Winbagility : Débogage furtif et introspection de machine virtuelle

Nicolas Couffin
nicolas.couffin@intradef.gouv.fr

DGA Maîtrise de l'information

Résumé. Lorsqu'un analyste souhaite réaliser l'étude du comportement d'un système sous Windows, peu de choix s'offrent à lui.Il doit passer le système en mode `/DEBUG` et se connecter avec le débogueur de Microsoft (WinDbg) au serveur KD qui se situe dans le noyau du système à analyser. WinDbg peut se connecter soit à distance via série, Ethernet, USB ou localement avec Live-KD. Quelle que soit la solution choisie, le comportement du système se retrouve alors modifié par l'environnement de débogage.
Dans cet article, nous présentons notre approche d'introspection et de débogage furtif de machine virtuelle et nous discutons de l'architecture et de l'implémentation de Winbagility. Winbagility est une réalisation de débogueur furtif fondé sur un hyperviseur, permettant d'utiliser la majeure partie des fonctionnalités de WinDbg sans passer le système en mode `/DEBUG`.

1 Introduction

Plusieurs débogueurs fondés sur la virtualisation ont vu le jour depuis une dizaine d'années, HyperDbg[1], Ramooflax [6], VMWare Debugger, VirtDbg, VMIDbg,... Même si certains de ces débogueurs présentent quelques similarités avec Winbagility, dont l'utilisation d'un hyperviseur, ils possèdent plusieurs inconvénients qui sont incompatibles avec les objectifs fixés que nous présentons un peu plus loin dans cet article. Dans la suite de ce chapitre, nous avons choisi de présenter le mode `/DEBUG` et les deux débogueurs qui se rapprochent le plus de Winbagility. Un lexique est disponible en annexe de cet article.

1.1 Mode /DEBUG

Le mode `/DEBUG` de Windows est un mode particulier de démarrage du système. Il permet d'attacher un débogueur au noyau afin d'analyser son comportement ou celui des pilotes. Il est souvent utilisé lors d'analyses de

[1] `https://code.google.com/archive/p/hyperdbg/`

vulnérabilité en espace noyau (noyau ou driver), pour étudier des drivers malveillants (*rootkit*, ...), pour aider à la rétro-conception de certaines fonctionnalités du noyau ou tout simplement pour le débogage au cours du dévelopement de composants noyaux.

Le comportement du système en mode `/DEBUG` n'est pas exactement le même qu'un système démarré normalement :

- Certaines structures habituellement chiffrées pour éviter leur modification ou leur lecture comme `KDBG` de type `_KDDEBUGGER_DATA64` ne le sont pas en mode `/DEBUG`.
- *PatchGuard* qui garantit l'intégrité du noyau n'est pas complètement fonctionnel, car certaines de ces vérifications ne sont pas compatibles avec un débogueur.
- Le serveur KD est chargé mais n'est pas utilisable.
- Etc...

1.2 État de l'art : VirtDbg

Lors du SSTIC 2010, Damien Aumaitre et Christophe Devine ont présenté VirtDbg [4], un débogueur fondé sur un hyperviseur minimaliste injecté dans la mémoire noyau du système en utilisant les DMA, DMA qui servent aussi à le piloter.

Ce débogueur nécessite l'utilisation d'une machine physique sur laquelle est connecté un équipement matériel dont il est difficile de faire l'acquisition. Ceci rend l'utilisation de ce débogueur fastidieuse (démarrage, absence de sauvegarde...) et couteuse (PCB, FireWire, FPGA,...).

Deuxièmement, Virtdbg modifie directement la mémoire du système en cours d'utilisation pour y injecter du code non signé, poser un point d'arrêt logiciel ou modifier une *Page Table Entrie* (PTE) pour ces points d'arrêt mémoire. Ces modifications de la mémoire du système à analyser rendent le débogueur non furtif vis-à-vis des drivers qui peuvent vérifier leur intégrité ou même de *PatchGuard*.

De plus, depuis l'arrivée des technologies VT-d et IOMMU, l'injection via DMA ne se fait plus aussi facilement. Empêchant l'accès à la mémoire du système depuis un équipement matériel, ces technologies rendent VirtDbg inutilisable sur un système contemporain (Windows 8.1 et 10).

Depuis Windows 8, certaines structures du noyau nécessaires au débogage sont chiffrées afin d'en empêcher leurs utilisations par des *rootkit* [2]. Virtdbg ne possédant pas les primitives de déchiffrement, il nous est impossible de l'utiliser sur ces nouveaux systèmes.

Finalement, le stub utilisé par Virtdbg est gdbserver. Même si cette API est suffisante pour faire du débogage noyau, elle n'est pas compatible avec WinDbg qui propose des capacités d'analyse évoluées concernant les systèmes Windows.

1.3 État de l'art : LibVMI + VMIDbg

LibVMI[2] fournit une API en C conçue pour l'introspection de machine virtuelle s'exécutant, au moment d'écriture de l'article, sur les hyperviseurs KVM[3] ou Xen[4].

Sur KVM modifié, les possibilités sont limitées à l'accès à la mémoire et aux registres de l'invité, et à la pause et reprise de la machine virtuelle, tandis que sur Xen les possibilités sont plus nombreuses. Connecté à un hyperviseur Xen, LibVMI permet d'utiliser les événements de type :

- Accès R/W/X à une zone mémoire
- Accès au CR0, CR3, CR4 et MSR
- Interruptions
- Single-Step

Présenté au 31c3, VMIDbg[5] s'appuie sur LibVMI pour exposer une interface gdbserver offrant à l'utilisateur la possibilité de déboguer des systèmes en cours d'utilisation. Malgré toutes ces fonctionnalités et sa furtivité, VMIDbg possède plusieurs inconvénients qui ne permettent pas de réaliser un débogage poussé des systèmes Windows. En effet, VMIDbg ne permet pas de :

- poser des points arrêts assez performants pour répondre à nos besoins
- déboguer avec WinDbg et ses extensions
- faire fonctionner l'hyperviseur sur un autre OS que Linux
- réaliser un retour rapide à une sauvegarde

1.4 Problèmes de discrétion ou de performance

Quelle que soit la solution choisie, le système en cours de débogage se retrouve modifié soit par le passage en mode `/DEBUG`, soit par le chargement d'un driver, et il devient alors difficile de garantir qu'une application ou

[2] `http://libvmi.com/`
[3] `http://www.linux-kvm.org/`
[4] `htpp://www.xenproject.org/`
[5] `https://github.com/Zentific/vmidbg`

un maliciel fonctionneront à l'identique dans ce nouvel environnement ainsi modifié.

Dans le cas de LibVMI, bien que l'interface offre des capacités intéressantes, l'absence de support d'un hôte Windows ou de WinDbg rend le débogage de driver Windows difficile et limité. De plus, les performances en lecture mémoire de LibVMI ne permettent pas de réaliser une détection efficace lors d'analyses dynamiques [9].

2 Objectifs

Lors de la phase de conception de l'outil présenté, plusieurs objectifs principaux ont été fixés.

2.1 Analyse avec *Patchguard* activé

Patchguard[6] est une fonctionnalité des noyaux NT 64-bits, excepté XP, qui les protège des modifications. *Patchguard* est partiellement désactivé lors du passage en mode `/DEBUG`, il est donc difficile d'analyser un driver malveillant qui tente de le désactiver [5].

De plus, au vu des points protégés par *Patchguard*[7], il est tout aussi difficile de développer un débogueur noyau qui fonctionne en parallèle de celui-ci.

Voici par exemple une liste non exhaustive des modifications non autorisées :

- Modification de la SSDT
- Modification de l'IDT
- Modification de la GDT
- Modification du code du noyau

2.2 Analyse de drivers malveillants

L'un des objectifs principaux de Winbagility est d'offrir la capacité d'analyser des drivers malveillants. Certains développeurs, ne souhaitant pas que leurs drivers soient analysés, empêchent le chargement de leurs drivers en mode `/DEBUG`.

Plusieurs techniques de détection de débogueur noyau existent et sont utilisées par les maliciels afin de contourner ou de rendre difficile

[6] `http://uninformed.org/index.cgi?v=8&a=5`
[7] `http://www.alex-ionescu.com/?p=290`

leur analyse. Par exemple, ils ont la possibilité de lire la Boot Configuration Data [1], qui contient les options de démarrage des systèmes Windows, à la recherche du mode `/DEBUG`. Ces drivers peuvent aussi utiliser `SystemKernelDebuggerInformation` [7] ou vérifier que la valeur des registres de débogage est bien à 0. Il est aussi envisageable qu'ils puissent vérifier qu'aucun driver non signé n'ait été chargé sur le système.

2.3 Analyse dynamique et détection de compromission de VM

La détection de compromission de VM lors d'analyses dynamiques se fait habituellement avec l'utilisation d'un agent provoquant la modification du système analysé. Par exemple, Cuckoo utilise des *hooks* pour superviser les appels du processus [3], certains utilisent un débogueur attaché au processus et d'autres utilisent des *hooks* d'appels systèmes.

Toutes ces techniques présentent l'inconvénient de reposer sur des méthodes facilement détectables par un processus (`IsDebuggerPresent`, checksum de code) ou un driver (vérification de la SSDT). De plus, la modification de la SSDT ne peut être réalisée sans désactiver *PatchGuard*, et donc sans passer le système en mode `/DEBUG`.

En mode `/DEBUG` certains composants du système ne sont pas chargés, ou possèdent un comportement différent qu'en mode normal. Ainsi, si un de ces composants est ciblé par un exploit, il semble difficile de réaliser une détection de la vulnérabilité exploitée.

Winbagility permet aussi de réaliser la supervision de certains appels systèmes critiques, à l'aide de points d'arrêts performants. Enfin, il a la capacité de vérifier l'intégrité du noyau et certaines de ses structures, ou de détecter le lancement d'applications non autorisées.

3 Techniques

Dans cette partie de l'article, nous présentons les différentes techniques qui ont été mises en œuvre pendant la conception de Winbagility.

3.1 KD

Le *Kernel Debugging protocol* (KD) est une interface bas-niveau utilisée par le noyau et son débogueur pour communiquer ensemble. Ce protocole propriétaire analogue à GDB, permet au noyau de présenter un stub utilisable à distance par WinDbg.

