noyau sous Windows ; il a inspiré de nombreux autres débogueurs comme rr0d [8] et Syser [9]. Il n'est malheureusement plus maintenu depuis 2007. SoftICE (et ses dérivés) présente l'avantage unique de pouvoir déboguer le système d'exploitation en local, avec cependant un prix à payer en termes de stabilité du système ; ainsi, il n'est pas rare en utilisant Syser d'avoir un écran bleu lors de manipulations simples (comme de faire défiler le code désassemblé).

Notons que WinDbg a la capacité de déboguer en local le noyau Windows, avec de nombreuses limitations cependant (pas de possibilité de poser des points d'arrêt, accès en lecture seulement à la mémoire).

1.2 Limites des débogueurs actuels

L'objectif majeur d'un débogueur est la transparence vis-à-vis du programme débogué, qui devrait en théorie s'exécuter dans les mêmes conditions qu'il soit débogué ou non. Ainsi, de nombreux

codes malveillants tentent d'identifier leur environnement d'exécution pour modifier leur comportement en fonction.

Débogueurs distants Ces derniers requièrent l'installation d'un "stub", petit module intégré (comme kgdb) au système d'exploitation. Plusieurs structures fondamentales seront modifiées telles que la table des gestionnaires d'interruptions (IDT) dans l'architecture x86.

L'activation du mode "/DEBUG" sous Windows Vista et 7 x64 désactive PatchGuard et modifie le comportement du système [10] ; par exemple, les protections contre la copie (DRM) sont désactivées pour interdire leur débogage. Le mode DEBUG, jamais utilisé par le grand public, pourrait de plus être détecté par un code malveillant qui voudrait rendre son analyse à rebours plus difficile. Notons que lorsque le drapeau /DEBUG n'est pas activé, PatchGuard vérifie l'intégrité de plusieurs structures importantes comme l'IDT et les MSR, empêchant effectivement le débogage via un module noyau tiers parti.

Un second désavantage est que le stub doit communiquer avec le débogueur ; pour cela, il fait appel aux routines de communications du système débogué. Il devient alors très difficile, voire impossible, de déboguer des fonctions de très bas niveau comme le traitement des requêtes d'interruption, car ces mêmes requêtes sont utilisées par le stub. Une alternative est la réimplémentation dans le stub du pilote pour le périphérique choisi, ce qui s'avère compliqué pour plusieurs protocoles haut débit (USB EHCI, FireWire et Ethernet 100baseT).

Débogueurs locaux Les limitations citées précédemment s'appliquent, sachant que l'empreinte d'un débogueur local est beaucoup plus importante (il faut par exemple intercepter les interruptions de gestion du clavier et de l'écran) et dépendante de plus de chemins d'accès dans le noyau comme les routines de traitement du pilote vidéo. Un tel débogueur en devient plus fragile et facile à détecter. Il est difficile de faire un débogueur local générique ; par exemple, l'adresse en mémoire du "framebuffer" dépendra du système d'exploitation et du pilote vidéo (si ce dernier est mappé en intégralité, ce qui n'est pas forcément le cas).

1.3 VirtDbg

Au vu des constats précédents, nous avons choisi d'implémenter un débogueur qui minimise l'impact sur les structures du système d'-exploitation. Pour cela deux techniques ont été choisies. Tout d'abord, la virtualisation dans les processeurs Intel [11] et AMD64 [12] fournit la possibilité d'intercepter le flux d'exécution du système en restant très discret vis-à-vis de ce dernier. VirtDbg est ainsi constitué d'un hyperviseur situé en "ring -1" et assurant une séparation nette entre stub de déboguage et système débogué.

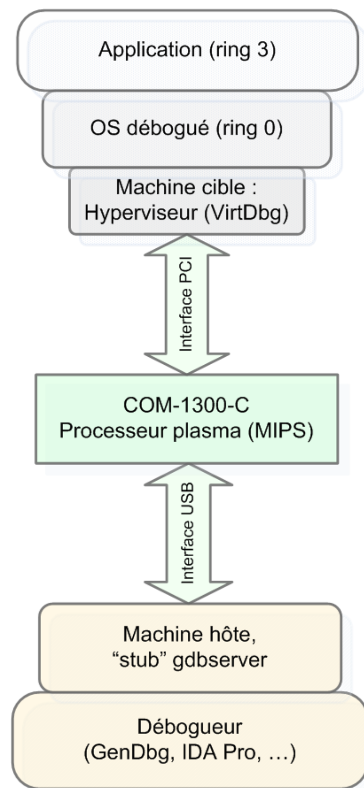
Lorsque l'hyperviseur prend la main sur le système invité, celui-ci est totalement figé. Il est donc nécessaire que l'hyperviseur propose un moyen de communication entre le serveur (résidant sur la machine à déboguer) et le client (résidant sur une autre machine). La séparation a aussi un autre avantage non négligeable : elle permet de garder le code de l'hyperviseur le plus simple possible, et évite d'utiliser les routines de communications de l'OS débogué.

Nous avons choisi d'utiliser les accès à la mémoire physique depuis le bus PCI (accès DMA) comme moyen d'installation de l'hyperviseur puis de communication avec ce dernier. L'accès à la mémoire physique se fait grâce à la carte COM-1300-C construite par ComBlock, qui est dotée d'un FPGA implémentant directement le protocole PCI (réimplémentation en repartant à zéro des travaux présentés à SSTIC en 2009 [13]). De la même manière, il serait possible d'injecter l'hyperviseur par un accès DMA en Firewire, ou bien en s'installant en mémoire lors du processus de démarrage, par exemple en modifiant le secteur de démarrage du disque dur.

Les fonctions plus complexes (reconstitution de la mémoire virtuelle, d'attachement à un processus, etc.) sont fondées sur les travaux présentés en 2008 à SSTIC [14], et sont externalisées dans le client. De cette façon, l'hyperviseur restera le plus simple possible en ne proposant que les primitives les plus fondamentales de lecture et écriture en mémoire physique et traitement des exceptions.

La figure 1.3 présente de manière synthétique le mode de fonctionnement du débogueur.

Dans un premier temps, nous exposerons le cahier des charges, puis seront présentées l'implémentation concrète de VirtDbg, les difficultés rencontrées et finalement nous aborderons les pistes futures.



2 Cahier des charges et architecture

Les débogueurs à distance se basent sur plusieurs composants pour mener à bien leurs tâches :

- un programme de chargement (injecteur) qui permet d'insérer le code du module ("stub") au sein du système d'exploitation (typiquement un module noyau) ;
- le stub qui implémente toutes les primitives offertes au débogueur (pose de points d'arrêts, lecture/écriture de registres, etc.) ;
- un module de communication pour envoyer des ordres au module.

Les débogueurs existants n'ont pas une séparation forte entre ces différents composants, il est donc difficile de les changer (ainsi, le stub kd est fortement intégré au noyau de Windows). Ils sont souvent trop intrusifs (nécessité de démarrer en mode debug, de modifier du code du noyau, etc.) facilitant leur détection.

Considérons l'exemple d'un rootkit qui se protègerait de l'analyse en vérifiant que les noms des débogueurs noyau classiques ne sont pas présents dans la liste des modules chargés ; il détecterait de la sorte Syser et WinDbg.

Supposons maintenant que le nom du module soit modifié. Le débogueur arrive à se charger et tente de tracer le fonctionnement de ce code malveillant. Celui-ci utilise des contrôles d'intégrité. Toute tentative de pose d'un point d'arrêt logiciel (remplacement de l'instruction courante pour l'opcode INT3 afin de déclencher une interruption) sera donc détectée. Il ne reste donc plus que les points d'arrêts matériels qui ne sont malheureusement qu'au nombre de quatre, limitant fortement l'analyse. Ce code pourrait de plus "hooker" l'interruption 1 afin de détecter et bloquer les points d'arrêts matériels). Une autre technique consiste à utiliser les particularités des registres DR ([15]) afin détecter leurs modifications.

Le code malveillant étant au même niveau que le débogueur, on retrouve les mêmes classes d'anti-debug que pour l'userland. Il est donc nécessaire de se situer un cran au dessus de la cible.

Classiquement un débogueur noyau utilise les interruptions 1 et 3 (points d'arrêts matériels et logiciels) et l'interruption 14 (gestionnaire de faute de pages) pour contrôler la cible. Il propose aussi un mécanisme pour prendre la main (*breakin*) c'est-à-dire stopper l'exécution de la cible afin d'inspecter la mémoire et/ou les registres pour ensuite poser si besoin est des breakpoints et reprendre l'exécution.

Sur un système d'exploitation récent comme Windows 7 x64, il existe des mécanismes de sécurité pour protéger l'intégrité du noyau. Par exemple, PatchGuard va surveiller l'IDT, la GDT, la table des appels systèmes, l'intégrité du code du noyau et de certains drivers (liste non exhaustive). Toute modification entraîne un écran bleu et un redémarrage de l'ordinateur.

Notre débogueur doit proposer les primitives suivantes sans altérer le système d'exploitation :

- stopper la cible (*breakin*) ;
- modifier la mémoire ;
- modifier les registres ;
- poser les points d'arrêts.

Tout le problème revient à réussir à se placer au dessus de la cible. Les processeurs actuels proposent des fonctionnalités de vir-

tualisation matérielle. Cela permet de mettre le processeur dans un mode privilégié permettant de contrôler la cible sans l'altérer. Nous avons donc un moyen de déboguer le système d'exploitation. L'hyperviseur possède un contrôle très complet sur le système invité. Il peut monitorer les interruptions, l'accès à la mémoire, les entrées sorties.

Il faut donc maintenant pouvoir charger l'hyperviseur. Si l'on utilise un module noyau, nous nous retrouvons dans le cas des débogueurs précédemment cités : la modification des structures du système d'exploitation. Comment faire pour exécuter du code en contournant les mesures de sécurité du système d'exploitation et du processeur ? Nous allons utiliser les accès dits "DMA" (Direct Memory Access). Il a été précédemment montré [14] qu'avec un accès à la mémoire physique nous pouvons exécuter du code arbitraire.

Nous avons plusieurs moyens pour accéder à la mémoire physique par le DMA : le bus FireWire, le bus CardBus, en fait tout ce qui est connecté au bus PCI.

Nous avons choisi d'utiliser le bus CardBus. Les travaux présentés au SSTIC 2009 [13] ont donné lieu à une preuve de concept. Nous nous somme inspirés de ces travaux, corrigé différents bogues liés à l'implémentation du protocole PCI et utilisé le code VHDL d'un processeur MIPS open-source [16]. Ainsi, il devient possible de compiler du code C en assembleur MIPS et de charger ce code sur le FPGA, ce qui facilite considérablement la tâche.

Finalement, il reste un point à élucider : comment l'hyperviseur communiquera avec le débogueur ? Il n'est pas possible d'utiliser le code du système d'exploitation, figé lorsque l'hyperviseur prend la main. Nous avons choisi d'utiliser les accès DMA. Plusieurs raisons ont mené à ce choix :

- nous utilisons déjà la mémoire physique pour l'injection de l'hyperviseur ;
- cela permet de contrôler l'hyperviseur sans l'intervention du système d'exploitation ;
- nous avons déjà tout le code permettant de reconstruire l'espace d'adressage virtuel en utilisant la mémoire physique.

Ensuite il faut pouvoir communiquer avec le FPGA. Nous avons choisi d'utiliser le protocole gdbserver. Celui-ci est en mode texte, il

est parfaitement documenté et extensible. De nombreux outils sont capables de le comprendre, autorisant une plus grande interopérabilité.

Les parties suivantes vont expliquer en détail chacun des composants de *virtdbg*. Tout d'abord, nous allons expliquer comment est chargé l'hyperviseur par l'intermédiaire d'un accès à la mémoire physique. Ensuite nous décrirons en détail comment réaliser un hyperviseur et comment s'en servir en tant que débogueur. Finalement, nous ferons un rappel sur le protocole PCI et expliquerons les différentes couches de communications.

Le schéma suivant montre *virtdbg* avec tous ses composants :

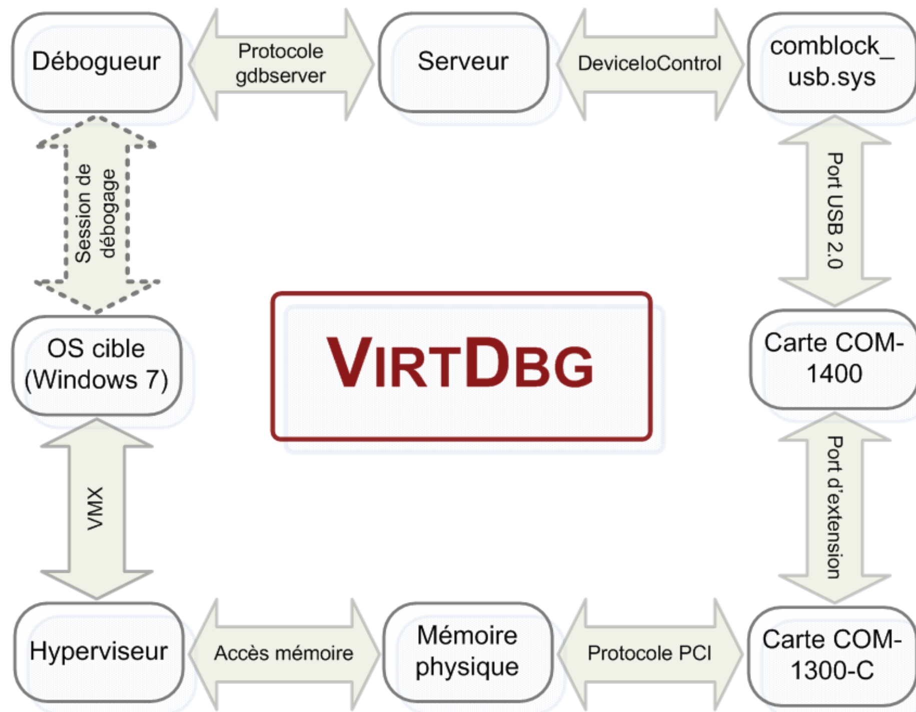


Figure 1. Schéma récapitulatif global

3 Loader

L'hyperviseur est conçu sous la forme d'un driver noyau classique, et peut donc être chargé en tant que tel. Cette méthode a néanmoins un certain nombre de désavantages :

– sous un système d'exploitation 64 bits comme Windows 7, les drivers doivent être signés. Il est possible de contourner cette limitation en désactivant cette vérification au démarrage du système (touche F8) ;
– charger un driver laisse des traces sur le système. Il apparait dans l'arborescence de l'object manager et dans la liste des drivers chargés. Nous pouvons utiliser des techniques de rootkits afin de nous enlever de ces listes, mais rien ne garantit que le chargement n'a pas laissé des traces ailleurs.

Ces désavantages sont facilement contournés par une injection de code en utilisant la mémoire physique. Pour la suite nous considérons que nous avons un accès en lecture-écriture à la mémoire physique. La méthode retenue est la suivante :

– injection d'un stager chargé d'allouer suffisamment de mémoire pour pouvoir copier section par section le driver ;
– chargement de l'hyperviseur de manière identique à un loader de PE (parcours de la table des imports du driver afin de remplir les adresses virtuelles des fonctions utilisées par celui-ci et application des relocations si nécessaire).

3.1 Stager

Windows 7 64 bits utilise Patchguard. Celui-ci protège le noyau de modifications intempestives. Il faut donc être très précautionneux lors du choix des pointeurs à écraser afin de détourner le flux d'exécution.

