Gunpack : un outil générique d'unpacking de malwares

Julien Lenoir
julien.lenoir@airbus.com

Airbus Group Innovation

Résumé. L'outil Gunpack n'est pas à proprement parler un unpacker mais plutôt un outil permettant de développer facilement des scripts d'unpacking en Python. Après une présentation détaillée des mécanismes internes de Windows et de l'implémentation de l'outil nous illustrerons sur des exemples concrets les possiblités offertes par Gunpack.

1 Introduction

1.1 C'est quoi un packer ?

De nombreux programmes, malveillants pour la plupart, utilisent des techniques dites de *packing*. Celles-ci consistent à compresser et/ou chiffrer tout ou partie du code original d'un logiciel et à décompresser et/ou déchiffrer ce code au démarrage du programme.

Cette technique est parfois utilisée à des fins légitimes pour réduire la taille d'un binaire en vue d'optimiser sa distribution par exemple. Le compresseur le plus connu et le plus utilisé est sans aucun doute UPX [7].

Cette technique est aussi utilisée à des fins malveillantes. En effet, en compressant et/ou en chiffrant la partie malveillante d'un programme, on dissimule alors sa charge virale. Le but est de rendre plus complexe sa détection, par un anti-virus notamment. Lorsqu'on souhaite étuder ce code viral il est nécessaire de disposer d'un outil permettant d'extraire ce code : un *unpacker*.

Nous distinguons ici les packers des virtualiseurs :

- *Packers* : le code du programme original est décompressé/déchiffré au démarrage puis executé
- *Virtualiseurs* : le code du programme original se présente, potentiellement partiellement, sous forme de *bytecode* exécuté par une machine virutelle.

L'outil présenté ici a pour but d'aider à l'analyse et à l'unpacking des packers et non des virtualiseurs. Il existe de nombreux packers, qu'ils soient COTS ou faits maison, chacun se présentant sous plusieurs variantes. Développer un unpacker ad-hoc pour chacun d'entre eux est une tâche titanesque, c'est pourquoi des initiatives visant à développer des outils automatiques d'*unpacking* ont vu le jour.

1.2 État de l'art

La démarche visant à développer des outils d'unpacking génériques et automatiques n'est pas nouvelle. De nombreux articles décrivant de tels outils ont été publiés depuis plus de 10 ans. Nous pouvons citer MutantX-S [8], Justin [2], Ether [6], Omniunpack [5], Renovo [11] ou encore Packer Attacker [1]. Leurs choix d'implémentation sont variés : depuis un hyperviseur, le noyau de l'OS ou alors en userland. Mis à part Packer Attacker [1], ils sont souvent anciens et rarement open-source.

Bien que cette problématique soit ancienne, elle est toujours d'actualité, notamment lorsqu'on souhaite faire de la classification de malware à grande échelle. C'est d'ailleurs dans ce cadre que le projet Gunpack a été initié.

Nous avons initialement tenté de réimplémenter le moteur d'unpacking décrit dans le papier MutantX-S [8] dont le code n'est pas public. Nous nous sommes alors rendu compte que l'architecture choisie n'était pas optimale, aussi nous avons décidé de l'améliorer, de la généraliser et de la publier. C'était l'origine du projet Gunpack. Finalement, nous avons décidé de faire évoluer l'outil pour qu'il ne soit plus un simple unpacker générique mais un outil facilitant de développement de scripts d'unpacking.

1.3 Choix de conception

L'outil Gunpack a été conçu avec les contraintes suivantes :

- Compatibilité : être utilisable sur une machine physique comme dans un environnement virtualisé, quel qu'il soit.
- Furtivité : être relativement furtif vis à vis du packer analysé.

L'outil Gunpack fonctionne à l'intérieur de l'OS cible,aujourd'hui Windows 7 32 bits PAE et à deux niveaux :

- Noyau : une partie importante de Gunpack se situe au niveau du système d'exploitation. Notamment pour intercepter les exceptions et modifier la mémoire des packers analysés.

– Userland : des scripts d'analyse peuvent être développés pour interagir avec le moteur.

Une toute autre approche aurait pû être choisie, en utilisant un outil permettant de virtualiser Windows "à chaud", comme SimpleVisor [4]. Cette option n'a pas été privilégiée pour deux raisons : cet outil n'existait pas au démarrage de notre projet et surtout l'approche reposant sur l'OS nous semblait plus simple à mettre en oeuvre.

Ce choix a des inconvénients, comme par exemple le fait qu'un programme analysé puisse détecter l'éventuel environnement de virtualisation ; ou encore réussir à élever ses privilèges, grâce à une faille du système d'exploitation par exemple, et contourner ainsi contourner l'outil.

1.4 Idée générale

Gunpack permet de suivre l'exécution d'un processus, afin de collecter suffisamment d'informations pour permettre à l'utilisateur d'étudier un packer et de développer un unpacker le cas échéant.

Pour cela, notre outil instrumente de façon fine le système d'exploitation Windows pour surveiller de manière précise l'exécution d'un processus (création de code dynamique, chargement de bibliothèques, etc.). L'implémentation des scripts d'*unpacking* peut alors se faire dans un langage de haut niveau : en Python.

Avant de présenter en détail l'implémentation et le fonctionnement de Gunpack, nous devons introduire deux mécanismes clés du système d'exploitation Windows : la gestion de la mémoire et le traitement des exceptions.

2 Un peu de Windows Internals

2.1 Traduction d'adresses virtuelles

Les processus s'exécutant sur le système Windows ne manipulent que des adresses virtuelles par opposition aux adresses dites physiques utilisées par le processeur. Aussi, ce dernier doit pouvoir traduire ces adresses virtuelles en adresses physiques de façon à accéder aux données en RAM, c'est l'objet de la traduction d'adresses.

Sur l'architecture x86-64, en mode PAE lorsque la pagination et le mode protégé sont activés, le processeur traduit une adresse virtuelle en adresse physique en utilisant des tables d'indirection stockées en mémoire physique. Ces mécanismes sont décrits dans la documentation Intel [3],

aussi nous ne rentrerons pas dans les détails. Il convient néanmoins de faire quelques rappels.

Le processeur manipule la mémoire par blocs, ou pages de mémoire. Ces pages ont une taille de 4 kilo-octets ou de 2 méga-octets. Dans la suite de cet article nous ne nous intéresserons qu'aux pages de 4 kilo-octets.

En mode PAE, comme le montre la figure 1, trois niveaux d'indirection sont utilisés.

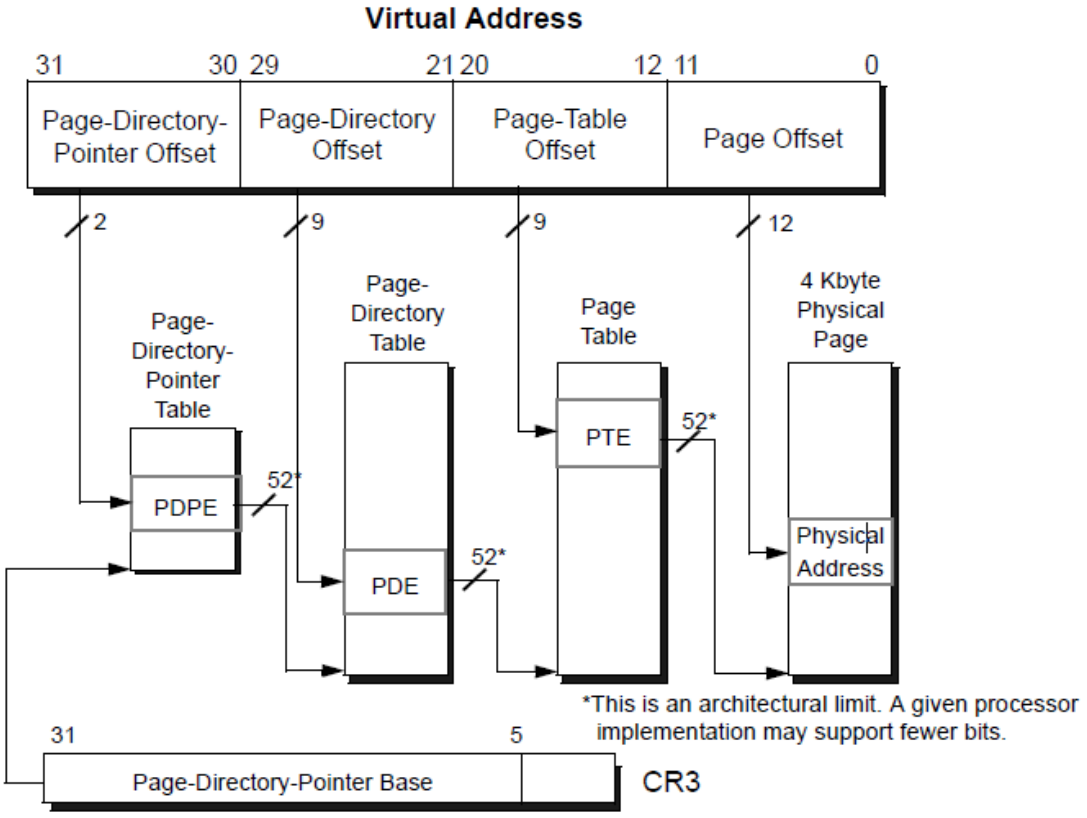

Fig. 1. Traduction d'adresse virtuelle vers adresse physique en mode PAE (documentation AMD)

Le dernier niveau d'indirection, la *Page Table Entry* (ou PTE), mérite une attention toute particulière car elle contient des informations essentielles concernant les droits d'accès à la mémoire. Le schéma 2 détaille le format des PTE.

