

Techniques utilisées par les rootkits

Etudiant : **Florent Wenger** florent_wenger@hotmail.com

Professeur : Gérald Litzistorf

Laboratoire de transmission de données, novembre 2006

En collaboration avec la société **ellisys**

TABLE DES MATIERES

1. PREFACE	1
1.1. ENONCE	1
1.1.1. DESCRIPTIF	1
1.1.2. TRAVAIL DEMANDE	1
2. FAMILIARISATION AVEC WINDOWS	2
2.1. PREAMBULE	2
2.1.1. INTRODUCTION	2
2.1.2. COMPLEXITE VS SECURITE	2
2.1.3. CADRE DE L'ETUDE	3
2.2. PREMIERE APPROCHE	3
2.2.1. SYSTEME D'EXPLOITATION	3
2.2.2. L'API WINDOWS	5
2.2.3. CONCEPTS FONDAMENTAUX	5
2.3. FORMAT PE, OCCUPATION MEMOIRE ET DLLS	7
2.3.1. FORMAT PE	7
2.3.2. IMPORT ADDRESS TABLE	12
2.3.3. OCCUPATION MEMOIRE : THEORIE	16
2.3.4. OCCUPATION MEMOIRE : PRATIQUE	19
2.3.5. DEPENDANCES	23
2.4. PROCESSUS, THREADS ET HANDLES	25
2.4.1. PROCESSUS	25
2.4.2. THREADS	28
2.4.3. HANDLES	30
2.4.4. FICHIERS MAPPEES ET MEMOIRE PARTAGEE	32
2.5. MODELE DE SECURITE	33
2.5.1. UTILISATEUR ET JETON D'ACCES	33
2.5.2. DROITS, HANDLES ET ACLS	35
2.5.3. PRIVILEGES	36
2.5.4. AUTRES MECANISMES : SRP, DEP	39
2.5.5. SOURCES	42
2.6. CONCLUSION	42
3. TECHNIQUES UTILISEES PAR LES ROOTKITS	43
3.1. QU'EST-CE QU'UN ROOTKIT?	43
3.1.1. DEFINITION	43
3.1.2. TYPES DE ROOTKITS	43
3.1.3. LE ROOTKIT NTILLUSION	44

3.2. GENERALITES	44
3.2.1. MOTIVATION	44
3.2.2. EVOLUTION DE LA TECHNIQUE	45
3.2.3. METHODE STATIQUE OU DYNAMIQUE?	45
3.2.4. PIRATAGE DE L'API	46
3.3. TECHNIQUES FONDAMENTALES	47
3.3.1. LOCALISATION DES PROCESSUS	47
3.3.2. VUE D'ENSEMBLE DES HOOKS	48
3.4. TECHNIQUES D'INJECTION	50
3.4.1. PROBLEMATIQUE DU <i>MAPPING</i>	50
3.4.2. INJECTION DE DLL VIA LA BASE DE REGISTRE	52
3.4.3. INJECTION DE DLL GRACE AUX HOOKS	53
3.4.4. INJECTION DE DLL AVEC <i>CREATEREMOTEThread & LOADLIBRARY</i>	53
3.4.5. INJECTION DE CODE AVEC <i>WRITEPROCESSMEMORY</i>	56
3.5. TECHNIQUES D'INTERCEPTION	56
3.5.1. A QUEL NIVEAU?	56
3.5.2. MODIFICATION DE L'IAT	58
3.5.3. <i>INLINE PATCHING</i>	61
3.5.4. EXPERIENCE AVEC DETOURS ET <i>WINDBG</i>	63
3.5.5. REDIRECTION DE DLL	67
3.6. TECHNIQUES ADDITIONNELLES	68
3.6.1. MODULE FANTOME	68
3.6.2. CONTRE LE <i>REVERSING</i>	69
3.6.3. MESURES ET CONTRE-MESURES	70
3.7. ET SOUS WINDOWS VISTA?	70
3.7.1. 1 ^{ER} TEST : TERMINAISON DES PROCESSUS	71
3.7.2. 2 ^{EME} TEST : MODIFICATION DE L'IAT	71
3.7.3. 3 ^{EME} TEST : <i>INLINE PATCHING</i> AVEC DETOURS	71
3.7.4. 4 ^{EME} TEST : INTERCEPTION DE MOTS DE PASSE	72
3.7.5. BILAN	74
4. POSTFACE	75
4.1. CONCLUSION	75
4.2. REMERCIEMENTS	76
4.3. REFERENCES	76

1. Préface

1.1. Enoncé

1.1.1. Descriptif

La compréhension des techniques d'attaque sur Windows peut être un moyen de mieux lutter contre certains *malwares*.

Dans un but didactique et en complément au labo Windows Forensics proposé par le laboratoire, l'étudiant va analyser diverses techniques (*hook, injection dll, ...*) utilisées par les rootkits en mode *user*.

Il doit identifier les outils (*debugger, ...*) à utiliser afin de construire des scénarios pédagogiques basés sur des exemples (code machine) précis complètement documentés.

Il comparera les résultats obtenus sous Windows XP SP2 et sous Windows Vista.

1.1.2. Travail demandé

Ce travail se compose des parties suivantes :

1) Familiarisation

5 semaines

Etudier l'architecture logicielle (module, *thread, handle, dll, ...*) ainsi que l'occupation mémoire.

Etudier les mécanismes de chargement (*loader, format PE, ...*) .

Etudier les mécanismes de protection mémoire (*private & not shareable page, shared memory & mapped files, data execution prevention*) ainsi que le modèle de sécurité (*token, privileges*).

2) Techniques utilisées par les rootkits

5 semaines

Etudier diverses techniques (*IAT patching, inline patching, dll injection, code injection, ...*).

Choisir des exemples présentant des risques importants pour l'utilisateur.

Mettre en œuvre.

3) A choix (hors mémoire)

2 semaines

Contournement d'un anti-virus / d'un firewall personnel / de Windows File Protection – Sécurité avec .NET 3.0 / ...

Sous réserve de modifications en cours de travail.

Liens

<http://www.rootkit.com>

<http://www.uninformed.org/>

http://www.phrack.org/62/p62-0x0c_Win32_Portable_Userland_Rootkit.txt

Gérald LITZISTORF, sept. 06

Travail en collaboration avec [Ellisys](#) (société créée par des diplômés du laboratoire).

2. Familiarisation avec Windows

2.1. Préambule

2.1.1. Introduction

La 1^{ère} partie de ce travail a pour but de se familiariser avec les concepts et les mécanismes fondamentaux du fonctionnement interne de Windows. Rédigée dans une perspective didactique, elle devrait permettre à un étudiant en fin d'études HES en filière informatique de s'initier aux fondements de ce système d'exploitation.

Le point de départ incontournable à ce sujet est le livre *Windows Internals*, écrit par Mark Russinovich, en parallèle avec la documentation officielle MSDN. Dans tous les cas, les références données indiqueront nos documents source pour vérification ou approfondissement.

Nous accompagneront notre propos par des expériences avec divers utilitaires existants, afin de montrer ce qui se passe vraiment dans le cœur de Windows. Nous incluons également des fragments de code C++ ou assembleur qui pourront servir de base à un développement ultérieur à l'aide, par exemple, de Microsoft Visual Studio 2005.

2.1.2. Complexité vs Sécurité

Tout d'abord, essayons d'avoir une vue globale de la situation. Windows est un **système d'exploitation** (*operating system*, OS), donc une suite logicielle chargée de gérer les ressources matérielles et logicielles d'un PC. La complexité de Windows vient du fait qu'il est :

<i>Caractéristique</i>	<i>Signification</i>
Multitâche	Plusieurs processus peuvent s'exécuter concurremment sur un même processeur
Multiprocesseur	Windows XP Pro prend en charge 1 ou 2 processeurs, (d'autres versions jusqu'à 64)
Multiutilisateur	Plusieurs sessions utilisateur peuvent être ouvertes simultanément

De plus, Windows XP descend de la famille NT. Il doit, dans une certaine mesure, assurer la compatibilité avec les versions antérieures. Ajoutons à cela la multiplicité d'architectures matérielles (32/64 bits) et d'environnements sous-système (Windows, POSIX).

Il y a encore les problématiques de l'internationalisation qui introduit des différences régionales et linguistiques, de la mise à jour qui permet un nombre théoriquement infini de combinaisons de versions installées des composants (DLLs, EXEs, etc.) et de la configuration qui est plus que déterminante pour la sécurité.

La complexité est l'ennemie de la sécurité : un tel OS est un casse-tête non seulement à comprendre, mais aussi à sécuriser. En effet, un logiciel malveillant (*malware*) peut exploiter la plus petite faille qui se trouve peut-être dans un recoin méconnu du système.

Heureusement pour nous, les *hackers* cherchent généralement des techniques d'attaque fonctionnant sur le plus grand nombre de machines semblables. Les risques les plus probables touchent donc les composants les plus répandus. Par conséquent, il vaut la peine d'étudier les concepts fondamentaux de Windows.

2.1.3. Cadre de l'étude

Ayant aperçu combien le domaine est à la fois vaste et complexe, il est nécessaire de bien délimiter le cadre de notre étude :

MS Windows XP Professional Service Pack 2, version anglaise

Architecture 32 bits x86, monoprocesseur

L'ordinateur n'appartient pas à un domaine

La plupart des observations qui suivent s'appliquent également à d'autres plates-formes, c'est-à-dire à d'autres architectures (p. ex. 64 bits ou multiprocesseur) et d'autres versions de Windows. Pour s'en assurer, se reporter au livre de référence [WI] qui ne manque pas d'indiquer les éventuelles différences.

Avant d'entrer dans le vif du sujet, il faut encore dire que notre préoccupation est la sécurité et non les performances. De plus, puisque nous nous intéressons à des techniques employées en mode utilisateur, nous ferons souvent abstraction du fonctionnement interne de Windows pour nous concentrer sur le point de vue d'une application lancée par un utilisateur limité.

2.2. Première approche

Cette partie s'inspire fortement des chapitres 1 et 2 de [WI], ainsi que du chapitre 3 de [REV] qu'il est recommandé de lire entièrement avant de passer à la suite.

2.2.1. Système d'exploitation

L'OS exécute les tâches fondamentales telles que :

- Allocation de la mémoire principale (RAM)
- Ordonnancement des tâches
- Contrôle des entrées / sorties
- Communication via le réseau
- Gestion du système de fichiers
- Affichage graphique et système de fenêtrage

L'OS gère l'accès aux ressources de la machine et constitue un passage obligé vers le matériel. Il faut se rappeler que Windows s'appuie sur deux modes d'exécution de code qui sont implémentés matériellement dans le processeur (Intel ou compatible) :

- Un mode noyau (kernel ou ring-0), qui est privilégié et permet d'exécuter n'importe quelle instruction.
- Un **mode utilisateur** (*user* ou ring-3) ou mode non privilégié, qui restreint la liberté du programme.