Contrairement à GDB, KD est un protocole binaire où l'unité d'échange est le message. Ces messages peuvent être échangés via divers supports comme le RS232, le NamedPipe, l'USB, le FireWire ou bien l'Ethernet. Pour chacun de ces vecteurs, une surcouche permet de gérer les spécificités du support choisi. Ainsi il existe KDCOM pour le RS232 ou le NamedPipe, KDUSB pour l'USB, KDNET pour l'Ethernet...

L'API offerte par KD est relativement complète et est centrée sur le débogage noyau, mais cette interface reste utilisable pour du débogage en mode utilisateur. Par exemple, l'existence des primitives `ReadPhysicalMemory`, `WritePhysicalMemory` et `QueryPhysicalMemory` permet de réaliser des opérations qui sont difficiles, voire impossibles, en utilisant GDB.

Initialement développé pour une utilisation sur RS232, un canal non robuste, KD possède une gestion d'acquittement, de détection d'erreur et d'ordonnancement des messages.

Divers personnes ou projets ont participé à l'analyse de ce protocole, j00ru[8] et ReactOS[9] jusqu'à KD6 et dernièrement VirtualKD[10] jusqu'à KD10, la dernière version à ce jour. Une ré-implémentation d'un serveur KD a été réalisée pour Winbagility afin d'y connecter WinDbg, cette implémentation pouvant être réutilisée pour d'autres outils.

Un exemple de message KD se trouve en annexe de cet article.

3.2 Virtualisation

Habituellement, le système d'exploitation a directement accès au matériel et aux composants comme le processeur, la mémoire ou les disques. Cela peut poser certains problèmes lors de l'analyse de malware (compromission), ou de vulnérabilité (BSOD).

La virtualisation utilise un *Virtual Machine Manager* (VMM) qui crée une couche d'abstraction entre l'OS et le matériel. Chaque *Virtual Machine* (VM) utilise alors des composants virtuels vCPU, vDisk... Cette capacité de virtualisation permet de faire depuis l'hôte des sauvegardes d'état et de faire fonctionner plusieurs systèmes sur une même machine physique.

De nos jours, la virtualisation repose sur les extensions de virtualisation matérielles développées par AMD (AMD-V) et Intel (VT-x) dans le milieu des années 2000. Dans la suite de cet article, nous nous focalisons sur

[8] `http://j00ru.vexillium.org/?p=405`
[9] `https://www.reactos.org/wiki/Techwiki:Kd`
[10] `http://virtualkd.sysprogs.org/`

VT-x, sachant que la majorité des mécanismes présentés par la suite sont identiques entre VT-x et AMD-V.

Avant toute chose, nous tenons à expliquer les mécanismes de VMEntry et de VMExit. Le VMEntry est réalisé lors de l'utilisation par l'hyperviseur des instructions `VMLAUNCH` ou `VMRESUME`. Ces instructions permettent à l'hyperviseur de passer le processeur dans le contexte de l'invité. Lorsqu'une exception est levée pendant que l'invité s'exécute et si la configuration de l'invité le permet, l'exception sera remontée au niveau de l'hyperviseur. L'hyperviseur exécutera alors un handler en fonction du type d'exception levé.

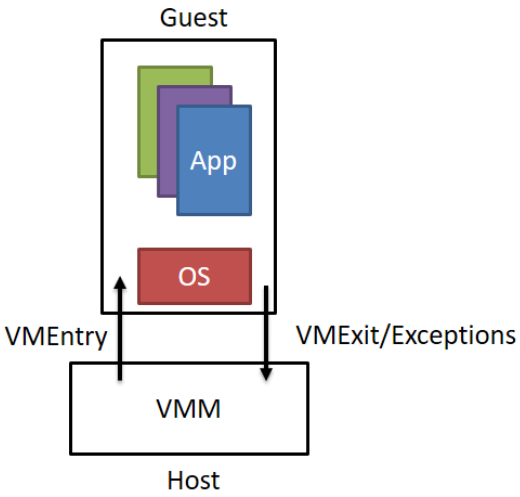

Fig. 1. VMEntry et VMExit entre l'invité et l'hôte

Une *Virtual Machine Control Structure*[11] (VMCS) est un espace mémoire physique de 4K qui définit et spécifie l'environnement de travail d'un vCPU d'une VM. Dans cette structure sont stockés différents éléments :

Guest/Host Save Area Lors d'un VMEntry, l'état du processeur physique hôte est sauvegardé dans la *Host Save Area* (HSA) puis le contexte du processeur est défini avec les valeurs que se trouvent dans la *Guest Save Area* (GSA) de la VMCS.

A contrario lors d'une exception générant un VMExit, l'état du CPU est sauvegardé dans la GSA puis chargé à partir de la HSA.

La VMCS étant dans l'espace d'adressage de l'hôte, l'état du vCPU de l'invité est lisible et inscriptible depuis la GSA.

[11] `http://lxr.free-electrons.com/source/arch/x86/kvm/vmx.c`

VM-Execution Control Field Le *Primary Processor-Based VM-Execution Control* permet de définir quand sera généré un VMExit, un VMExit étant le fait de passer de l'invité à l'hôte. Si un bit associé à un type d'exception est à 0 dans ce champ de 32 bits, l'exception n'est pas levée et l'invité continuera de s'exécuter sur le processeur. Dans le cas contraire, si le bit est positionné, le processeur repassera en mode hyperviseur pour l'exécution d'un handler d'exception spécifique.

Voici une liste non exhaustive des exceptions utilisées dans le cadre de ce projet :

```
/** VM-exit when executing the HLT instruction. */
#define VMX_VMCS_CTRL_PROC_EXEC_HLT_EXIT            RT_BIT(7)
/** VM-exit when executing a MOV DRx instruction. */
#define VMX_VMCS_CTRL_PROC_EXEC_MOV_DR_EXIT         RT_BIT(23)
/** Monitor trap flag. */
#define VMX_VMCS_CTRL_PROC_EXEC_MONITOR_TRAP_FLAG   RT_BIT(27)
```

Listing 1. Exemples d'exceptions VT-x

Le *Monitor Trap Flag* (MTF) est un drapeau spécialement conçu pour exécuter l'invité en mode pas à pas. Quand le MTF est à 1, l'invité va générer à chaque instruction une exception de type #MTF et donc redonner la main au handler d'exception en mode hyperviseur.

VM-Entry Control Field Un champ de la VMCS, la `ENTRY_INTERRUPTION_INFO` définit quelles sont les interruptions qui sont levées dans l'invité lors du prochain VMEntry, permettant d'injecter des interruptions dans la VM depuis l'hôte. Nous expliquons dans la suite de cet article comment nous les utilisons pour créer des points d'arrêt.

3.3 *Second Level Address Translation*

Les processeurs modernes utilisent les concepts de mémoire physique et de mémoire virtuelle ; les processus utilisent la mémoire virtuelle et le processeur utilise lui, la mémoire physique. Cette conversion est réalisée par un composant matériel, la *Memory Management Unit* (MMU) et se fait à l'aide d'une *Page Table*. Les détails de ce mécanisme sont expliqués dans un des manuels Intel [8].

Ce mécanisme permet entre autres de protéger les pages du noyau ou d'interdire la lecture des pages d'un processus depuis un autre processus.

Quand un système est virtualisé, un mécanisme identique est mis en place, la SLAT pour *Second Level Address Translation*. Le nom SLAT

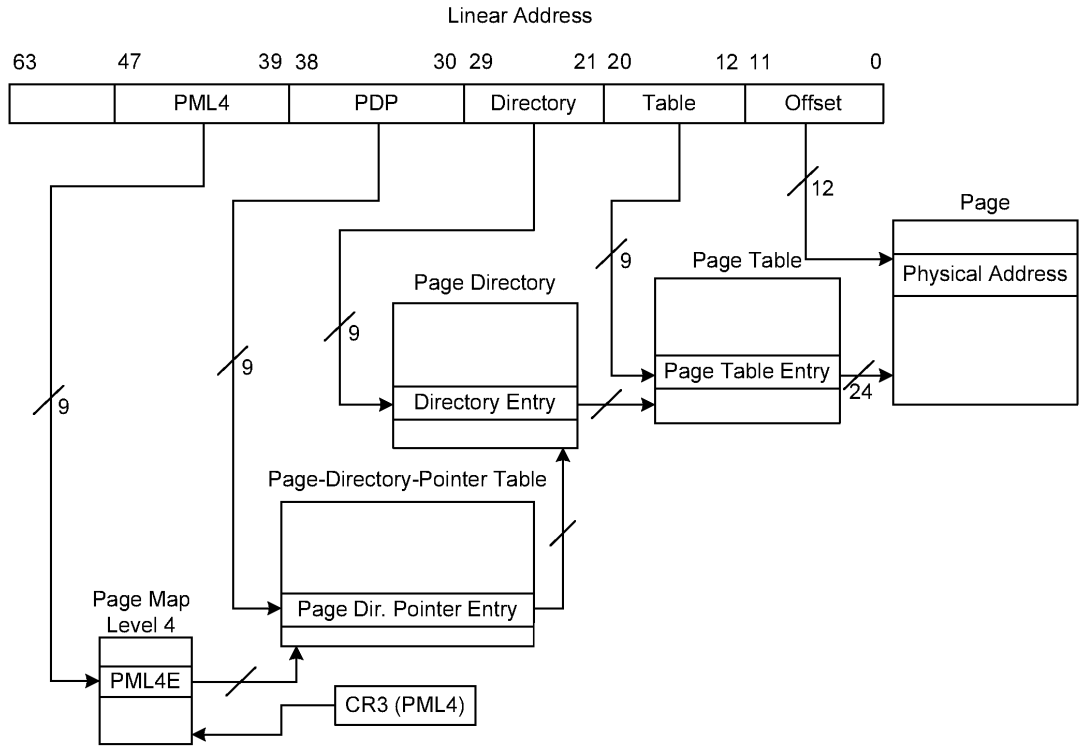

Fig. 2. Schéma d'une *Page Table* x64

est le nom générique de cette technologie, la solution d'Intel est nommée *Extended Page Table* (EPT), tandis que celle d'AMD s'appelle *Rapid Virtualisation Indexation* (RVI) ou *Nested Page Table* (NPT). L'EPT permet de protéger les pages de l'hyperviseur et des accès mémoire entre VM.