On peut voir sur ce schéma que, du point de vue du processeur, il n'existe que deux types de PTE :

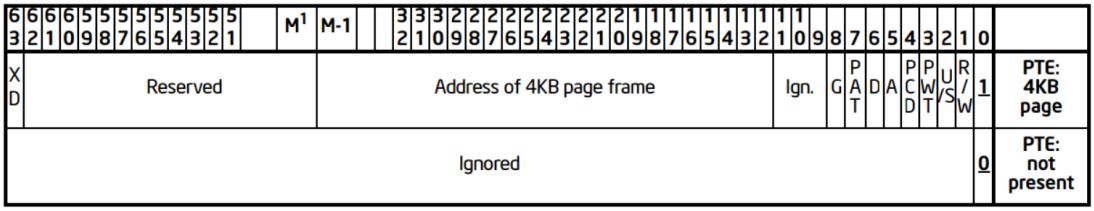

Fig. 2. Format du PTE en mode PAE (documentation AMD)

- valides : bit 0 à **vrai**, une traduction d'adresse virtuelle vers adresse physique existe
- invalides : bit 0 à **faux**, pas de traduction d'adresse

Parmi les nombreux champs d'une PTE valide, certains nous intéressent tout particulièrement :

- **bit 1** (R/W) : Si 0 les données ne sont pas inscriptibles, si 1 elles le sont.
- **bits 12 to 62** : adresse où sont stockées les données en mémoire physique (RAM)
- **bit 63** (XD) : Si 1 les données ne sont pas exécutables, si 0 elles le sont.

Comme nous l'avons vu, lorsqu'une PTE est invalide, les 63 bits de poids fort sont ignorés par le processeur et laissés à la discrétion de l'OS. Windows en tire profit et les utilise pour définir plusieurs types de PTEs invalides dévolues à la gestion de la mémoire. On peut notamment citer :

- *Software PTE* : le contenu de la page a été stocké sur le disque dans le *pagefile* (swap).
- *Transition PTE* : la page est en "transit" vers le *pagefile* mais ses données se trouvent toujours en mémoire.
- *Prototype PTE* : la page est partagée avec un autre processus.

L'exemple des *Software PTE*, qui concernent les pages de mémoire ayant été placées par Windows dans le *pagefile*, illustre bien le type d'information stockées à bas niveau par Windows :

```
typedef struct _MMPTE_SOFTWARE {
    ULONGLONG Valid : 1;
    ULONGLONG PageFileLow : 4;
    ULONGLONG Protection : 5;
```

```
    ULONGLONG Prototype : 1;
    ULONGLONG Transition : 1;
    ULONGLONG PageFileHigh : 52;
} MMPTE_SOFTWARE;
```

La signification de ces champs est la suivante :

- Valid : toujours à 0 car c'est une PTE invalide
- PageFileLow : nombre qui référence le numéro du *pagefile* où se trouve le contenu de la page
- Protection : protection, au sens Windows, qu'avait la page de mémoire avant son stockage dans le *pagefile*
- Prototype : toujours à 0
- Transition : toujours à 0
- PageFileHigh : offset de la page dans le *pagefile*

Il faut retenir de cette section que les PTEs contiennent, à bas niveau, les droits d'accès qu'autorise le processeur sur les données mais également des informations essentielles au fonctionnement de Windows.

2.2 Exceptions et fautes de pages

Lorsqu'un processus ou le noyau accèdent à de la mémoire virtuelle, le processeur effectue une traduction d'adresse pour accéder à la mémoire physique correspondante. Si la PTE est invalide ou si l'accès tenté n'est pas compatible avec les bits de protection (*writable* et *executable*), le processeur génère une faute de page. Il transfère le contrôle au noyau en appelant le gestionnaire de faute de page *KiTrap0E* pour que celui-ci tente de gérer la faute.

Comme nous l'avons vu dans la section précédente, Windows utilise les PTE invalides comme des mécanismes de gestion de la mémoire. Ceci a pour effet de faire générer au processeur des fautes de pages qui ne sont pas causées par un comportement incorrect mais par le fonctionnement normal de l'OS. Pour comprendre comment Windows distingue ces deux types de fautes de pages, intéressons nous au chemin d'exécution suivi par la fonction *KiTrap0E*. La figure 3 détaille ce chemin.

C'est la fonction *ntoskrnl!MmAccessFault* qui est chargée de faire cette distinction. Elle a le prototype suivant :

```
NTSTATUS MmAccessFault (
    ULONG_PTR FaultStatus,
    PVOID VirtualAddress,
    KPROCESSOR_MODE PreviousMode,
    PVOID TrapInformation
    )
```

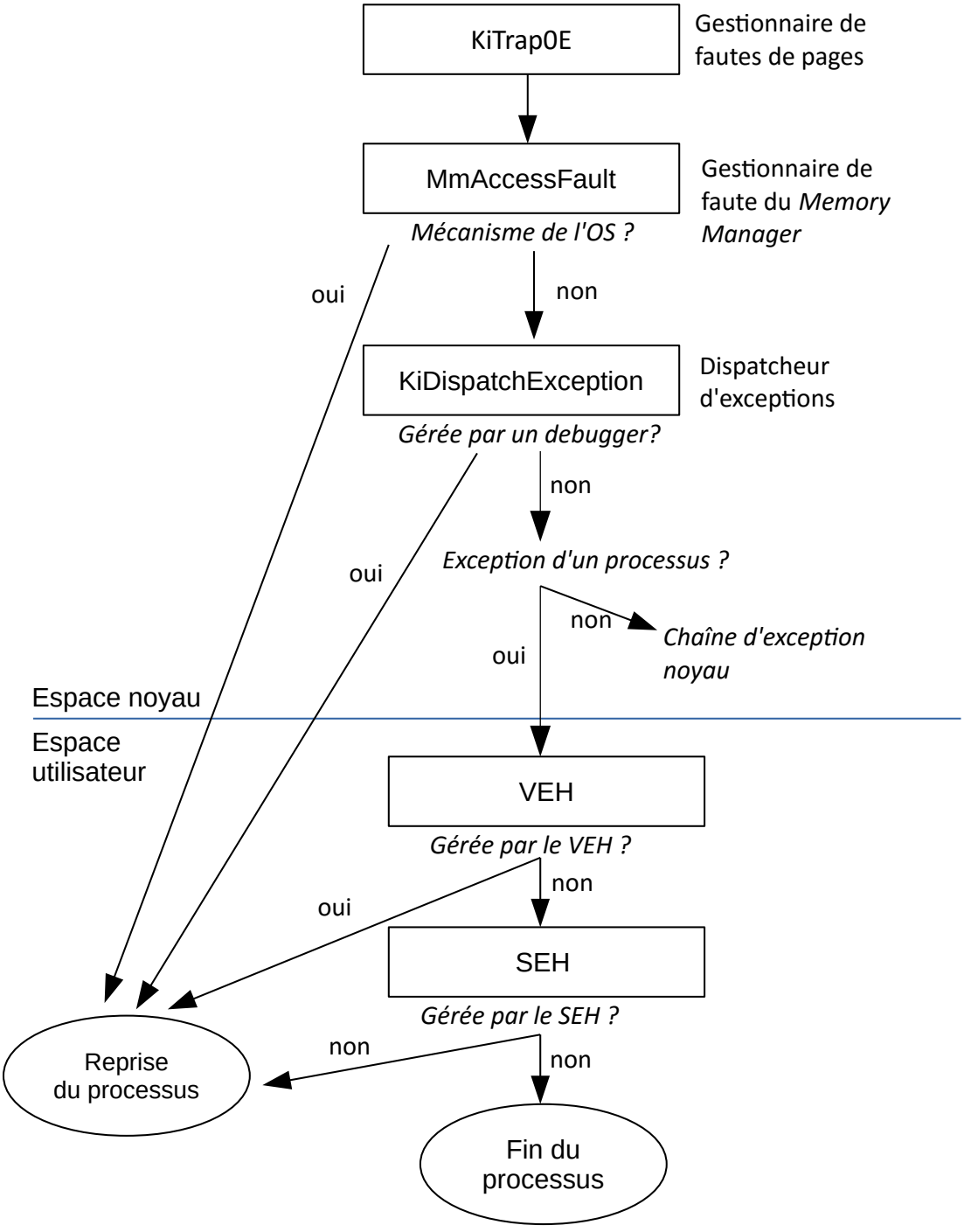

Fig. 3. Chemin du gestionnaire de faute de page de Windows

Ses arguments sont :

- *FaultStatus* : type d'accès ayant entraîné la faute (lecture, écriture, exécution,...)
- *VirtualAddress* : adresse virtuelle de la donnée accédée
- *PreviousMode* : mode du processeur lors de la faute (noyau ou utilisateur)
- *TrapInformation* : pointeur vers, entre autres, le contexte du processeur lors de la faute

Ses valeurs de retour sont :

- Success : code *NTSTATUS* positif indiquant la gestion de la faute et le type de faute géré
- Error : code *NTSTATUS* négatif indiquant à *KiTrap0E* la cause de l'erreur

Si la faute est liée à un mécanisme légitime de l'OS, *MmAccessFault* renvoie donc un code de succès après avoir effectué les actions appropriées suivant le cas :

- *Page file* : la page physique correspondant à l'adresse virtuelle demandée n'est pas en RAM mais dans le fichier de pagination. La page est alors extraite de celui-ci pour être chargée en RAM.
- *Demand zero* : premier accès par le processus sur une page virtuelle allouée. La page physique n'est allouée qu'à ce moment là par soucis d'optimisation

KiTrap0E reprend alors l'exécution du processus en restaurant le contexte *KTRAP_FRAME* passé en argument à *MmAccessFault*. Ceci est totalement transparent pour le processus ayant généré la faute (ou le noyau s'il s'agit d'un accès noyau).