Nous avons décidé de stocker le stager dans la première page d'un driver. En manipulant les PTE de la page, nous ajustons les permissions pour les mettre en RWX. Ce stager va classiquement allouer de la mémoire avec `ExAllocatePoolWithTag` et sauvegarder l'adresse physique du résultat.

3.2 PE Loader

Le loader utilise l'adresse physique renvoyée par le stager pour copier les sections du PE en respectant l'alignement des sections indiqué dans le header du PE. Il parcours ensuite la liste des exports des drivers chargés afin de remplir l'IAT du driver. Puis il applique les relocations si celles-ci sont nécessaires. Finalement, il modifie le pointeur utilisé par le stager pour le faire pointer sur le point d'entrée du driver. Une fois le driver chargé, le hook est restauré.

3.3 Reconstruction de l'espace virtuel du noyau sous Windows 7 64 bits

Afin de réaliser les deux étapes précédentes, il faut reconstruire le lien entre les adresses virtuelles et les adresses physiques. Dans nos travaux précédents[14] nous avons montré comment reconstruire l'espace d'adressage virtuel sous Windows XP. Sous Windows 7, en particulier avec la version 64 bits, la méthode n'est plus la même. Précédemment la méthode utilisée était la suivante :

- avec une signature (dépendante de la version de l'OS), nous retrouvions l'adresse physique d'une structure `_EPROCESS` ;
- celle-ci contenait une sauvegarde de la valeur du registre `cr3` permettant de convertir une adresse virtuelle en adresse physique ;
- les structures `_EPROCESS` sont liées entre-elles grâce à une liste doublement chaînée. Nous pouvions donc reconstruire l'espace d'adressage de tous les processus.

Cependant, des informations très intéressantes (la liste des drivers chargés, la racine de l'object directory par exemple) ne sont pas présentes dans la structure `_EPROCESS`. Celles-ci sont stockées dans une structure `_KDDEBUGGER_DATA64` (voir annexe pour sa définition) atteignable à partir du champ `KdVersionBlock` de la structure `_KPCR`. Sous Windows XP, cette structure est toujours mappée à une adresse virtuelle fixe (`0xffdff000`). Il était ainsi possible de parcourir la liste des drivers chargés.

Malheureusement cette technique ne fonctionne plus sous Windows 7. En effet, même si nous pouvons appliquer la même méthode pour retrouver les structures `_EPROCESS`, il est beaucoup plus difficile de retrouver les structures `_KPCR`. Leurs adresses sont différentes

à chaque démarrage, car elles sont situées dans la section `.data` du noyau dont l'adresse de base est rendue aléatoire.

La méthode que nous avons conçu et implémenté est la suivante :

- nous cherchons le fichier PE du noyau dans la mémoire physique ;
- avec les informations de debug contenues dans le PE nous sommes capables d'identifier précisément la version du noyau et ainsi l'offset vers la première structure `_KPCR` ;
- dans celle-ci se trouve une sauvegarde des registres du processeur (`cr0`, `cr3`, `cr4`) permettant de connaître le mode de pagination utilisé par le processeur. Il devient alors possible de traduire les adresses virtuelles de l'espace noyau.
- en bonus dans le `_KPCR` se trouve le nombre de processeurs logiques actifs sur le système. Cela permet de savoir si le système est en mode SMP ou pas.

Chaque fichier PE peut contenir des informations de debug. Elles servent à donner des indications au débogueur afin de lui permettre d'identifier avec précision la version du binaire. Elles sont situées dans le *Debug Directory* qui possède la structure suivante (voir figure 1.1).

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD  MajorVersion;
    WORD  MinorVersion;
    DWORD Type;
    DWORD SizeOfData;
    DWORD AddressOfRawData;
    DWORD PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

Listing 1.1. structure `IMAGE_DEBUG_DIRECTORY`

Il faut examiner le champ `Type` pour déterminer le type d'informations de debug. Dans notre cas ces informations sont présentes dans des fichiers PDB et le type est `IMAGE_DEBUG_TYPE_CODEVIEW`. Les 4 premiers octets de la structure pointée par le champ `AddressOfRawData` permettent de déterminer la structure à utiliser. Si ceux-ci valent 'RSDS' alors nous avons affaire à une structure `_CV_INFO_PDB70` (voir figure 1.2).

```
typedef struct _CV_INFO_PDB70 {
```

```
        DWORD  CvSignature;
        GUID Signature;
        DWORD Age;
        BYTE PdbFileName[1];
    } CV_INFO_PDB70, *PCV_INFO_PDB70;
```

Listing 1.2. structure `CV_INFO_PDB70`

Le champ `Signature` est un `GUID` généré par le compilateur et identifiant de manière unique le binaire. Le champ `Age` est utilisé avec le champ `Signature` par le gestionnaire de symboles de Microsoft. Le champ `PdbFileName` donne le chemin vers le fichier PDB.

La méthode utilisée nous oblige à utiliser plusieurs offsets "en dur" ce qui est, en règle générale, une mauvaise chose. Cependant dans notre cas, en tant que débogueur il faut pouvoir identifier avec précision la cible du débogage. En utilisant les informations de debug c'est chose faite ! De plus nous pouvons utiliser les bibliothèques fournies par Microsoft afin de récupérer dynamiquement les symboles nécessaires. Le lecteur intéressé pourra se reporter à l'excellente analyse faite par DebugInfo[17].

Revenous maintenant à la structure `_KPCR`. Malheureusement, le champ `KdVersionBlock` de la structure `_KPCR` n'est plus rempli et vaut toujours 0. Nous avons besoin d'un autre offset permettant de trouver l'adresse de la structure `_KDDEBUGGER_DATA64`. Nous avons ensuite la liste des drivers chargés dans le champ `PsLoadedModuleList`, celle des processus dans le champ `PsActiveProcessHead` et la racine de l'object manager dans le champ `ObpRootDirectoryObject`. Les offsets requis pour la suite sont les suivants (symboles de debug) : `KiInitialPCR` pour la première structure `_KPCR`, `KdDebuggerDataBlock` pour la structure `_KDDEBUGGER_DATA64`.

Il faut ensuite trouver des pointeurs à écraser pour pouvoir exécuter du code arbitraire. Si PatchGuard est actif, nous ne pouvons pas modifier les tables du processeur (IDT, GDT, etc.) ni le code du noyau, de `hal.dll` et de `ndis.sys` ; il faut utiliser d'autres pointeurs. Nous allons vous présenter deux endroits où se situent des pointeurs de fonctions (en principe) non surveillés par PatchGuard.

Injection de code arbitraire en détournant les fonctions gérant les IRP des drivers Le premier est constitué par la table de fonctions gérant les différentes IRP envoyées au driver. Celle-ci se trouve

dans la structure `_DRIVER_OBJECT` (voir figure 1.3) dans le champ `MajorFunction`.

```
typedef struct _DRIVER_OBJECT
{
/*0x000*/     INT16        Type;
/*0x002*/     INT16        Size;
/*0x004*/     UINT8        _PADDING0_[0x4];
/*0x008*/     struct _DEVICE_OBJECT* DeviceObject;
/*0x010*/     ULONG32      Flags;
/*0x014*/     UINT8        _PADDING1_[0x4];
/*0x018*/     VOID*        DriverStart;
/*0x020*/     ULONG32      DriverSize;
/*0x024*/     UINT8        _PADDING2_[0x4];
/*0x028*/     VOID*        DriverSection;
/*0x030*/     struct _DRIVER_EXTENSION* DriverExtension;
/*0x038*/     struct _UNICODE_STRING DriverName;
/*0x048*/     struct _UNICODE_STRING* HardwareDatabase;
/*0x050*/     struct _FAST_IO_DISPATCH* FastIoDispatch;
/*0x058*/     ULONG64 DriverInit;
/*0x060*/     ULONG64 DriverStartIo;
/*0x068*/     ULONG64 DriverUnload;
/*0x070*/     ULONG64 MajorFunction[28];
}DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Listing 1.3. structure `_DRIVER_OBJECT`

Il nous reste plus qu'à trouver ces structures `_DRIVER_OBJECT`. Lors du parcours de la liste doublement chaînée des drivers (`PsLoadedModuleList`), nous suivons des structures de type `_LDR_DATA_TABLE_ENTRY` (voir figure 1.4). Nous n'avons pas de pointeurs vers une structure `_DRIVER_OBJECT`. Par contre, il y a tout ce qu'il faut pour parcourir l'image "PE" (adresse de base et taille de l'image). Ces informations serviront ensuite pour écrire le loader.

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
/*0x000*/     struct _LIST_ENTRY InLoadOrderLinks;
/*0x010*/     struct _LIST_ENTRY InMemoryOrderLinks;
/*0x020*/     struct _LIST_ENTRY InInitializationOrderLinks;
/*0x030*/     VOID*        DllBase;
/*0x038*/     VOID*        EntryPoint;
/*0x040*/     ULONG32      SizeOfImage;
/*0x044*/     UINT8        _PADDING0_[0x4];
/*0x048*/     struct _UNICODE_STRING FullDllName;
/*0x058*/     struct _UNICODE_STRING BaseDllName;
/*0x068*/     ULONG32      Flags;
/*0x06C*/     UINT16       LoadCount;
/*0x06E*/     UINT16       TlsIndex;
              union
              {
```

```
/*0x070*/         struct _LIST_ENTRY HashLinks;
                  struct
                  {
/*0x070*/             VOID*        SectionPointer;
/*0x078*/             ULONG32      CheckSum;
/*0x07C*/             UINT8        _PADDING1_[0x4];
                  };
              };
              union
              {
/*0x080*/         ULONG32      TimeDateStamp;
/*0x080*/         VOID*        LoadedImports;
              };
/*0x088*/     struct _ACTIVATION_CONTEXT*
    EntryPointActivationContext;
/*0x090*/     VOID*        PatchInformation;
/*0x098*/     struct _LIST_ENTRY ForwarderLinks;
/*0x0A8*/     struct _LIST_ENTRY ServiceTagLinks;
/*0x0B8*/     struct _LIST_ENTRY StaticLinks;
/*0x0C8*/     VOID*        ContextInformation;
/*0x0D0*/     UINT64       OriginalBase;
/*0x0D8*/     union _LARGE_INTEGER LoadTime;
}LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Listing 1.4. structure `_LDR_DATA_TABLE_ENTRY`

Pour trouver la liste des structures `_DRIVER_OBJECT` il faut s' aventurer sur le terrain de l'object manager. Nous voyons l'arborescence de l'object manager quand nous utilisons un outil de type WinObj. La racine de cette arborescence est une structure de type `_OBJECT_DIRECTORY` (voir figure 1.5) dont l'adresse est stockée dans le champ `ObpRootDirectoryObject` de la structure `_KDDEBUGGER_DATA64`.

```
typedef struct _OBJECT_DIRECTORY
{
/*0x000*/     struct _OBJECT_DIRECTORY_ENTRY* HashBuckets[37]
/*0x128*/     struct _EX_PUSH_LOCK Lock;
/*0x130*/     struct _DEVICE_MAP* DeviceMap;
/*0x138*/     ULONG32      SessionId;
/*0x13C*/     UINT8        _PADDING0_[0x4];
/*0x140*/     VOID*        NamespaceEntry;
/*0x148*/     ULONG32      Flags;
/*0x14C*/     UINT8        _PADDING1_[0x4];
}OBJECT_DIRECTORY, *POBJECT_DIRECTORY;
```

Listing 1.5. structure `_OBJECT_DIRECTORY`

Les `HashBuckets` contiennent une table de structures `_OBJECT_DIRECTORY_ENTRY` (voir figure 1.6) qui sont elles-mêmes liées entre elles par le champ `ChainLink`. Le champ `Object` contient un pointeur vers l'objet en question.

```
typedef struct _OBJECT_DIRECTORY_ENTRY
{
/*0x000*/      struct _OBJECT_DIRECTORY_ENTRY* ChainLink;
/*0x008*/      VOID*        Object;
/*0x010*/      ULONG32      HashValue;
/*0x014*/      UINT8        _PADDING0_[0x4];
}OBJECT_DIRECTORY_ENTRY, *POBJECT_DIRECTORY_ENTRY;
```

Listing 1.6. structure `_OBJECT_DIRECTORY_ENTRY`

Les drivers sont situés dans le répertoire `\Driver` de l'object manager. Munis de ces informations nous sommes en mesure de retrouver le driver qui nous intéresse, de copier dans sa première page exécutable le code du stager et d'exécuter celui-ci en détournant le flux d'exécution par un hook sur la table des fonctions du driver.

Exécution de code arbitraire avec les structures `_OBJECT_TYPE`
Une autre méthode pour trouver des pointeurs de fonctions consiste à utiliser ceux contenus dans les structures de type `_OBJECT_TYPE_INITIALIZER`. Avant d'aborder ceux-ci, regardons de plus près les structures utilisées par l'object manager. Chacun des objets gérés par l'object manager possède une structure `_OBJECT_HEADER` (voir figure 1.7) située avant le contenu de l'objet (la liste des objets se trouve dans le répertoire `\ObjectTypes`).

```
typedef struct _OBJECT_HEADER
{
/*0x000*/      INT64        PointerCount;
               union
               {
/*0x008*/          INT64        HandleCount;
/*0x008*/          VOID*        NextToFree;
               };
/*0x010*/      struct _EX_PUSH_LOCK Lock;
/*0x018*/      UINT8        TypeIndex;
/*0x019*/      UINT8        TraceFlags;
/*0x01A*/      UINT8        InfoMask;
/*0x01B*/      UINT8        Flags;
/*0x01C*/      UINT8        _PADDING0_[0x4];
               union
               {
/*0x020*/          struct _OBJECT_CREATE_INFORMATION*
    ObjectCreateInfo;
/*0x020*/          VOID*        QuotaBlockCharged;
               };
/*0x028*/      VOID*        SecurityDescriptor;
/*0x030*/      struct _QUAD Body;
}OBJECT_HEADER, *POBJECT_HEADER;
```

Listing 1.7. structure `_OBJECT_HEADER`

Le champ `TypeIndex` contient l'index d'une structure `_OBJECT_TYPE` (voir figure 1.8) dans un tableau contenant tous les types connus (symbole `ObTypeIndexTable`).

```
typedef struct _OBJECT_TYPE
{
/*0x000*/     struct _LIST_ENTRY TypeList;
/*0x010*/     struct _UNICODE_STRING Name;
/*0x020*/     VOID*          DefaultObject;
/*0x028*/     UINT8          Index;
/*0x029*/     UINT8          _PADDING0_[0x3];
/*0x02C*/     ULONG32        TotalNumberOfObjects;
/*0x030*/     ULONG32        TotalNumberOfHandles;
/*0x034*/     ULONG32        HighWaterNumberOfObjects;
/*0x038*/     ULONG32        HighWaterNumberOfHandles;
/*0x03C*/     UINT8          _PADDING1_[0x4];
/*0x040*/     struct _OBJECT_TYPE_INITIALIZER TypeInfo;
/*0x0B0*/     struct _EX_PUSH_LOCK TypeLock;
/*0x0B8*/     ULONG32        Key;
/*0x0BC*/     UINT8          _PADDING2_[0x4];
/*0x0C0*/     struct _LIST_ENTRY CallbackList;
}OBJECT_TYPE, *POBJECT_TYPE;
```

Listing 1.8. structure `_OBJECT_TYPE`

Celle-ci contient une structure `_OBJECT_TYPE_INITIALIZER` (voir figure 1.9) dans le champ `TypeInfo`. Nous trouvons 8 pointeurs de fonctions dans cette structure (champs finissants par `Procedure`).