Il existe 3 types d'adresse lors du mécanisme de traduction :

- *Guest Virtual Address* (GVA), adresses virtuelles manipulées par les processus de l'invité.
- *Guest Physical Address* (GPA), adresses physiques manipulées par la machine virtuelle.
- *Host Physical Address* (HPA), adresses physiques manipulées par l'hôte.

Prenons l'exemple d'un driver qui s'exécute dans une machine virtuelle et qui souhaite écrire `0x67` à l'adresse (GVA) `0xFFFFF6FB7DBEDF68`.

1. La GVA sera traduite par la MMU en une GPA en utilisant le CR3 du CPU. Si une erreur arrive pendant la conversion (ex. : Page Fault), l'interruption reste dans la VM.
2. La GPA acquise sera traduite par l'EPT en une HPA en utilisant l'*EPT Pointer* (EPTP) qui se trouve dans la VMCS. Si une erreur

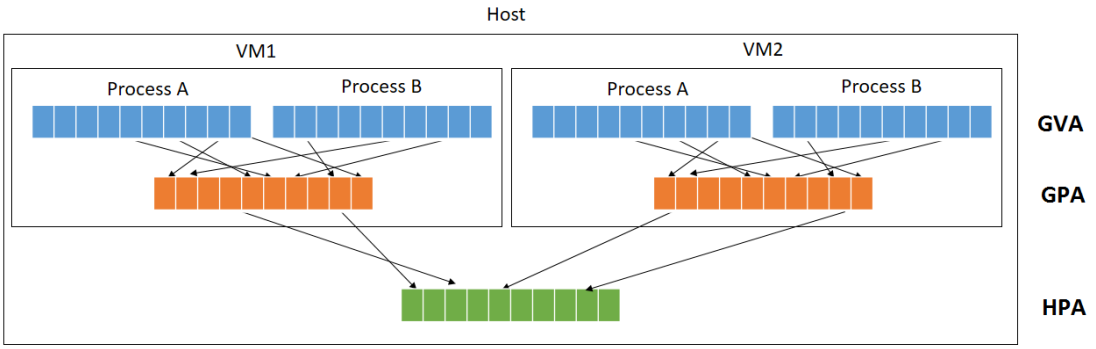

Fig. 3. Différents niveaux d'adressage utilisés pendant la virtualisation

survient pendant la conversion, l'exception (#EPTViolation) remonte jusqu'à l'hyperviseur.

Pour information :

- Depuis l'hôte, une GVA d'un processus peut être traduite en GPA en lisant les GPA de la *Page Table* du processus.
- Depuis l'hôte, la GPA ne représente rien et n'est utilisable qu'en association avec l'EPT.
- Depuis l'invité, une HPA n'est pas atteignable.

4 Architecture

Dans cette partie de l'article, nous présentons rapidement l'architecture générale de Winbagility et certains composants qu'il utilise.

L'architecture (Figure 4) de Winbagility a été pensée pour être robuste et performante. Cette robustesse provient essentiellement du fait que la majeure partie des traitements est réalisée hors de l'hyperviseur.

4.1 VirtualBox

VirtualBox[12] à l'instar de VMWare est un hyperviseur de type 2[13], ce qui signifie qu'il s'exécute en tant que driver d'un système hôte.

Pour ce projet, le choix s'est tourné assez rapidement vers VirtualBox. Comparé à d'autres solutions de virtualisation, VirtualBox possède plusieurs avantages. Premièrement, son code est diffusé ce qui nous autorise

[12] `http://virtualbox.org/`
[13] `http://searchservervirtualization.techtarget.com/definition/hosted-hypervisor-Type-2-hypervisor`

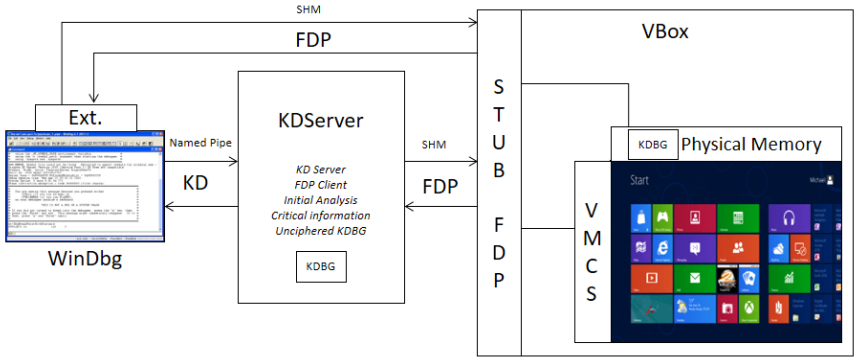

Fig. 4. Architecture générale de Winbagility

à étendre ses fonctionnalités plus facilement. Ensuite, VirtualBox est compatible avec des systèmes hôtes différents (Windows, Linux ou Mac OS) et il peut aussi bien être lancé avec une interface utilisateur qu'en mode démon.

Le montage de la chaîne de compilation[14] est assez long et fastidieux, car elle nécessite beaucoup de dépendances dans des versions très précises (Visual Studio 2010, KMK, OpenSSL, QT,...). Une fois cette chaîne de compilation montée, nous avons la possibilité de compiler des versions modifiées de VirtualBox. Les fichiers de sortie sont nombreux (service, driver, hyperviseur, DLL, GUI, ...) et ne sont pas signés. Cela nous oblige à démarrer le système hôte en `/TESTSIGNING`, ce qui permet de charger dans le noyau de l'hôte des drivers non signés.

Dans le cadre de l'intégration au sein de VirtualBox de l'API FDP, présentée plus tard dans cet article, nous nous sommes efforcés à modifier le minimum de fichiers. Cela nous permet de maintenir le patch plus facilement et de l'intégrer dans les nouvelles versions de VirtualBox sans trop de difficultés. Les modifications que nous avons réalisées dans VirtualBox portent essentiellement sur le *Virtual Machine Manager*, en particulier sur le :

- VMMr0 (VMM en ring-0)
  - *Hardware Manager VMX* (HMVMX) qui gère les VMEntry, VMExit et exceptions
- VMMr3 (VMM en ring-3)
  - *Execution Manager* (EM) qui gère la boucle principale de l'hyperviseur
  - *Page Manager* (PGM) qui gère les pages de l'invité

[14] `https://www.virtualbox.org/wiki/Windowsbuildinstructions`

- *Save State Manager* (SSM) qui gère les sauvegardes et restaurations d'état

L'utilisation par Winbagility de VirtualBox comme hyperviseur de base peut le rendre détectable. En effet plusieurs techniques de détection de VM sont disponibles publiquement, les outils comme ScoopyNG[15] ou Pafish[16] en sont les parfaits exemples. Ceci peut être résolu par l'utilisation du patch[17] de VirtualBox qui contourne les techniques de détections de VM, mais cela sort du cadre du projet présenté.

4.2 WinDbg

Habituellement lorsqu'un analyste souhaite déboguer le noyau Windows, il fera appel à WinDbg. WinDbg est un débogueur (noyau et utilisateur) interactif et gratuit, développé et distribué par Microsoft dans les Debugging Tools[18].

WinDbg possède plusieurs fonctionnalités que nous jugeons importantes ; la gestion du format PDB[19], la spécificité des commandes, la vaste collection des extensions disponibles et la flexibilité de son langage de scripting. Toutes ces possibilités font de WinDbg un excellent choix pour l'étude du comportement de drivers chargés dans le noyau NT.

WinDbg peut être utilisé pour déboguer un système qui s'exécute en mode `/DEBUG`, soit à distance avec KD ou soit en local avec Live-KD. Il a aussi la capacité d'analyser des *crashdumps* qui sont des images du système générées lors d'une erreur critique de celui-ci.

Voici des exemples d'extensions disponibles avec WinDbg :

- !pool, permet d'avoir l'état des pools et d'y détecter des corruptions
- !pte, permet d'avoir les PML4E, PDPE, PDE, PTE et les droits d'une adresse virtuelle
- !process, permet d'avoir la liste des processus avec leurs informations détaillées
- !analyze, permet d'avoir des informations concernant un crash système
- lm, permet d'avoir la liste des drivers chargés par le noyau

[15] `http://www.trapkit.de/research/vmm/scoopyng/`
[16] `https://github.com/a0rtega/pafish`
[17] `http://www.kernelmode.info/forum/viewtopic.php?f=11&t=3478`
[18] `https://msdn.microsoft.com/en-us/library/windows/hardware/mt219729(v=vs.85).aspx`
[19] `https://msdn.microsoft.com/en-us/library/yd4f8bd1(vs.71).aspx`

4.3 *Fast Debugging Protocol*

Après plusieurs essais d'API, *VBox TCP Ascii* et *VBox IMachineDebugger*, nous nous sommes vite rendu compte qu'aucune d'entre elles ne nous permettrait de réaliser les actions que nous souhaitions (lecture de GPA, modifications des EPTPTE, performance...).

Nous avons donc pris la décision de développer une interface en C, minimaliste et performante pour le débogage noyau, appelée FDP pour *Fast Debugging Protocol*.

Cette API permet de :

- Lire/écrire de la mémoire Guest (physique ou virtuelle)
- Lire/écrire des registres Guest
- Installer/désinstaller des points d'arrêt
- Sauvegarder/restaurer l'état de l'invité
- Suspendre/reprendre l'exécution de l'invité
- Redémarrer l'invité

FDP fonctionne avec des mémoires partagées, ce qui permet d'éviter les appels systèmes pour une simple lecture ou écriture, garantissant de bonnes performances et un portage rapide vers d'autres systèmes d'exploitation. L'interface est relativement simple, comparée à KD ou GDB, mais permet de répondre à nos besoins. L'intégration du stub dans VirtualBox passe obligatoirement par une phase de recompilation, mais cela était le prix à payer pour que l'API de débogage réponde à nos besoins.

En annexe se trouve des exemples de fonctions proposées par FDP.

4.4 Analyse initiale

Winbagility fonctionne dans deux modes, le mode *Crash* et le mode *Live*. Dans le mode *Crash*, il peut autoriser WinDbg à analyser un dump mémoire physique, dans l'autre mode, il se connecte à VirtualBox pour déboguer une machine virtuelle. Dans la suite de ce chapitre, nous nous attardons essentiellement sur le mode *Live*.