Si, en revanche, la faute n'est pas due à un mécanisme interne de l'OS mais bel et bien à un accès invalide, le noyau doit alors gérer cette faute comme une exception. Il construit alors des structures décrivant l'exception et les passe à *KiDispatchException*, dont le prototype est le suivant :

```
void KiDispatchException (
    PEXCEPTION_RECORD ExceptionRecord,
    PKEXCEPTION_FRAME ExceptionFrame,
    PKTRAP_FRAME TrapFrame,
    KPROCESSOR_MODE PreviousMode,
    BOOLEAN FirstChance)
```

Ses arguments sont :

- *ExceptionRecord* : structure contenant le type d'exception (Single-step, Access violation, etc.), l'adresse virtuelle où elle s'est produite, etc.
- *ExceptionFrame* : similaire à TrapFrame
- *TrapFrame* : pointeur vers, entre autres, le contexte du processeur lors de la faute
- *PreviousMode* : mode du processeur lors de la faute (noyau ou utilisateur)
- *FirstChance* : indique si c'est une FirstChance ou SecondChance exception

Le rôle de *KiDispatchException*, comme son nom l'indique, est d'aiguiller l'exception vers le bon gestionnaire. En premier lieu, *ntoskrnl!KiDispatchException* détermine si un débuggeur est attaché au processus ou au noyau. Si tel est le cas, l'exception est transmise au débugger pour traitement. Si ce dernier a traité l'exception, l'exécution du processus ou du noyau est reprise à partir des informations de contexte contenues dans la structure *KTRAP_FRAME*. En revanche, si aucun débuggeur n'est présent ou si celui-ci n'a pas sû gérer l'exception, *KiDispatchException* aiguille l'exception vers la chaîne de gestion des exceptions enregistrées pour le thread courant.

Dans le cas d'une exception en *userland*, la suite de la gestion des exceptions se déroule en *userland* dans le contexte du processus ayant généré la faute. Des *callbacks* peuvent y être enregistrés pour gérer spécifiquement certains types d'exceptions.

Deux types de gestionnaires peuvent être définis et sont appelés dans cet ordre :

- *Vectored Exception Handlers* : gestionnaires définis pour l'ensemble du processus, appelés très tôt dans la chaîne en mode utilisateur. [10]
- *Structured Exception Handlers* : gestionnaires définis pour chaque thread du processus. Ils constituent la fin de la chaîne de gestion des exceptions. [9]

Chacun des *VEH* et *SEH* peut aussi restaurer le contexte du processus pour reprendre son exécution. Si finalement aucun gestionnaire d'exception, en mode utilisateur comme en mode noyau, ne gère l'exception, Windows interrompt le processus.

Ce qu'il faut retenir de cette section est qu'il est possible de gérer une faute à plusieurs niveaux. Aussi bien au début de la chaine d'exception,

dans le noyau, que dans le contexte du processus lui-même. Le processus reprend son exécution à partir d'un contexte potentiellement modifié par le gestionnaire d'exception.

2.3 Espace d'adressage d'un processus

Les processus *userland* disposent d'un espace de 2 giga-octets de mémoire virtuelle dans lequel réside toutes les données du processus (code et données). Cette mémoire virtuelle est manipulée par l'OS et le processeur avec la granularité d'une page.

Le système d'exploitation gère l'espace d'adressage du processus par blocs de pages contigües aussi appelées régions. À chaque région de mémoire virtuelle d'un processus sont associés les attributs suivants :

- Addresse (virtuelle) de base de la région ;
- Taille de la région ;
- État : mémoire allouée, libre ou réservée ;
- Protection : combinaison de droits comme **readable**, **writable**, **executable**, etc ;
- Type : mémoire privée, partagée ou image d'une section (dll).

L'attribut *protection* est un combinaison de constantes comme :

- *PAGE_NOACCESS*
- *PAGE_READWRITE*
- *PAGE_WRITECOPY*
- *PAGE_EXECUTE*

Windows permet 32 combinaisons de constantes pour représenter la protection d'une page de mémoire virtuelle ; cet attribut protection est stocké sur 5 bits.

Un processus peut manipuler son espace d'adressage à l'aide de différents appels systèmes, notamment :

- NtAllocateVirtualMemory : allocation une région de mémoire
- NtProtectVirtualMemory : changement de l'attribut *Protection* d'une région
- NtQueryVirtualMemory : récupération des attributs d'une région
- NtFreeVirtualMemory : libération d'une région
- NtMapViewOfSection : chargement d'une région étant la vue d'un section (fichier par exemple)
- NtUnmapViewOfSection : déchargement de la vue d'une section

Pour représenter cet espace d'adressage Windows maintient une structure appelée *Virtual Address Descriptor* ou VAD. À chaque processus est associé un VAD, qui est un arbre binaire dans lequel chaque noeud est une région de la mémoire du processus, comme le montre l'exemple de la figure 4.

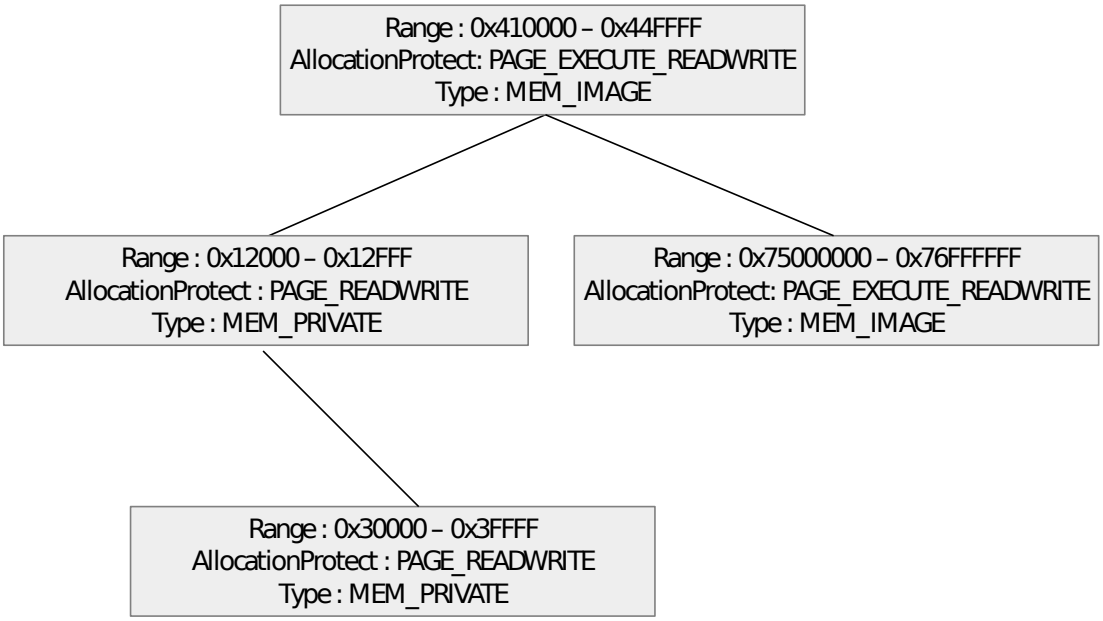

Fig. 4. Exemple de VAD

Les attributs de chaque région, à l'exception de l'attribut protection, sont contenus dans le VAD. Lorsque Windows recherche des informations sur une page de mémoire virtuelle, il parcourt le VAD du processus. Si une région contenant la page existe, le système consulte alors les attributs contenus dans le noeud représentant la région. Si aucun noeud (région) contenant la page n'existe dans le VAD, l'adresse virtuelle est considérée comme libre.

L'ensemble des pages virtuelles d'un processus chargées en RAM, c'est à dire ayant une traduction d'adresse virtuelle vers physique valide, constitue le *Working Set* du processus. Ce *Working Set* évolue constamment, le système chargeant des pages en mémoire physique lorsque celles-ci sont accédées et les déchargeant de la mémoire physique lorsque la ressource se fait rare. Ces informations sont dans un tableau, le *Working Set List*, qui référence chaque page du *Working Set* pour ce processus.

Afin d'optimiser les performances du système, Windows ne place pas directement dans le fichier de pagination les pages de mémoire physique qui quittent le *Working Set*. Ces pages passent d'abord par une liste temporaire en mémoire, la *Modified page list*. D'une manière générale, le gestionnaire de mémoire utilise plusieurs listes pour optimiser l'utilisation de la mémoire physique de la machine. De plus amples informations à ce sujet sont disponibles dans le chapitre *Memory Management* de *Windows Internals*. Par soucis de simplicité nous considérerons deux cas : le contenu d'un page de mémoire virtuelle se trouve soit en mémoire (RAM), soit dans le fichier de pagination (sur le disque).