L'architecture de Windows est composée de multiples couches, dont nous voyons les principales à la Figure 1 ci-dessous.

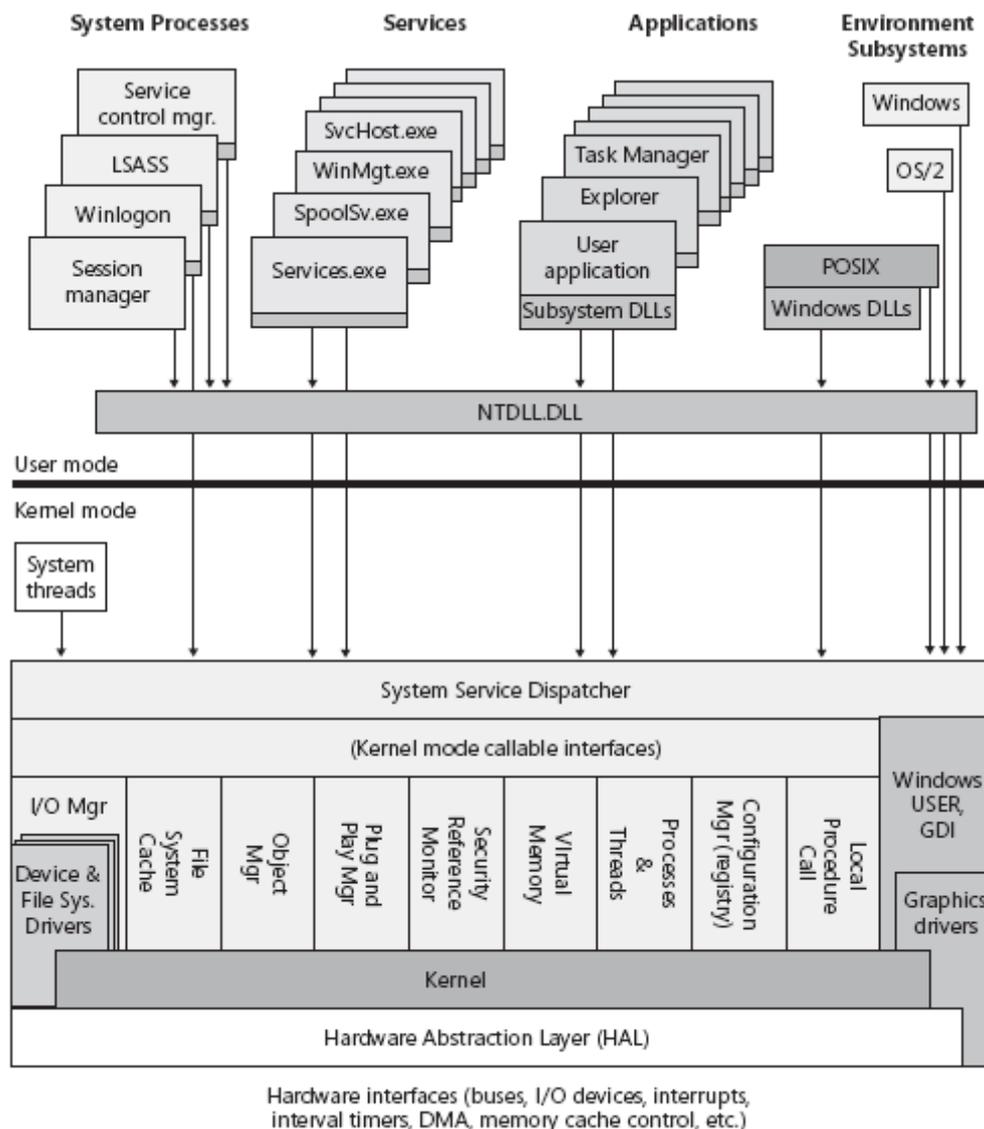


Figure 1: Architecture en couches de Windows

© 2005 by D. Solomon & M. Russinovich

Le noyau, l'exécutif Windows et les pilotes de périphériques tournent en mode noyau dans le même espace d'adresses. Un pilote peut donc consulter (et éventuellement corrompre) toutes les structures de données de l'OS.

Il faut être bien conscient de cette barrière entre le mode *user* et le mode *kernel*. Une application utilisateur emploie les services de l'OS et

manipule les ressources du système à travers des interfaces de programmation déterminées par Microsoft (API). Selon le principe de l'encapsulation, ce n'est pas à l'utilisateur de se préoccuper quel composant de l'OS fait quoi et comment, mais plutôt qu'est-ce qui lui est mis à disposition par l'intermédiaire de l'API.

2.2.2. L'API Windows

L'API (*Application Programming Interface*) Windows est l'interface de programmation du système. Sa version 32 bits est appelée API Win32, tandis que l'API Windows regroupe aussi les architectures 64 bits. Chaque version de l'OS implémente un sous-ensemble différent de l'API Windows et met ainsi à disposition plusieurs milliers de fonctions. Il faut bien distinguer :

- Les **fonctions de l'API** Windows, sous-programmes documentés et appelables. Par exemple : les fonctions `CreateProcess`, `CreateFile` et `GetMessage`.
- Les **fonctions natives** du système: non documentées mais appelables en mode utilisateur, ces sous-programmes sont sous-jacents à l'API Windows. Par exemple, la fonction Windows `CreateProcess` appelle le service interne `NtCreateProcess`, qui transmet l'appel au noyau avec un passage en mode kernel.

Bien qu'une application puisse appeler directement tant les fonctions de l'API que les fonctions natives, ces dernières sont à éviter car elles sont sujettes à des modifications. Nous retrouvons le principe de l'encapsulation : seules les fonctions de l'API Windows sont garanties (nous en reparlerons au § 3.5.1).

2.2.3. Concepts fondamentaux

Posons quelques définitions primordiales. Nous reviendrons en détails sur tous ces concepts l'un après l'autre par la suite.

Un **programme** (*program*) est une séquence statique d'instructions. On peut exécuter plusieurs **instances** du même programme simultanément en mémoire, alors qu'il n'y a qu'une seule **image** (fichier) sur le disque.

Un **processus** (*process*) est un conteneur stockant les ressources utilisées lors de l'exécution de l'instance d'un programme. Un processus Windows comporte¹ :

1. Un **espace d'adresses virtuelles ou EAV** (*virtual address space*). Le mécanisme de mémoire virtuelle donne l'illusion au processus de disposer d'un grand espace d'adresses pour lui seul, comme s'il était le seul à s'exécuter en mémoire. De plus, ce mécanisme isole les processus entre eux. Telle adresse d'un processus peut désigner une structure de données quelconque, tandis que la même adresse

¹ Cf. [WI] p. 6

dans un autre processus peut désigner tout autre chose, ou bien être invalide. Ainsi, les processus n'entrent pas en collision l'un avec l'autre et ne peuvent pas non plus corrompre les données du système d'exploitation.

2. Un programme exécutable. Il définit le code et les données de départ. Il est inséré (*mapped*) dans l'espace d'adresses virtuelles. Son chargement en mémoire est effectué par le chargeur de programme (*loader*).
3. Un ou plusieurs threads d'exécution (voir ci-dessous).
4. Une liste des ressources, telles que sémaphores, ports de communication et fichiers, utilisées par le programme et accessibles à tous les threads. Ces ressources ne sont pas manipulables directement par leur adresse, mais à travers le système d'exploitation par un numéro appelé *handle* (voir § 2.4.3).
5. Un identificateur unique appelé *process ID* ou PID (nommé *client ID* dans les structures internes de Windows).
6. Le PID de son père, c'est-à-dire du processus qui l'a démarré.
7. Un contexte de sécurité nommé jeton d'accès (*access token*) qui identifie l'utilisateur, les groupes de sécurité ainsi que les droits associés au processus.

Un **module** est une instance d'une image en mémoire ; nous pouvons trouver dans la RAM plusieurs instances du même programme. Il peut s'agir d'un exécutable (fichier EXE), d'une DLL ou autre (NLS, OCX, SCR, etc.).

Un **thread** modélise un flux de contrôle exécutant un programme. Il appartient à un processus qui peut en contenir plusieurs. Tous les threads d'un même processus ont accès à toutes les ressources du processus et partagent le même espace d'adresses virtuelles. C'est l'ordonnanceur (*scheduler*) de Windows qui répartit le(s) processeur(s) entre les threads, i.e. décide où, à quel moment et jusqu'à quand chaque thread peut s'exécuter. Un thread Windows englobe² :

1. L'état du processeur (**contexte**), c'est-à-dire le contenu d'un ensemble de registres de ce dernier. Il contient en particulier le compteur ordinal (*program counter*), qui pointe vers la prochaine instruction à exécuter.
2. Deux piles, pour l'exécution du thread respectivement en mode *kernel* et en mode *user*.

² Cf. [WI] p. 12

3. Une zone de stockage privé appelé *thread-local storage* (TLS), pour être utilisée par les sous-systèmes³ (Windows, OS/2, POSIX), les bibliothèques d'exécution et les DLLs.
4. Un identificateur unique appelé *thread ID* (également nommé *client ID* dans les structures internes de Windows).
5. Parfois, un thread dispose de son propre contexte de sécurité.

Une **DLL** (*dynamic-link library* ou *dynamically linked library*) est une :

- Bibliothèque (*library*), i.e. un ensemble de routines utilisées pour développer du logiciel. Une bibliothèque contient du code et des données qui fournissent des services à des programmes indépendants. Ce code et ces données peuvent ainsi être partagées et modifiées de façon modulaire (on peut modifier leur implémentation en conservant l'interface externe).
- Liée : un lien est simplement une référence entre un exécutable et une bibliothèque, permettant au premier d'utiliser la seconde.
- Dynamiquement : la bibliothèque est présente une seule fois sur le disque dur et en mémoire, économisant ainsi de l'espace. Par opposition, une bibliothèque statique est insérée dans chaque exécutable qui l'utilise par l'éditeur de liens (*linker*). Un programme peut référencer une DLL lors de son chargement (*loadtime*) ou de son exécution (*run-time*).

Le système d'exploitation fournit la majorité de ses services sous forme de DLLs.

2.3. Format PE, occupation mémoire et DLLs

Pour présenter les diverses notions à aborder, nous allons étudier un cas pratique, depuis son stockage sur le disque jusqu'à son exécution en mémoire. L'application choisie est le Wordpad, qui se trouve dans "%ProgramFiles%\Windows NT\Accessories\wordpad.exe".