Afin de faire fonctionner WinDbg avec un système démarré normalement, il faut réaliser une analyse préalable à son attachement au noyau, attachement au noyau qui n'est pas réel avec Winbagility. Dans la suite de ce chapitre nous présentons les grandes étapes de cette analyse initiale.

Traduction d'adresses WinDbg comme tout autre débogueur manipule essentiellement des adresses virtuelles (GVA). En revanche, depuis l'hôte, les adresses lisibles d'une machine virtuelle sont des GPA.

La première étape consiste donc à trouver une solution pour traduire les GVA en GPA et vice versa. Pour cela, il faut trouver une `DirectoryTableBase` valide, une `DirectoryTableBase` est l'adresse physique d'une PML4 (Figure 2).

Sous Windows, l'espace noyau est toujours valide, quel que soit le processus et quel que soit le contexte (utilisateur ou noyau) dans lequel le système s'exécute.

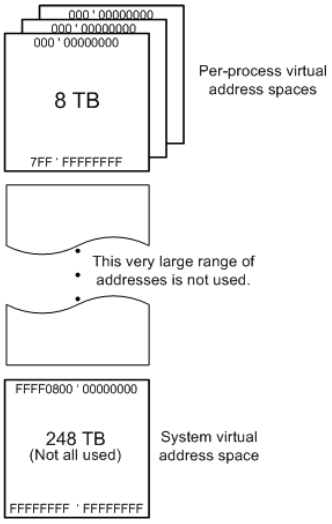

Fig. 5. Espaces d'addressage sous Windows x64

Sur les processeurs x64, le registre CR3 pointe vers une PML4. En lisant depuis l'hôte, la valeur du registre CR3 à partir de la VMCS, il est possible de récupérer la GPA d'une PML4 valide et à partir de là simuler le comportement d'une MMU.

KDBG Une fois que nous avons la possibilité de traduire des adresses GVA en GPA, la deuxième étape consiste à trouver et déchiffrer la structure `KDBG` de type `_KDDEBUGGER_DATA64`.

Lors de l'attachement de WinDbg au système en mode `/DEBUG`, le serveur KD délivre immédiatement au débogueur le `KDBG`. Cette structure est essentielle au bon fonctionnement du débogueur, car elle lui livre les adresses et les valeurs de plusieurs éléments importants du système (`KernelBase`, `PsLoadedModuleList`, `PsActiveProcessHead`...).

Comme nous avons pu l'expliquer dans un précédent chapitre, cette structure est chiffrée en mode normal et n'est donc pas délivrable telle quelle. Le chiffrement de cette structure est effectué dans la phase d'initialisation du noyau à l'aide de 3 clefs (`nt!KiWaitNever`, `nt!KiWaitAlways` et `nt!KdpDataBlockEncoded`)

A noter que si un BSOD se produit pendant que le noyau est configuré pour écrire un crahsdump, cette structure est déchiffrée puis écrite dans le rapport de crash. La fonction dans le noyau chargé de déchiffrer cette structure est `nt!KdCopyDataBlock` qui utilise la primitive suivante :

```
uint64_t UncipherData(
                uint64_t Data,
                uint64_t KiWaitNever,
                uint64_t KiWaitAlways,
                uint64_t KdpDataBlockEncoded){
    Data = Data^KiWaitNever;
    Data = _rol64(Data, KiWaitNever & 0xFF);
    Data = Data^KdpDataBlockEncoded;
    Data = BYTESWAP64(Data);
    Data = Data^KiWaitAlways;
    return Data;
}
```

Listing 2. Déchiffrement de 64 bits appartenant au KDBG

Pour trouver les GVA de ces différents éléments (Clefs, `KDBG`...), Winbagility utilise les PDB des serveurs de Microsoft qui donnent pour une version spécifique du noyau l'offset d'un symbole depuis la `KernelBase`.

Reste donc à trouver `KernelBase`. Pour cela, nous pouvons lire le *Model Specific Register* (MSR) LSTAR[20] qui pointe sur la GVA de `nt!KiSystemCall64`. Une fois la GVA de `nt!KiSystemCall64` en main, et avec le PDB du noyau (ou en bruteforçant de manière itérative les PDB), nous pouvons calculer `KernelBase`, déduire la GVA des clefs, déduire la GVA de `KDBG` pour enfin le déchiffrer.

5 *Breakpoints*

Un point d'arrêt est un mécanisme qui permet à un analyste d'intentionnellement mettre en pause un programme. Pendant la pause ainsi générée, l'utilisateur du débogueur a la possibilité de réaliser une inspection ou une modification de l'environnement du programme afin d'en avoir une vision plus claire, de réaliser des tests approfondis et de valider ou invalider une hypothèse. Il pourra lire ou écrire des registres, lire ou écrire de la mémoire, modifier la pile, dumper un module ou une zone mémoire...

[20] toujours depuis la VMCS

Deux grands types de points d'arrêts cohabitent, les points d'arrêt matériels (*Hardware Breakpoint*) et les points d'arrêt logiciels (*Software Breakpoint*). Chacun de ces types de points d'arrêt a ses avantages ainsi que ses inconvénients. Nous en faisons une rapide introduction dans les sections suivantes de cet article.

5.1 *Software Breakpoint*

Le *Software Breakpoint* est le point d'arrêt le plus communément utilisé par les analystes. Il consiste à modifier les instructions du programme à déboguer en y insérant une instruction particulière à l'endroit exact où l'on souhaite mettre l'exécution en pause. Sur x86 et x64, l'instruction est un `0xCC` (`INT3`).

```
0:  59                      pop    rcx
1:  48 c7 c0 05 00 00 0c    mov    rax,0xc000005
8:  c3                      ret

0:  59                      pop    rcx
1:  cc                      int3
X:  <c7 c0 05 00 00 0c>     ?
8:  c3                      ret
```

Listing 3. Exemple d'installation de *breakpoint* logiciel

Lorsque le processeur exécutera l'instruction `0xCC`, une interruption logicielle (`INT3`) sera levée, le flot d'exécution sera détourné vers la routine d'interruption dont l'adresse se trouve dans l'entrée 3 de l'IDT. En mode noyau, cette routine donnera la main au serveur KD qui se chargera de prévenir le débogueur distant qu'un point d'arrêt a été atteint.

Lorsque l'analyste souhaitera reprendre l'exécution, le débogueur distant remettra l'instruction originale, positionnera le 8e bit du registre EFLAGS et relancera l'exécution. Le 8e bit du registre EFLAGS est le *Trap Flag* (TF), il permet au processeur de lever une interruption matérielle (`INT1`) après l'exécution d'une seule instruction.

De ce fait, lorsque l'interruption (`INT1`) est levée, le débogueur distant sera alerté que l'instruction a été exécutée, il pourra ensuite remettre le `0xCC`, retirer le TF et relancer le fil d'exécution.

Pour retirer ce point d'arrêt, WinDbg peut à tout moment replacer l'instruction originale en lieu et place du `0xCC`.

L'avantage de ce type de point d'arrêt est qu'il est disponible en grand nombre (ex. : 32 dans WinDbg), mais il possède l'inconvénient majeur de ne pouvoir être atteint que par une exécution.

5.2 *Hardware Breakpoint*

Le second type de point d'arrêt est le *breakpoint* matériel. Comme son nom l'indique, il repose sur des registres physiques et spécifiques de débogage que l'on peut trouver sur x86 et x64. Sur les processeurs Intel et AMD, 6 *Debug Registers* (DR) existent, nommés de DR0 à DR3 et DR6 à DR7. Les DR sont des ressources privilégiées, ils ne peuvent être définis que si le ring courant est à 0 (noyau). Lorsqu'une application *user land* (Ring-3) souhaite définir un DR, elle doit passer par un appel système particulier (`nt!NtSetContextThread`) au risque de générer une exception de type *General Protection*.

Il y a trois types de DR, les DR d'adresse, les DR de contrôle et les DR de statut.

Les DR d'adresse (DR0, DR1, DR2 et DR3) définissent les adresses qui généreront un point d'arrêt.

Le DR7 est un DR de contrôle, il permet de spécifier les conditions du point d'arrêt. Les conditions peuvent être l'exécution d'une instruction, la lecture ou l'écriture à une adresse définie dans un DR d'adresse. Il existe aussi une autre condition qui est la lecture ou l'écriture sur un port IO.

Finalement, le DR6 est un DR de statuts, qui permet de savoir quel point d'arrêt a été atteint.

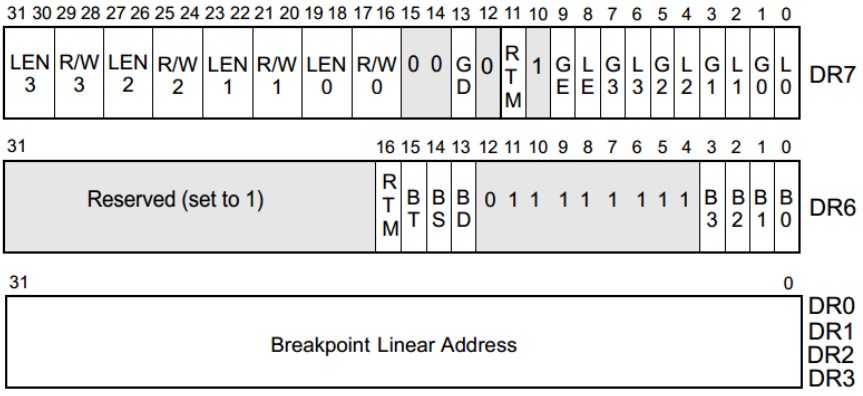

Fig. 6. Debug Registers

Lorsqu'un point d'arrêt de ce type est atteint, le processeur place les bits adéquats dans le DR6 et lève une interruption matérielle (`INT1`). À ce moment sur un système en mode `/DEBUG`, KD averti le débogueur distant que le processeur a atteint un *breakpoint*. En lisant la valeur de

DR6, le débogueur distant peut déduire quel est le ou les points d'arrêt responsables.

Pour reprendre l'exécution, le débogueur distant met à 0 dans DR7 les bits spécifiques aux points d'arrêt atteints, positionne le TF [21], relance l'exécution, attend un autre `INT1`. Quand l'interruption `INT1` est levée, il repositionne les bits dans DR7 pour réactiver le point d'arrêt, retire le TF et enfin, relance l'exécution.

Finalement, pour retirer ce type de *breakpoint*, le débogueur distant met les DR d'adresse et les bits spécifiques à ces DR dans DR7 à 0.