Afin de suivre à tout moment le contenu de la mémoire physique, le noyau maintient un structure de données globale au système, la *Page Frame Number Database*. C'est en réalité un tableau indexé par le *Page Frame Number*, qui se calcule en conservant la partie significative de l'adresse physique de la page : en ôtant les 12 bits de poids faible. Les entrées de la *Page Frame Number Database* ont un champ (*OriginalPte*) qui stocke le champ protection associé à chaque page.

Afin de comprendre l'interaction entre tous ces composants, examinons la figure 5.

Tous les attributs d'une page de mémoire virtuelle d'un processus se trouvent, à l'exception de l'attribut protection, dans le VAD. L'emplacement du champ protection varie suivant que la page est dans le *Working Set* ou non. Si la page se trouve dans le *Working Set*, l'attribut protection se trouve à la fois dans la *Working Set List* et dans la *Page Frame Number Database*. En revanche, lorsque la page ne se trouve pas dans le *Working Set*, quand elle est dans le *pagefile*, le champ protection est stocké parmi les 63 bits du PTE ignorés par le processeur, comme nous l'avons vu dans la section 2.1.

3 L'outil

3.1 Surveillance de la mémoire

Le mécanisme clé de Gunpack est celui permettant de suivre l'évolution de la mémoire du processus analysé. Non seulement l'évolution des régions de son espace d'adressage (allocation, libération, etc.) mais aussi, et surtout, la génération de code dynamique. Gunpack permet de suivre tous les accès en écriture et en exécution effectués par le processus durant son exécution avec la granularité d'une page. Il devient alors possible de surveiller la génération dynamique de code, très employée par de nombreuses familles de packers et de codes malveillants.

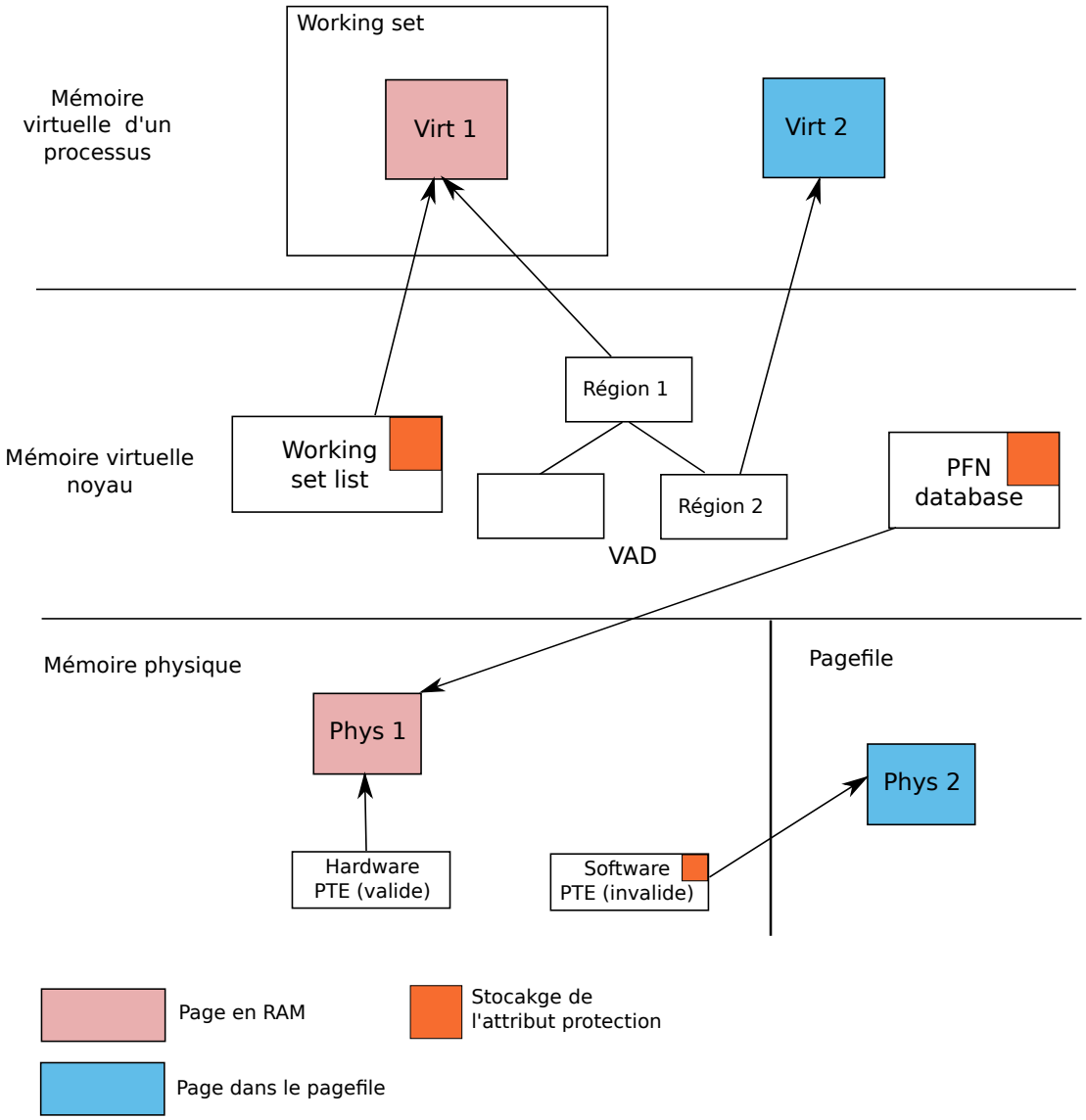

Fig. 5. Structures du gestionnaire de mémoire Windows

Pour cela, notre outil modifie les droits (protection) des pages du processus à son insu. Lorsque celui-ci accède soit en écriture, soit en exécution à la mémoire, des exceptions sont générées et traitées très tôt dans la chaîne de gestion, l'évènement est alors enregistré puis les droits de la mémoire restaurés. Tout ceci de façon totalement invisible pour le processus analysé.

Afin de garantir qu'aucun accès n'échappe au monitoring, Gunpack s'assure qu'à tout instant la mémoire puisse être que **writable** ou **executable**, de façon exclusive.

3.2 Modification du noyau

Nous avons besoin de modifier l'attribut *protection* des pages de mémoire du processus analysé. Nous aurions pu utiliser une méthode simple et efficace : se servir de l'appel système dédié *NtProtectVirtualMemory*. Ceci aurait eu pour effet de mettre à jour l'attribut *protection* dans toutes les structures de données internes du noyau présentées précédemment (figure 5). Il serait alors aisé pour le processus analysé de se rendre compte de cette modification en utilisant, par exemple, l'appel système *NtQueryVirtualMemory*. Sauf à modifier également les structures du noyau et fournir à *NtQueryVirtualMemory* une "vue" de ce qu'aurait dû être la mémoire du processus analysé, tâche qui nous paraît pour le moins complexe.

Nous avons choisi une toute autre approche : celle de laisser ces structures intactes et de ne modifier l'attribut *protection* que dans le PTE de la page en mémoire physique. On crée alors une "dé-synchronisation" entre la protection des pages dans les PTE et la vision qu'a le noyau de la mémoire du processus. On doit cependant s'assurer que :

- le processus ne peut modifier ses pages de sa propre son initiative ;
- le noyau ne re-synchronisera pas ces deux vues.

Pour que le processus analysé ne puisse modifier sa mémoire, nous avons utilisé un méthode classique visant à intercepter les appels systèmes suivants :

- NtAllocateVirtualMemory
- NtProtectVirtualMemory
- NtMapViewOfSection

Lorsque celui-ci effectue un de ces appels système, Gunpack laisse dans un premier temps l'appel système s'exécuter afin de permettre au

noyau de mettre à jour ses structures internes (VAD, PFN, WSL,...). Dans un second temps, les droits modifiés son appliqués ou ré-appliqués par Gunpack aux PTE des pages correspondantes.

Concernant la "re-synchronisation" des deux vues (noyau et mémoire physique), celle-ci se produit lorsque le gestionnaire *MmAccessFault* est appelé. En effet, si celui-ci est appelé à cause d'une faute liée à la "dé-synchronisation" décrite précédemment, cette incohérence est traitée comme une faute liée à la gestion de l'OS (*Demand Zero*, *Page File*, etc.). L'attribut *protection* du PTE est alors restauré ("re-synchronisé") et le processus reprend son exécution. Si on laissait cela se produire cela aurait pour effet de rendre très rapidement les modifications effectuées par Gunpack inopérantes. Il donc a fallu intercepter les exceptions très tôt dans la chaine : au niveau de *MmAccessFault*.