```
typedef struct _OBJECT_TYPE_INITIALIZER
{
/*0x000*/     UINT16         Length;
              union
              {
/*0x002*/         UINT8          ObjectTypeFlags;
                  struct
                  {
/*0x002*/             UINT8          CaseInsensitive : 1;
/*0x002*/             UINT8          UnnamedObjectsOnly : 1;
/*0x002*/             UINT8          UseDefaultObject : 1;
/*0x002*/             UINT8          SecurityRequired : 1;
/*0x002*/             UINT8          MaintainHandleCount : 1;
/*0x002*/             UINT8          MaintainTypeList : 1;
/*0x002*/             UINT8          SupportsObjectCallbacks : 1;
                  };
              };
/*0x004*/     ULONG32        ObjectTypeCode;
/*0x008*/     ULONG32        InvalidAttributes;
/*0x00C*/     struct _GENERIC_MAPPING GenericMapping;
/*0x01C*/     ULONG32        ValidAccessMask;
/*0x020*/     ULONG32        RetainAccess;
/*0x024*/     enum _POOL_TYPE PoolType;
/*0x028*/     ULONG32        DefaultPagedPoolCharge;
/*0x02C*/     ULONG32        DefaultNonPagedPoolCharge;
```

```
/*0x030*/      ULONG64 DumpProcedure;
/*0x038*/      ULONG64 OpenProcedure;
/*0x040*/      ULONG64 CloseProcedure;
/*0x048*/      ULONG64 DeleteProcedure;
/*0x050*/      ULONG64 ParseProcedure;
/*0x058*/      ULONG64 SecurityProcedure;
/*0x060*/      ULONG64 QueryNameProcedure;
/*0x068*/      ULONG64 OkayToCloseProcedure;
}OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;
```

Listing 1.9. structure `_OBJECT_TYPE_INITIALIZER`

Ces pointeurs sont appelés lorsque l'object manager manipule ces objets. Par exemple la fonction `OpenProcedure` est appelée lorsque l'objet est ouvert (sur un processus par exemple avec `OpenProcess`). Avec WinDbg il est facile de voir ce mécanisme.

Tout d'abord, nous affichons la liste de tous les types connus afin de récupérer l'adresse de la structure `_OBJECT_TYPE` correspondant aux objets de type `Process`.

```
0: kd> !object \ObjectTypes
Object: fffff8a000006830  Type: (fffffa8001787de0) Directory
    ObjectHeader: fffff8a000006800 (new version)
    HandleCount: 0  PointerCount: 44
    Directory Object: fffff8a000004740  Name: ObjectTypes

    Hash Address          Type          Name
    ---- -------          ----          ----
     00  fffffa8001820570 Type          TmTm
     01  fffffa8001821de0 Type          Desktop
         fffffa8001810f30 Type          Process
     03  fffffa8001810440 Type          DebugObject
     04  fffffa8001821c90 Type          TpWorkerFactory
     05  fffffa8001821b40 Type          Adapter
         fffffa8001787a40 Type          Token
     08  fffffa8001824570 Type          EventPair
     09  fffffa8002253f30 Type          PcwObject
         fffffa80018673a0 Type          WmiGuid
     11  fffffa80018683a0 Type          EtwRegistration
     12  fffffa80018332b0 Type          Session
         fffffa8001870230 Type          Timer
     13  fffffa8001822570 Type          Mutant
     16  fffffa8001821600 Type          IoCompletion
     17  fffffa8001821f30 Type          WindowStation
         fffffa800187e700 Type          Profile
     18  fffffa8001820080 Type          File
     21  fffffa8001870380 Type          Semaphore
     23  fffffa800186a3a0 Type          EtwConsumer
     25  fffffa8001820420 Type          TmTx
         fffffa8001787c90 Type          SymbolicLink
     26  fffffa800220d080 Type          FilterConnectionPort
         fffffa8001836390 Type          Key
```

```
            fffffa800187e5b0 Type          KeyedEvent
            fffffa80018625d0 Type          Callback
       28   fffffa8001810c90 Type          UserApcReserve
            fffffa8001810080 Type          Job
       29   fffffa80018219f0 Type          Controller
            fffffa8001810b40 Type          IoCompletionReserve
       30   fffffa80018218a0 Type          Device
            fffffa8001787de0 Type          Directory
       31   fffffa80018338e0 Type          Section
            fffffa8001833f30 Type          TmEn
            fffffa8001810de0 Type          Thread
       32   fffffa8001787f30 Type          Type
       33   fffffa800220d3c0 Type          FilterCommunicationPort
            fffffa800183c460 Type          PowerRequest
       35   fffffa80018202d0 Type          Choucroute
            fffffa8001825570 Type          Event
       36   fffffa800183cf30 Type          ALPC Port
            fffffa8001821750 Type          Driver
```

Examinons maintenant cette structure afin de récupérer les adresses des différentes procédures concernant les objets de type `Process`.

```
0: kd> dt -v _OBJECT_TYPE TypeInfo. fffffa8001810f30
nt!_OBJECT_TYPE
struct _OBJECT_TYPE, 12 elements, 0xd0 bytes
   +0x040 TypeInfo  : struct _OBJECT_TYPE_INITIALIZER, 25
       elements, 0x70 bytes
      +0x000 Length    : 0x70
      +0x002 ObjectTypeFlags : 0x4a 'J'
      +0x002 CaseInsensitive : Bitfield 0y0
      +0x002 UnnamedObjectsOnly : Bitfield 0y1
      +0x002 UseDefaultObject : Bitfield 0y0
      +0x002 SecurityRequired : Bitfield 0y1
      +0x002 MaintainHandleCount : Bitfield 0y0
      +0x002 MaintainTypeList : Bitfield 0y0
      +0x002 SupportsObjectCallbacks : Bitfield 0y1
      +0x004 ObjectTypeCode : 0
      +0x008 InvalidAttributes : 0xb0
      +0x00c GenericMapping : struct _GENERIC_MAPPING, 4
          elements, 0x10 bytes
      +0x01c ValidAccessMask : 0x1fffff
      +0x020 RetainAccess : 0x101000
      +0x024 PoolType  : Enum _POOL_TYPE,  15 total enums
0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0x1000
      +0x02c DefaultNonPagedPoolCharge : 0x528
      +0x030 DumpProcedure : (null)
      +0x038 OpenProcedure : 0xfffff800'02d99d58        long
          nt!PspProcessOpen+0
      +0x040 CloseProcedure : 0xfffff800'02d843c4        void
          nt!PspProcessClose+0
      +0x048 DeleteProcedure : 0xfffff800'02d84090        void
           nt!PspProcessDelete+0
      +0x050 ParseProcedure : (null)
      +0x058 SecurityProcedure : 0xfffff800'02d8cb50
          long  nt!SeDefaultObjectMethod+0
```

```
        +0x060 QueryNameProcedure : (null)
        +0x068 OkayToCloseProcedure : (null)
```

Avec un breakpoint mémoire, nous obtenons avec beaucoup d'aisance la fonction qui utilise ce pointeur. Elle vérifie d'abord que celui-ci est non nul et l'apelle :

```
nt!ObpIncrementHandleCountEx
fffff800`02dbda37 33db            xor     ebx,ebx
...
fffff800`02dbdb1a 48395d78        cmp     qword ptr [rbp+78h],
    rbx
...
fffff800`02dbddb9 ff5578          call    qword ptr [rbp+78h]
```

En détournant ce pointeur, nous avons facilement un moyen d'exécuter du code arbitraire en mode noyau. Maintenant que l'injection de code arbitraire en mode noyau a été présentée, nous allons entrer dans le vif du sujet et expliquer comment créer un hyperviseur.

4 Hyperviseur

4.1 Conception d'un hyperviseur

Tout d'abord, introduisons un peu de terminologie. Nous avons basé nos travaux sur la virtualisation des processeurs Intel. Celle-ci est supportée par le jeu d'extension VMX (Virtual Machine eXtensions). Lorsqu'il est activé se rajoute un niveau de séparation logicielle supplémentaire, similaire à la séparation entre ring 0 et ring 3, appelé parfois "ring -1". Il y a donc deux contextes d'exécution :

- le VMM (Virtual Machine Monitor). Il s'agit de l'hyperviseur proprement dit. Il a le contrôle total des machines virtuelles.
- les VM (Virtual Machines) aussi appelées "guest".

Lorsque le VMM est en cours d'exécution, il est dit en mode "VMX root". Les VM s'exécutent en mode "VMX non-root". La transition entre le mode VMX root et le mode VMX non-root est appelée "VM-Entry". Sa réciproque est appelée "VM-Exit".

Plusieurs choses sont à noter : premièrement, il n'existe pas de dispositifs logiciels pour savoir si un processeur est en mode VMX non-root, secondement le système d'exploitation que nous désirons virtualiser n'a besoin d'aucune modification. Cela assure donc par conception une grande furtivité à notre débogueur.

L'hyperviseur possède un contrôle très complet sur le système invité. Il peut monitorer les interruptions, l'accès à la mémoire, les entrées-sorties. Chaque machine virtuelle est contrôlée par une zone appelée VMCS (Virtual Machine Control data Structure). Celle-ci indique entre autres les conditions qui entraînent une sortie vers l'hyperviseur (VM-Exit) et contient l'état du processeur (hôte et invité).

Support des extensions VMX Avant de lancer l'hyperviseur, il faut s'assurer que celui-ci supporte la virtualisation. Pour cela est utilisée l'instruction `CPUID`. Elle prend en paramètre le registre `eax`. Nous l'appelons une première fois avec `eax` valant 0 pour vérifier qu'il s'agit bien d'un processeur Intel. Ensuite est récupéré dans `eax` l'index maximum utilisable avec l'instruction `CPUID` et dans `ebx`, `edx`, `ecx` les constantes `0x756e6547` ('Genu'), `0x49656e69` ('ineI') et `0x6c65746e` ('ntel') ce qui donne en concaténant le tout 'GenuineIntel'.

```
_CpuId(0, &eax, &ebx, &ecx, &edx);

/* Intel Genuine */
if (!(ebx == 0x756e6547 && ecx == 0x6c65746e && edx == 0
    x49656e69))
{
    DbgLog(("vmx: error not an INTEL processor\n"));
    return STATUS_UNSUCCESSFUL;
}
```

Listing 1.10. Utilisation de `CPUID`

Nous appelons ensuite de nouveau l'instruction `CPUID` avec `eax` valant 1 afin de savoir si le processeur supporte le jeu d'instruction VMX. Si c'est le cas alors le bit 5 du registre `ecx` est à 1. Une fois ces vérifications faites on peut passer à l'instruction `VMXON`.

Mode VMX-root L'instruction permettant de passer en mode VMX-root est `VMXON`. Avant de pouvoir l'utiliser, il est nécessaire d'examiner le MSR `IA32_FEATURE_CONTROL` (voir figure 1.11) qui contrôle celle-ci :

– le bit 0 est appelé “lock bit”. S’il vaut 0 alors l’instruction VMXON provoque une faute de protection générale. Généralement le BIOS emploie ce bit pour autoriser le support du VMX ;
– le bit 1 permet l’utilisation de VMXON en mode SMX. S’il vaut 0 alors l’instruction VMXON provoque une faute de protection générale si elle est employée en mode SMX ;
– le bit 2 permet l’utilisation de VMXON en dehors du mode SMX. S’il vaut 0 alors l’instruction VMXON provoque une faute de protection générale lorsqu’elle est utilisée en mode non-SMX.

```
typedef struct _IA32_FEATURE_CONTROL_MSR {
    unsigned Lock            :1;        // Bit 0 is the lock
        bit - cannot be
                                        // modified once lock
                                            is set, controled
                                            by BIOS
    unsigned VmxonInSmx      :1;
    unsigned VmxonOutSmx     :1;
    unsigned Reserved2       :29;
    unsigned Reserved3       :32;
} IA32_FEATURE_CONTROL_MSR;
```

Listing 1.11. MSR IA32_FEATURE_CONTROL

Le mode SMX est l’interface de programmation du mode TXT (Trusted eXecution Technology). Cela sort du propos de notre article et nous considérons pour la suite qu’il est soit désactivé soit non supporté. En résumé pour pouvoir faire appel au mode VMX, le BIOS doit avoir mis à 1 le bit 0 et le bit 2.

Lorsque nous utilisons les extensions VMX, il existe des restrictions sur les valeurs des registres cr0 et cr4. Les premiers processeurs supportant la virtualisation demandaient à être en mode protégé (c.-à-d. les bits PE et PG du registre cr0 doivent être à 1). Cela n’est plus vrai avec des processeurs récents qui supportent la virtualisation des modes réels ou protégés sans la pagination. Les bits qui doivent être à 0 (ou à 1) dans les registres cr0 et cr4 sont indiqués par plusieurs MSR (IA32_VMX_CR0_FIXED0, IA32_VMX_CR0_FIXED1, IA32_VMX_CR4_FIXED0 et IA32_VMX_CR4_FIXED1). Les “FIXED0” indiquent les bits qui doivent être à 1 tandis que les “FIXED1” indiquent les bits qui doivent être à 0. Cela signifie que chaque bit de cr0 (respectivement cr4) doit être fixé à 0 (les bits des deux MSR

sont à 0), fixé à 1 (les bits des deux MSR sont à 1) ou alors peuvent prendre n'importe quelle valeur (0 dans "FIXED0" et 1 dans "FIXED1").

Il ne reste plus qu'à mettre à 1 le bit 13 (appelé VMXE) du registre `cr4`, sinon l'instruction `VMXON` provoque une exception #UD (Invalid Opcode). Nous pouvons maintenant faire appel à l'instruction `VMXON`. Elle prend en argument l'adresse d'une zone de mémoire physique alignée sur 4Ko appelée *VMXON region*. En assembleur cela se traduit par les instructions suivantes :

```
_VmxOn PROC
    push rcx
    mov rax, rsp
    vmxon qword ptr [rax]
    pop rcx
    ret
_VmxOn ENDP
```

Ça y est, nous sommes passés en mode VMX-root !. L'étape suivante consiste à configurer correctement la VMCS (Virtual Machine Control data Structures) pour pouvoir lancer la virtualisation du système d'exploitation.

VMCS La VMCS contrôle les transitions entre les modes VMX root et VMX non-root (VM-Entry et VM-Exit). Elle se manipule à l'aide de nouvelles instructions (`VMCLEAR`, `VMPTRLD`, `VMREAD` et `VMWRITE` entre autres). De la même façon que pour la région VMXON, la VMCS est allouée en mémoire physique. Cette zone de mémoire doit avoir des propriétés spécifiques qui sont indiquées par le MSR `IA32_VMX_BASIC` (voir figure 1.12).