2.3.1. Format PE

Il faut d'abord étudier l'**image** au format PE, c'est-à-dire le fichier tel qu'il est stocké sur le disque. C'est instructif car, comme le montre la Figure 2, on retrouve la structure du fichier PE (en vert) dans la mémoire (en violet). Les mêmes sections y sont présentes dans le même ordre, elles sont simplement alignées (espacées) différemment.

³ Pour les sous-systèmes, voir [WI] p. 53ss. Nous considérons uniquement le sous-système Windows.

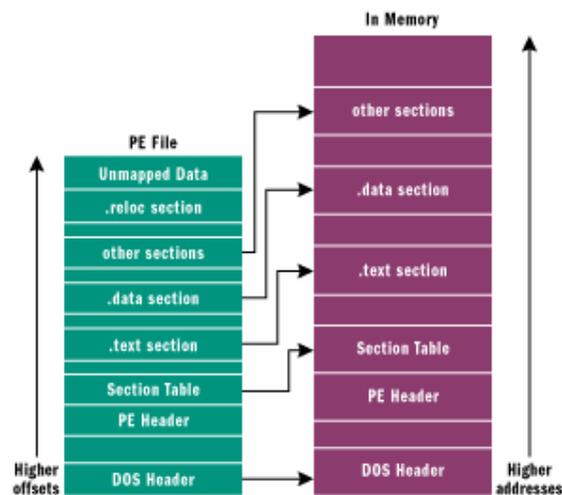


Figure 2 : Fichier PE en mémoire

© 2002 by Marc Pietrek

Sources

Eldad Eilam donne un bon survol du format PE dans [REV] pp. 93-102.

Pour un aperçu plus détaillé, se référer à l'article en 2 parties de Mark Pietrek intitulé "An In-Depth Look into the Win32 Portable Executable File Format" :

1^{ère} partie : <http://msdn.microsoft.com/msdnmag/issues/02/02/PE/>

2^{ème} partie : <http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/>

Enfin, les spécifications officielles des formats PE & COFF (structure détaillée et valeurs des champs) se trouvent à l'URL suivante :

<http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.msp>

Outils

- PE Browse Professional, de SmidegonSoft (fait aussi office de désassembleur) : <http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html>
- LordPE⁴ écrit par y0da : <http://y0da.cjb.net/>
- Une liste d'outils (dont PESam) pour manipuler les fichiers PE : <http://programmerstools.org/taxonomy/term/56>

Analysons le fichier PE wordpad.exe à l'aide de PE Browse Pro. Nous passons sur l'en-tête DOS (*DOS Header*), qui se contente d'afficher le message d'erreur "This program cannot be run in DOS mode" si l'exécutable est démarré depuis MS-DOS. Dans la partie gauche (Figure 3) se trouve les différentes parties du fichier. Elles peuvent être

⁴ Installation : extraire LordPE Deluxe (<http://scifi.pages.at/yoda9k/LordPE/LPE-DLX.ZIP>) puis remplacer le fichier LordPE.exe par celui contenu dans la micro-update depuis http://scifi.pages.at/yoda9k/LordPE/LPE-DLXb_UPD.zip

parcourues en choisissant *Dump* (données brutes), *Details* ou encore *Structure* dans le menu contextuel de chaque entrée.



Figure 3 : En-tête de fichier (PEBrowse Pro)

Regardons les détails de l'**en-tête de fichier** COFF. Nous voyons dans *Number of Sections* la présence de 3 sections, et dans *Characteristics* les attributs suivants :

- "*File is executable*": le fichier est un exécutable (EXE), par opposition à une DLL qui n'est pas conçue pour être exécutée seule.
- "*Relocation info stripped...*" : les informations de **relogement** (*relocation*) ne sont pas présentes. Ces informations sont nécessaires lorsque le module est chargé à une adresse différente de son **adresse de départ favorite** (ADF) ; on dit qu'il est relogé. Puisqu'en principe l'exécutable est chargé en premier dans l'EAV, il est chargé à son ADF. Par contre, ce n'est pas toujours le cas pour les DLLs.
- "*Line number / Local symbols stripped...*" : le format PE dérive du format COFF qui concerne les fichiers objets, produits de la compilation. Certains champs (table de symboles, numéro de ligne) ne sont pas pertinents pour les fichiers PE.

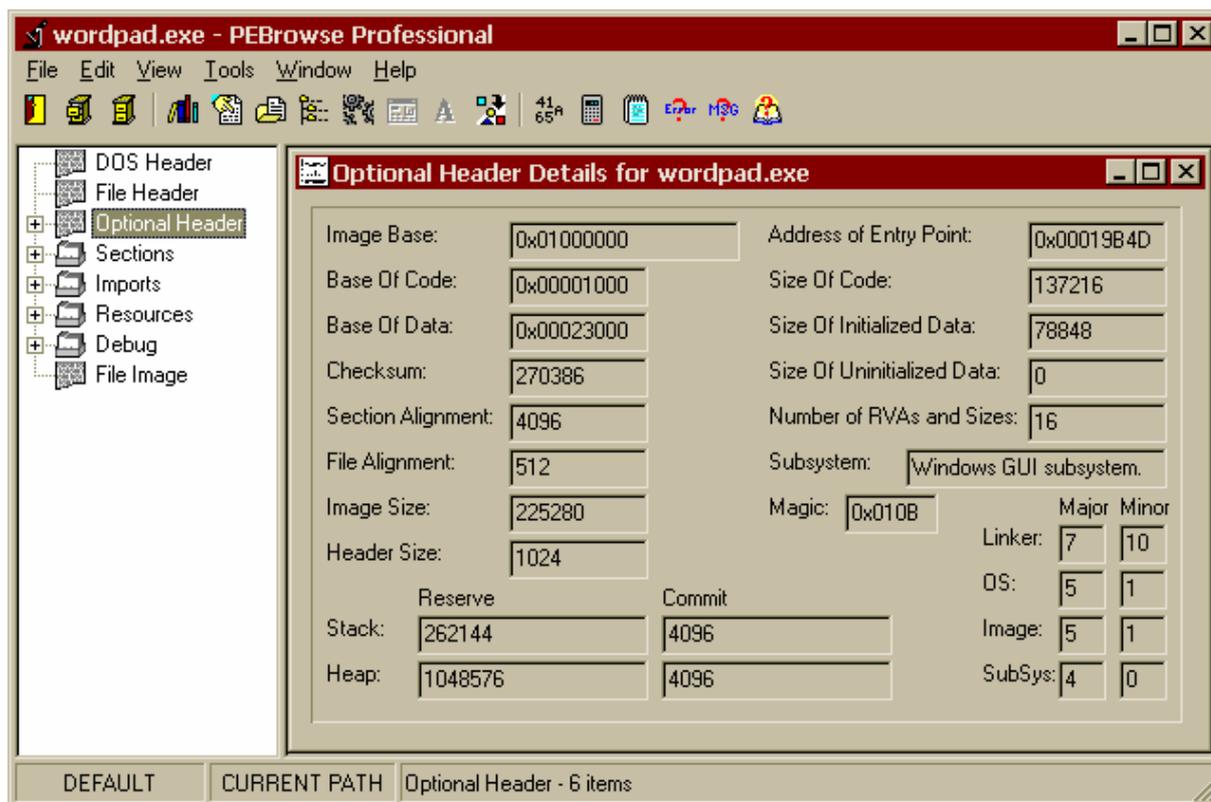


Figure 4 : En-tête facultatif (PEBrowse Pro)

L'**en-tête facultatif** (*optional header*) fournit des informations au chargeur de programmes, dont (Figure 4) :

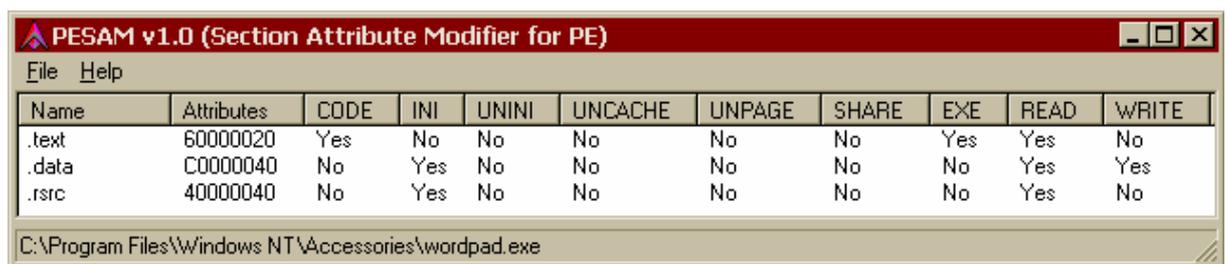
- "**Image Base**" : valeur de l'adresse de chargement favorite.
- "**Address of Entry Point**" : l'adresse du point d'entrée, c'est-à-dire de la 1^{ère} instruction machine à exécuter. Ce champ est facultatif pour les DLLs et peut valoir zéro.
- "**Checksum**" : il s'agit d'une somme de contrôle d'intégrité⁵ du fichier image. Sont vérifiés au chargement le *checksum* de tous les pilotes (mode *kernel*), de toute DLL chargé au démarrage (*boot*) et de toute DLL chargée dans un processus Windows critique. En bref, Windows ne contrôle pas l'intégrité de toutes les images.
- "**Subsystem**" : nous apprend que cette image est prévue pour le sous-système Windows (et non POSIX ou autre) et que l'interface homme-machine est en mode graphique (GUI = *Graphical User Interface*) par opposition au mode caractère (appelé par Microsoft CUI = *Console User Interface*).

⁵ Les spécifications ne mentionnent pas l'algorithme employé. Les fonctions de calcul de *checksum* sont incorporées à la DLL *imaghelp.dll*, documentée ici : <http://msdn2.microsoft.com/en-us/library/ms680181.aspx>

- "Stack / Heap Reserve & Commit" : quantités de mémoire réservées et "engagées"⁶, respectivement pour la pile (*stack*) et le tas (*heap*).

Voyons maintenant les **sections**, qui sont les unités fondamentales de code ou de données du fichier PE. Leur nom est libre même si l'éditeur de liens Microsoft attribue des noms éloquents débutant par un point. La section `.text` contient habituellement le code et les données, la section `.data` les variables (initialisées ou non) et la section `.rsrc` les ressources (menus, icônes, bitmaps, curseurs, boîtes de dialogues, etc.).

Chaque section a ses propres **attributs** (*characteristics*) qui indiquent notamment si elle contient du code, est accessible en lecture et/ou écriture, est exécutable ou est partageable. La Figure 5 montre les attributs des 3 sections de `wordpad.exe`. Dans notre cas, aucune de ces sections n'est initialement partageable avec d'autres processus.