Notre outil maintient une liste des pages suivies du processus, indiquant si la page est monitorée par Gunpack et si oui quels sont les droits du PTE associé (**writable** ou **executable**). On peut alors écrire le pseudo-code du notre fonction *MmAccessFault* modifiée comme suit :

```
NTSTATUS MmAccessFault_hook (
    ULONG_PTR FaultStatus,
    PVOID VirtualAddress,
    KPROCESSOR_MODE PreviousMode,
    PVOID pTrap
    )
{
    if ( !IsTrackedProcess() )
        goto not_hooked;

    if ( !IsTrackedPage(VirtualAddress) )
        goto not_hooked;

    MmAccessFault(FaultStatus, VirtualAddress, PreviousMode, pTrap);

    if ( FaultStatus == READ )
    {
        GetStoredPTEAccess(&writable,&executable);
        SetAccessOnPTE(VirtualAddress,writable,executable);
    }
    else if ( FaultStatus == WRITE )
    {
        SetStoredPTEAccess(WRITABLE,NOT_EXECUTABLE);
        SetAccessOnPTE(VirtualAddress,WRITABLE,NOT_EXECUTABLE);
    }
    else if ( FaultStatus == EXECUTE )
    {
        SetStoredPTEAccess(NOT_WRITABLE,EXECUTABLE);
        SetAccessOnPTE(VirtualAddress,NOT_WRITABLE,EXECUTABLE);
    }

    return 0;

not_hooked:

    return MmAccessFault(FaultStatus, VirtualAddress, PreviousMode, pTrap);
}
```

Les fautes concernant un processus non monitoré par Gunpack ainsi que les fautes sur des pages d'un processus monitoré mais non marquées comme suivies sont ignorées, c'est à dire transmises à la chaine de traitement

des exceptions. Ensuite, la fonction *MmAccessFault* originelle est appelée pour s'assurer que les PTE manipulés ensuite seront valides puisque cette dernière chargera la page en mémoire physique si celle-ci n'y réside pas déjà. Vient ensuite une disjonction des cas sur les raisons de la faute. Deux primitives sont utilisées :

- *SetStoredPTEAccess* : change les *flags* (*writable*, *executable*) du PTE associés à une adresse (suivi de Gunpack).
- *SetAccessOnPTE* : modifie les droits (writable, executable) dans le PTE d'une page

On observe la bascule qui permet de s'assurer que chaque page peut être writable *xor* executable. Il est important de noter que les fautes en lecture sont elles aussi gérées. Lorsqu'une faute en lecture se produit sur une page "monitorée" par notre outil, cela est nécessairement dû au fait que la page a été sortie du *Working Set* par le noyau (mise dans le *pagefile* par exemple). Il est alors important de restaurer les droits du PTE modifiés par l'outil car l'appel à *MmAccessFault* ligne 14 a, lui, "re-synchronisé" les PTE avec la vue du noyau. Il faut donc ré-appliquer la modification des droits de la page, sans quoi nous risquons de rater des accès ultérieurs à cette page. Grâce à cette modification de *MmAccessFault*, on s'assure que le processus analysé manipulera toujours des pages dont on a modifié le PTE, et ainsi que tous les accès à ces pages seront monitorés par Gunpack.

Sur la figure 6 on peut voir que Gunpack intercepte également les exceptions au niveau de *KiDispatchException*. C'est par ce que, contrairement à *MmAccessFault*, *KiDispatchException* permet de gérer les exceptions liées au *single-step* (ou exécution pas à pas). Nous allons voir dans les sous-sections suivantes pourquoi c'est nécessaire.

3.3 Cas des pages auto-modifiantes

Il est un cas que l'on rencontre souvent lorsque l'on analyse des malwares ou des packers en tous genres : celui de code auto-modifiant situé dans une même page. Prenons l'exemple du code suivant :

```
00401009  MOV EAX,0040101D
0040100E  XOR ECX,ECX
00401010  XOR BYTE PTR DS:[EAX+ECX],42
00401014  INC ECX
00401015  CMP ECX,100
0040101B  JNE SHORT 00401010
0040101D  db 0D2h
```

Ce bout de code, situé dans la page *0x00401000*, effectue une boucle pour décoder, à l'exécution, la suite des instructions à l'adresse *0x0040101D*.

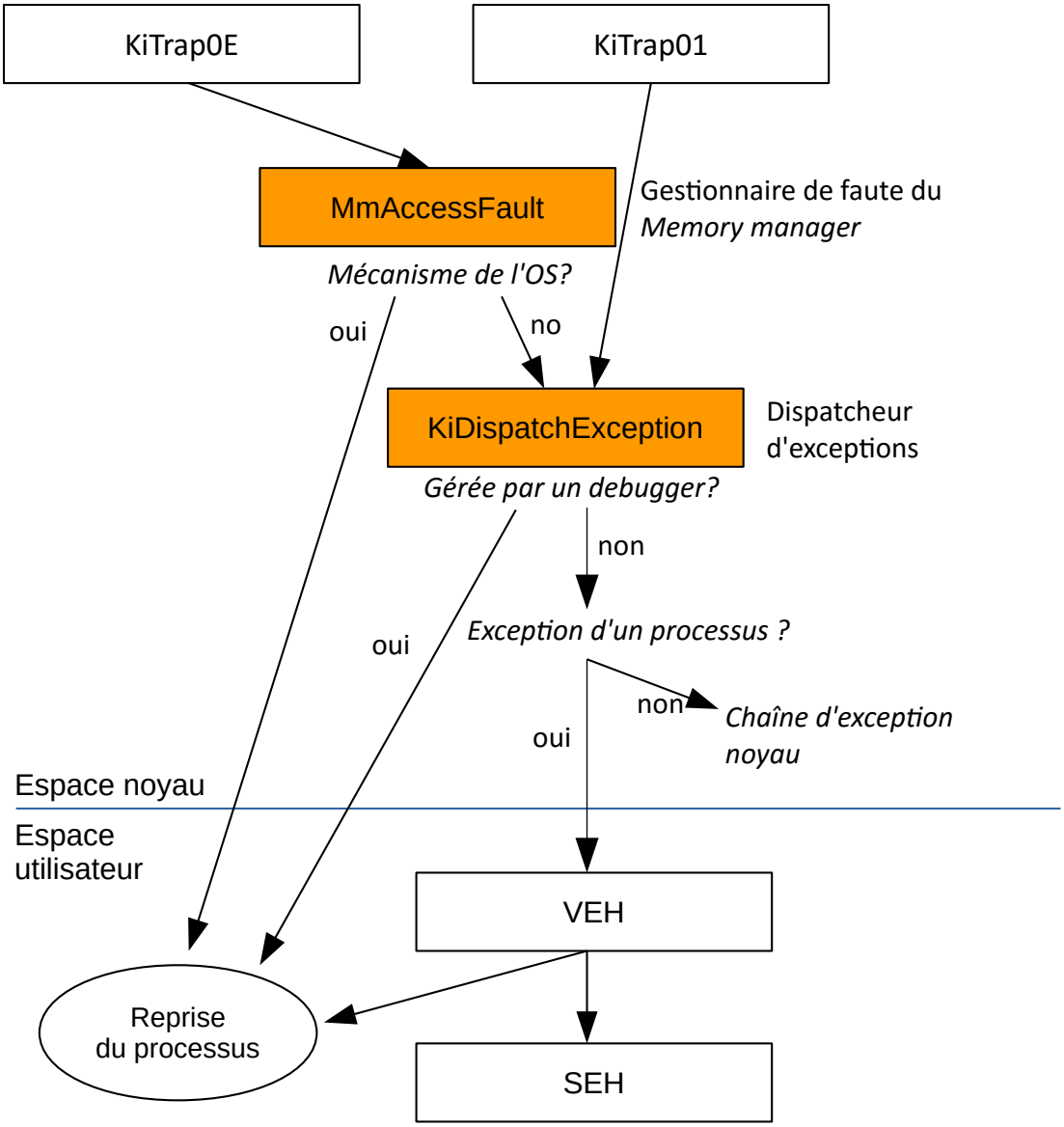

Fig. 6. Fonctions hookées par Gunpack

Si on analysait ce programme en appliquant strictement l'invariant qui interdit à une page d'être à la fois writable et executable, nous rentrerions dans une boucle infinie. Déroulons notre algorithme sur cet extrait de code pour s'en rendre compte. La page ayant initialement les droits RX :

- 1. Exception : tentative d'écriture à *0x0040101D* (Eip = 0x00401010) alors que la page est RX. La page devient RW, reprise de l'exécution en *0x00401010*.
- 2. Exception : tentative d'exécution à *0x00401010* alors que la page est RW. La page devient RX, reprise de l'exécution en *0x00401010*.
- 3. On se retrouve dans le cas 1.
- ...

Il faut donc effectuer un traitement spécial pour les pages auto-modifiantes. Nous avons choisi de proposer deux stratégies, le choix de la stratégie est paramétrable dans le moteur de Gunpack.

La première stratégie consiste à activer le *single-step* du processeur. Lorsqu'une instruction du programme analysé nécessite que la page soit RWX, le moteur de Gunpack passe les droits de la page à RWX puis active l'exécution pas à pas du processeur dans le contexte du thread avant de le laisser reprendre son exécution. Une fois l'instruction exécutée, notre gestionnaire d'exception sera à nouveau appelé, mais pour gérer l'exception *single-step* cette fois. Le moteur restaure les droits originels de la page (RX) et désactive le mode pas à pas. Cette stratégie a pour avantage de garantir l'invariant, puisque l'exception à cet invariant n'est autorisée que sur une seule instruction. Elle a pour inconvénient majeur d'augmenter considérablement le temps d'analyse d'un programme utilisant les pages auto-modifiantes de façon massive.

C'est pourquoi nous avons décidé de proposer une stratégie alternative. Cette fois lorsqu'une instruction nécessite que sa page soit RWX, lors de l'exception le moteur passe la page en RWX avant de rendre la main au programme analysé. Cette page restera RWX jusqu'à ce qu'une faute en écriture se produise dans le processus. La page reprendra alors ses droits initiaux (RX). Le moteur n'autorise qu'une seule page en RWX pour l'ensemble du processus. Bien que l'utilisation de la page RWX soit très restreinte, cette technique a pour inconvénient de rompre l'invariant, en revanche elle a le gros avantage d'accélérer grandement l'analyse.