Le grand avantage de ce type de *breakpoint* est qu'il permet de mettre l'exécution en pause lors d'un accès à une adresse spécifique, chose impossible avec les *breakpoints* logiciels. En revanche, les points d'arrêt matériels sont limités au nombre de 4, contrairement aux points d'arrêts logiciels.

6 Difficulté d'utilisation des *breakpoints* conventionnels sans /DEBUG

Dans cette section de l'article, nous présentons rapidement les raisons qui nous empêchent d'utiliser les points d'arrêt conventionnels sur un système démarré sans mode `/DEBUG`.

6.1 Sofware Breakpoint

Une des fonctions essentielles de *Patchguard* est de garantir l'intégrité du code noyau afin de débusquer les trampolines qu'aurait pu installer un *rootkit* ou un antivirus.

Pour cela la protection noyau se charge de calculer régulièrement la somme d'intégrité de certaines fonctions critiques du noyau et de la comparer avec la somme de contrôle calculée à l'initialisation de ce même noyau.

Ainsi, si le *checksum* n'est pas identique à celui d'origine, *PatchGuard* appelle la fonction `nt!KeBugCheck`, générant par la suite un écran bleu (BSOD) avec comme code d'erreur `CRITICAL_STRUCTURE_CORRUPTION` (Figure 7).

Nous avons vu que pour fonctionner, les points d'arrêt logiciels nécessitent la modification du code à analyser en y remplaçant une instruction par un `0xCC`. Ainsi, si *Patchguard* calcule le *checksum* du code pendant qu'un *breakpoint* est installé, le BSOD est garanti.

En mode `/DEBUG`, cette fonctionnalité de *Patchguard* n'est pas activée.

[21] pour réaliser un *Single Step*

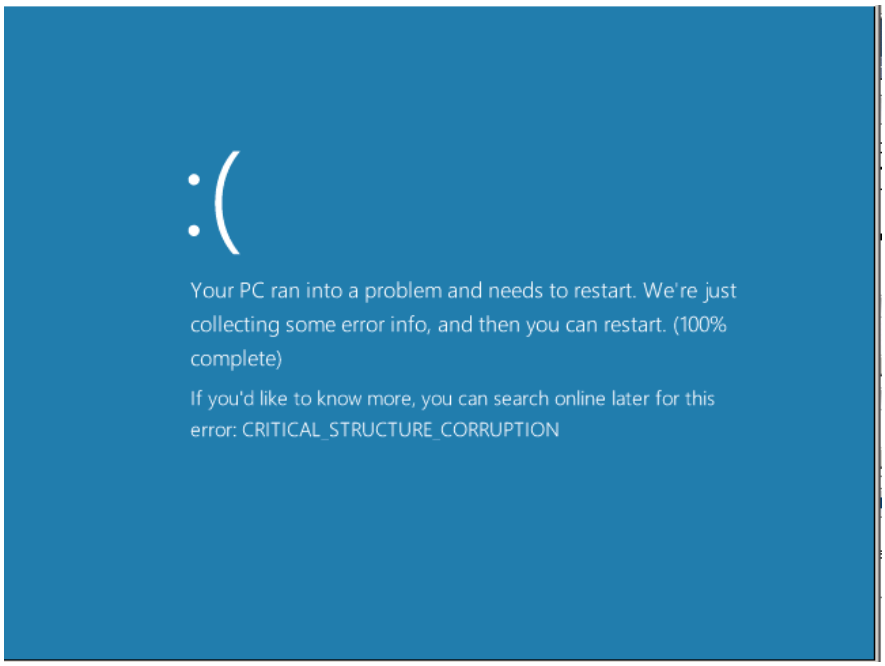

Fig. 7. BSOD suite au passage de PatchGuard

6.2 Hardware Breakpoint

Lorsque Windows est démarré normalement, les routines d'interruption `INT1` et `INT3` ne sont pas utilisables en espace noyau. Si un *Hardware Breakpoint* ou un *Software Breakpoint* est atteint, les interruptions `INT1` et `INT3` provoqueront inévitablement un BSOD avec pour code d'erreur `KMODE_EXCEPTION_NOT_HANDLED`.

Une solution possible aurait pu être de modifier les entrées correspondantes dans l'IDT afin intercepter ces interruptions, mais *Patchguard* la supervise elle aussi et cette technique n'est absolument pas furtive.

7 *Breakpoints* développés

Dans cette sous-partie sont présentées les solutions techniques qui ont été développées afin de faire fonctionner des *breakpoint* même si *Patchguard* est activé et que le système est démarré sans `/DEBUG`. Ces solutions reposent essentiellement sur les capacités de virtualisation matérielle offertes par les processeurs contemporains.

7.1 *PageHyperBreakpoint*

Le premier type de point d'arrêt présenté fonctionne au niveau des droits sur les EPT. Il permet de réaliser un *breakpoint* assez souple d'utilisation, mais possédant un *overhead* relativement élevé.

Chaque *Guest Physical Page* (GPP) peut être traduite en une ou plusieurs *Host Physical Page* (HPP) via l'EPT, en fonction des tailles de page :

- 1 GPP de 2M donnera 1 HPP de 2M ou 512 HPP de 4K
- 1 GPP de 4K donnera 1 HPP de 4K.

L'*EPT Page Table Entry* (EPTPTE) possède trois bits (R, W, X) qui définissent les droits accès d'une GPA sur une HPA.

```
typedef struct EPTPTEBITS {
    /** Present/Readable bit. */
    uint64_t    u1Present       : 1;
    /** Writable bit. */
    uint64_t    u1Write         : 1;
    /** Executable bit. */
    uint64_t    u1Execute       : 1;
    /** Available for software. */
    uint64_t    u9Available     : 9;
    /** HPA of page.*/
    uint64_t    u40PhysAddr     : 40;
    /** Available for software. */
    uint64_t    u12Available    : 12;
} EPTPTEBITS;
```

Listing 4. Structure représentant une EPTPTE

Les bits sont définis comme suit :

- R, si la page est lisible
- W, si la page est inscriptible
- X, si la page est exécutable

Pendant que l'invité s'exécute et s'il souhaite réaliser un accès à une GPA, le processeur va vérifier l'existence d'une entrée dans le *Translation Lookaside Buffer* (TLB) concernant cette GPA. Si l'entrée n'existe pas, la MMU va réaliser la traduction et le TLB sera rempli avec la clé (GPA, vPID), la HPA et les droits d'accès de la GPA sur cette HPA. Cette opération n'est pas réalisée si l'entrée est déjà dans le TLB. Ensuite, le type d'accès souhaité est confronté aux bits de droits trouvés dans le TLB.

Si les bits de droits autorisent l'accès courant, aucune exception ne sera levée, l'invité continuera à s'exécuter sur le processeur et pourra terminer l'accès en cours sur la GPA.

En revanche, si les bits de droits interdisent l'action demandée sur la mémoire (ex. : exécution d'une page non exécutable), un VMExit se produira et un handler d'exception (#EPTViolation) sera exécuté en tant qu'hôte. Cet handler d'exception (*Nested Page Fault Handler*) peut redéfinir les droits sur la GPA et relancer l'exécution de l'invité. À l'aide de ce mécanisme, il est possible de créer un type de *breakpoint* en retirant certains droits sur les GPA souhaitées.

Prenons l'exemple d'un *breakpoint* en exécution sur `nt!NtWriteFile`.

Tout d'abord la première étape consiste à récupérer la GPA de la fonction. Pour réaliser cette première traduction, nous pouvons parcourir les *Page Tables* de l'invité, à partir de son CR3, pour récupérer la GPA de `nt!NtWriteFile`. Puis, à partir de cette GPA et en parcourant l'EPT, il est possible de récupérer la HPA, mais surtout l'adresse physique hôte de l'EPTPTE qui définit les droits de la GPA de `nt!NtWriteFile` sur sa HPA.

Une fois l'adresse de l'EPTPTE acquise, il est possible de mettre à 0 le bit X de cette entrée, rendant l'ensemble de la page physique de l'invité non exécutable. L'action suivante sera de retirer du TLB l'entrée concernant la GPA, le TLB ayant conservé les anciens droits de la page lorsqu'elle était encore exécutable. L'instruction `INVEPT` depuis l'hôte permet de réaliser cette opération.

À partir d'ici, toute exécution sur la GPA de `nt!NtWriteFile` provoquera une #EPTViolation et donc l'exécution en mode hôte du *Nested Page Fault Handler*. Il sera ainsi possible de bloquer l'exécution au *breakpoint*.

Un problème se pose : pour le moment le *breakpoint* ne peut avoir qu'une seule taille, celle d'une page. Pour différencier l'exécution de `nt!NtWriteFile` par rapport à l'exécution de `nt!NtWriteFile+0x10` (qui se trouve sur la même page), le *handler* d'exception à la possibilité de lire dans la *VM-Exit Information Field* de la VMCS l'adresse (*Faulting Address*) qui a généré l'exception, ainsi que le type d'accès qui y a été réalisé.

Avec ces deux champs de la VMCS, il est concevable, en faisant un tri, d'avoir sur la même page des *breakpoints* de type et taille différents.

Dans le *Nested Page Fault Handler*, l'exécution est reprise en réalisant cette suite d'actions :

1. restauration des droits d'origine de la GPA,
2. invalidation de la GPA (`INVEPT`),
3. activation du MTF,
4. relancement de l'exécution (1 seule instruction),

5. restauration des droits altérés de la GPA,
6. désactivation du MTF,
7. invalidation de la GPA avec `INVEPT`,
8. relancement de l'exécution.