```
typedef struct _VMX_BASIC_MSR {
    unsigned RevId:32;
    unsigned szVmxOnRegion:12;
    unsigned ClearBit:1;
    unsigned Reserved:3;
    unsigned PhysicalWidth:1;
    unsigned DualMonitor:1;
    unsigned MemoryType:4;
    unsigned VmExitInformation:1;
    unsigned Reserved2:9;
} VMX_BASIC_MSR, *PVMX_BASIC_MSR;
```

Listing 1.12. MSR `IA32_VMX_BASIC`

Les quatre premiers octets de la VMCS doivent contenir la valeur du champ `RevId`. Le champ `szVmxOnRegion` indique la taille des régions VMXON et VMCS. Le champ `MemoryType` indique le type de mémoire utilisée. Sous Windows, ces allocations se traduisent par le code suivant :

```
PVOID AllocateContiguousMemory(ULONG size)
{
    PVOID Address;
    PHYSICAL_ADDRESS l1, l2, l3;

    l1.QuadPart = 0;
    l2.QuadPart = -1;
    l3.QuadPart = 0x200000;

    Address = MmAllocateContiguousMemorySpecifyCache(size, l1,
        l2, l3, MmCached);

    if (Address == NULL)
    {
        return NULL;
    }

    RtlZeroMemory(Address, size);
    return Address;
}
```

Listing 1.13. Allocation des régions VMXON et VMCS

Avant de pouvoir utiliser la VMCS il est nécessaire d'initialiser son état, ce qui est fait avec les instructions `VMCLEAR` et `VMPTRLD`. Ces dernières prennent en argument un pointeur sur l'adresse physique de la structure VMCS.

Une fois la VMCS allouée et configurée, il faut maintenant la remplir. Il n'est pas conseillé d'accéder aux champs de la structure en accédant directement à la mémoire ; il est préférable de faire appel aux instructions `VMREAD` et `VMWRITE`. Chaque champ de la VMCS est accessible par un *encoding* qui indique le type de champ, sa taille, etc. Heureusement pour nous Intel fournit l'encodage de tous ces champs. Il suffit d'écrire deux wrappers permettant de les manipuler.

```
_ReadVMCS PROC
    vmread rdx, rcx
    mov rax, rdx
    ret
_ReadVMCS ENDP
```

Listing 1.14. Wrapper sur l'instruction `VMREAD`

```
_WriteVMCS PROC
    vmwrite rcx, rdx
    ret
_WriteVMCS ENDP
```

Listing 1.15. Wrapper sur l'instruction VMWRITE

La structure complète de la VMCS est détaillée dans le manuel Intel 3B. Celle-ci est organisée en six zones :

- Guest-state area : elle contient l'état de la machine virtuelle (sauvegardé à chaque VM-exit et restauré à chaque VM-entry) ;
- Host-state area : elle contient l'état de l'hyperviseur ;
- VM-execution control fields : elle contrôle le comportement du processeur en mode VMX non-root. Cela conditionne certaines VM-exit ;
- VM-exit control fields : comme son nom l'indique ;
- VM-entry control fields : idem ;
- VM-exit information fields : partie en lecture seule qui contient les informations sur les VM-exit.

Comme nous virtualisons "à chaud" un système d'exploitation, une grande partie du travail de remplissage des zones Guest-state area et Host-state area est déjà fait. Dans la plupart des cas, il suffit de recopier les valeurs courantes du système.

Il y a cependant une subtilité au niveau de la gestion des registres `RSP` et `RIP`. Ceux-ci sont contrôlés par les champs `GUEST_RIP` et `GUEST_RSP` pour la machine virtuelle et `HOST_RIP` et `HOST_RSP`. En effet nous partons d'une seule instance de l'OS pour aboutir à deux instances : l'hyperviseur et le guest.

Étant donné que les VM-Exit peuvent se produire n'importe quand, il est nécessaire d'allouer une pile spécifique pour l'hyperviseur. Tout en haut de la pile est mise la structure représentant notre CPU virtualisé. Cela nous permet de retrouver notre structure lors des VM-Exit.

Pour gérer la première VM-Entry a été implémentée une fonction assembleur en deux parties : la première partie est `_StartVirtualisation`. Elle se charge de sauvegarder le contexte courant et d'appeler la fonction `StartVirtualization`. Celle-ci va mettre en place la région VMXON , la structure VMCS, la pile de l'hyperviseur et appeler virtualiser l'OS avec l'instruction `VMLAUNCH`. Nous passons donc en

mode vmx non-root. Comme il s'agit d'une VM-Entry, le contexte est restauré et RIP pointe sur `_GuestEntryPoint`.

```
_StartVirtualization PROC
    push    rax ; sauvegarde du contexte
    push    rcx
    ...
    push    r14
    push    r15

    sub rsp, 28h

    mov rcx, rsp ; StartVirtualisation est appele
                 ; avec le rsp du guest en parametre
    call StartVirtualization ; pas de ret on revient dans
        _GuestEntryPoint
_StartVirtualization ENDP
```

Listing 1.16. Lancement de la virtualisation

`_GuestEntryPoint` se contente quant à elle de restaurer le contexte. Nous avons finalement virtualisé notre premier processeur logique !

```
_GuestEntryPoint PROC

    call    ResumeGuest ; on est en mode vmx non-root

    add rsp, 28h

    pop r15
    pop r14
    ...
    pop rcx
    pop rax
    ret

_GuestEntryPoint ENDP
```

Listing 1.17. Première VM-Entry

Gestion des erreurs Bien évidemment, il peut arriver que les instructions VMX échouent. Intel spécifie deux types d'erreurs :

– `VMFailInvalid` : le bit CF du registre `RFLAGS` est mis à 1 ;

```
VmFailInvalid PROC
    pushfq
    pop rax
    xor rcx, rcx
    bt eax, 0 ; RFLAGS.CF
```

```
    adc cl, cl
    mov rax, rcx
    ret
_VmFailInvalid ENDP
```

Listing 1.18. VmFailInvalid

- `VMFailValid` : le bit ZF du registre `RFLAGS` est mis à 1, dans ce cas le numéro de l'erreur est accessible dans la VMCS avec le champ `VM-instruction error`.

```
_VmFailValid PROC
    pushfq
    pop rax
    xor rcx, rcx
    bt eax, 6 ; RFLAGS.ZF
    adc cl, cl
    mov rax, rcx
    ret
_VmFailValid ENDP
```

Listing 1.19. VmFailValid

Gestion des VM-Exit Par défaut l'hyperviseur doit gérer un certain nombre de VM-Exit qui sont inconditionnels. Les autres se configurent en manipulant des champs de la structure VMCS. Les VM-Exit inconditionnels sont les suivants :
- plusieurs instructions provoquent des VM-Exit : `CPUID`, `INVD`, `GETSEC` et `XSETBV` (ces dernières concernent les extensions SMX) ;
- toutes les instructions VMX (`VMCALL`, `VMREAD`, `VMWRITE` etc.).

Il est donc nécessaire d'émuler ces instructions. Ensuite suivant les valeurs des champs `VM-execution controls`. Un événement VM-Exit est déclenché dans les cas ci-après (liste non exhaustive) :
- modification des registres CR ;
- modification des registres DR ;
- entrées-sorties avec les instructions de la famille IN et OUT ;
- lecture-écriture des MSR ;
- instructions utilisant les tables de descripteurs du processeur (LIDT, SIDT, LGDT et SGDT pour les plus connues).

À chaque fois, le principe pour gérer ces VM-Exit est le même : la routine (spécifiée par le champ `HOST_RIP` dans la VMCS) est appelée. La raison de la sortie est récupérée dans le champ `VM_EXIT_REASON`.

Nous récupérons le pointeur de pile du guest avec le champ `GUEST_RSP`. Si la raison de la sortie est due à une instruction qui doit être émulée, nous ajustons le registre `RIP` du guest. Considérons par exemple l'instruction `CPUID` (voir 1.20) :

```
BOOLEAN HandleCpuid(PVIRT_CPU pCpu, PGUEST_REGS pGuestRegs)
{
    ULONG32 Function, eax, ebx, ecx, edx;
    ULONG64 InstructionLength;

    Function = (ULONG32)pGuestRegs->rax;
    _CpuId(Function, &eax, &ebx, &ecx, &edx);
    pGuestRegs->rax = eax;
    pGuestRegs->rbx = ebx;
    pGuestRegs->rcx = ecx;
    pGuestRegs->rdx = edx;

    InstructionLength = _ReadVMCS(VM_EXIT_INSTRUCTION_LEN);
    _WriteVMCS(GUEST_RIP, _ReadVMCS(GUEST_RIP)+
        InstructionLength);

    return TRUE;
}
```

Listing 1.20. Émulation de l'instruction `CPUID`

Nous récupérons la taille de l'instruction avec le champ `VM_EXIT_INSTRUCTION_LEN`. Cela est très pratique, car un LDE (Length Disassembler Engine) n'est plus nécessaire.

Une des raisons des VM-Exit est la lecture ou la modification du registre `CR3`. Cette sortie sera mise à profit ultérieurement pour vérifier à intervalle régulier si le débogueur demande d'arrêter le fonctionnement du système d'exploitation, permettant de "breaker" dans le contexte d'un processus donné.

```
typedef struct _MOV_CR_QUALIFICATION {
    unsigned ControlRegister:4;
    unsigned AccessType:2;
    unsigned LMSWOperandType:1;
    unsigned Reserved1:1;
    unsigned Register:4;
    unsigned Reserved2:4;
    unsigned LMSWSourceData:16;
    unsigned Reserved3:32;
} MOV_CR_QUALIFICATION, *PMOV_CR_QUALIFICATION;
```

Listing 1.21. Champ `EXIT_QUALIFICATION` pour un accès aux registres CR

Pour émuler correctement l'accès aux registres CR est consulté le champ EXIT_QUALIFICATION (voir figure 1.21). Le champ AccessType spécifie le type d'accès (MOV_TO_CR pour une écriture dans un registre) et le champ ControlRegister donne le numéro du registre (3 pour le registre CR3).

4.2 Mise en œuvre d'un hyperviseur en tant que débogueur

Dans la partie précédente, nous avons vu comment mettre en place et lancer un hyperviseur minimal. Pour l'instant celui-ci ne fait rien. Il se contente de prendre la main lors des VM-Exit obligatoires, de virtualiser ce qui doit être virtualisé puis de faire un VMRESUME pour de rendre la main à la machine virtuelle. Étudions maintenant quels sont les champs de la VMCS à mettre en place afin d'obtenir des primitives de debug. De façon classique un débogueur noyau fonctionne de la manière suivante :

- il réagit à une requête *Breakin* en stoppant l'exécution de la machine déboguée ;
- une fois la machine stoppée, il est possible d'examiner son état (registres, mémoire), de poser des points d'arrêts (logiciels, matériels, mémoire) ;
- en recevant une requête *Continue* il reprend l'exécution ;
- si jamais un point d'arrêt est atteint il le signale par un évènement.

L'architecture Intel propose plusieurs mécanismes pour faire du débogage. Le premier est appelé point d'arrêt logiciel. Il consiste en l'utilisation de l'instruction INT 3 directement dans le code du logiciel. Lorsque le processeur exécute cette instruction il génère une interruption appelée #BP (Breakpoint Exception). Le vecteur numéro 3 de l'IDT est donc appelé.

Le deuxième dispositif est appelé point d'arrêt matériel. Il met en jeu des registres particuliers du processeur : les registres DR. Les registres DR0, DR1, DR2, DR3 contiennent les adresses virtuelles pour lesquelles nous désirons *breaker*. Leurs validités (et d'autres paramètres comme la taille et le type d'accès) sont stockées dans le registre DR7.

```
typedef struct _DR7 {
    unsigned L0:1;
    unsigned G0:1;
    unsigned L1:1;
    unsigned G1:1;
    unsigned L2:1;
    unsigned G2:1;
    unsigned L3:1;
    unsigned G3:1;
    unsigned LE:1;
    unsigned GE:1;
    unsigned Reserved1:3;
    unsigned GD:1;
    unsigned Reserved2:2;
    unsigned RW0:2;
    unsigned LEN0:2;
    unsigned RW1:2;
    unsigned LEN1:2;
    unsigned RW2:2;
    unsigned LEN2:2;
    unsigned RW3:2;
    unsigned LEN3:2;
} DR7, *PDR7;
```

Listing 1.22. Registre `DR7`

Lorsque le processeur accède à une adresse (que ce soit en lecture, écriture ou exécution) configurée à l'aide des registres précédents, il émet une interruption appelée #DB (Debug Exception) qui a pour vecteur d'interruption 1 dans l'IDT. Cette routine consulte le registre `DR6` pour savoir pourquoi l'interruption a été déclenchée.

L'interruption 1 est aussi déclenchée sur chaque instruction lorsque le flag TF (Trap Flags) du registre `RFLAGS` est mis à 1. Nous pouvons aussi utiliser le flag BTF (single-step on branches) du MSR `IA32_DEBUGCTL` pour obtenir une interruption 1 sur chaque branche prise (au lieu de chaque instruction).

```
typedef struct _IA32_DEBUGCTL_MSR
{
    unsigned LBR:1;
    unsigned BTF:1;
    unsigned Reserved1:4;
    unsigned TR:1;
    unsigned BTS:1;
    unsigned BTINT:1;
    unsigned BTS_OFF_OS:1;
    unsigned BTS_OFF_USR:1;
    unsigned FREEZE_LBRS_ON_PMI:1;
    unsigned FREEZE_PERFMON_ON_PMI:1;
    unsigned Reserved2:1;
    unsigned FREEZE_WHILE_SMM_EN:1;
```

```
} IA32_DEBUGCTL_MSR , *PIA32_DEBUGCTL_MSR;
```

Listing 1.23. MSR IA32_DEBUGCTL

Afin de poser des points d'arrêts en mémoire, il est possible de faire appel aux registres DR. Nous sommes cependant limités : au maximum quatre points d'arrêts sont utilisables, et la zone monitorée est au maximum de 8 octets pour un processeur 64 bits.

Pour outrepasser ces limitations, une technique couramment utilisée est la manipulation des PDE (Page Directory Entry) et PTE (Page Table Entry). Ces descripteurs (situés en mémoire physique) contiennent les permissions d'accès aux pages de mémoires virtuelles. Ils spécifient si ce sont des pages accessibles depuis l'userland ou pas, si elles sont en lecture seule, etc. Ils spécifient aussi si la page est valide (c.-à-d. il y a une correspondance entre l'adresse virtuelle et l'adresse physique). Imaginons que nous voulons tracer un accès en écriture sur une plage de mémoire donnée. Il est nécessaire de retrouver le descripteur utilisé pour cette adresse, et le modifier afin de passer la page en lecture seule. Lorsque le processeur tentera d'accéder à la page, il va déclencher ce qu'on appelle une faute de page. Celle-ci est appelée #PF (Page Fault Exception) et a pour vecteur d'interruption 14. Les débogueurs qui emploient cette technique "hookent" donc l'interruption 14.

Au final nous voyons que pour implémenter un débogueur nous avons besoin de pouvoir :
- hooker les interruptions 1, 3 et 14 ;
- manipuler les PTE ;
- manipuler les registres DR.

Il est de cette façon possible d'implémenter un débogueur **sans** hooker l'IDT (pré-requis indispensable pour cohabiter avec Windows Seven 64 bits). Présentons maintenant comment mettre ces techniques en action.

Interceptions d'interruptions Lorsque de l'interception d'une interruption, nous devons lire le champ `VM_EXIT_INTR_INFO`. Celui-ci est une structure `INTERRUPT_INFO_FIELD` (voir figure 1.24).