3.4 Cas des appels système

Intéressons nous à la façon dont le noyau s'assure de la validité des pointeurs passés aux appels systèmes. Lorsqu'un processus effectue un

appel système, le noyau s'assure que ceux-ci se trouvent bien en *userland* et qu'ils ont les bons attributs de protection. C'est à dire que le noyau pourra lire leur contenu dans le cas de données de type *input* et qu'il pourra écrire dans ces buffers s'ils sont de type *output*. Pour cela, le noyau tente d'abord de lire ou d'écrire la totalité du buffer avec une primitive du type *ProbeForWrite* ou *ProbeForWrite*, qu'on peut simplifier comme suit :

```
void ProbeForWrite ( BYTE * outbuffer , SIZE_T out_size )
{
    SIZE_T i;
    BYTE c;
    for (i=0; i< out_size ;i++)
    {
        c = outbuffer[i];
        outbuffer[i] = c;
    }
}
```

Dans chaque gestionnaire d'appel système, le noyau enregistre un gestionnaire d'exception local de façon à retourner une erreur si les arguments reçus n'ont pas les droits de protection adéquats. Prenons l'appel système *NtProtectVirtualMemory* en exemple :

```
NTSTATUS NtProtectVirtualMemory(HANDLE ProcessHandle,
                                PVOID *BaseAddress,
                                PULONG NumberOfBytesToProtect,
                                ULONG NewAccessProtection,
                                PULONG OldAccessProtection)
{
    try
    {
        ProbeForWrite(OldAccessProtection, sizeof(ULONG));
    }
    __except( EXCEPTION_EXECUTE_HANDLER )
    {
        return GetExceptionCode();
    }

    //syscall stuff...
    return result;
}
```

Si le dernier argument, passé par le processus, référence de la mémoire non inscriptible alors la primitive *ProbeForWrite* génère une exception en écriture. Comme nous l'avons vu sur la figure 3, cette exception atteint alors *KiDispatchException* qui aiguille cette exception vers le gestionnaire noyau défini dans l'appel système concerné ; qui ici renvoie un code d'erreur au processus sans que l'appel système ne se soit exécuté correctement. Lorsque Gunpack analyse un processus, il modifie les droits de la mémoire à l'insu de ce dernier. Il arrive donc que le processus analysé fournisse au noyau des buffers qui devraient être inscriptible mais qui en réalité ne le sont pas car modifiés par Gunpack. Pour pallier ce problème, lorsque pareil cas se produit, les droits de la page sont passés temporairement à RW, le noyau est exécuté pas à pas pendant une instruction, puis les

droits de la page sont restaurés, le mode pas à pas désactivé et l'exécution du noyau relancée. En d'autres termes, on *single-step* le noyau.

3.5 Architecture générale

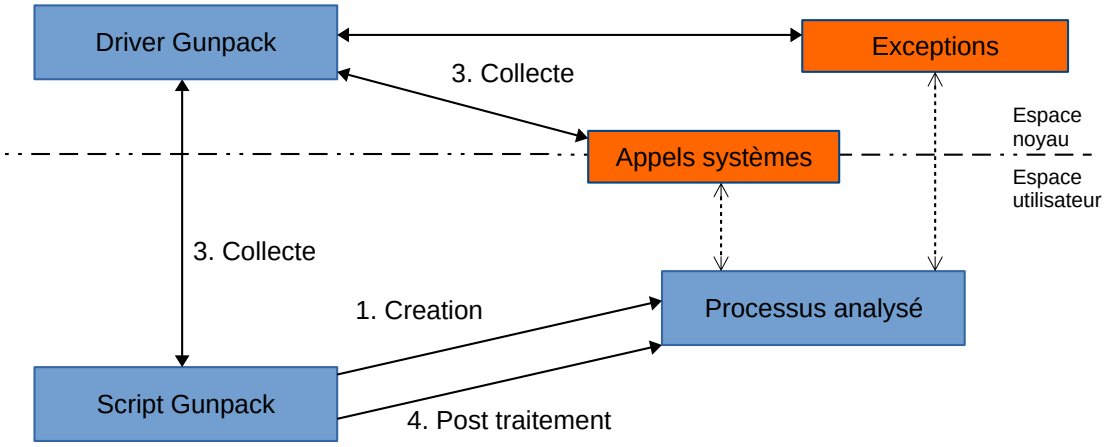

Fig. 7. Architecture de Gunpack

Gunpack se compose de deux éléments (figure 7) :

- Un pilote noyau qui se charge de collecter les évènements liés au processus analysé (Exceptions, appels système, ...)
- Des scripts, développés en python, rapatrient les informations collectées par le pilote et implémentent l'intelligence de l'outil

Le pilote de Gunpack transmet divers objets aux scripts suivant l'évènement qui s'est produit dans le processus analysé :

Exception :

- Adresse virtuelle de l'exception
- Adresse physique de l'exception
- Adresse virtuelle de l'instruction ayant généré l'exception (Eip)
- Type d'exception (WRITE ou EXECUTE)
- Etat interne du loader windows : en cas de chargement d'une DLL (Adresse de base de la Dll, Chemin, ...)

Création de processus :

– PID du processus père
– PID du processus fils

Création d'un thread :

– PID du processus
– TID du nouveau thread

Chargement d'une bibliothèque :

– Adresse de base de la bibliothèques
– PID du processus
– Chemin de la bibliothèque

Mort d'un processus :

– PID du processus
– PID du processus tué

Mais également toutes les informations relatives aux appels système mémoire :

– allocation mémoire
– libération mémoire
– chargement d'une section
– protection de la mémoire

L'analyse d'un processus s'effectue en plusieurs phases. Tout d'abord Gunpack crée le processus analysé dans l'état suspendu. Le driver de Gunpack parcourt alors l'espace d'adressage du processus et met toutes les pages de mémoire en RX, puis démarre l'exécution du processus.

Durant l'exécution du processus, chaque évènement décrit ci dessus est envoyé au script Python, la fonction *event_handler* est alors appelée :

```
    def event_handler(self, event_type, event_obj):
        None
```

Elle reçoit en paramètre l'évènement qui s'est produit pour pouvoir le traiter. Le thread courant du processus cible reste suspendu tant que la fonction *event_handler* n'a pas terminé son traitement. Suivant la valeur retournée, l'exécution reprend ou non :

– 1 : l'analyse continue

– 0 : l'analyse s'arrête

Si la fonction *event_handler* décide de stopper le processus ou si un certain temps s'est écoulé (paramétrable à l'initialisation du moteur), le driver de Gunpack suspend le processus et notifie le script d'unpacking pour lui permettre d'effectuer un post-traitement. La fonction *post_treatment* est alors appelée :

```
    def post_treatment(self):
        None
```

Il est donc relativement aisé de développer son propre script d'unpacking. Pour cela, il suffit de régler les paramètres du moteur (temps maximum d'unpacking, gestion des pages automodifiantes, ...) et de surcharger les fonction *event_handler* et *post_treatment*.

4 Exemples d'application

Cette section présente des exemples de scripts d'*unpacking* qui peuvent être développés aisément avec Gunpack. Il ont tous pour objectif de retrouver l'OEP (*original entry point*), c'est-à-dire le point où le packer a fini son travail et passe la main au programme original.

4.1 Unpacker pour Upack

Upack est un packer très simple. Il embarque le code du programme original compressé en LZMA, le décompresse puis l'exécute après avoir reconstruit ses imports. C'est un bon exemple car il montre pourquoi le réglage du moteur de Gunpack est important. En effet, la page du packer qui décompresse le programme original est censée être RWX. A chaque écriture d'un groupe d'instructions décompressées, la morceau de code suivant est exécuté :

```
00C8999F  43          INC EBX
00C899A0  59          POP ECX
00C899A1  895D 0C     MOV DWORD PTR SS:[EBP+C],EBX
00C899A4  56          PUSH ESI
00C899A5  8BF7        MOV ESI,EDI
00C899A7  2BF3        SUB ESI,EBX
00C899A9  F3:A4       REP MOVS BYTE PTR ES:[EDI],BYTE PTR>
00C899AB  AC          LODS BYTE PTR DS:[ESI]
00C899AC  5E          POP ESI
00C899AD  B1 80       MOV CL,80
00C899AF  AA          STOS BYTE PTR ES:[EDI]
00C899B0  3B7E 24     CMP EDI,DWORD PTR DS:[ESI+24]
00C899B3  73 03       JNB SHORT wscript.00C899B8
```

L'instruction située à la ligne 3 écrit dans la page *0x00C89000* à chaque tour de boucle. Si on choisi d'activer la stratégie de *single-step* dans le moteur, l'instruction de la ligne 3 sera exécutée pas à pas à chaque tour de boucle, faisant augmenter le temps d'analyse à un niveau inacceptable. Dans ce cas, l'utilisation de la stratégie RWX rend l'unpacking beaucoup plus efficace.