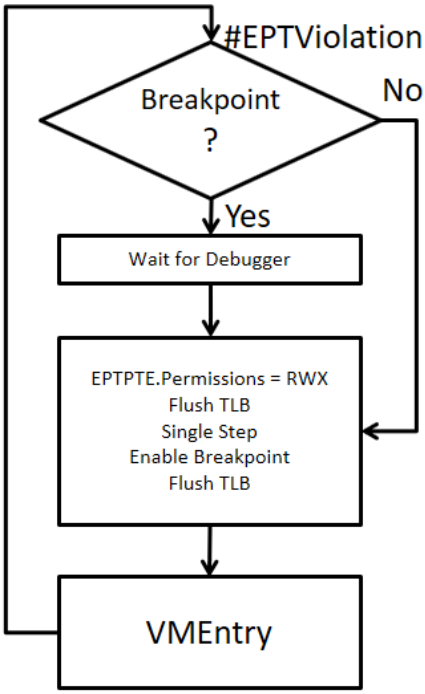

Fig. 8. Control flow d'un PageHyperBreakpoint

L'enchainement d'actions, précédemment décrit, est relativement coûteux en temps processeur, ce qui en fait un *breakpoint* à n'utiliser que dans certaines conditions (ex. : page peu accédée...). En revanche, les possibilités de ce type de point d'arrêt sont très intéressantes dans le sens où il permet de faire des points d'arrêt sur des zones mémoire complètes là où les points d'arrêt matériels sont limités au maximum à 8 octets. Par exemple, cela peut s'avérer très utile pour faire de l'*unpacking* ou positionner un point d'arrêt lors d'accès à de la mémoire IO.

7.2 *SoftHyperBreakpoint*

Comme nous l'avons expliqué dans la section précédente, le TLB agit comme un cache pour les traductions d'adresses. Les adresses relatives aux instructions sont mises en cache dans le I-TLB et les adresses relatives aux données le sont dans le D-TLB. Ce détail d'implémentation est important,

car il autorise une désynchronisation des deux caches, cela ayant été démontré par *Shadow Walker* [13].

En revanche, depuis la génération Intel Nehalem (~2010), quand les deux caches partagent des entrées communes, ces entrées sont regroupées dans un cache commun, le S-TLB, empêchant les désynchronisations.

Lors de Black-Hat 2014 une preuve de concept de *rootkit* utilisant une pseudo-désynchronisation a été présentée, *More Shadow Walker* [14]. Pour simuler cette désynchronisation, *More Shadow Walker* exploite les exceptions de type #EPTViolation pour diriger les accès aux données sur une page physique hôte et les accès aux instructions sur une deuxième page physique hôte.

En utilisant ce mécanisme, il est possible en fonction du type d'accès (lecture ou exécution) que le contenu d'une même GPA ait deux valeurs distinctes, permettant de dissimuler à un driver, le code réellement exécuté.

Pour mettre en place un point d'arrêt sur une page physique de l'invité (GPA), nous récupérons la page physique hôte P1 (HPA) associée à l'aide de l'EPT. Puis nous allouons une deuxième page physique hôte P2 (HPA) et nous y copions les données de la page originale P1. Dans cette nouvelle page, nous plaçons ensuite une instruction `0xF4` (`HLT`) à l'endroit où nous souhaitons un point d'arrêt. Une fois la page dupliquée, nous faisons pointer l'EPTPTE de la GPA ciblée vers P2 avec le seul flag X (exécutable) à 1, invalidons la GPA à l'aide de `INVEPT` et relançons l'exécution. La GPA est maintenant devenue seulement exécutable. Tant que l'invité ne fait que des exécutions sur cette page, aucune #EPTViolation ne sera levée.

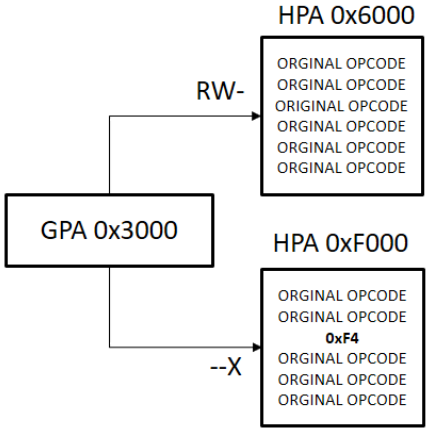

Fig. 9. Une GPA possédant deux HPA

Nous avons décidé d'utiliser l'instruction `HLT` car comme le `INT3`, sa taille est de 1 octet facilitant son installation ou son retrait. De plus, cette instruction est l'une des seules à générer depuis l'invité une exception dans les deux modes d'exécution d'un processeur (noyau ou utilisateur).

Si le processeur exécute une instruction `HLT` pendant qu'il est en mode invité, deux cas peuvent se présenter ; soit l'invité est en mode noyau et une exception #HLTException est levée, soit l'invité est en mode utilisateur et une exception #GPException est levée. Tout ceci permet d'avoir un handler d'exception exécuté en mode hyperviseur avec la machine virtuelle stoppée à l'endroit choisi.

Le déroulement de reprise de l'exécution est le suivant :

1. Réinstallation de l'opcode original.
2. Activation du MTF.
3. *Single-Step*.
4. Désactivation du MTF.
5. Réinstallation du `0xF4`.

Si un driver, s'exécutant dans la machine virtuelle, calcule régulièrement la somme d'intégrité de son code et si une partie de son code se trouve dans une page modifiée par notre point d'arrêt, l'accès provoquera une #EPTViolation, car cette page n'est plus accessible en lecture. Nous pouvons dans l'*EPTViolation Handler* :

1. faire pointer l'EPTPTE de la GPA vers la HPA originale P1 avec le droit de lecture seule,
2. invalider la GPA avec `INVEPT`,
3. reprendre l'exécution.

Les données lues par le driver seront les instructions originales et ce pilote ne pourra pas réaliser, par une simple lecture, qu'un *breakpoint* est positionné dans son code. Quand le driver exécutera de nouveau son code, dans lequel se situe le point d'arrêt, une nouvelle #EPTViolation sera levée car la GPA est à ce moment en lecture seule. Dans ce nouvel appel à l'*EPTViolation Handler*, nous pouvons :

1. faire pointer la GPA vers la HPA P2
2. mettre la GPA en exècution seulement
3. invalider la GPA
4. relancer l'exècution de l'invité.

L'inconvénient majeur de cette technique est le changement d'une HPA à l'autre lors du changement du type d'accès sur la GPA. Dans la majorité

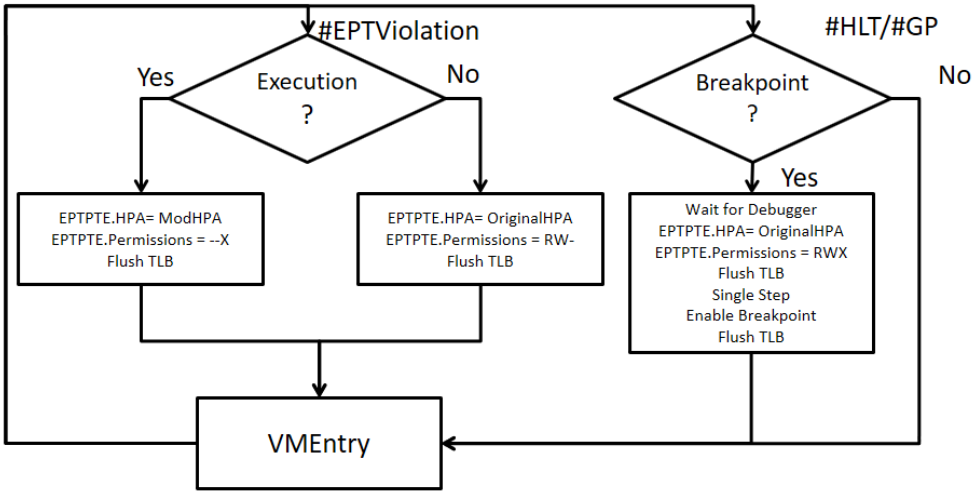

Fig. 10. Control flow d'un SoftHyperBreakpoint

des cas, les pages sont soit de type *"données"*, soit de type *"instructions"* réduisant au maximum le changement de type d'accès. Néanmoins, les programmes qui vérifieraient très souvent les instructions qu'ils sont sur le point d'exécuter pourraient faire baisser de manière significative les performances générales du système, mais ce cas semble très marginal.

Dans certains cas, les pages peuvent contenir des données en écriture et du code. Pour gérer ce cas, lors d'accès en écriture, la page originale est la page de destination. A la fin de chaque écriture, les données modifiées par l'écriture de la page originale sont copiées dans la seconde page et le `0xF4` est réinstallé à la fin de la copie.

7.3 *HardHyperBreakpoint*

Les *HardHyperBreakpoint* sont les équivalents des breakpoints matériels standards tout en étant furtifs et difficilement détectables par un driver malveillant ou *PatchGuard*. Ils utilisent eux aussi les *Debug Registers* de l'invité.

Si une interruption `INT1` est levée pendant que l'invité s'exécute quand le bit 1 (#DB) de son *Primary Processor-Based VM-Execution Control* est défini à 1, alors un VMExit se produit, donnant la main à l'hyperviseur pour exècuter le *#DBException Handler*.

Depuis le *#DBException Handler*, il est possible de savoir si un *Debug Register* est la cause de cette exception en regardant le champ `EXIT_QUALIFICATION` de la VMCS. Si un des DR est la cause de l'exception, nous alertons WinDbg par KD qu'un point d'arrêt a été atteint. Pour rependre l'exécution, il faut :

- désactiver dans la VMCS l'interruption `INT1` en attente pour éviter le BSOD de l'invité,
- désactiver dans DR7 les *Debug Registers* responsables,
- single-step avec le MTF,
- réactiver les *Debug Registers* précédemment désactivés,
- relancer l'exécution.

Si les *Debug Registers* ne sont pas responsables de l'interruption, nous laissons l'interruption en attente pour qu'elle soit propagée au niveau de l'invité.

Nous avons vu que nous utilisons les *Debug Registers* de l'invité pour poser nos propres points d'arrêt, une question pourrait rapidement venir en tête : comment l'invité ne peut-il pas détecter l'utilisation de ses *Debug Registers* ? En effet, sur un système non débogué, les DR0, DR1, DR2 et DR3 sont habituellement à 0. Toutes valeurs différentes peuvent être la preuve que quelque chose d'inhabituel se produit. De plus pendant son exécution, *PatchGuard* va régulièrement mettre DR7 à 0 pour s'assurer qu'aucun point d'arrêt ne se lève lors de sa vérification de l'IDT [12]. Finalement lors du passage entre mode utilisateur et mode noyau, les DR sont sauvegardés puis restaurés, par `nt!KiSaveDebugRegisterState` et `nt!KiRestoreDebugRegisterState`.