```
typedef struct _INTERRUPT_INFO_FIELD {
    unsigned Vector:8;
```

```
        unsigned InterruptionType:3;
        unsigned ErrorCodeValid:1;
        unsigned NMIUnblocking:1;
        unsigned Reserved:18;
        unsigned Valid:1;
    } INTERRUPT_INFO_FIELD, *PINTERRUPT_INFO_FIELD;
```

Listing 1.24. structure `INTERRUPT_INFO_FIELD`

Le champ `Vector` représente le vecteur de l'interruption, le champ `InterruptionType` précise le type d'interruption (matériel, logiciel, NMI etc.).

Breakpoints logiciels La pose d'un point d'arrêt logiciel est décrite ci-après :

- on vérifie que l'adresse est valide (dans le contexte courant) en parcourant la table des descripteurs de page du processeur ;
- le premier octet de l'instruction originale est sauvegardé, et remplaçé par l'octet '`\xcc`' qui correspond à l'instruction `int 3` ;
- la VMCS étant configurée pour faire des VM-Exit pour l'interruption 3, l'hyperviseur prend la main dès que le processeur exécute l'instruction `int 3` ;
- dans le gestionnaire de l'hyperviseur nous vérifions que l'adresse correspond à un point d'arrêt ayant été posé ;
- si c'est le cas l'instruction originale est restaurée et le flag TF du registre RFLAGS est positionné ;
- on reprend l'exécution de la machine virtuelle ;
- comme le flag TF est à 1, le processeur émet une interruption #DB dès que l'instruction originale a été exécutée ;
- le gestionnaire de VM-Exit de l'hyperviseur prend la main ;
- en interrogeant la VMCS nous sommes en mesure de connaître la source de l'interruption ;
- dans notre cas, le point d'arrêt est restauré si besoin est et l'exécution reprend.

Si l'interruption provient d'un point d'arrêt que nous n'avons pas posé, on peut avertir la machine qui débogue en générant un évènement, ou encore transférer directement l'interruption à la machine virtuelle.

Nous pouvons générer n'importe quelle exception dans le contexte de la VM. Pour cela Intel met à notre disposition un champ de taille 32 bits dans la structure VMCS appelé `VM_ENTRY_INTR_INFO_FIELD`. Il correspond à une structure de type `INTERRUPT_INJECT_INFO_FIELD`. Le champ `Vector` représente le vecteur de l'interruption, le champ `InterruptionType` précise le type d'interruption (matériel, logiciel, NMI etc.), le champ `DeliverErrorCode` précise si un code d'erreur doit être poussé sur la pile.

```
typedef struct _INTERRUPT_INJECT_INFO_FIELD{
    unsigned Vector:8;
    unsigned InterruptionType:3;
    unsigned DeliverErrorCode:1;
    unsigned Reserved:19;
    unsigned Valid:1;
} INTERRUPT_INJECT_INFO_FIELD, *PINTERRUPT_INJECT_INFO_FIELD;
```

Dans le cas où nous voulons forwarder l'exception à la machine virtuelle, il faut remplir cette structure de la manière suivante :

```
pInjectEvent->Vector = BREAKPOINT_EXCEPTION;
pInjectEvent->InterruptionType = SOFTWARE_INTERRUPT;
pInjectEvent->DeliverErrorCode = 0;
pInjectEvent->Valid = 1;
_WriteVMCS(VM_ENTRY_INTR_INFO_FIELD, InjectEvent);
_WriteVMCS(VM_ENTRY_INSTRUCTION_LEN, 1);
_WriteVMCS(GUEST_RIP, GuestRip);
```

Listing 1.25. Forward de l'exception au guest

Breakpoints matériels Pour gérer les point d'arrêts matériels sont à notre disposition plusieurs champs dans la structure VMCS. Le registre `DR7` est virtualisé ainsi que le MSR `IA32_DEBUGCTL`. Lorsqu'une interruption #DB arrive, nous pouvons interroger la VMCS afin de connaître les raisons de cette interruption.

Une #DB ne met pas à jour le registre `DR6`, le flag `GD` de `DR7` ainsi que le flag `LBR` du MSR `IA32_DEBUGCTL`. C'est la responsabilité de l'hyperviseur de mettre à jour ces champs. Pour cela il faut consulter le champ `EXIT_QUALIFICATION` de la VMCS.

```
typedef struct _DEBUG_EXIT_QUALIFICATION {
    unsigned B0:1;
    unsigned B1:1;
    unsigned B2:1;
    unsigned B3:1;
```

```
        unsigned Reserved:9;
        unsigned BD:1;
        unsigned BS:1;
        unsigned Reserved2:39;
    } DEBUG_EXIT_QUALIFICATION, *PDEBUG_EXIT_QUALIFICATION;
```

Listing 1.26. champ `EXIT_QUALIFICATION`

Les champs `B0` à `B3` indique quel est le registre DR responsable de l'interruption. Le champ `BD` indique si la cause est un accès aux registres DR. Finalement, le champ `BS` sert à savoir si l'interruption est due à un single step (`RFLAGS.TF=1` et `IA32_DEBUGCTL.BTF=0`) ou parce qu'une branche a été prise (`RFLAGS.TF=1` et `IA32_DEBUGCTL.BTF=1`).

Si nous décidons de forwarder l'interruption au guest il est nécessaire de mettre à jour le registre `DR6` avec les informations présentes dans le champ `EXIT_QUALIFICATION` de la VMCS. Ensuite est construit l'évènement à injecter de la même façon que précédemment.

```
    pInjectEvent->Vector = DEBUG_EXCEPTION;
    pInjectEvent->InterruptionType = HARDWARE_EXCEPTION;
    pInjectEvent->DeliverErrorCode = 0;
    pInjectEvent->Valid = 1;
    _WriteVMCS(VM_ENTRY_INTR_INFO_FIELD, InjectEvent);
    _WriteVMCS(GUEST_RIP, _ReadVMCS(GUEST_RIP));
```

Listing 1.27. Injection d'un int1

Breakpoints mémoire Deux choix s'offrent à nous pour poser des point d'arrêts en mémoire. Le premier est d'utiliser les registres DR qui proposent directement cette fonctionnalité, mais seuls huit octets dans le meilleur des cas peuvent être surveillés. Une façon d'outrepasser cette limitation consiste à manipuler les descripteurs de pages du processeur, PDE et PTE, afin de marquer les pages surveillées comme étant non-présentes (point d'arrêts en exécution) ou en lecture-seule (point d'arrêts en écriture).

La figure 2 décrit le fonctionnement de la translation entre une adresse virtuelle et une adresse physique.

Pour faire cette translation, nous avons normalement besoin de connaître la valeur du registre `CR3` et de parcourir la mémoire physique pour lire les descripteurs de page et trouver l'adresse physique correspondante. Cependant dans un souci de performances l'OS maintient un mapping en mémoire virtuelle des différentes tables.

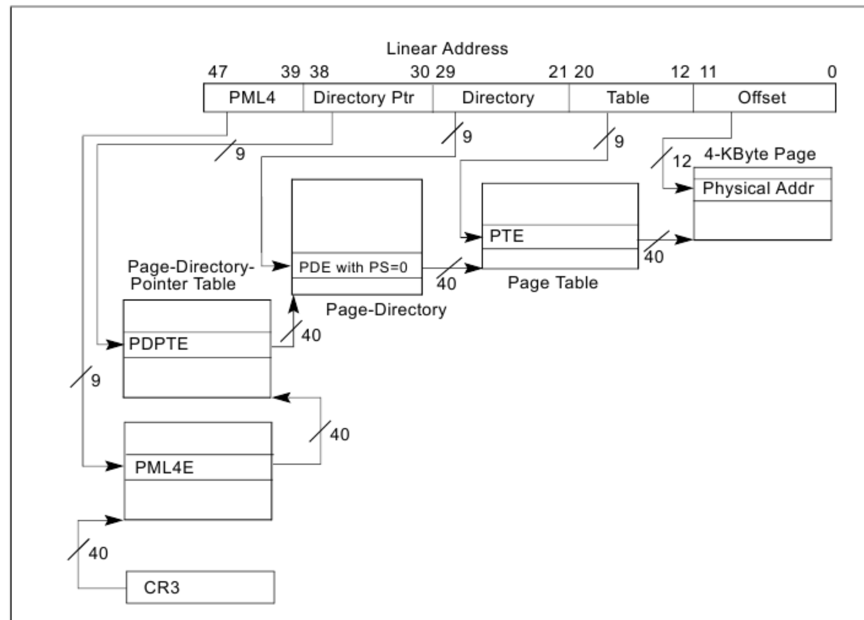


Figure 2. Pagination en 64 bits (source : manuel Intel)

Nous trouvons les adresses des différentes tables en désassemblant la fonction `MmGetPhysicalAddress`, ce qui donne les définitions suivantes :

```
#define PML4_BASE   0xfffff6fb7dbed000
#define PDP_BASE    0xfffff6fb7da00000
#define PD_BASE     0xfffff6fb40000000
#define PT_BASE     0xfffff68000000000
```

Listing 1.28. Adresses des différentes tables du processeur

En décomposant l'adresse virtuelle il devient possible de retrouver l'adresse physique correspondante, et surtout de manipuler ses descripteurs.

Nous allons virtualiser l'interruption 14 (#PF) qui gère les fautes de pages. De façon analogue à l'interception de l'interruption 1 (#DB) le registre `CR2` n'est pas mis à jour en cas de VM-Exit. L'adresse qui a provoqué la faute de pages est située dans le champ `EXIT_QUALIFICATION` de la VMCS. Le code d'erreur qui indique plusieurs informations sur la faute de page (accès en lecture, en exécution, etc.) n'est pas poussé

sur la pile mais se situe dans le champ VM_EXIT_INTR_ERROR_CODE de la VMCS. Ce coup-ci pour faire passer l'exception au guest il faut spécifier que l'interruption contient un code d'erreur (celui-ci est stocké dans le champ VM_ENTRY_EXCEPTION_ERROR_CODE. Cela donne le code suivant :

```
_SetCr2(ExitQualification);
_WriteVMCS(VM_ENTRY_EXCEPTION_ERROR_CODE, ErrorCode);
InjectEvent = 0;
pInjectEvent->Vector = PAGE_FAULT_EXCEPTION;
pInjectEvent->InterruptionType = HARDWARE_EXCEPTION;
pInjectEvent->DeliverErrorCode = 1;
pInjectEvent->Valid = 1;
_WriteVMCS(VM_ENTRY_INTR_INFO_FIELD, InjectEvent);
_WriteVMCS(GUEST_RIP, _ReadVMCS(GUEST_RIP));
```

Listing 1.29. Forward d'une faute de page

Breakin et continue Maintenant que nous sommes capables de poser des point d'arrêts, la dernière fonctionnalité restant à implémenter est la capacité d'arrêter l'exécution de la machine virtualisée. Pour cela nous avons choisi d'utiliser les VM-Exit induites par le changement du registre CR3. Celui-ci est modifié à chaque changement de contexte afin de charger l'espace d'adressage virtuel correspondant au processus en train de s'exécuter. À ce moment là, on regarde si l'utilisateur a demandé d'arrêter l'exécution (requête *Breakin*). Dans ce cas l'hyperviseur rentre dans une boucle pendant laquelle le guest est stoppé (puisque le mode VMX root est toujours actif) et attend les requêtes de l'utilisateur. Il pourra examiner la mémoire, les registres etc. Lorsqu'il a fini il envoie une requête *Continue* en spécifiant s'il désire poursuivre l'exécution, transférer l'exception au guest ou arrêter le debug et décharger l'hyperviseur.

Sur les machines multiprocesseurs, cas de figure de plus en plus répandu, il faut rajouter une étape. Il est nécessaire de stopper les autres processeurs logiques afin que l'état de la mémoire ne soit pas altéré pendant que l'utilisateur inspecte la machine. Pour cela dès qu'un processeur rencontre une requête de breakin il la signale par une variable partagée aux autres processeurs. Ceux-ci lors de la prochaine VM-Exit rencontrée (changement de contexte ou autre) vont rentrer dans une boucle d'attente non interruptible en mode

VMX root (“spinlock”) jusqu’à ce que le processeur qui a reçu la requête leur signale de reprendre l’exécution.

Protocole de communication Nous avons choisi de séparer le protocole de communication en couches : une première couche gère le protocole de debug et une seconde gère le transport. (présentée dans la partie suivante). Cette distinction permet de séparer les différents composants et de pouvoir ainsi changer facilement de moyen de transport. Pour l’instant la couche de transport est fondée sur l’utilisation d’un FPGA qui communique via une mémoire partagée avec l’hyperviseur. Cependant, l’hyperviseur n’a pas à s’en préoccuper. Il utilise 2 primitives `SendDebugPacket` et `ReceiveDebugPacket` qui assureront de manière transparente le transport des paquets par le moyen de transport sélectionné.

Le protocole retenu est inspiré par celui utilisé par Kd (partie noyau de l’API de debug de Microsoft). Il emploie différents paquets servant à la fois au contrôle du protocole (reset de la communication, bonne réception des paquets, etc.) et au transport des données (lecture de la mémoire, des registres, pose de points d’arrêts, etc.).

Chaque paquet contient un header (structure `_DEBUG_PACKET`) et des données (matérialisées par une structure dépendante du type de paquet) concaténées au header.

```
typedef struct _DEBUG_PACKET {
    ULONG32 Magic;
    ULONG32 Type;
    ULONG32 Size;
    ULONG32 Id;
    ULONG32 Checksum;
} DEBUG_PACKET, *PDEBUG_PACKET;
```

Listing 1.30. structure `DEBUG_PACKET`

Le champ `Size` contient la taille des données, le champ `Id` contient un numéro de séquence permettant de déterminer la rupture de connexion et le champ `Checksum` contient une somme de contrôle.

Voici une brève description du protocole lorsqu’une requête de breakin est demandée :

- envoi d’un paquet `PACKET_TYPE_BREAKIN` par la machine qui débogue ;

- l'hyperviseur répond par un paquet `PACKET_TYPE_ACK` lorsque tous les processeurs logiques ont pris en compte la demande ;
- l'utilisateur peut alors poser des points d'arrêt (`PACKET_TYPE_SOFTWARE_BREAKPOINT` par exemple pour un point d'arrêt logiciel) ou examiner la mémoire (`PACKET_TYPE_READ_VIRTUAL_MEMORY` pour examiner la mémoire virtuelle) ;
- chaque paquet doit être validé par l'envoi d'un packet `PACKET_TYPE_ACK` avec le champ `Id` correspondant au paquet reçu ;
- lorsque l'utilisateur a fini il envoie un paquet `PACKET_TYPE_CONTINUE` qui contient un status indiquant la suite à donner aux opérations (continuer l'exécution, forwarder l'exception, arrêter le debug, etc.).

Si la somme de contrôle contenue dans le champ `Checksum` est invalide alors le paquet `PACKET_TYPE_RESEND` est renvoyé. Si jamais la connexion est interrompue les deux parties se resynchronisent en envoyant un paquet `PACKET_TYPE_RESET` qui fixe la valeur du champ `Id`.

Lorsque l'hyperviseur doit signaler un évènement à l'utilisateur, il envoie un paquet (par exemple `PACKET_TYPE_HIT_SOFTWARE_BREAKPOINT` pour un int 3).

Le schéma 3 résume les interactions entre les différents participants.

4.3 Déchargement de l'hyperviseur

Il peut arriver que l'utilisateur désire décharger l'hyperviseur. Le mécanisme mis en place est similaire à celui déroulé lors de la virtualisation initiale. En mode VMX non-root les instructions VMX provoquent des VM-Exit. En particulier il existe une instruction `VMCALL` qui sert de moyen de communication avec l'hyperviseur. Pour décharger l'hyperviseur nous procédons de manière suivante :

- la routine `_StopVirtualization` (en mode VMX non-root) exécute l'instruction `VMCALL` avec des paramètres bien particuliers dans le contexte ;