Name	Attributes	CODE	INI	UNINI	UNCACHE	UNPAGE	SHARE	EXE	READ	WRITE
.text	60000020	Yes	No	No	No	No	No	Yes	Yes	No
.data	C0000040	No	Yes	No	No	No	No	No	Yes	Yes
.rsrc	40000040	No	Yes	No	No	No	No	No	Yes	No

Figure 5 : Attributs de sections (PESam)

Pour comprendre le mécanisme de chargement, il faut bien distinguer entre une adresse virtuelle relative (**RVA** = *relative virtual address*) en mémoire et un pointeur (**offset**) vers les données de l'image sur le disque. Dans le cas d'une valeur relative, il faut toujours se demander quel est le point de référence : début du fichier, ADF, début de section.

Sans parler des contraintes sur la taille des sections, remarquons juste que leur taille en mémoire (*VirtualSize*) peut dépasser celle sur le disque (*Size of Raw Data*). Le complément est simplement rempli avec des zéros ; il pourrait correspondre à des variables non-initialisées.

⁶ [WI] p. 384 : la mémoire virtuelle d'un processus est divisée en pages qui peuvent être libres (*free*), réservées (*reserved*) ou engagées (*committed*). Réserver une page revient seulement à réserver une plage d'adresses consécutives ; on ne peut pas encore y accéder. Une fois que la mémoire est *committed*, on peut y accéder, i.e. il y a une traduction possible entre mémoire virtuelle et mémoire physique.

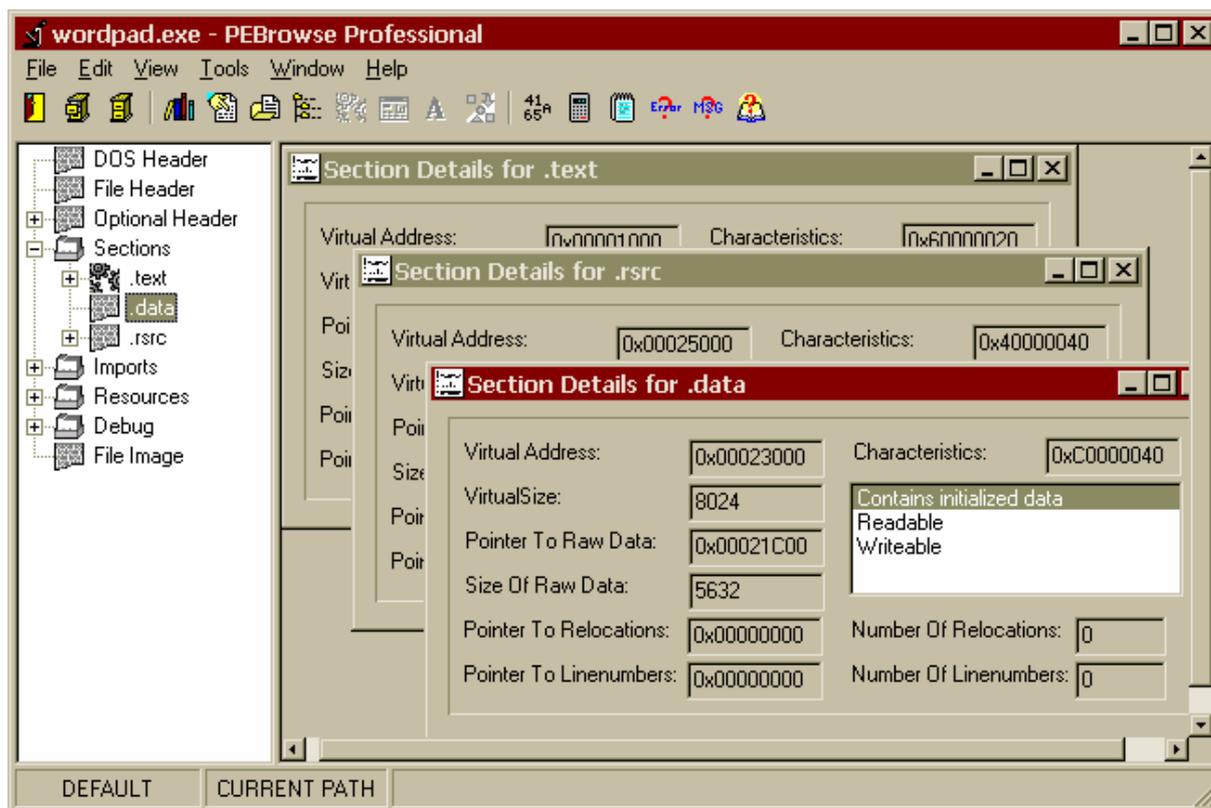


Figure 6 : Détails de la section .data (PEBrowse Pro)

Laissant de côté les ressources et les informations facultatives de débogage (*debugging information*), voyons encore la table d'imports, qui est d'une importance capitale.

2.3.2. Import Address Table

Un module peut employer une fonction externe, respectivement mettre à disposition une de ses fonctions grâce au mécanisme d'imports / exports. Un symbole exporté est identifié par son nom ou un nombre ordinal. Typiquement, les exécutables n'exportent pas de fonctions mais se contentent d'en importer depuis diverses DLLs.

A priori, une application ne peut pas être sûre de l'adresse à laquelle une DLL est chargée. Pour utiliser une fonction importée, un module doit connaître l'adresse virtuelle (VA) de cette fonction à l'intérieur de la DLL, qu'il peut ensuite appeler via un **pointeur de fonction**. Il faut à un moment ou un autre opérer la liaison entre le module utilisateur et le module fournisseur : c'est ce qu'on appelle la *binding*.

Il existe plusieurs types de *binding*⁷ qui peuvent être mis en œuvre conjointement pour les applications :

⁷ Il y a encore une autre forme d'import appelé "*forward*" dans le cas où une DLL met à disposition une fonction d'une DLL tierce.

1. "**Explicit import**" : c'est l'application qui demande explicitement l'adresse des fonctions importées. Nous trouvons dans le code des appels aux fonctions de l'API LoadLibrary et GetProcAddress.
2. "**Implicit import**" : le *binding* est effectué implicitement lors du chargement en mémoire. C'est le chargeur de programmes Windows qui appelle automatiquement les fonctions mentionnées en 1.
3. "**Bound import**" : c'est une variante de 2. On fait l'hypothèse que la DLL de la version connue à la compilation sera bien chargée à son ADF et on code en dur l'adresse de ses fonctions utilisées dans l'image de l'EXE lors de l'édition de liens.

Le chargeur de programme vérifie l'hypothèse lors du chargement de l'EXE. Si elle est infirmée (adresse de départ ou version différentes), les pointeurs de fonction sont faux et il faut procéder comme en 2. Si elle est confirmée, le *binding* est déjà fait, ce qui gagne du temps.

4. "**Delayload import**" : pour améliorer les performances, le *binding* est effectué juste avant le premier appel à la fonction importée. Cette variante, implémentée dans le CRT (C runtime), est transparente pour Windows.

Ce type d'import est aussi appelé *lazy binding*, car on ne fait les choses qu'au dernier moment. Il a pour but d'accélérer le chargement d'une application (qui peut être long en raison du chargement récursif des DLLs, voir dépendances § 2.3.5).

Ce qu'il est important de comprendre, c'est que dans tous les cas à l'exception du 1, l'adresse de la fonction importée est stockée⁸ dans une table appelée **IAT** (*import address table*).

En fait, la variante 4 utilise une autre table similaire : la DIAT. Mais dans les cas 2, 3 et 4, tous les appels à une fonction importée passent par un pointeur de fonction stocké dans une table. Voyons comment à la Figure 7 ci-dessous.

⁸ Attention! Il faut distinguer l'IAT sur le disque de l'IAT en mémoire. Selon le type d'import, ses pointeurs de fonctions sont soit chargés depuis le fichier image, soit initialisés dynamiquement.

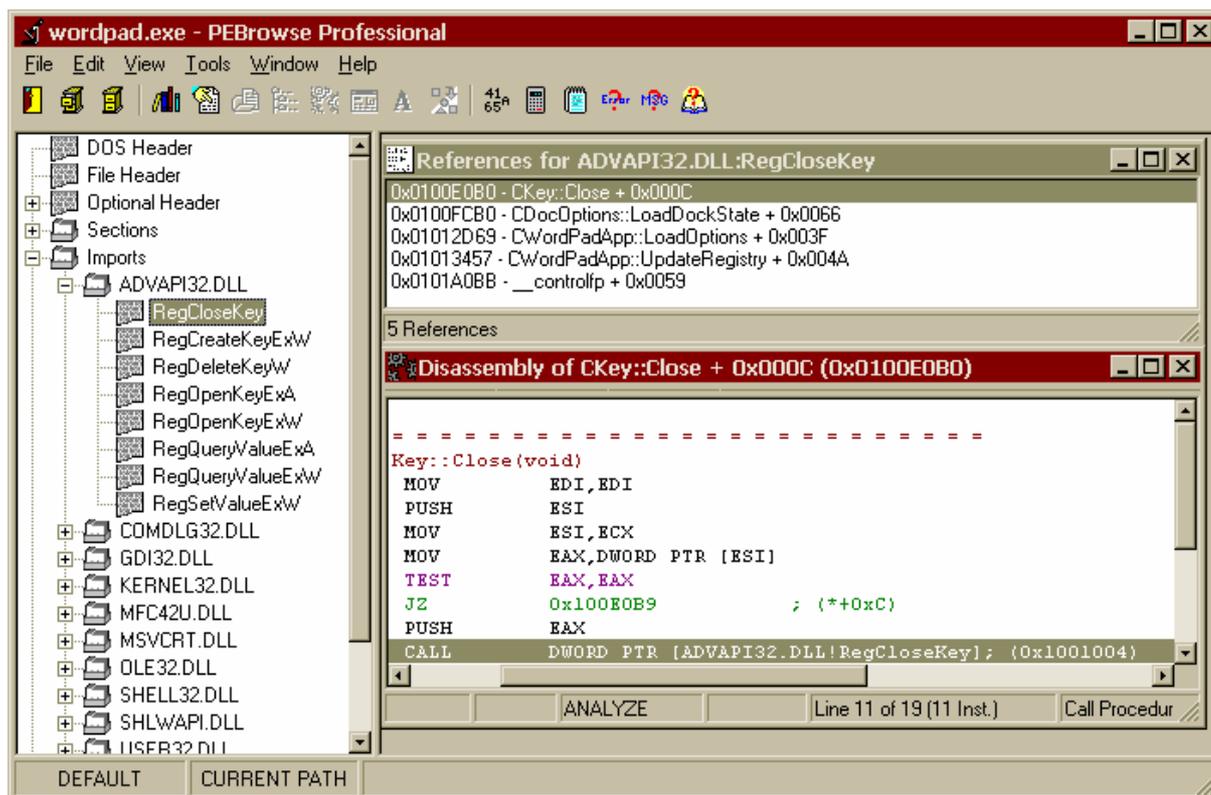


Figure 7 : Appel à une fonction importée (PEBrowse Pro)

Nous constatons dans la partie gauche que wordpad.exe référence des fonctions provenant de 10 DLLs, listée dans l'arbre des imports. Prenons l'exemple de la fonction ADVAPI32.DLL!RegCloseKey (qui ferme une clé de la base de registres). Nous pouvons dénombrer dans la fenêtre en haut à droite 5 références à cette fonction, c'est-à-dire que le code contient 5 appels vers cette fonction.