Pour répondre à ce problème, nous utilisons les exceptions de type #MovDR qui sont levées quand l'invité réalise une action de lecture ou d'écriture sur un des DR. Ces exceptions se nomment ainsi, car le noyau (seulement) ne peut accéder aux *Debug Registers* qu'à l'aide de l'instruction `MOV`. Lors d'une lecture ou d'une écriture d'un des registres de débogage par l'invité, notre *#MovDR Exception Handler* est appelé et se charge :

1. de sauvegarder les valeurs des DR invisibles pour l'invité,
2. de restaurer les valeurs des DR visibles pour l'invité,
3. de désactiver la levée des #MovDR,
4. d'exécuter (à l'aide du MTF) l'instruction `MOV` ayant généré le #MovDR,
5. de réactiver la levée des #MovDR,
6. de sauvegarder les valeurs des DR visibles pour l'invité et
7. de restaurer les valeurs des DR invisibles pour l'invité.

Cette manipulation permet de tromper un code privilégié s'exécutant dans l'invité sur les valeurs réelles des registres de débogage et donc de lui dissimuler le positionnement d'un point d'arrêt.

Finalement, pour faire fonctionner les *breakpoints* matériels à l'intérieur de la machine virtuelle, nous analysons les valeurs écrites pendant le

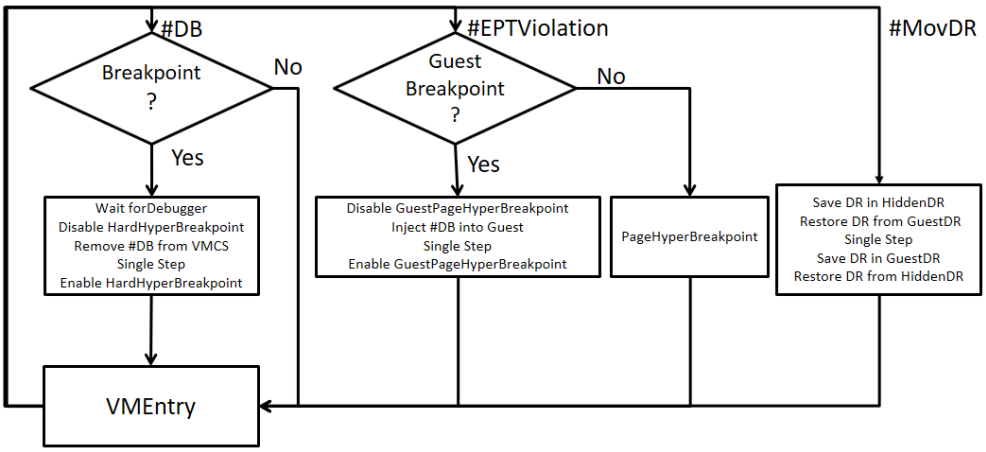

Fig. 11. Control flow d'un HardHyperBreakpoint

#MovDR et nous remplaçons le *Hardware Breakpoint* invité par un *PageHyperBreakpoint*. Lorsqu'un *PageHyperBreakpoint* est atteint alors qu'il remplace un *Hardware Breakpoint* invité, nous injectons une interruption `INT1` dans la VMCS et nous reprenons le fil d'exécution, permettant ainsi de faire fonctionner un débogueur à l'intérieur de la machine virtuelle.

8 Exemples, tests & performances

Dans cette partie de l'article, nous présentons les résultats de différents tests que nous avons conduits avec Winbagility ainsi qu'une partie du code des tests. Les tests on été réalisés à l'aide de l'API FDP et sans WinDbg.

```
CPU: i5 4200M
Memoire: 8Go DDR3-1333
HDD: 500Go 4200 tours/min
Hyperviseur: VirtualBox 5.0.2 (Modifie)
OS: Windows 10 TH2
```

Listing 5. Configuration de la machine hôte

```
CPU: 1
Memoire: 2Go
Disque: 50Go
OS: Windows 10 TH2
```

Listing 6. Configuration de la machine virtuelle

8.1 Pas à pas

```
    FDP_Pause(pFDP);
    printTime();
    for(uint32_t SingleStepCount = 0; SingleStepCount < 1000000;
        SingleStepCount++){
        FDP_SingleStep(pFDP, 0);
    }
    printTime();
    FDP_Resume(pFDP);
```

Listing 7. Code utilisé

Résultat : 105000 Instructions/sec

8.2 Lecture de mémoire

```
printTime();
    uint32_t ReadMemoryCount = 0;
    char Buffer[4096];
    while (ReadMemoryCount < 10000000){
        if(bIsPhysicalTest == true)
            FDP_ReadPhysicalMemory(pFDP, Buffer, 4096,
                physicalAddress);
        else
            FDP_ReadVirtualMemory(pFDP, 0, Buffer, 4096,
                virtualAddress);
        ReadMemoryCount++;
    }
    printTime();
```

Listing 8. Code utilisé

Résultats :

Type de lecture	Lectures de 4K/s	Go/s
FDP_ReadVirtualMemory	625000	2,38
FDP_ReadPhysicalMemory	714000	2,72

8.3 Parcours de liste de processus

Pour cet essai, nous avons utilisé un code proche de celui qui se trouve en annexe de cet article. Il consiste à calculer le nombre de parcours complet de la liste des processus d'un système qui peut être réalisé par seconde. Pendant ce test, le système analysé n'a pas été mis en pause et l'essai a été réalisé après l'installation du système.

Résultats :

Parcours/sec	Temps par parcours(ms)
2439	0,41

8.4 HyperBreakpoints

Pour ces tests, nous avons développé une petite application qui réalise en boucle l'incrémentation d'un entier et nous y avons positionné un *HyperBreakpoint*. Nous avons mesuré le temps pour que 100000 *breakpoints* soient atteints. L'opération a été réalisée 3 fois et nous avons fait la moyenne des 3 résultats. Le code pour positionner les différents types de *breakpoints* est disponible dans les sources du projet dans le fichier « testFDP.cpp ».

Résultats :

Type de breakpoint	Breakpoint/sec	Temps (s)
HardHyperBreakpoint	11235	8.9
SoftHyperBreakpoint	7812	12.8
PageHyperBreakpoint	5347	18.7

8.5 Analyse d'appels systèmes

Pour ce test, nous avons installé un point d'arrêt sur `nt!NtWriteFile` et nous avons mesuré le temps pour réaliser une compression avec 7z. En moyenne, 789 *breakpoints* ont été atteints pendant les tests. Le dossier à compresser contenait 50 fichiers pour un total de 25 Mo.

Résultats :

Type de breakpoint	Temps (s)
-	11,7
HardHyperBreakpoint	11.8
SoftHyperBreakpoint	12,3
PageHyperBreakpoint	14,4

8.6 WinDbg

Le dernier test que nous avons réalisé permet de comparer les performances de WinDbg avec deux canaux. Pour réaliser cet essai, nous avons exécuté une machine virtuelle en mode `/DEBUG` dans VirtualBox non modifié et une machine virtuelle sans mode `/DEBUG` dans notre version modifiée de VirtualBox. Le premier canal choisi est KDNET, l'un des stub WinDbg les plus performants, et nous l'avons connecté à la machine en mode `/DEBUG`. L'autre canal est celui que nous avons développé avec FDP et que nous avons connecté sur la machine virtuelle démarrée normalement.

Nous avons exécuter le script WinDbg suivant :

```
.echotime;
.for(r @$t0 = 0; @$t0 < 1000; r @$t0=@$t0+1)
{
    t;
    r @$t1 = @rax;
    .for(r @$t2 = 0; @$t2 < 100; r @$t2=@$t2+1)
    {
        r @$t3 = poi(@rsp+@$t2);
    }
}
.echotime;
```

Listing 9. Code utilisé

Résultats :

Canal	Temps (s)
KDNET	123
Winbagility	70

Les résultats des tests que nous avons réalisés sont cohérents avec le choix de l'architecture que nous avons fait. Les performances de FDP lui confèrent la possibilité d'être utilisé pour faire de l'introspection furtive et temps réel de machine virtuelle. Nous laissons au lecteur imaginer ces propres scénarios de détection basés sur FDP. Les résultats des tests précédents sont donnés à titre d'indicatif, car nous n'avons pas réussi à faire fonctionner LibVMI et Xen. D'après plusieurs tests disponibles [9–11] FDP semble plus performant que LibVMI. Enfin Winbagility offre des performances supérieures à KDNET, la limite des performances semble être due à WinDbg.

9 Bonus

9.1 !save & !restore

Lors d'analyse de driver, les occasions de se tromper sont nombreuses (choix de branches, suite d'actions...) et ont souvent un impact fort sur les systèmes (BSOD, inconsistance de structure...). Pour réaliser des tests plus rapidement, nous avons développé des extensions pour WinDbg pour réaliser des sauvegardes et des restaurations de contextes. Ces extensions utilisent FDP pour communiquer avec notre version modifiée de VirtualBox. L'avantage des sauvegardes intégrées à WinDbg est qu'il n'est pas nécessaire de reconnecter le débogueur au noyau après la restauration, comme cela peut être fait lors d'une analyse avec KDNET et VMWare.

L'utilisation de `!save` lors d'un *breakpoint* permet à la machine virtuelle être sauvegardée pour une restauration ultérieure. Depuis cette sauvegarde, il est envisageable de tester plusieurs valeurs pour un registre sans devoir relancer la machine. Ainsi, si une valeur particulière engendre un BSOD, `!restore` permet de retourner à l'état initial sans relancer la machine, le driver ou WinDbg.

9.2 !hba

`!hba` est une extension que nous avons développé pour WinDbg permettant d'utiliser la capacité des *PageHyperBreakpoint*. Elle permet de définir dans WinDbg des points d'arrêt lors d'accès à des zones mémoire GVA ou GPA, et ce sans limite de taille, les points d'arrêt matériels conventionnels étant limités au maximum à 4x8 octets.

Un exemple concret pourrait être le souhait de stopper la machine lorsque n'importe quel processus accède en lecture à la structure `USER_SHARED_DATA`. Cette structure a une adresse virtuelle fixe (0x7ffe0000) et une page physique identique (imaginons 0xdead0000) pour tous les processus du système. La commande `!hba p r 0xdead0000 4096` permet de rendre la page sensible aux lectures et de mettre la VM en pause lors de l'accès à `USER_SHARED_DATA`.