```
_StopVirtualization PROC
    push rax
    push rbx
    xor rax, rax
    xor rbx, rbx
```

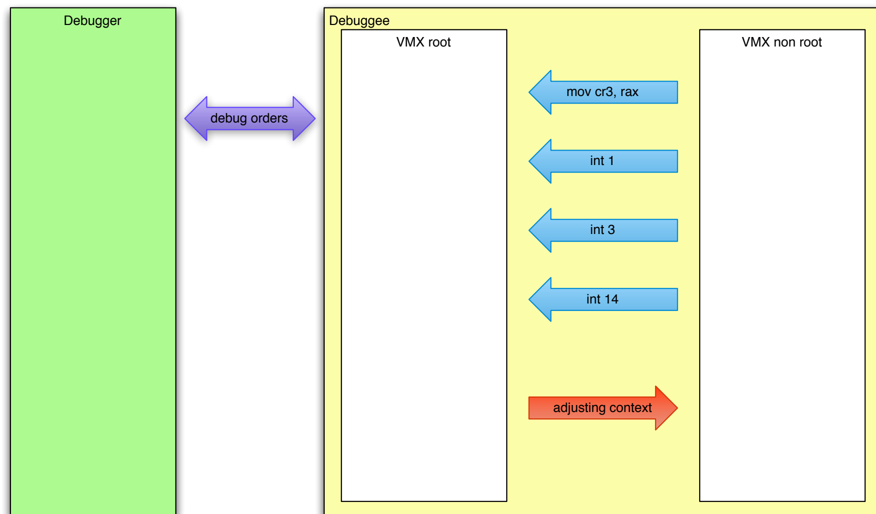


FIGURE 3. Transition entre l'hyperviseur et le guest

```
        mov eax, 42424242h
        mov ebx, 43434343h
        vmcall
    _StopVirtualization ENDP
```

Listing 1.31. Routine `_StopVirtualization`

– l'appel à l'instruction `VMCALL` est intercepté par l'hyperviseur qui appelle la routine chargée de gérer les appels à `VMCALL` ;

```
    BOOLEAN HandleVmCall(PVIRT_CPU pCpu, PGUEST_REGS
        pGuestRegs)
    {
        ULONG64 InstructionLength, Rip, Rsp;
        ...
        if ((pGuestRegs->rax == 0x42424242) && (pGuestRegs->
            rbx == 0x43434343))
        {
            Rip = (ULONG64)_GuestExit;
            Rsp = pGuestRegs->rsp;
            _VmxOff(Rip, Rsp);
        ...
```

Listing 1.32. Routine gérant les appels à `VMCALL`

– nous appelons la routine `_VmxOff` qui restaure `RIP` et `RSP`, l'exécution va reprendre sur la routine `_GuestExit` ;

```
_VmxOff PROC
    vmxoff
    mov rsp, rdx
    push rcx
    ret
_VmxOff ENDP
```

Listing 1.33. Routine `_VmxOff`

- celle-ci restaure la pile de la fonction `_StopVirtualization` ;

```
_GuestExit PROC
    pop rbx
    pop rax
    ret
_GuestExit ENDP
```

Listing 1.34. Routine `_GuestExit`

- en exécutant la routine `_StopVirtualization` sur chacun des processeurs logiques le mode VMX root est désactivé.

5 Communications

Dans cette partie nous allons détailler l'interfaçage des différents composants : tout d'abord l'accès à la mémoire physique par le Cardbus et ensuite la communication avec l'extérieur.

5.1 Interfaçage CardBus ⇔ système débogué

Rappels sur le protocole PCI Les bases de ce protocole ont été présentées dans l'article “Compromission physique par le bus PCI” [13]. Nous ferons donc seulement un bref rappel des grandes lignes ; plus d'informations sont disponibles sur Internet en cherchant les mots-clefs “PCI Local Bus Specification”.

Un périphérique est relié au bus PCI par un ensemble de fils électriques : 32 sont dédiés au transport de l'adresse ou des données (signal AD), et plusieurs autres fils véhiculent des informations de commande et de contrôle (`CLK`, `RST#`, Command/Byte Enable `CBE#`, etc.). Le niveau de voltage 3.3V correspond à l'état inactif, et le niveau de voltage 0 correspond à l'état actif : les signaux PCI sont dits “actifs bas”, dénoté par un dièse après le nom du signal.

Ces fils sont communs à tous les périphériques, mis à part ceux transportant les signaux de demandes d'accès aux bus (`REQ` / `GNT`)

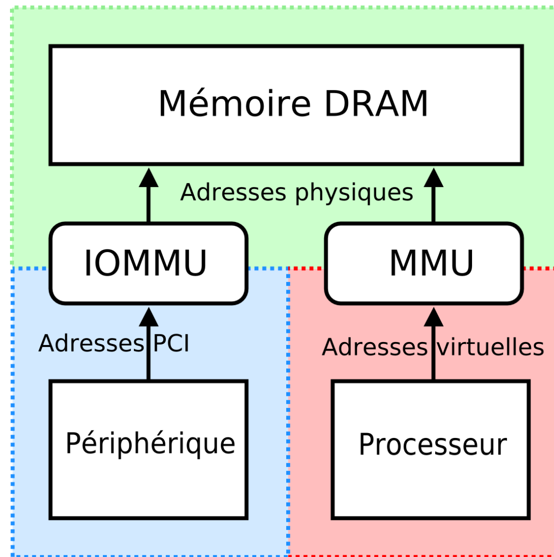


FIGURE 4. Requêtes vers la mémoire physique (source : Wikipedia)

et d'interruption (`INT`) ; ils permettent au contrôleur de distinguer les périphériques entre eux. Le protocole décrit principalement comment demander l'accès au bus, envoyer ou recevoir les données, et finalement vérifier l'absence d'erreur lors de la transmission.

Seize types d'ordres sont définis par la norme ; parmi ceux-ci, on s'intéresse en particulier aux commandes de lecture et d'écriture dans l'espace d'adressage de 32-bit (accès dits "DMA", pour Direct Memory Access). Une partie de cet espace d'adressage est réservée à la mémoire interne des cartes PCI ; les autres requêtes sont redirigées par le contrôleur vers la mémoire vive de l'ordinateur comme indiqué sur le schéma suivant (voir figure 4) :

L'absence de vérification par le contrôleur mémoire des accès vers la mémoire vive constitue une faille de sécurité pouvant donner lieu à la compromission de l'ordinateur ; différentes preuves de concepts faisant intervenir cette vulnérabilité ont été présentées ([18]). Dans le cas ou l'ordinateur cible dispose de plus de 3 Giga-octets de mémoire, l'attaque devient moins fiable car le haut de la mémoire vive devient inaccessible depuis le bus PCI. Par ailleurs, les processeurs Intel et AMD récents incluent une IOMMU, c'est-à-dire un gestion-

naire d'entrées/sorties capable de filtrer les requêtes vers la mémoire physique. Cette protection n'est pas infaillible pour autant ([19]).

Le PCI-Express Le protocole PCI Express est une évolution du PCI, Il en conserve les grandes lignes, en particulier les accès DMA. L'espace d'adressage passe à 64 bits ; en l'absence d'IOMMU, il devient alors possible d'accéder depuis ce bus à la totalité de la RAM, hormis les trous constitués par la mémoire des périphériques PCI / PCI Express. Notons que des convertisseurs CardBus vers ExpressCard sont vendus dans le commerce, ce qui permettrait d'utiliser la carte COM-1300-C sur un portable seulement doté d'un port ExpressCard.

PCI, suite (et fin ?) Par rapport à l'article publié en 2009 [g], notre compréhension du protocole PCI a quelque peu évolué, en grande partie grâce à la lecture du livre "PCI System Architecture" [20] et de nombreuses expérimentations.

Un point important est que les accès en lecture et écriture ne sont pas symétriques. Les demandes d'écriture sont "postées", c'est-à-dire que le contrôleur mémoire les met en cache et peut agréger plusieurs demandes successives afin d'optimiser les accès à la mémoire vive. Au contraire, un accès initial en lecture n'est pas prédictible, et la mémoire est rarement disponible de suite : les DRAM ont une latence de plusieurs cycles pour changer de colonne ou de ligne d'adresse (latences dites RAS et CAS). Ainsi, une première lecture vers la mémoire physique sera avortée par le contrôleur mémoire pour ne pas monopoliser le bus dans l'attente d'une réponse de la DRAM. Le périphérique PCI doit impérativement détecter l'erreur (signal `STOP#`, message "Retry") pour réémettre une demande de lecture identique quelques cycles d'horloge plus tard, jusqu'à ce que l'opération réussisse ; faute de quoi, le contrôleur mémoire se retrouvera dans un état non prévu par la norme. Le schéma (voir figure 5) illustre une lecture initiale ayant échoué au cycle 4.

Le protocole CardBus, bien que très proche du PCI, présente quelques différences significatives liées à l'économie d'énergie et le branche-et-joue. Ces différences compliquent l'implémentation du moteur de DMA. En premier lieu, il faut forcer la valeur du signal `CCLKRUN# à 0` pour que le contrôleur CardBus ne désactive pas

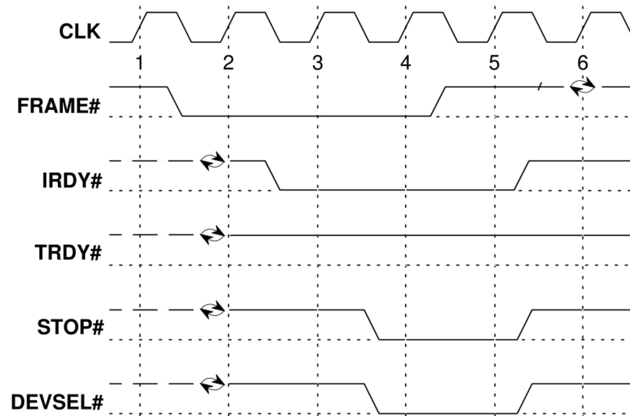


Figure 5. Lecture PCI échouée (source : norme PCI 2.2)

l'horloge de la carte, mais ce n'est pas suffisant : il faut de plus émettre un accès DMA tous les quelques milliers de cycles, sinon le contrôleur coupe de la même façon l'horloge – on peut voir cela comme un battement de cœur.

Un second problème est relatif à la détection de la carte par le système d'exploitation. Sous Windows, le pilote de périphérique du contrôleur CardBus détecte l'insertion d'une carte et fait une demande vers l'espace de configuration pour connaître le `VENDOR_ID`, `DEVICE_ID` et `SUBSYSTEM_ID`. Ces trois données déterminent alors le pilote à charger ; si aucun pilote ne correspond, la carte est désactivée et une boîte de dialogue s'affiche. L'obstacle est donc que le pilote doit se charger correctement : si par exemple nous renvoyons l'identifiant d'une carte réseau Realtek 8189, il faut implémenter correctement le protocole de communication avec RTL8189.SYS. En cas d'erreur, le pilote de Realtek signale au pilote du contrôleur CardBus que la carte est non fonctionnelle, et son accès au bus PCI est coupé.

L'astuce que nous avons identifiée est que le pilote CardBus ne coupe pas l'horloge de la carte lors de la première installation d'un nouveau périphérique si un pilote est connu, et même si ce dernier reporte une erreur. Nous avec donc modifié le moteur DMA pour rendre aléatoire le `SUBSYSTEM_ID`, de sorte que chaque nouvelle insertion donne lieu à la détection d'un nouveau périphérique. La génération de ce nombre aléatoire se fait simplement en lisant signal AD

lorsque le bus PCI est inactif. Le périphérique choisi est un modem de marque Lucent, qui présente l'avantage d'être présent dans les pilotes par défaut depuis Windows XP jusqu'à Windows 7. L'inconvénient de cette méthode est la création d'une nouvelle clef de registre à chaque insertion ; à terme, il sera souhaitable d'implémenter un sous-ensemble minimal du protocole cible pour que le pilote reste correctement chargé, et conserver de la sorte l'accès au bus.

Finalement, nous n'avons pas encore élucidé un souci relatif au signal `FRAME#`. Ce dernier est positionné par l'initiateur d'un accès au bus lorsque les données sont prêtes à être lues ou écrites, suivant le type de commande. Durant les tests sur un portable Lenovo (modèle T400), `FRAME#` doit être laissé flottant (valeur 'Z'), conformément à la norme ; sinon, la carte COM-1300-C est bloquée par le contrôleur CardBus. Mais étrangement, sur un portable plus ancien il a fallu mettre un "pull-up" sur `FRAME#` (valeur 'H') afin d'éviter un gel complet du système. Plus de tests sont donc nécessaires pour comprendre l'origine de ce phénomène.

Notons que la programmation de la carte avec le logiciel de ComBlock semble nécessiter un portable doté d'un contrôleur CardBus de la marque Ricoh ; d'autres modèles ont été testés sans succès.

Le processeur plasma Lors de travaux précédents relatifs au protocole PCI, un processeur nommé "BlackCPU" a été créé par Guillaume Vissian dans l'optique de programmer avec plus de souplesse les accès DMA. Une limitation majeure restait l'absence de compilateur C, obligeant à programmer en assembleur le microcode chargé dans la RAM interne du FPGA.

Nous avons alors fait le choix de repartir à zéro et d'architecturer le code VHDL autour du processeur "plasma", conçu et implémenté par l'américain Steve Rhoads [16]. Plasma présente l'avantage majeur d'implémenter le jeu d'instructions MIPS I, et son architecture s'inspire fortement des travaux de recherche de John Hennessy à l'université de Stanford [21]. Le code source de plasma est dans le domaine public ; de plus, une chaîne de développement ("toolchain") précompilée basée sur gcc/binutils a été rendue disponible par Steve Rhoads.

Plasma est composé des éléments suivants :

- u1_pc_next : détermine l'adresse suivante du pointeur d'instruction (program_counter), qui dépende des signaux pause_in, reset_in et take_branch ;
- u2_mem_ctrl : contrôleur mémoire, décode le type d'accès (lecture/écriture, grand/petit indien, 8/16/32 bits) et envoie la requête correspondante à mem_dispatch (voir partie xxx) ;
- u3_control : décode l'instruction courante sur 32-bits et génère les signaux de contrôle (index des registres, etc.) à destination des autres composants ;
- u4_reg_bank : banque de registres à double canal ; ce composant est dans notre cas basé sur une RAM interne de type XILINX_16X ;
- u5_bus_mux : redirige les accès aux bus généraux A/B/C suivant le type d'instruction, et contrôle par ailleurs le signal take_branch ;
- u6_alu : unité arithmétique classique, gère les opérations '+', '-', '< signé', '< non signé', AND, OR, XOR et NOR ;
- u7_shifter : unité de calcul des opérations à décalage de bits (gauche et droite, sur 1/2/4/8/16 bits) ;
- u8_mult : implémente la multiplication et la division – 32 cycles sont requis par instruction.

Plusieurs versions de plasma sont disponibles sur le site opencores.org. Durant nos tests, la version 3.0 s'est le mieux comportée (absence de bugs lors de simulations avec ModelSim) et a été retenue pour la suite.

Redirection des accès mémoire En sus du processeur qui vient d'être présenté (composant nommé "p1_cpu") ont été implémentés les composants suivants :
- p2_ram, RAM interne ;
- p3_dma, moteur d'accès DMA ;
- p4_usb, chargé des entrées/sorties USB.

Le code source originel de plasma implémente une RAM de 8 Ko qui contient le code MIPS de démarrage et les données (pile, variables globales, etc.). La carte COM-1300-C est dotée d'un FPGA xc3s400, contenant une RAM interne (BlockRAM) de 32 Ko ; le code du composant p2_ram a été modifié pour prendre en compte cette taille.

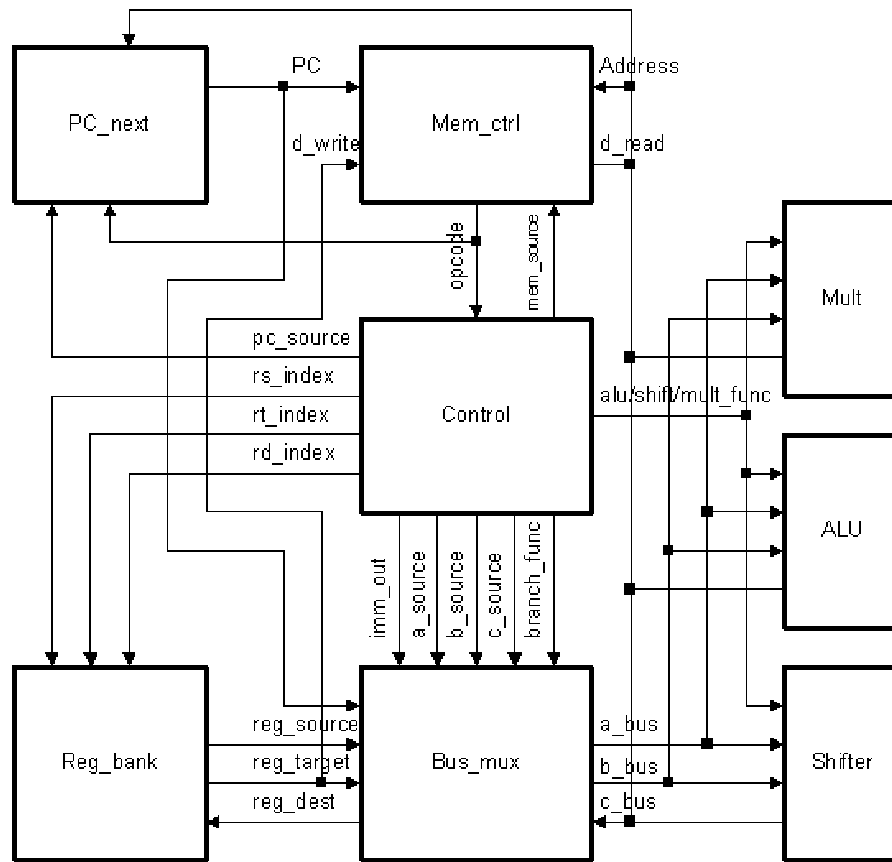


FIGURE 6. Processeur Plasma (source : S. Rhoads)

En pratique, seize composants RAMB16_S2 sont implémentés, pour un total de 16 * 2 * 8192 bits (il n'existe malheureusement pas de composant unique de 32 Ko).

Par défaut, les accès mémoire via les instructions load/store sont dirigés vers p2_ram. Comment faire alors pour déclencher des accès DMA ou USB depuis le processeur ?

Une première solution serait d'étendre le jeu d'instructions avec par exemple ld_pci et st_pci. L'avantage est de pouvoir adresser l'ensemble de la mémoire PCI, cependant il aurait fallu apporter des modifications importantes au processeur lui-même ainsi qu'à la chaine de compilation. Une autre solution, de même non retenue, est

l'utilisation d'une adresse mémoire pour accéder de manière indirecte au PCI :