En désassemblant le code de ces appels (en bas à droite), nous constatons qu'il est toujours identique:

```
CALL DWORD PTR [ADVAPI32.DLL!RegCloseKey]
```

Cette instruction machine en fait appelle le pointeur de fonction stocké dans une certaine entrée de l'IAT (à l'adresse 0x0100'1004 dans notre cas). Modifier ce seul pointeur permet donc de dérouter tous les appels à cette fonction importée!

En anticipant sur la suite, indiquons simplement comment afficher le contenu de l'IAT en mémoire. Il nous faut trouver son adresse en mémoire, en ajoutant la RVA de l'IAT à la valeur de l'ADF de la DLL. Nous avons déjà vu comment extraire l'ADF ou *base address*. Pour trouver la RVA de l'IAT, nous allons lancer un autre outil : LordPE.

Cliquons sur le bouton *PE Editor*, puis ouvrons le fichier PE voulu, wordpad.exe en l'occurrence. Enfin, cliquons sur *Directories* pour obtenir la fenêtre suivante (Figure 8) qui liste des couples (RVA, taille) pour divers répertoires d'informations.

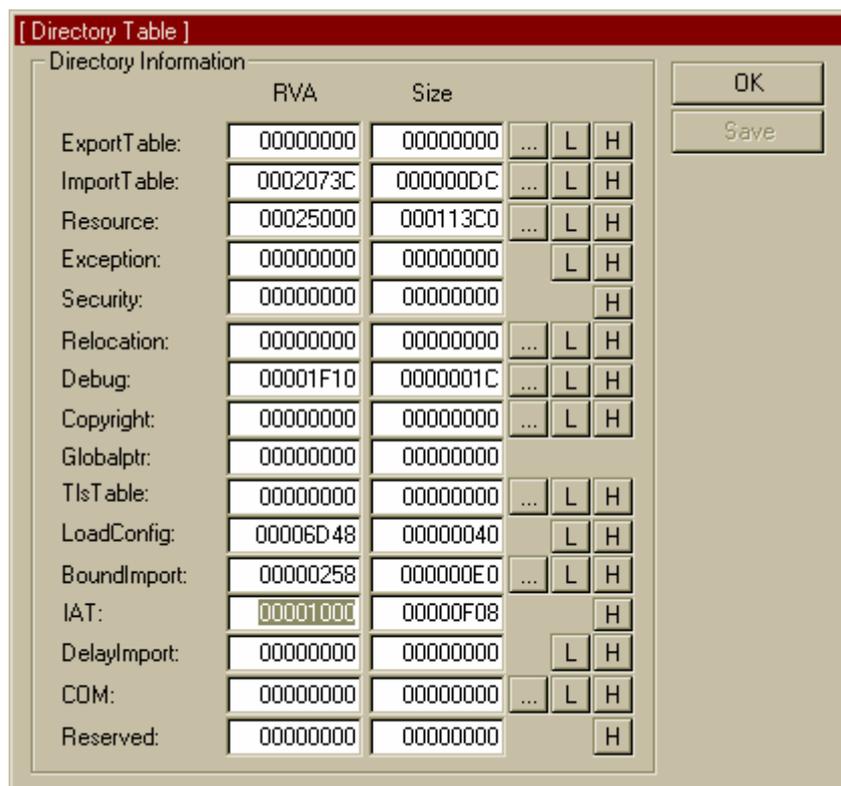


Figure 8 : Répertoires de données, dont IAT (LordPE)

Calculons maintenant l'adresse virtuelle en mémoire de l'IAT :

Virtual Address = base adress + RVA

$$= 0x1000'0000 + 0x0000'1000 = 0x1000'1000$$

Puisqu'un pointeur tient sur 32 bits soit 4 octets, la première entrée se trouve à l'adresse virtuelle 0x1000'1000, la seconde à la VA 0x1000'1004, et ainsi de suite.

Déboignons maintenant l'application wordpad.exe avec OllyDbg (dont nous reparlerons sous peu). Ouvrons la fenêtre *Memory* (Alt+M), choisissons *Dump* dans le menu contextuel (clic droit) puis allons à l'adresse voulue (Ctrl+G).

Enfin, sélectionnons la représentation de données suivantes (clic droit) : Long -> Address. OllyDbg affiche maintenant une adresse 32 bits par ligne, en effectuant même la résolution des noms de fonction. Voici le début de l'IAT de Wordpad à la Figure 9. Nous trouvons l'adresse virtuelle de la fonction Advapi32!RegCloseKey dans la deuxième entrée.

Remarquons que l'emplacement mémoire 0x1000'1020, qui vaut zéro, marque la limite entre les fonctions importées depuis 2 DLLs, ici advapi32.dll et gdi32.dll.

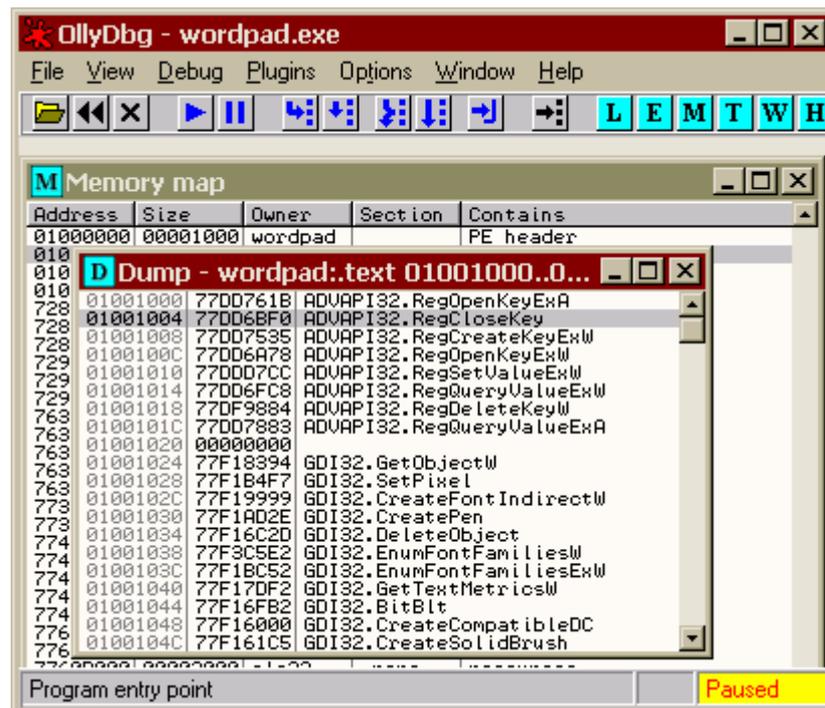


Figure 9 : Dump mémoire de l'IAT (OllyDbg)

Pour terminer, disons qu'il existe de nombreux outils offrant de manipuler les fichiers PE. On peut bien sûr faire ce qu'on veut avec un éditeur hexadécimal, mais c'est moins pratique... Outre l'analyse des champs, ces outils permettent notamment de:

- Changer le point d'entrée d'un EXE, c'est-à-dire l'adresse de la 1ère instruction à exécuter
- Modifier du code machine existant (*patching*)
- Remplacer une fonction importée d'une DLL par une autre
- Repérer un espace libre (rempli de zéros) où insérer du code
- Si une section n'est pas exécutable, modifier ses attributs de section
- Ajouter une section pour créer plus d'espace disponible

Toutes ces modifications statiques ont un impact réel sur la sécurité mais elles impliquent d'altérer le fichier sur le disque. Elles peuvent donc être détectées aisément, par exemple par un pare-feu qui maintiendrait une table d'empreintes (MD5, SHA-1 ou autre) de ces fichiers.

2.3.3. Occupation mémoire : théorie

Chaque processus s'exécute dans un **espace d'adresses virtuelles** (EAV) de 4 Go. Les adresses de 0x0000'0000 à 0x7FFF'FFFF correspondent à la zone utilisateur (propre à chaque processus), alors

que les adresses de 0x8000'0000 à 0xFFFF'FFFF correspondent à la zone système (commune à tous les processus).⁹

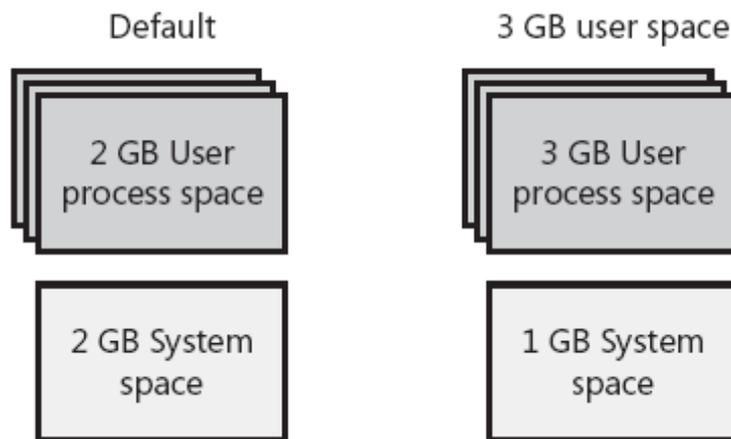


Figure 10 : Espaces d'adresses virtuelles des processus et du système

© 2005 by D. Solomon & M. Russinovich

A noter que la limite entre espace utilisateur et espace système peut être déplacée à 3 Go en changeant les options de démarrage de Windows (cf. [WI] p. 15).

Chaque processus dispose de son propre EAV qui est une vue logique sur la mémoire. Par exemple, l'adresse 0x0BAD'BEEF d'un processus pointe sur une fonction de l'exécutable, alors que la même adresse d'un autre processus pointe sur une structure de données ou est invalide (n'a pas de correspondance physique). Ceci est possible grâce à un mécanisme de traduction dont nous ferons abstraction.