Cette extension peut aussi permettre un *breakpoint* lorsqu'un processus particulier accède en écriture à son PEB, `!hba v w <PEB Address> <PEB Size>` ou tente d'exécuter du code situé dans sa pile, `!hba v e <Stack Address> <Stack Size>`.

```
!hba <Address Type> <Access type> <Address> <Breakpoint Length>
<Address Type>
    v for virtual address
    p for physical address
<Access Type>
    r to break on read access
    w to break on write access
    e to break on execute access
<Address>
    Virtual/Physical address to break on
<Breakpoint Length>
    Length of the breakpoint
```

Listing 10. Aide de !hba

10 Conclusion

Dans cet article, nous avons présenté Winbagility, un outil de débogage furtif de noyau Windows. Winbagility peut difficilement être détecté ou

contourné depuis la machine virtuelle analysée ce qui le rend intéressant pour l'étude de drivers malveillants. Nous avons discuté de l'implémentation de différents *breakpoints* dans l'outil démontrant leur performance et leur furtivité.

Winbagility permet actuellement l'analyse de système Windows x64, mais l'ensemble des techniques utilisés par ses *breakpoints* peut aussi l'être pour déboguer un noyau Linux. Beaucoup de travail reste à faire pour le rendre fiable avec plusieurs processeurs, pour améliorer les performances, pour ajouter le support du 32bits...

Le développement d'un convertisseur FDP vers GDB permettrait rapidement l'utilisation de la partie VirtualBox (points d'arrêt furtifs, save et restore) pour réaliser un débogage avec d'autres débogueurs.

Un proxy FDP vers VID pourrait quant à lui permettre d'utiliser Hyper-V à la place de VirtualBox, mais il ne pourrait pas autoriser l'utilisation de point d'arrêt furtif, car VID ne possède pas (à notre connaissance) de fonction de modification des droits sur les SLAT.

Enfin ARM offre lui aussi des instructions de virtualisation et un mécanisme équivalant à la SLAT qui pourraient être utilisés dans le but de créer un outil similaire qui autoriserait de débogage furtif d'Android.

L'idée d'utiliser les extensions de virtualisation à des fins de débogage n'est pas nouvelle, mais très peu de projets se sont réellement attardés sur Windows, KD et WinDbg pour être utilisables sur des cas concrets et réels d'analyses de driver.

Références

1. Eugene Rodionov Aleksandr Matrosov. Defeating x64 : The evolution of the tdl rootkit. *Confidence*, 2011.
2. Jason Sean Alexander. A counter-intelligence method for spying while hiding in (or from) the kernel with apcs. 2012.
3. Cuckoo. Cuckoo sandbox - open source automated malware analysis. 2013.
4. Christophe Devine Damien Aumaitre. Virtdbg. *SSTIC*, 2010.
5. tecamac@gmail.com deresz@gmail.com. Uroburos : the snake rootkit. 2014.
6. Stephane Duverger. ubervisor. *SSTIC*, 2011.
7. Peter Ferrie. The "ultimate"anti-debugging reference. 2011.
8. Intel. Tlbs, paging-structure caches, and their invalidation. 2008.
9. Kousuke Nakamura Kenichi Kourai. Efficient vm introspection in kvm and performance comparison with xen. 2014.
10. Bryan D. Payne. Simplifying virtual machine introspection using libvmi. 2012.
11. Bryan D. Payne. An introduction to virtual machine introspection using libvmi. 2014.
12. skywing@valhallalegends.com. Patchguard reloaded : A brief analysis of patchguard version 3. 2007.

13. Sherri Sparks. Shadow walker : Raising the bar for rootkit detection. *Black Hat*, 2005.
14. Jacob Torrey. More shadow walker : Tlb-splitting on modern x86. *Black Hat*, 2014.

A Annexes

A.1 Lexique

- DMA : Direct Memory Access, Mécanisme d'accès direct à la mémoire sans passer par le processeur.
- SLAT : Second Level Address Translation, Mécanisme de translation entre GPA et HPA.
- SSDT : System Service Dispatch Table, Table de dipatch des appels systèmes.
- IDT : Interrupt Descriptor Table, Table contenant les pointeurs vers les routines d'interruption.
- GDT : Global Descriptor Table, Table qui décrit les segments de mémoire.
- MMU : Memory Management Unit, Composant responsable de l'accès à la mémoire par le processeur.
- PML4 : Page Map Level 4, Première table utilisée lors de la translation d'adresses virtuelles.
- PTE : Page Table Entry, Entrée dans la PT qui définit les droits d'une adresse virtuelle.
- PDB : Program Database, Fichier contenant les informations de debug d'un programme.
- MSR : Model Specific Register, Registre qui définit une partie de la configuration du CPU.
- LSTAR : Long System Target-Address Register, MSR qui définit l'adresse de la fonction noyau appelée lors d'un syscall.
- TLB : Translation Lookaside Buffer, Mémoire cache contenant les résultats des dernière translation d'adresses.
- vPID : Virtual PID, Identifiant d'une VM dans le TLB.
- VID : Virtualization Infrastructure Driver, API de communication entre le client et le driver Hyper-V.

A.2 Exemple d'utilisation de FDP

```
    printf("Saving safe state...");
    FDP_Pause(pFDP);
    FDP_Save(pFDP);
    FDP_Resume(pFDP);
```

```
    printf("DONE !\n");

    while (bRunning == true){
        uint64_t PsActiveProcessHead = GetPsActiveProcessHead();
        uint64_t FirstProcess = PsActiveProcessHead -
            EPROCESS_ACTIVEPROCESSLIST_OFF;
        while (true){
            if (FDP_ReadVirtualMemory(
                    pFDP,
                    0,
                    &FirstProcess,
                    8,
                    FirstProcess + EPROCESS_ACTIVEPROCESSLIST_OFF
                    ) == false){
                break;
            }
            FirstProcess = FirstProcess -
                EPROCESS_ACTIVEPROCESSLIST_OFF;

            char ProcessName[EPROCESS_PROCESSNAME_SIZE];
            if (FDP_ReadVirtualMemory(
                    pFDP,
                    0,
                    ProcessName,
                    EPROCESS_PROCESSNAME_SIZE,
                    FirstProcess + EPROCESS_PROCESSNAME_OFF) ==
                        false){
                break;
            }

            if (strcmp(ProcessName, "calc.exe") == 0){
                printf("Restoring safe state...");
                FDP_Pause(pFDP);
                FDP_Restore(pFDP);
                FDP_Resume(pFDP);
                printf("DONE !\n");
            }

            if (ProcessName[0] == 0){
                break;
            }
        }
        Sleep(10);
    }
```

Listing 11. Liste de processus

A.3 Exemple de message KD

```
typedef struct _DBGKD_READ_MEMORY64 {
    UINT64 TargetBaseAddress;
    UINT32 TransferCount;
    UINT32 ActualBytesRead;
    UINT64 Unknown1;
    UINT64 Unknown2;
    UINT64 Unknown3;
```

```
    UINT8 Data[0];
} DBGKD_READ_MEMORY64, *PDBGKD_READ_MEMORY64;

typedef struct _DBGKD_MANIPULATE_STATE64 {
    UINT32 ApiNumber;
    UINT16 ProcessorLevel;
    UINT16 Processor;
    NTSTATUS ReturnStatus;
    union{
        DBGKD_READ_MEMORY64 ReadMemory;
        DBGKD_QUERY_MEMORY QueryMemory;
        DBGKD_WRITE_MEMORY64 WriteMemory;
        DBGKD_GET_CONTEXT GetContext;
        DBGKD_SET_CONTEXT SetContext;
        DBGKD_RESTORE_BREAKPOINT RestoreBreakPoint;
        DBGKD_CONTINUE2 Continue2;
        DBGKD_GET_VERSION64 GetVersion64;
        DBGKD_BREAKPOINTEX BreakPointEx;
        DBGKD_READ_WRITE_MSR ReadWriteMsr;
        //...
    };
} DBGKD_MANIPULATE_STATE64, *PDBGKD_MANIPULATE_STATE64;

typedef struct _KD_PACKET_T{
    UINT32 Leader;
    UINT16 Typer;
    UINT16 Length;
    UINT32 Id;
    UINT32 Checksum;
    union{
        DBGKD_MANIPULATE_STATE64 ManipulateState64;
        DBGKD_WAIT_STATE_CHANGE64 StateChange;
        //...
    };
}KD_PACKET_T, *PKD_PACKET_T;
```

Listing 12. Structures KD

A.4 Exemples de fonctions de l'API FDP

```
    FDP_SHM*    FDP_OpenSHM(const char *pShmName);
    bool        FDP_Init(FDP_SHM *pShm);
    bool        FDP_Pause(FDP_SHM *pShm);
    bool        FDP_Resume(FDP_SHM *pShm);
    bool        FDP_WritePhysicalMemory(
                    FDP_SHM *pShm,
                    const void *pSrcBuffer,
                    uint32_t WriteSize,
                    uint64_t PhysicalAddress);
    bool        FDP_ReadVirtualMemory(
                    FDP_SHM *pShm,
                    uint32_t CpuId,
                    void *pDstBuffer,
                    uint32_t ReadSize,
                    uint64_t VirtualAddress);
    bool        FDP_ReadRegister(
```

```
                    FDP_SHM *pShm,
                    uint32_t CpuId,
                    FDP_Register RegisterId,
                    uint64_t *pRegisterValue);
int         FDP_SetBreakpoint(
                    FDP_SHM *pShm,
                    uint32_t CpuId,
                    FDP_BreakpointType BreakpointType,
                    uint8_t BreakpointId,
                    FDP_Access BreakpointAccessType,
                    FDP_AddressType BreakpointAddressType,
                    uint64_t BreakpointAddress,
                    uint64_t BreakpointLength);
bool        FDP_UnsetBreakpoint(
                    FDP_SHM *pShm,
                    uint8_t BreakpointId);
bool        FDP_SingleStep(
                    FDP_SHM *pShm,
                    uint32_t CpuId);
bool        FDP_GetPhysicalMemorySize(
                    FDP_SHM *pShm,
                    uint64_t *pPhysicalMemorySize);
bool        FDP_Reboot(FDP_SHM *pShm);
bool        FDP_Restore(FDP_SHM *pShm);
bool        FDP_InjectInterrupt(
                    FDP_SHM *pShm,
                    uint32_t CpuId
                    uint32_t InterruptCode,
                    uint32_t ErrorCode,
                    uint64_t Cr2);
```

Listing 13. Sous-partie de FDP