On peut alors implémenter le script de la façon suivante :

```
class Upack(Gunpack):

    def __init__(self, file_path):
        self.command_line = file_path
        self.set_rwe_policy(1)
        self.set_max_unpack_time(120)

        self.bin_name = file_path.split("\\")[-1]
        self.written_pages = []
        self.librairies_loaded = False

    def event_handler(self, event_type, event_obj):

        if ( event_type == EVENT_EXCEPTION ):
            e = event_obj

            if ( e.OwningThread != None):
                return 1

            PageBase = e.AccessedAddress >> 0xC

            if ( e.AccessedType == WRITE_ACCESS ):
                self.written_pages.append(PageBase)

            if ( (self.librairies_loaded == True) and (e.AccessedType == EXECUTE_ACCESS ))
                 :
                if ( PageBase in self.written_pages ):
                    self.oep = e.AccessedAddress
                    return 0

        if ( event_type == EVENT_LOAD_LIBRARY ):
            library = event_obj
            lib_name = ConvertString(library.DllName)

            if ( lib_name.find("System32\\ntdll.dll") != -1 ):
                return 1

            if ( lib_name.find("System32\\kernel32.dll") != -1 ):
                return 1

            if ( lib_name.find("System32\\KernelBase.dll") != -1 ):
                return 1

            if ( lib_name.find(self.bin_name) != -1 ):
                return 1

            self.librairies_loaded = True

        return 1

    def post_treatment(self):
        self.process.DumpPE( self.process.pid, self.oep, "dumped.exe" )
        print "Process dump with Oep = 0x%x" % self.oep
```

L'OEP se trouve dans une page de code générée dynamiquement, dans l'ensemble des pages écrites. Plus précisément, l'OEP se trouve juste après le chargement des librairies, engendré par la résolution des imports. Il correspond donc au premier accès exécution après le chargement des

librairies dans une page préalablement écrite. le script n'a donc besoin de traiter, dans la méthode *event_handler*, que les évènements de type *EVENT_EXCEPTION* et *EVENT_LOAD_LIBRARY*.

4.2 Unpacker générique

Intéressons nous maintenant à l'outil d'unpacking générique décrit dans le papier MutantX-S [8]. L'algorithme utilisé, extrait du papier, est présenté sur la figure 8.

Algorithm 1 `MutantX-S` unpacking algorithm

Input: A packed binary program B
Output: A unpacked PE file with original program codes

Load the packed program into memory
for all p in the program's memory pages **do**
  $Permission(p)| = \tilde{W}$ //remove write permission
end for

while B is running **and** $T_{runtime} < T_{thresh}$ **do**
  a: The address of the page fault
  t: The page fault type $t \in$ {WRITE, EXECUTE}
  $p \leftarrow Page(a)$
  if t = WRITE **then**
    $Permission(p)| = (W|\tilde{X})$ // Writable but non-executable
    $last\_written(p) \leftarrow$ current time
  end if
  if t = EXECUTE **then**
    $Permission(p)| = (\tilde{W}|X)$ //non-writable but executable
    $last\_exec(p) \leftarrow$ current time
    $addr\_exec(k) \leftarrow a$
  end if
end while

Dump process memory
reconstruct B' by setting OEP to be $addr\_exec(k)$ where:
$k = \arg\min_k(last\_exec(k) > \max(last\_written(i))$
return B'

Fig. 8. Algorithme d'unpacking générique utilisé par MutantX-S

L'algorithme permettant de retrouver l'OEP est en fait assez simple. Le programme cible est exécuté de façon à générer une trace de tous ses accès mémoire. Pour chaque accès, sont enregistrés le type d'accès

(écriture ou exécution), le moment où il se produit ainsi que l'adresse en mémoire où il s'est produit. Le programme est exécuté durant un temps qu'on estime suffisamment long pour que le packer ait fini son travail, pour que l'OEP ait été appelé.

Finalement, cette trace est parcourue pour déterminer lequel de ces accès correspond à l'OEP. L'OEP étant issu d'une page générée dynamiquement par le code du packer, il se trouve nécessairement dans une page appartenant au sous-ensemble des pages ayant été écrites puis exécutées. Puisque le programme original démarre son exécution une fois le travail du packer terminé, cet OEP est la première page exécutée après la dernière écriture dans ce sous-ensemble.

On peut écrire cet unpacker générique de façon relativement simple avec Gunpack. Tout d'abord avec une classe *Generic* qui hérite de la class *Gunpack* et qui collecte toutes les exceptions jusqu'à ce que le processus ne termine ou n'atteigne son temps d'exécution maximal autorisé :

```
class Generic(Gunpack):

    def __init__(self, command_line):
        self.command_line = command_line
        # set Gunpack engine params
        self.set_max_unpack_time(30)
        self.set_rwe_policy(RWE_SINGLE_STEP)
        self.exception_list = []

    def filter_execption(self,e):
        if e.OwningThread != None:
            return 0
        if self.process.is_address_in_dll(e.AccessedAddress):
            return 0

        return 1

    def event_handler(self, event_type, event_obj):
        #We are interested only in EXCEPTION events
        if ( event_type == EVENT_EXCEPTION ):
            exception = event_obj

            #Filter out loader related exceptions
            if self.filter_execption(exception):
                self.exception_list.append(exception)

        #The process tries to exit, stop unpacking
        if ( event_type == EVENT_TERMINATE_PROCESS ):
            terminate = event_obj

            if (terminate.TargetProcessId == self.process.pid):
                return 0

    def post_treatment(self):
        # Apply MutantX algorithm on exception list to compute OEP
        (Oep_virtual, Oep_physical) = mutantX.mutantX(self.exception_list)
        if (Oep_virtual == 0):
            print "[x] Error : unable to determine Oep"

        # Dump PE with new OEP
        self.process.DumpPE( self.process.pid, Oep_virtual, "dumped.exe" )
```

Puis ces exceptions sont passées à la fonction *MutantX* qui détermine l'OEP. Nous avons appliqué l'algorithme sur les adresses physiques plutôt que virtuelles :

```
def mutantX(ExceptionsList):

    PageAccessDict = {}
    i = 0

    for e in ExceptionsList:
        e.PhysicalAccessedAddress = e.Physicalhigh * 0x100000000 + e.Physicallow

        i = i + 1
        PageBase = e.PhysicalAccessedAddress & 0xFFFFF000

        if PageBase in PageAccessDict.keys():
            CurrentPage = PageAccessDict[PageBase]
        else:
            CurrentPage = PageAccess()
            PageAccessDict[PageBase] = CurrentPage

        if e.AccessedType == WRITE_ACCESS:
            CurrentPage.LastWritten = i
        elif e.AccessedType == EXECUTE_ACCESS:
            CurrentPage.LastExec = i
            CurrentPage.ExecAddress = e.PhysicalAccessedAddress
        else:
            self.log.error("[AddPageAccess] ERROR : unexpected page access : %d!" %
                 AccessedType)
            raise UnpackerException("unexpected page access" % AccessedType)

    type1_pages = []
    page_access = 0

    #Build the written and executed pages set
    for page_base in PageAccessDict.keys():
        page_access = PageAccessDict[page_base]
        if (page_access.LastWritten != 0) and (page_access.LastExec != 0):
            type1_pages.append(page_access)

    #Find the last written page in the set
    last_write = 0
    last_write_page = page_access
    for page_access in type1_pages:
        if page_access.LastWritten > last_write:
            last_write = page_access.LastWritten

    #Compute physical Oep
    Physical_Oep = 0
    min_exec = 0xFFFFFFFF
    for page_access in type1_pages:
        if (page_access.LastExec > last_write) and (page_access.LastExec < min_exec):
            min_exec = page_access.LastExec
            Physical_Oep = page_access.ExecAddress

    #Match virtual Oep
    Oep = 0
    if Physical_Oep != 0:
        for exp in ExceptionsList:
            if exp.PhysicalAccessedAddress == Physical_Oep :
                Oep = exp.AccessedAddress

    return (Oep,Physical_Oep)
```

Il est difficile de donner des statistiques précises d'un tel *unpacker* générique. Tout d'abord par ce que disposer d'un base de malwares packés variées, complète et correctement triée n'est pas chose aisée. Mais aussi car l'algorithme présenté ici est dépendant du comportement de la charge virale qui peut parfois induire l'algorithme en erreur. Si le *malware* génère du code dynamiquement après l'OEP par exemple. Le lecteur intéressé est encouragé à tester le script, à l'améliorer et à le partager avec la communauté.

4.3 Unpacking du malware Dyre

Ce dernier exemple vise à montrer qu'il est possible de suivre des processus arbitraires pendant la phase d'unpacking ; ce qui peut se révéler particulièrement utile si le malware analysé crée un processus fils pour continuer son exécution. C'est le cas de Dyre, un malware bancaire qui a sévi en 2015 et dont on peut trouver une analyse sur Internet.

Dans les grandes lignes, ce malware utilise plusieurs couches d'unpacking à l'intérieur de son propre processus. La première couche de code est construite dans de la mémoire allouée dynamiquement. Cette couche déchiffre alors un PE contenu dans le binaire principal et le copie en lieu et place du PE original puis l'exécute. Finalement, il crée un processus fils, *svchost.exe*, suspendu, charge une section aditionnelle dans ce processus, modifie le point d'entrée de façon à ce qu'il saute dans cette section.