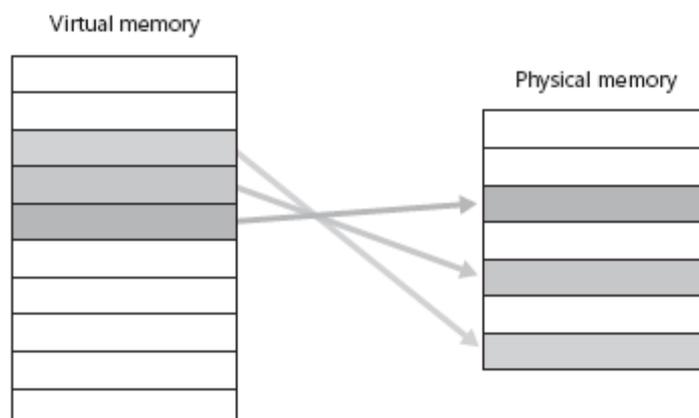


Figure 11 : Traduction entre mémoire virtuelle et mémoire physique

© 2005 by D. Solomon & M. Russinovich

⁹ Cf. *Virtual Address Space* dans MSDN :

http://msdn.microsoft.com/library/en-us/memory/base/virtual_address_space.asp

Par conséquent, les processus sont isolés entre eux : le processus A ne peut pas accéder directement à la mémoire (virtuelle) du processus B. Un processus peut partager explicitement de la mémoire au moyen de fichiers mappés en mémoire (*memory mapped files*) ou de sections partagées. Néanmoins, un processus détient l'accès total à ses processus fils créés grâce à la fonction `CreateProcess`¹⁰.

De plus, un processus s'exécutant en mode utilisateur ne peut accéder à l'espace système (les 2 Go supérieurs). S'il tente de le faire, il y aura une violation d'accès. En voici une démonstration : considérons le code assembleur suivant qui va lire le mot de 32 bits se trouvant à l'adresse `0x8001'0000` (qui est l'adresse de chargement favorite de `HAL.DLL`, une *well-known DLL* chargée lors du *boot*).

```
_asm {  
  mov eax,    0x80010000      // EAX <- adresse de base de HAL.DLL  
  mov ebx,    dword ptr [eax] // Violation d'accès en lecture  
  call dword ptr [eax]       // Idem lors de la tentative d'appel  
  mov dword ptr [eax], ecx   // Violation d'accès en écriture  
}
```

Insérons ce code dans Visual Studio 2005 au sein du fichier source d'un nouveau projet C++ "*Win32 Console Application*", exécutons-le puis déboguons-le pour avoir les détails de l'exception. Voici le résultat:

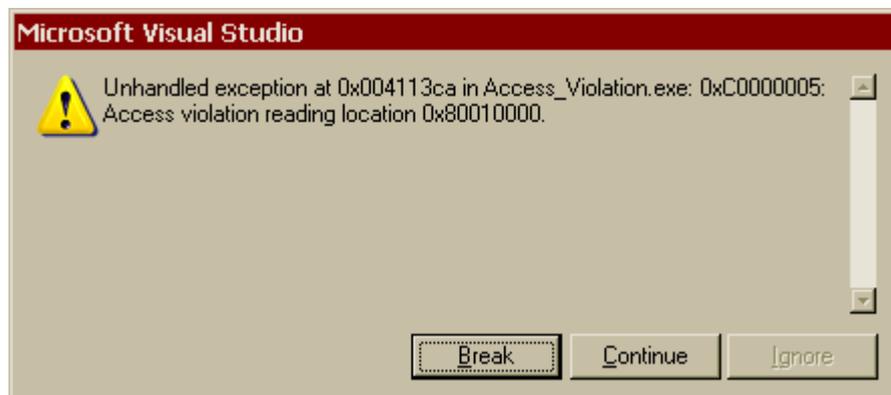


Figure 12 : Violation d'accès à l'espace système (Visual Studio 2005)

Nous avons dit que l'espace d'adresses virtuelles (EAV) s'étend sur 4 Go. Mais la quantité de mémoire physiquement présente sur la machine peut varier : par exemple, mon PC de test comporte seulement 1 Go de RAM.

Pour y remédier, le gestionnaire de mémoire transfère ou pagine (*pages out*) des portions de la mémoire (appelée pages) sur le disque pour

¹⁰ Cf. "*Process Security and Access Rights*" dans MSDN : [http://msdn.microsoft.com/\(...\)/dllproc/base/process_security_and_access_rights.asp](http://msdn.microsoft.com/(...)/dllproc/base/process_security_and_access_rights.asp)

libérer de l'espace en mémoire. Sitôt qu'un thread doit accéder à une page qui se trouve sur le disque, le gestionnaire de mémoire doit à nouveau charger (*page in*) la page en mémoire. Si toute la mémoire est occupée, alors une autre page doit être paginée pour libérer de l'espace pour la première¹¹.

Ainsi, nous ne pouvons pas prédire si le contenu de telle portion de la mémoire virtuelle se trouvera réellement en mémoire physique ou sur le disque dur. Mais nous n'avons pas besoin de nous en soucier car le gestionnaire de mémoire de Windows s'en charge d'une façon transparente au programme utilisateur.

La **page** est la plus petite unité de protection au niveau matériel, c'est-à-dire qu'on peut uniquement définir les droits d'accès par page, et pas pour chaque adresse. Selon nos hypothèses (architecture x86 avec Windows XP), les petites pages font 4 ko.

En mode *user*, on ne peut allouer qu'un multiple de la taille de page (4 ko). De plus, l'allocation de mémoire doit respecter une certaine granularité¹². Cela veut dire que l'adresse de départ des blocs de mémoire alloués doit être un multiple de 64 ko. Ces deux paramètres du système, la taille de page et la granularité d'allocation, peuvent être déterminés grâce à l'API `GetSystemInfo` :

```
SYSTEM_INFO sSysInfo;  
GetSystemInfo( &sSysInfo ); // Initialise la structure.  
DWORD dwPageSize = sSysInfo.dwPageSize;  
DWORD dwAllocGran = sSysInfo.dwAllocationGranularity;
```

En passant, mentionnons que la pagination de la mémoire peut faire en sorte que le contenu de la mémoire se retrouve sur le disque. Les pages qui ont été sorties de la mémoire sont stockées dans `\pagefile.sys`. De même, lors d'une mise en veille prolongée (*hibernation*), le contenu entier de la RAM est transféré dans `\hiberfile.sys`. Ce phénomène peut compromettre la confidentialité des informations présentes en mémoire à ce moment.

2.3.4. Occupation mémoire : pratique

OllyDbg¹³, que nous avons aperçu plus haut, est un excellent débogueur gratuit. Il est important d'en maîtriser les diverses fenêtres pour savoir où trouver les informations sur le processus débogué. Tout d'abord, ouvrons l'exécutable du Wordpad. Etudions ce qu'il vient de se passer

¹¹ Cf. [WI] p. 15

¹² Il existe aussi des grandes pages de 4 Mo (*large page memory*) pour des questions de performances. Rechercher "*large page support*" dans MSDN pour aller plus loin.

¹³ OllyDbg (Olly pour les intimes) est écrit par Oleh Yuschuk et téléchargeable sur : <http://www.ollydbg.de/>

lors de l'ouverture de l'EXE. Nous avons à disposition un journal des événements (*log*) qui s'ouvre en pressant Alt+L. Nous le voyons à la Figure 13.

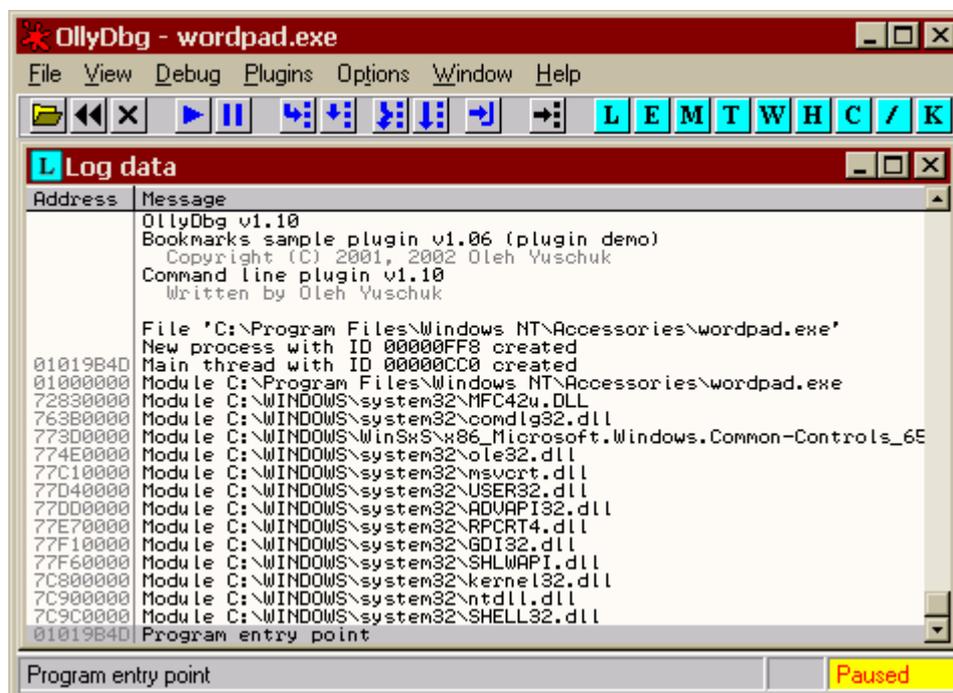


Figure 13 : Journal des événements initial (OllyDbg)

Outre la création du processus (*new process*) et de son thread principal (*main thread*), le journal initial indique le chargement d'une liste de modules avec leur adresse virtuelle de base en regard. C'est bien l'EXE qui est chargé en premier, puis il est suivi d'une série de DLLs.

La dernière ligne (*Program entry point*) indique que le programme est complètement chargé et prêt à s'exécuter. Le pointeur d'instruction EIP stocke désormais le point d'entrée du programme qui est 0x0101'9B40 dans notre cas. Nous nous trouvons juste avant le début de l'exécution.

Intéressons-nous de plus près aux modules (EXE et DLLs) cités ici. Pour ce faire, affichons la vue *Executable modules* en pressant Alt+E (voir Figure 14).

Le chargeur de programme (*Windows loader*) charge d'abord en mémoire l'exécutable à son adresse de départ favorite (ADF). Puis les DLLs nécessaires sont chargées à leur tour, si possible à leur ADF, sinon le chargeur procède à un relogement (*relocation*).