```
*(uint *)(0x10000) = 0x10001000; /* on renseigne l'adresse PCI
     */
*(uint *)(0x20000) = 0xaabbccdd;  /* on ecrit a l'adresse pre-
    citee */
```

Finalement, nous avons choisi de rediriger l'ensemble des accès mémoire au dessus de l'adresse 0x8000 (limite de la RAM interne) vers le bus PCI. Le code VHDL de plus haut niveau est un multiplexeur d'adresses qui détermine quel composant activer selon le prochain accès mémoire :

```
-- processus de selection du composant a activer

device_select: process(pci_clk, mem_address)
begin
   case mem_address(31 downto 15) is
   when ("00000000000000000") => -- acces < 32K
      case mem_address(14 downto 2) is
      when "1111111111111" => -- 0x7ffc sert aux lectures/
          ecritures usb
         ram_enable  <= '0'; pci_enable  <= '0'; usb_enable  <=
              '1';
      when others =>          -- BlockRAM interne
         ram_enable  <= '1'; pci_enable  <= '0'; usb_enable  <=
              '0';
      end case;
   when others =>             -- tous les autres acces vont sur
       le PCI
         ram_enable  <= '0'; pci_enable  <= '1'; usb_enable  <=
              '0';
   end case;
   -- mise en registre de la prochaine adresse
   if rising_edge(pci_clk) then
      if cpu_pause = '0' then
         address_next    <= mem_address;
      end if;
   end if;
end process;
```

Le processeur est mis en pause tant que l'accès USB ou PCI ne s'est pas terminé :

```
cpu_pause <= pci_busy or usb_busy;
```

Une fois l'accès terminé, les donnés éventuellement lues sont redirigées à bon port :

```
-- processus de redirection des donnees lues depuis le
    composant cible

data_read: process(address_next, data_read_ram, data_read_pci,
    data_read_usb)
begin
  case address_next(31 downto 15) is
  when ("00000000000000000") =>
    case address_next(14 downto 2) is -- ram interne 32 KB
    when "1111111111111" =>
      mem_data_read(7 downto 0) <= data_read_usb;
    when others =>
      mem_data_read <= data_read_ram;
    end case;
  when others =>
    mem_data_read <= data_read_pci;
  end case;
end process;
```

De la sorte, une écriture vers la mémoire physique de l'ordinateur s'écrit simplement :

```
*(uint *)(0x10001000) = 0xaabbccdd;
```

5.2 Interfaçage Débogueur ⇔ CardBus

On a vu que les communications entre la carte COM-1300-C et la machine déboguée font intervenir le bus PCI. Maintenant, comment faire communiquer le FPGA avec le monde extérieur ? Heureusement, la carte dispose d'un connecteur d'extension à 40 pins, électriquement relié au FPGA. Nous avons envisagé plusieurs pistes.

Port série Le protocole RS-232 est d'une grande simplicité d'implémentation (de nombreux codes VHDL sont disponibles sur Internet), et ne requiert que trois fils : RX, TX et masse. Les contrôleurs série actuels supportent couramment des vitesses de l'ordre d'un méga-baud, soit environ 100 Ko de données transférées par seconde. Problème majeur cependant, les niveaux de voltage attendus sont de l'ordre de +/- 7 à 12 volts ; on ne peut donc pas brancher directement un Spartan 3 dessus car les ports d'I/O bruleraient.

Un convertisseur de niveaux de voltage est nécessaire. Parmi les puces disponibles, le MAX3232 est compatible 3.3V (c'est le cousin du MAX232, compatible 5V) et semble être un candidat sérieux. Nous n'avons pas poursuivi dans cette voie, le débit maximal étant

trop insuffisant pour certains cas d'usage (dump de la RAM, ...). Par ailleurs, le port série tend à disparaitre ; son utilisation aurait requis l'achat d'un convertisseur USB ⇔ série.

Interface USB sans PHY Mi-2009, Robert E. Jenkins, chercheur à l'université de Johns Hopkins publie sur le site du constructeur Xess le code source d'un contrôleur USB ne nécessitant pas d'interface physique (ou PHY) supplémentaire [22], ce qui permet de relier directement un FPGA de type Spartan 3 à un câble USB. En effet, les niveaux de voltage USB, soit 3.3V sont directement compatibles avec ce FPGA. Une limitation importante est que la conception de M. Jenkins limite la fréquence d'horloge à une fraction de la fréquence principale – seul l'USB 1.1 (1.5 MHz) est supporté. Le débit tombe alors au plus à 300 Ko par seconde environ. Nous avons testé ce composant, avec des résultats décevants : les lectures ou écritures ponctuelles fonctionnent, mais des requêtes soutenues donnent lieu au bout de quelques secondes à une désynchronisation USB. Il faut alors réinitialiser le FPGA pour revenir dans un état normal, et nous avons abandonné cette piste.

Puce FTDI La société FTDI commercialise des puces convertissant directement le protocole série côté FPGA en USB 2.0 (côté PC) – un driver spécifique est requis. Peu coûteux, facile à implémenter et supportant de débits jusqu'à 3 Mbauds (environ 300 Ko/s), cette solution est une alternative intéressante au module USB que nous allons maintenant présenter.

Module ComBlock USB Outre sa carte COM-1300-C, la société Mobile Satellite Services (MSS) commercialise pour un prix équivalent, le module COM-1400. Il est doté à l'identique d'un FPGA Xilinx xc3s400, et dispose en sus d'une interface physique USB 2.0 (voir figure 7.

Cette carte est compatible électriquement et logiquement avec les autres modules de la société MSS. Cependant, le code source complet des modules USB ainsi que CardBus ne sont pas disponibles : les parties cruciales sont fournies sous forme de boîte noire précompilée, car le code source VHDL est vendu en sus. Or nous n'utilisons

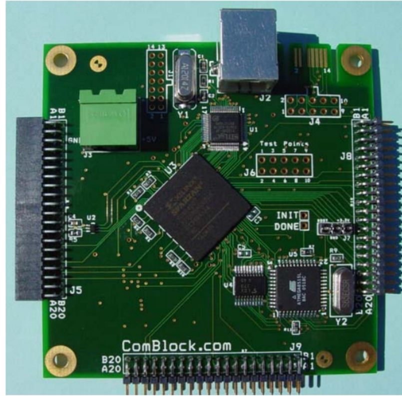


Figure 7. Module COM-1400 (source : Mobile Satellite Services)

pas ici le composant CardBus de MSS, ce dernier ne gérant pas les transferts DMA. Il a donc été nécessaire de redévelopper un protocole de communication simplifié entre les cartes COM-1300-C et COM-1400.

Sur les 40 pins du connecteur d'extension, 32 sont disponibles. Nous avons utilisé ici 16 pins pour les données (8 bits, en bidirectionnel), plus deux pins pour les requêtes de lecture ou écriture. Le problème de la métastabilité, courant en électronique, se pose lors de l'interconnexion de composants logiques se mettant à jour chacun sur une horloge différente (CLK_A et CLK_B). Même si la fréquence d'horloge est identique des deux côtés, le changement d'un signal synchrone avec CLK de 0 à 1 peut se produire au moment ou CLK_B envoie un coup d'horloge. Le système synchronisé sur CLK_B peut de voir la valeur 0 au lieu du 1 attendu (phénomène de "race condition").

Différentes solutions sont envisageables :

- utiliser la même horloge sur les deux systèmes ; cela ne semble pas possible ici car le PCI fonctionne à 33 MHz, et le module COM-1400 à 90 MHz. En pratique, rien n'empêche de faire fonctionner le moteur DMA à 33 MHz et le reste de plasma à 90 MHz, mais se pose alors le problème de synchroniser le moteur DMA avec plasma. De plus, le compilateur de Xilinx indique que plasma ne fonctionnerait pas correctement à plus de 40 MHz ;

– les deux domaines d'horloge se mettent d'accord sur une fréquence très inférieure aux deux, par exemple 2 MHz, et échantillonnent chacun le signal de l'autre à une fréquence quatre fois supérieure, un bit de départ permettant la synchronisation initiale. Ce protocole est en fait celui du RS-232. Le désavantage est un débit réduit du fait de fréquences plus faibles ;
– les composants BlockRAM de Xilinx offrent deux ports d'accès asynchrone, et fonctionnent comme une liste "premier entré, premier sorti". Chacun des deux ports est conçu pour fonctionner dans un domaine d'horloge différent ;
– lorsque l'une des deux horloges est au moins deux fois plus rapide que l'autre, il est possible d'assurer que la lecture est toujours fiable en échantillonnant deux fois le signal par l'horloge la plus rapide. Cette contrainte est respectée dans le cas présent ($90 > 33 * 2$), nous avons donc retenu cette solution qui offre un bon débit (au plus 33 Mo/s).

Côté COM-1400 (USB) Les changements apportés au code d'exemple de ComBlock sont peu importants ; dans un premier temps, les signaux d'entrées/sorties sont reliés au connecteur d'extension J8 :

```
RX_1300_ENABLE <= RIGHT_CONNECTOR(5);
RIGHT_CONNECTOR(18) <= RX_1300(6); RIGHT_CONNECTOR(17) <=
    RX_1300(7);
RIGHT_CONNECTOR(16) <= RX_1300(4); RIGHT_CONNECTOR(15) <=
    RX_1300(5);
RIGHT_CONNECTOR(14) <= RX_1300(2); RIGHT_CONNECTOR(13) <=
    RX_1300(3);
RIGHT_CONNECTOR(12) <= RX_1300(0); RIGHT_CONNECTOR(11) <=
    RX_1300(1);

TX_1300_ENABLE <= RIGHT_CONNECTOR(6);
TX_1300(6) <= RIGHT_CONNECTOR(28); TX_1300(7) <=
    RIGHT_CONNECTOR(27);
TX_1300(4) <= RIGHT_CONNECTOR(26); TX_1300(5) <=
    RIGHT_CONNECTOR(25);
TX_1300(2) <= RIGHT_CONNECTOR(24); TX_1300(3) <=
    RIGHT_CONNECTOR(23);
TX_1300(0) <= RIGHT_CONNECTOR(22); TX_1300(1) <=
    RIGHT_CONNECTOR(21);
```

L'inversion des bits deux à deux est normale. En effet, la nappe a été retournée pour éviter de devoir la plier (cf. photo ci-après).

Ces signaux sont par ailleurs connectés au module propriétaire USB de MSS :

```
-- charge finale disponible pour l'utilisateur ("Data stream
    2")
DATA2_OUT_SAMPLE_CLK_REQ => RX_1300_SAMPLE_CLK_REQ, -- demande
    de lecture
DATA2_OUT => RX_1300, -- donnees USB vers CardBus

DATA2_IN_SAMPLE_CLK => TX_1300_SAMPLE_CLK, -- demande d'
    ecriture
DATA2_IN => TX_1300_D2, -- donnees CardBus vers USB
```

Enfin est défini le processus d'échantillonnage, qui déclenche une lecture ou écriture en cas de changement des signaux de contrôle (enable) :

```
RECLOCK_CB: process(CLK_P)
begin
   if rising_edge(CLK_P) then
      TX_1300_D  <= TX_1300;
      TX_1300_D2 <= TX_1300_D;
      TX_1300_ENABLE_D  <= TX_1300_ENABLE;
      TX_1300_ENABLE_D2 <= TX_1300_ENABLE_D;
      RX_1300_ENABLE_D  <= RX_1300_ENABLE;
      RX_1300_ENABLE_D2 <= RX_1300_ENABLE_D;
      if (RX_1300_ENABLE_D = '0') and (RX_1300_ENABLE_D2 = '1')
           then
          RX_1300_SAMPLE_CLK_REQ <= '1';
      else
          RX_1300_SAMPLE_CLK_REQ <= '0';
      end if;
      if (TX_1300_ENABLE_D = '0') and (TX_1300_ENABLE_D2 = '1')
           then
          TX_1300_SAMPLE_CLK <= '1';
      else
          TX_1300_SAMPLE_CLK <= '0';
      end if;
   end if;
end process;
```

Côté COM-1300 (CardBus) Un module nommé p4_usb a été ajouté au code source VHDL. Il constitue seulement une glu entre plasma et le connecteur d'extension ; sa description ne présente pas d'intérêt particulier (l'échantillonnage n'est pas nécessaire dans le domaine d'horloge le plus lent). Ce module est activé lors d'une lecture ou d'une écriture vers l'adresse “magique” 0x7FFC.

Protocole de communication (couche de transport) Comment déterminer si des données sont disponibles dans le tampon de lecture, ou si le tampon d'écriture est plein ? Le composant USB propose deux signaux à cet effet : `DATA2_OUT_BUFFER_EMPTY` et `DATA2_IN_SAMPLE_CLK_REQ`. Néanmoins lors de tests ces signaux se sont avérés non fiables, et l'absence du code source du module USB de MSS n'a pas facilité le débogage de ce problème.

Envoi et réception Lorsque le tampon de lecture est vide, la lecture sur la carte COM-1300 d'un octet USB donne une valeur identique au dernier octet lu. Afin de pallier à l'absence de contrôle de flot fiable (signal `DATA2_OUT_BUFFER_EMPTY`), les donnés à envoyer ou recevoir sont coupés en deux morceaux ("nibbles", cf. [23]), et la valeur du bit de poids fort bascule entre 0 et 1 pour indiquer la présence de quatre nouveaux bits. Considérons l'exemple suivant :