Grâce à un script de Log de Gunpack, on peut observer ce comportement d'un point de vue macroscopique :

```
...
Process (3272) allocates 0x1f000 bytes, address 0x1210000, rights 0x40
Process (3272) EXCEPTION write, address 0x1210693
Process (3272) EXCEPTION execute, address 0x1210000
Process (3272) self modifying code : 0x1210000
Process (3272) protects 0x1f000 bytes, address 0x400000, rights 0x40
Process (3272) allocates 0x1000 bytes, address 0x1230000, rights 0x40
Process (3272) EXCEPTION write, address 0x1230000
Process (3272) free 0x1000 bytes, address 0x1230000, free type 0x8000
Process (3272) allocates 0x3000 bytes, address 0x1230000, rights 0x40
Process (3272) allocates 0x1000 bytes, address 0x1240000, rights 0x40
Process (3272) allocates 0x1000 bytes, address 0x1250000, rights 0x40
Process (3272) allocates 0x1000 bytes, address 0x1260000, rights 0x40
Process (3272) EXCEPTION write, address 0x400000
Process (3272) EXCEPTION write, address 0x401000
Process (3272) EXCEPTION write, address 0x402000
Process (3272) EXCEPTION write, address 0x403000
Process (3272) EXCEPTION write, address 0x404000
...
Process (3272) EXCEPTION write, address 0x41b000
Process (3272) EXCEPTION write, address 0x41c000
Process (3272) EXCEPTION write, address 0x41d000
Process (3272) EXCEPTION write, address 0x41e000
Process (3272) EXCEPTION write, address 0x1260000
Process (3272) EXCEPTION write, address 0x1250000
Process (3272) EXCEPTION write, address 0x1230004
Process (3272) EXCEPTION write, address 0x1231000
Process (3272) EXCEPTION write, address 0x1232000
...
Process (3272) creates a child process (1936)
Process (1936) creates a thread (2756)
Process (3272) allocates 0x2000 bytes, address 0x2bc000, rights 0x4
Process (3272) EXCEPTION write, address 0x2bdff8
Process (3272) EXCEPTION write, address 0x2bc84a
Process (3272) load image \Windows\System32\apphelp.dll, base address 0x74f20000
Process (3272) maps 0x4c000 bytes in Pid 3272, address 0x74f20000, protect 0x4, alloc type
      0x0
Process (3272) protects 0x1000 bytes, address 0x74f21000, rights 0x4
Process (3272) EXCEPTION write, address 0x74f21000
Process (3272) protects 0x1000 bytes, address 0x74f21000, rights 0x20
Process (3272) EXCEPTION write, address 0x74f5d000
Process (1936) allocates 0x1000 bytes, address 0x50000, rights 0x4
Process (3272) allocates 0x6000 bytes, address 0x1230000, rights 0x40
Process (3272) maps 0x2000 bytes in Pid 3272, address 0x1240000, protect 0x40, alloc type
     0x0
Process (3272) EXCEPTION write, address 0x401355
Process (3272) self modifying code : 0x401355
```

```
Process (3272) EXCEPTION write, address 0x1240000
Process (3272) EXCEPTION write, address 0x1241000
Process (3272) EXCEPTION write, address 0x401061
Process (3272) self modifying code : 0x401061
Process (3272) maps 0x2000 bytes in Pid 1936, address 0x60000, protect 0x40, alloc type 0
     x0
Process (3272) EXCEPTION write, address 0x1232104
Process (3272) maps 0x8000 bytes in Pid 3272, address 0x1250000, protect 0x40, alloc type
     0x0
Process (3272) EXCEPTION write, address 0x1250000
Process (3272) EXCEPTION write, address 0x1251000
Process (3272) EXCEPTION write, address 0x1252000
Process (3272) EXCEPTION write, address 0x1253000
Process (3272) EXCEPTION write, address 0x1254000
Process (3272) EXCEPTION write, address 0x1255000
Process (3272) maps 0x8000 bytes in Pid 1936, address 0x810000, protect 0x40, alloc type 0
     x0
Process (3272) free 0x40000 bytes, address 0x12c0000, free type 0x8000
Process (3272) EXCEPTION write, address 0x6f3513f8
Process (3272) EXCEPTION execute, address 0x71721280
Process (3272) self modifying code : 0x71721280
Process (3272) terminates process (3272)
Process (1936) load image \Device\HarddiskVolume1\Windows\System32\svchost.exe, base
     address 0x810000
Process (1936) load image \SystemRoot\System32\ntdll.dll, base address 0x76ed0000
...
Process (1936) EXCEPTION write, address 0x601a0
Process (1936) self modifying code : 0x601a0
Process (1936) load image \Windows\System32\wininet.dll, base address 0x75390000
...
```

On peut observer dans cette trace :

- La création dynamique de la première couche, à l'adresse *0x1210000*
- La réécriture complète du PE des adresse *0x400000* à *0x41e000*
- L'exécution du nouveau PE au point d'entrée *0x401000*
- La création du processus fils (Pid 1936)
- Le chargement d'une section à l'adresse *0x60000* dans le processus fils par le père
- Le démarrage du processus fils *svchost.exe*
- L'exécution du code dans la section à l'adresse de base *0x601a0* qui charge des APIs web.

Il est possible de dumper les trois premiers *stages* du malware avec le script suivant :

```
class Dyre(Gunpack):

    def __init__(self, file_path):

        self.command_line = file_path
        self.set_rwe_policy(1)
        self.set_max_unpack_time(120)

        self.final_base_address = 0
        self.final_size = 0
        self.first_stage = 0
        self.new_pid = 0

    def event_handler(self, event_type, event_obj):
        if ( event_type == EVENT_EXCEPTION ):
            exception = event_obj
            if ( exception.AccessedType == WRITE_ACCESS ):
                ac_type = "write"
```

```
            if ( exception.AccessedType == EXECUTE_ACCESS ):
                ac_type = "execute"

            AccessedPageBase = exception.AccessedAddress >> 0xC
            EipPageBase = exception.Eip >> 0xC

            if ( exception.AccessedType == EXECUTE_ACCESS ):
                #Dump first stage once executed
                if ( exception.AccessedAddress ==  self.first_stage ):
                    self.process.DumpBufferToFile(exception.ProcessId, "first_stage.bin",
                         self.first_stage, 0x1f000)
                #Dump second stage as a PE, when OEP is executed once again
                if ( exception.AccessedAddress == 0x401000 ):
                    self.process.DumpPE(self.process.pid,0x401000,"second_stage.exe")

        #Track child process
        if ( event_type == EVENT_CREATE_PROCESS ):
            process = event_obj
            if (process.Create != 0):
                self.add_tracked_pid(process.ProcessId)
                self.new_pid = process.ProcessId
                return 1

        if ( event_type == EVENT_LOAD_LIBRARY ):
            library = event_obj
            lib_name = library.DllName
            #Dump third stage when wininet is loaded
            #and then stop analysis
            if ( lib_name.find("System32\\wininet.dll") != -1 ):
                self.process.DumpBufferToFile(event_obj.ProcessId, "third_stage.bin", self
                     .final_base_address, self.final_size)
                return 0

        if ( event_type == EVENT_VIRTUALALLOC ):
            allocation = event_obj
            if ( allocation.result == 0 ):
                #Remember address of this big allocation
                if ( allocation.RegionSize == 0x1f000 and allocation.Protect == 0x40 ):
                    self.first_stage = allocation.BaseAddress

        if ( event_type == EVENT_MAPVIEWOFSECTION ):
            section = event_obj
            if ( section.result == 0 ):
               #Remember first section mapped in child process
               if ( section.ProcessId == self.process.pid and self.new_pid == section.
                    TargetPid ):
                    if (self.final_base_address == 0):
                        self.final_base_address = section.BaseAddress
                        self.final_size = section.ViewSize

        return 1

    def post_treatment(self):

        None
```

L'utilisation de la méthode *add_tracked_pid* de Gunpack permet d'ajouter un processus à la liste des processus monitorés à tout moment de l'analyse. Cet exemple mériterait d'être complété, en particulier si on souhaite extraire du troisième *stage* les informations utiles de ce malware : domaines C&C, etc.

5 Conclusion

Notre outil n'est pas parfait et est toujours en développement. Il ne faut pas le voir comme un outil clé en main permettant d'unpacker n'importe quel malware mais plutôt comme un allié pouvant faciliter la tâche. Il ne

dispense pas d'une première étape d'analyse du *packer* ou du *malware* qu'on souhaite analyser mais peut permettre d'en automatiser l'analyse.

Références

1. Nick Cano. *Packer Attacker*. https ://github.com/BromiumLabs/PackerAttacker, 2015.
2. Fanglu Guo , Peter Ferrie , Kent Griffin , Tzi cker Chiueh. *A Study of the Packer Problem and Its Solutions*. RAID, 2008.
3. Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual*. http ://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.
4. Alex Ionescu. *Simple Visor*. `https://github.com/ionescu007/SimpleVisor`.
5. Lorenzo Martignoni , Mihai Christodorescu , Somesh Jha. *OmniUnpack : Fast, Generic, and Safe Unpacking of Malware*. Computer Security Applications Conference, 2007.
6. Artem Dinaburg , Paul Royal , Monirul Sharif , Wenke Lee. *Ether : Malware Analysis via Hardware Virtualization Extensions*. CCS, 2008.
7. Markus F.X.J. Oberhumer, László Molnár , John F. Reiser. *Ultimate Packer for eXecutables*. `http://upx.sourceforge.net/`.
8. Xin Hu , Sandeep Bhatkar , Kent Griffin , Kang G. Shin. *MutantX-S : Scalable Malware Clustering Based on Static Features*. USENIX Annual Technical Conference, 2013.
9. Microsoft Windows. *Structured Exception Handling*. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms679270(v=vs.85).aspx`.
10. Microsoft Windows. *Vectored Exception Handling*. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx`.
11. Min Gyung Kang , Pongsin Poosankam, Heng Yin. *Renovo : A Hidden Code Extractor for Packed Executables*. Workshop on Recurring Malcode, 2007.