Nous pouvons vérifier, à l'aide de l'un des éditeurs de fichier PE présentés ci-dessus, qu'aucun des modules n'a été relogé. En effet, toutes ces DLLs font partie de Windows et leurs concepteurs ont choisi leur ADF de sorte que ces dernières ne se collisionnent pas dans l'EAV, évitant un relogement coûteux en temps.

De plus, nous constatons que tous ces modules respectent les contraintes d'alignement. Leur adresse de base est alignée sur 64 ko (en hexadécimal, les 4 chiffres de droite sont à zéro), tandis que leur taille est un multiple de 4 ko (les 3 chiffres de droites sont à 0).

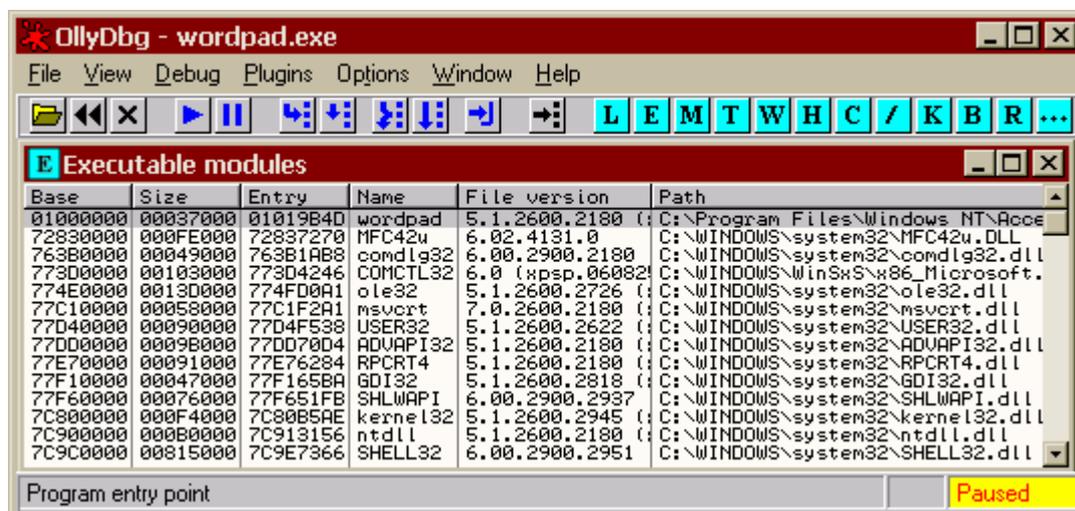


Figure 14 : Liste des modules exécutables (OllyDbg)

Nous pouvons aussi afficher une vue complète du *mapping* mémoire de la zone utilisateur. Pour ce faire, ouvrons la fenêtre *Memory map* en pressant Alt+M (voir Figure 15). Il est possible ici de savoir à quoi correspondent les adresses 0x0000'0000 à 0x7FFF'FFFF. L'observateur attentif découvrira qu'il y a parfois des plages d'adresses invalides entre deux zones de mémoire (i.e. 2 lignes); ceci est dû aux valeurs des contraintes d'alignement et au fait que tout l'espace virtuel n'est pas occupé.

Nous retrouvons, pour chaque module, l'en-tête PE et les diverses sections. A noter que les attributs de protection de page, affichés dans la colonne *Access*, semblent n'être pas actualisés¹⁴. En comparant ce qui est montré dans OllyDbg avec les attributs des sections contenu dans le fichier PE (cf. l'outil PESam), nous voyons bien qu'il y a un problème. Par exemple, les sections *.text* devraient être dénotées comme exécutables (E).

¹⁴ Ce bug est mentionné dans le forum d'OllyDbg à l'adresse suivante : <http://www.woodmann.com/forum/showthread.php?t=8698>

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped
01000000	00001000	wordpad	01000000 (itself)		PE header	Imag	R	RWE
01001000	00022000	wordpad	01000000	.text	code, import	Imag	R	RWE
01023000	00002000	wordpad	01000000	.data	data	Imag	R	RWE
01025000	00012000	wordpad	01000000	.rsrc	resources	Imag	R	RWE
72830000	00001000	MFC42u	72830000 (itself)		PE header	Imag	R	RWE
72831000	000A3000	MFC42u	72830000	.text	code	Imag	R	RWE
728D4000	00033000	MFC42u	72830000	.rdata	imports, exp	Imag	R	RWE
72907000	0000B000	MFC42u	72830000	.data	data	Imag	R	RWE
72912000	0000B000	MFC42u	72830000	.rsrc	resources	Imag	R	RWE
7291D000	00011000	MFC42u	72830000	.reloc	relocations	Imag	R	RWE
763B0000	00001000	comdlg32	763B0000 (itself)		PE header	Imag	R	RWE
763B1000	00030000	comdlg32	763B0000	.text	code, import	Imag	R	RWE
763E1000	00004000	comdlg32	763B0000	.data	data	Imag	R	RWE
763E5000	00011000	comdlg32	763B0000	.rsrc	resources	Imag	R	RWE
763F6000	00003000	comdlg32	763B0000	.reloc	relocations	Imag	R	RWE
773D0000	00001000	COMCTL32	773D0000 (itself)		PE header	Imag	R	RWE
773D1000	00090000	COMCTL32	773D0000	.text	code, import	Imag	R	RWE
77461000	00001000	COMCTL32	773D0000	.data	data	Imag	R	RWE
77462000	0006A000	COMCTL32	773D0000	.rsrc	resources	Imag	R	RWE
774CC000	00006000	COMCTL32	773D0000	.reloc	relocations	Imag	R	RWE
774E0000	00001000	ole32	774E0000 (itself)		PE header	Imag	R	RWE
774E1000	0011F000	ole32	774E0000	.text	code, import	Imag	R	RWE
77600000	00006000	ole32	774E0000	.orpc	code	Imag	R	RWE
77606000	00007000	ole32	774E0000	.data	data	Imag	R	RWE
7760D000	00002000	ole32	774E0000	.rsrc	resources	Imag	R	RWE
7760F000	0000E000	ole32	774E0000	.reloc	relocations	Imag	R	RWE
77C10000	00001000	msvcrt	77C10000 (itself)		PE header	Imag	R	RWE
77C11000	0004C000	msvcrt	77C10000	.text	code, import	Imag	R	RWE
77C5D000	00007000	msvcrt	77C10000	.data	data	Imag	R	RWE
77C64000	00001000	msvcrt	77C10000	.rsrc	resources	Imag	R	RWE
77C65000	00003000	msvcrt	77C10000	.reloc	relocations	Imag	R	RWE
77D40000	00001000	USER32	77D40000 (itself)		PE header	Imag	R	RWE
77D41000	0005F000	USER32	77D40000	.text	code, import	Imag	R	RWE
77DA0000	00002000	USER32	77D40000	.data	data	Imag	R	RWE

Figure 15 : Occupation initiale de la mémoire (OllyDbg)

Mais il y a deux choses intéressantes à remarquer concernant les modules. Premièrement, toutes ces DLLs se trouvent dans \Windows\system32, sauf une qui provient de \Windows\WinSxS\ . Cette dernière DLL trahit un mécanisme introduit dans Windows XP pour gérer simultanément plusieurs versions d'une même DLL¹⁵. En fouillant le répertoire \Windows\WinSxS, on trouve (Figure 16) 3 dossiers qui contiennent chacun la DLL comctl32.dll, avec les versions respectives 6.0.0.0, 6.0.10.0 et la version utilisée par wordpad.exe : 6.0.2600.2180.

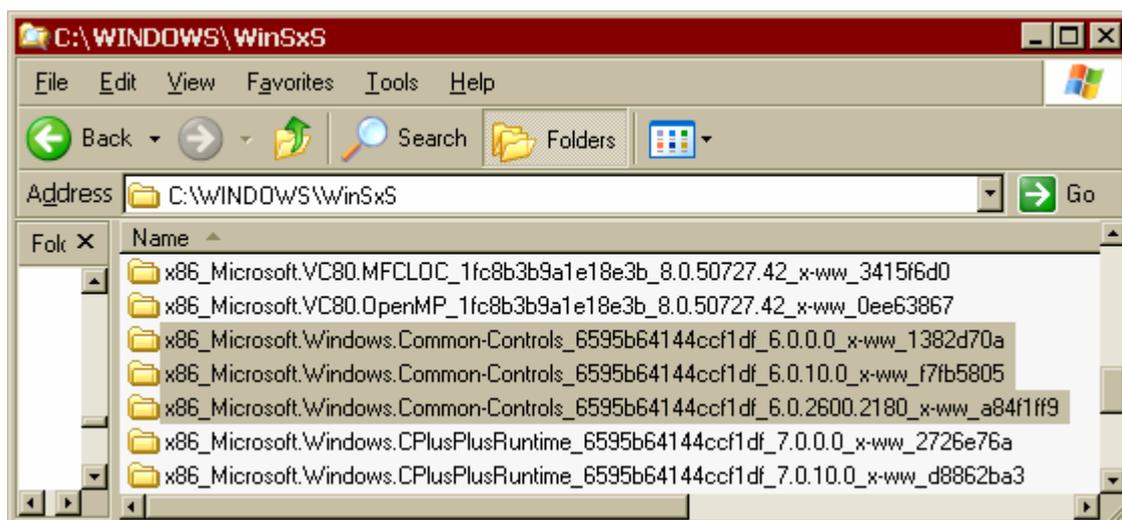


Figure 16 : Versions concurrentes d'une DLL (Explorer)

¹⁵ Pour plus d'informations, voir l'encart "Side-by-side Assemblies" dans [WI] p. 311.

Deuxièmement, pour l'instant, seules les DLLs importées implicitement sont présentes dans l'EAV. Or nous pouvons compter que 13 DLLs ont été chargées, tandis que le fichier PE n'en référence que 10 (voir Figure 7) ! Comment cela se fait-il? Il faut aborder le problème des dépendances.

2.3.5. Dépendances

Nous avons déjà parlé au § 2.3.2 des différents types d'imports. On dit que l'exécutable dépend des DLLs qu'il importe. Mais ces dernières dépendent aussi peut-être d'autres DLLs. Ainsi le chargeur de programmes doit-il vérifier ces dépendances récursivement afin de charger toutes les DLLs nécessaires.