```
Donnees a envoyer               : CC 81 00 FC
Donnees apres decoupage en deux : 0C 0C 08 01 00 00 0F 0C
Donnees avec bit n.8 positionne : 8C 0C 88 01 80 00 8F 0C
```

Remarque : cette solution est simple à implémenter mais divise le débit effectif par deux. Une alternative plus efficace serait l'utilisation de l'encodage base64, ou encore l'ajout d'un 9$^{\text{ème}}$ bit.

Est ajouté un en-tête contenant une séquence de synchronisation initiale et finale (0xAA répété quatre fois) plus un identifiant de paquet (ici, 0x5F en grand indien). Le message final envoyé par USB devient :

```
AA AA AA AA 85 0F 00 00 00 00 00 00 8C 0C 88 01 80 00 8F 0C AA
    AA AA AA
```

Ce paquet sera envoyé de manière répétée grâce à un timeout configurable jusqu'à réception du message "ACK" (charge finale de taille nulle) correspondant au même identifiant (0x5F) :

```
AA AA AA AA 85 0F 00 00 00 00 00 00 AA AA AA AA
```

Le lecteur avisé constatera dans ce format de trame l'absence d'une somme de contrôle ; en effet un "checksum" est déjà présent dans les paquets du protocole de débogage encapsulés. Néanmoins,

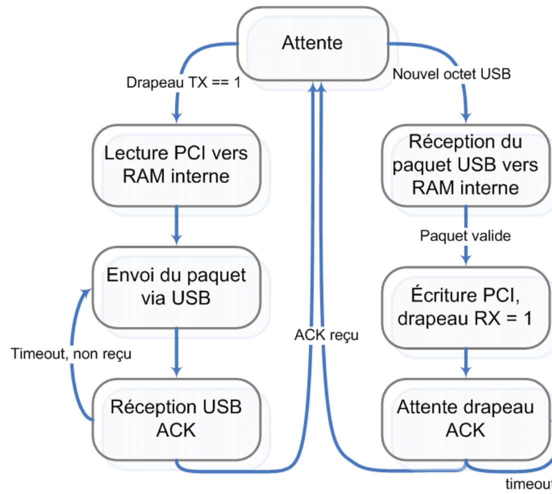


Figure 8. Machine à état du composant COM-1300-C

trois bits sur sept sont inutilisés, et pourraient à terme servir comme bits de contrôle d'erreur.

La réception d'un paquet se fait suivant le protocole décrit précédemment. Une fois le paquet lu en entier, un ACK est envoyé. Notons que les messages ACK inattendus sont simplement ignorés.

Machine à état finale Le schéma suivant (voir figure 8) décrit le fonctionnement de la carte COM-1300-C après chargement du code MIPS de communication :

5.3 Montage final

La figure 9 présente le montage après mise en place des deux cartes :

6 Conclusion

Nous avons démarré ce projet fin 2009 car l'idée de créer un débogueur à base d'hyperviseur semblait novatrice et utile à différentes tâches ; de plus, il n'existait pas de débogeur similaire : *SoftICE* n'est plus développé, *Syser* fonctionne de manière locale et n'est pas extensible, *WinDbg* oblige l'utilisateur à booter en mode

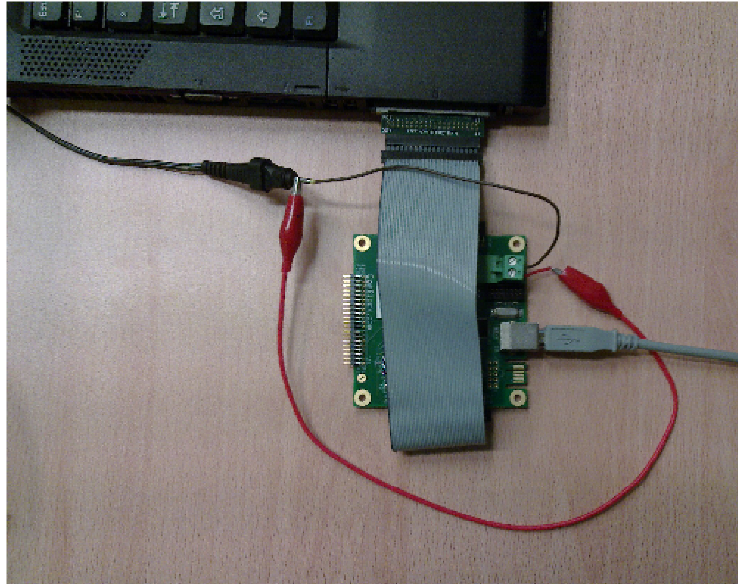


FIGURE 9. Montage final

debug. Un code malicieux présent dans le noyau serait capable de détecter facilement ces derniers.

Dans cette optique, nous avons décidé d'employer les méthodes de furtivité des rootkits dits "*ring -1*" comme BluePill afin d'implémenter un débogueur noyau ne nécessitant aucun soutien du système d'exploitation. Par conception *virtdbg* est furtif et modifie très peu le système ; c'est donc un outil adapté pour étudier des codes malicieux ou des protections logicielles.

Ont été présentés tout au long de l'article les différents composants de *virtdbg*, les problèmes rencontrés et les solutions retenues. *virtdbg* a été pour nous un projet très formateur de part la large palette de domaines qu'il a été nécessaire de comprendre et maîtriser : technologies de virtualisation et de débogage noyau, développement matériel, interfaçage électronique, etc.

L'objectif à court terme de *virtdbg* est le débogage d'un système sous Windows 7 64 bits avec PatchGuard activé ; par ailleurs, il est prévu de porter l'hyperviseur en 32 bits. Nous envisageons de plus l'ajout d'une couche de transport basée sur les sockets kernel, ce

qui permettra de déboguer via le réseau, sans dispositif matériel supplémentaire.

Références

1. GNU gdb : http://www.gnu.org/software/gdb/.
2. Eymery, D., Eymery, O., Borello, J.M., Fraygefond, J.M., Bion, P. : GenDbg : un débogueur générique. In : Actes du 6ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC). (2008) 136–159
3. IDA Pro : http://www.hex-rays.com/idapro/.
4. kgdb : http://en.wikipedia.org/wiki/KGDB.
5. I/O kit : http://developer.apple.com/sdk/.
6. WinDbg : http://www.microsoft.com/whdc/devtools/debugging/.
7. SoftICE : http://en.wikipedia.org/wiki/SoftICE.
8. rr0d : http://rr0d.droids-corp.org/.
9. Syser : http://www.sysersoft.com/.
10. Skywing : PatchGuard Reloaded : A Brief Analysis of PatchGuard Version 3. uninformed (2007)
11. Intel(R) 64 and IA-32 Architectures Software Developer's Manuals : http://www.intel.com/products/processor/manuals/.
12. AMD Developer Guides and Manuals : http://developer.amd.com/documentation/guides/.
13. Devine, C., Vissian, G. : Compromission physique par le bus PCI. In : Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC). (2009) 169–193
14. Aumaitre, D. : Voyage au coeur de la mémoire. In : Actes du 6ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC). (2008) 378–437
15. DR RootKit : (2008) http://www.immunitysec.com/downloads/linux_rootkit_source.tbz2.
16. Rhoads, S. : http://plasmacpu.no-ip.org:8080/.
17. matching debug information : http://www.debuginfo.com/articles/debuginfomatch.html.
18. Dornseif, M. : 0wned by an iPod
19. Lacombe, E., Deswarte, Y., Sang, F.L., Nicomette, V. : Analyse de l'efficacité du service fourni par une IOMMU. In : Actes du 8ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC). (2010)
20. Shanley, T., Anderson, D. : PCI system architecture. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1995)
21. Hennessy, J., Jouppi, N., Baskett, F., Gill, J. : MIPS : a VLSI processor architecture. In : Proc. CMU Conference on VLSI Systems and Computations. (1981) 337–346
22. Jenkins, R. : FPGA-based USB interface http://www.xess.com/projects/FPGA-USB-V2/readmeusb-v2.php.
23. Nibbles microblog : http://nibbles.tuxfamily.org/.

A Annexes

A.1 Définition de la structure _KDDEBUGGER_DATA64

```
typedef struct _KDDEBUGGER_DATA64 {

    struct _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.
    //

    UINT16  ThCallbackStack;            // offset in thread
        data

    //
    // these values are offsets into that frame:
```

```
        //

        UINT16  NextCallback;               // saved pointer to
            next callback frame
        UINT16  FramePointer;               // saved frame pointer

        //
        // pad to a quad boundary
        //
        UINT16  PaeEnabled;

        //
        // 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;

        ULONG64   ExpSystemResourcesList;
        ULONG64   ExpPagedPoolDescriptor;
        ULONG64   ExpNumberOfPagedPools;

        ULONG64   KeTimeIncrement;
        ULONG64   KeBugCheckCallbackListHead;
        ULONG64   KiBugcheckData;

        ULONG64   IopErrorLogListHead;

        ULONG64   ObpRootDirectoryObject;
        ULONG64   ObpTypeObjectType;

        ULONG64   MmSystemCacheStart;
        ULONG64   MmSystemCacheEnd;
        ULONG64   MmSystemCacheWs;

        ULONG64   MmPfnDatabase;
        ULONG64   MmSystemPtesStart;
        ULONG64   MmSystemPtesEnd;
        ULONG64   MmSubsectionBase;
        ULONG64   MmNumberOfPagingFiles;

        ULONG64   MmLowestPhysicalPage;
```

```
        ULONG64   MmHighestPhysicalPage;
        ULONG64   MmNumberOfPhysicalPages;

        ULONG64   MmMaximumNonPagedPoolInBytes;
        ULONG64   MmNonPagedSystemStart;
        ULONG64   MmNonPagedPoolStart;
        ULONG64   MmNonPagedPoolEnd;

        ULONG64   MmPagedPoolStart;
        ULONG64   MmPagedPoolEnd;
        ULONG64   MmPagedPoolInformation;
        ULONG64   MmPageSize;

        ULONG64   MmSizeOfPagedPoolInBytes;

        ULONG64   MmTotalCommitLimit;
        ULONG64   MmTotalCommittedPages;
        ULONG64   MmSharedCommit;
        ULONG64   MmDriverCommit;
        ULONG64   MmProcessCommit;
        ULONG64   MmPagedPoolCommit;
        ULONG64   MmExtendedCommit;

        ULONG64   MmZeroedPageListHead;
        ULONG64   MmFreePageListHead;
        ULONG64   MmStandbyPageListHead;
        ULONG64   MmModifiedPageListHead;
        ULONG64   MmModifiedNoWritePageListHead;
        ULONG64   MmAvailablePages;
        ULONG64   MmResidentAvailablePages;

        ULONG64   PoolTrackTable;
        ULONG64   NonPagedPoolDescriptor;

        ULONG64   MmHighestUserAddress;
        ULONG64   MmSystemRangeStart;
        ULONG64   MmUserProbeAddress;

        ULONG64   KdPrintCircularBuffer;
        ULONG64   KdPrintCircularBufferEnd;
        ULONG64   KdPrintWritePointer;
        ULONG64   KdPrintRolloverCount;

        ULONG64   MmLoadedUserImageList;

        // NT 5.1 Addition

        ULONG64   NtBuildLab;
        ULONG64   KiNormalSystemCall;

        // NT 5.0 hotfix addition

        ULONG64   KiProcessorBlock;
        ULONG64   MmUnloadedDrivers;
        ULONG64   MmLastUnloadedDriver;
        ULONG64   MmTriageActionTaken;
        ULONG64   MmSpecialPoolTag;
```

```
    ULONG64   KernelVerifier;
    ULONG64   MmVerifierData;
    ULONG64   MmAllocatedNonPagedPool;
    ULONG64   MmPeakCommitment;
    ULONG64   MmTotalCommitLimitMaximum;
    ULONG64   CmNtCSDVersion;

    // NT 5.1 Addition

    ULONG64   MmPhysicalMemoryBlock;
    ULONG64   MmSessionBase;
    ULONG64   MmSessionSize;
    ULONG64   MmSystemParentTablePage;

    // Server 2003 addition

    ULONG64   MmVirtualTranslationBase;

    UINT16    OffsetKThreadNextProcessor;
    UINT16    OffsetKThreadTeb;
    UINT16    OffsetKThreadKernelStack;
    UINT16    OffsetKThreadInitialStack;

    UINT16    OffsetKThreadApcProcess;
    UINT16    OffsetKThreadState;
    UINT16    OffsetKThreadBStore;
    UINT16    OffsetKThreadBStoreLimit;

    UINT16    SizeEProcess;
    UINT16    OffsetEprocessPeb;
    UINT16    OffsetEprocessParentCID;
    UINT16    OffsetEprocessDirectoryTableBase;

    UINT16    SizePrcb;
    UINT16    OffsetPrcbDpcRoutine;
    UINT16    OffsetPrcbCurrentThread;
    UINT16    OffsetPrcbMhz;

    UINT16    OffsetPrcbCpuType;
    UINT16    OffsetPrcbVendorString;
    UINT16    OffsetPrcbProcStateContext;
    UINT16    OffsetPrcbNumber;

    UINT16    SizeEThread;

    ULONG64   KdPrintCircularBufferPtr;
    ULONG64   KdPrintBufferSize;

    ULONG64   KeLoaderBlock;

    UINT16    SizePcr;
    UINT16    OffsetPcrSelfPcr;
    UINT16    OffsetPcrCurrentPrcb;
    UINT16    OffsetPcrContainedPrcb;

    UINT16    OffsetPcrInitialBStore;
    UINT16    OffsetPcrBStoreLimit;
```

```
    UINT16    OffsetPcrInitialStack;
    UINT16    OffsetPcrStackLimit;

    UINT16    OffsetPrcbPcrPage;
    UINT16    OffsetPrcbProcStateSpecialReg;
    UINT16    GdtR0Code;
    UINT16    GdtR0Data;

    UINT16    GdtR0Pcr;
    UINT16    GdtR3Code;
    UINT16    GdtR3Data;
    UINT16    GdtR3Teb;

    UINT16    GdtLdt;
    UINT16    GdtTss;
    UINT16    Gdt64R3CmCode;
    UINT16    Gdt64R3CmTeb;

    ULONG64   IopNumTriageDumpDataBlocks;
    ULONG64   IopTriageDumpDataBlocks;

    // Longhorn addition

    ULONG64   VfCrashDataBlock;
    ULONG64   MmBadPagesDetected;
    ULONG64   MmZeroedPageSingleBitErrorsDetected;

} KDDEBUGGER_DATA64, *PKDDEBUGGER_DATA64;
```

Listing 1.35. structure _KDDEBUGGER_DATA64