L'utilitaire Dependency Walker¹⁶ trace les dépendances d'une image (EXE ou DLL). Pour mieux comprendre ce qu'il affiche, consulter l'aide de l'application, en particulier la rubrique "*Types of Dependencies Handled...*". En ouvrant wordpad.exe avec depends, nous pouvons faire les observations suivantes sur la Figure 17 :

1. La zone en haut à gauche montre l'arbre des dépendances. La racine est sans surprise l'image étudiée. Le second niveau comporte toutes les DLLs importées implicitement (les 10 référencées implicitement dans le fichier PE), et ainsi de suite. La relation père-fils dénote la dépendance.
2. En développant le sous-arbre d'ADVAPI32.DLL, nous trouvons d'abord deux DLLs dupliquées  (c'est-à-dire qui ont déjà été identifiées plus tôt lors du parcours de l'arbre) NTDLL.DLL et KERNEL32.DLL.
3. Puis nous rencontrons RPCRT4.DLL, qui ne fait pas partie des 10 DLLs mentionnées plus haut mais qui se trouve dans la *Memory Map* initiale de OllyDbg. Cette DLL est chargée parce qu'ADVAPI32.DLL en a besoin pour fonctionner.
4. Les deux dernières DLLs sont à chargement retardé, d'où le symbole de sablier présent dans leur icône (). WINTRUST.DLL et SECURE32.DLL ne seront chargées qu'au moment où elles seront réellement utilisées. Elles sont actuellement absentes de l'EAV du Wordpad, puisqu'il n'y a aucun appel à des fonctions de ces DLLs avant que l'exécution du code du processus ne débute.

¹⁶ L'exécutable depends.exe est fourni avec les Windows XP Service Pack 2 Support Tools, téléchargeable sur <http://www.microsoft.com>.

5. Dans la partie médiane se trouve une liste de toutes les DLLs importées par le module wordpad.exe, directement ou non. Nous pouvons ignorer l'avertissement affiché en rouge en bas de la fenêtre. S'il y avait vraiment un problème de fonction non résolue, alors l'exécution de Wordpad le signalerait.
6. Les deux zones semblables en haut à droite sont des listes de fonctions. La liste du bas contient les fonctions exportées par la DLL qui est sélectionnée dans l'arbre des dépendances (à gauche). La liste du haut indique les fonctions effectivement importées par la DLL parente de la DLL actuellement sélectionnée. La liste du haut contient donc un sous-ensemble des fonctions de la liste du bas.

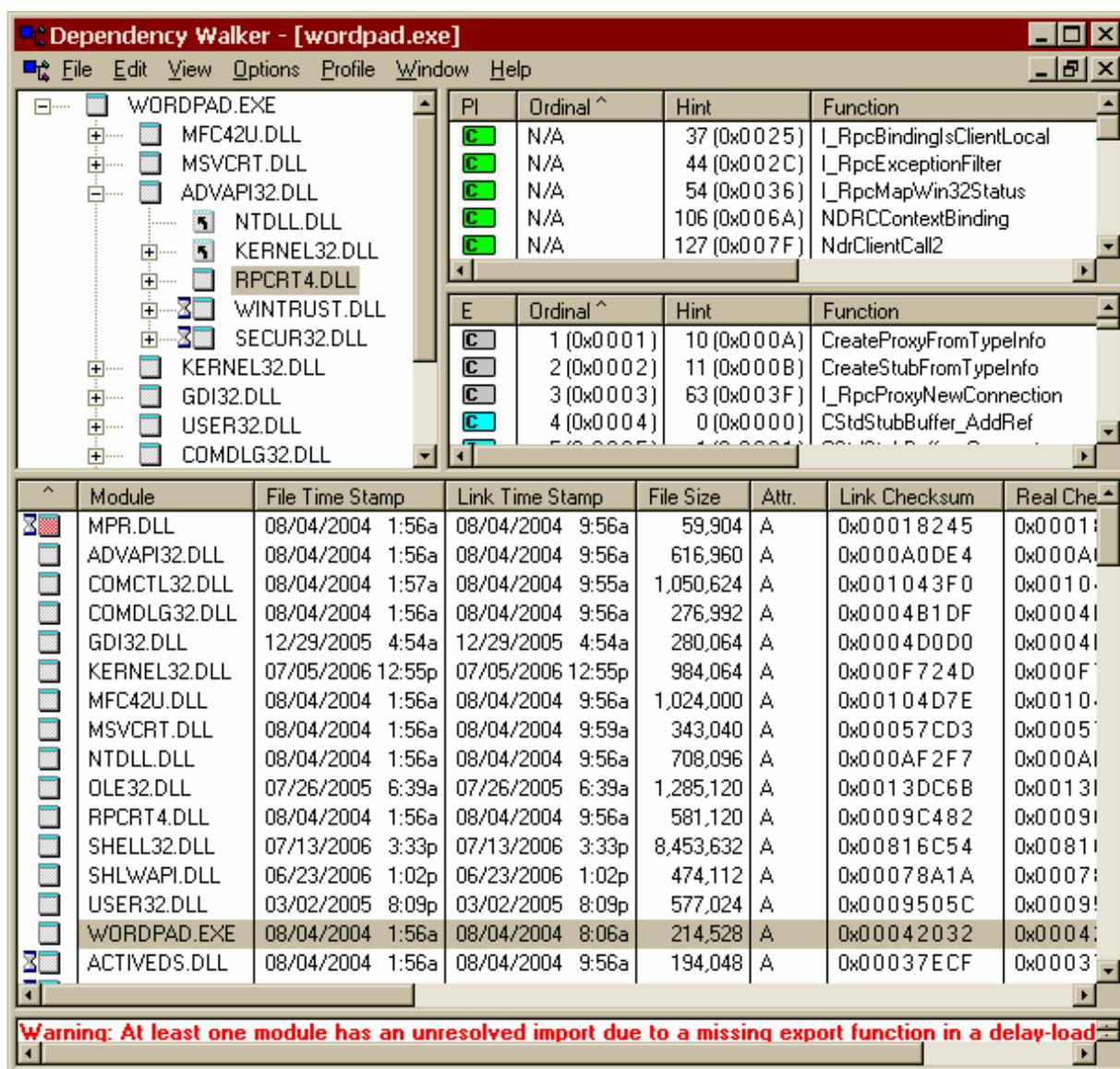


Figure 17 : Dépendances de DLLs et fonctions importées (Dependency Walker)

Par la suite, d'autres DLLs seront chargées dynamiquement, à cause d'un appel soit à l'API LoadLibrary, soit à des fonctions de *delay-load DLLs*¹⁷. Si nous lançons le programme en pressant sur F9 et que nous ouvrons la fenêtre *Log*, nous voyons le chargement (ligne débutant par *Module*) et le déchargement de DLLs (ligne débutant par *Unload*). Remarquons encore à la Figure 18 que le module hpcjrui.dll est chargé puis déchargé 3 fois de suite. Cette dernière fenêtre nous amène à la partie suivante, qui est centrée sur l'exécution du programme par les threads.

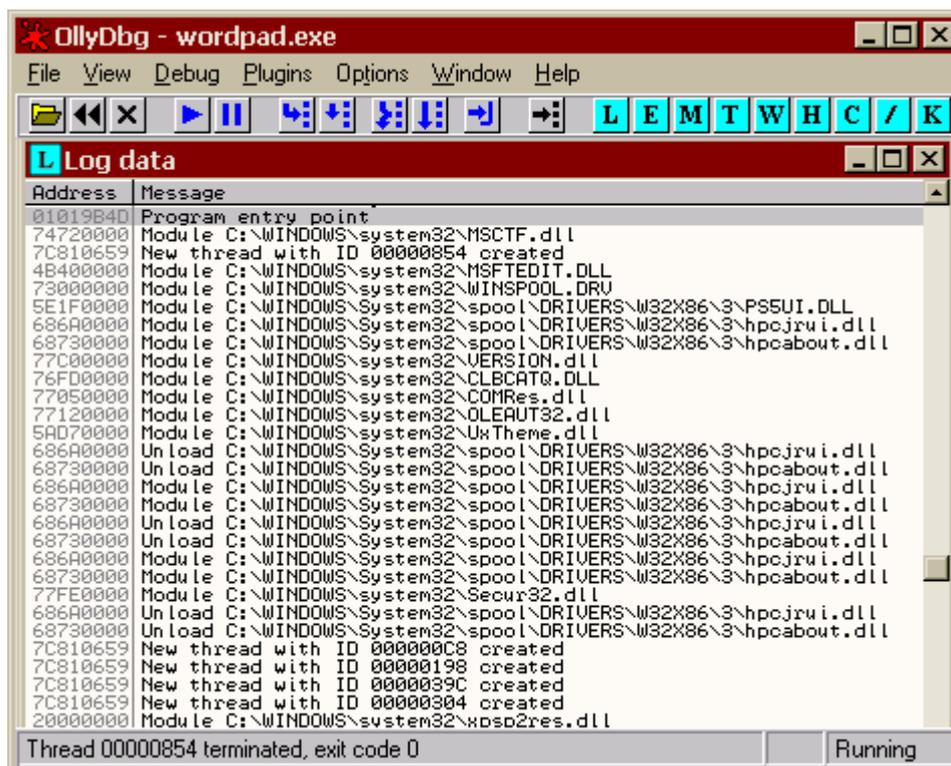


Figure 18 : Journal des événements suivant (OllyDbg)

2.4. Processus, threads et handles

2.4.1. Processus

Nous allons nous pencher maintenant sur l'exécution de wordpad.exe. Un processus est un objet vivant, qui naît, évolue puis meurt. Il ne suffit pas de disposer des bons outils pour l'observer, il faut encore regarder au bon moment. Pour cette partie, la référence est le chapitre 6 de [WI], intitulé "*Processes, Threads and Jobs*". Nous ne parlerons pas des jobs.

¹⁷ Il s'agit de DLLs qui sont chargées juste avant le premier appel à l'une de leurs fonctions, comme nous l'avons évoqué à la section sur l'IAT. Il s'agit d'un mécanisme paresseux (*lazy binding*), qui ne fait le travail que lorsqu'il est vraiment nécessaire. Le fait de retarder le chargement de DLLs permet à l'application de démarrer plus vite.

Le premier outil à notre disposition est le gestionnaire de tâches de Windows (*Task Manager*). Sous l'onglet *Processus*, nous pouvons voir tous les processus existants, leur PID, l'utilisateur qui les a lancés, leurs compteurs de threads et de handles, etc. (aller dans le menu *View | Select columns* pour afficher toutes les informations).

Remarquons en passant les valeurs que prennent les PIDs. Sous UNIX, les PIDs se sont attribués consécutivement (0, 1, 2, 3, etc.) et indiquent la chronologie des créations de processus. Sous Windows au contraire, les PIDs ne peuvent être prédits et ne présentent aucun ordre intelligible hormis le fait qu'ils sont des multiples de 4.

Sur la Figure 19, la valeur des PIDS varie de 0 à plus de 1412. Le PID du pseudo-processus *Idle* vaut 0 ; celui de *System*, 4. Le processus *wordpad.exe* a reçu l'ID 404, bien qu'il ait été lancé le dernier.

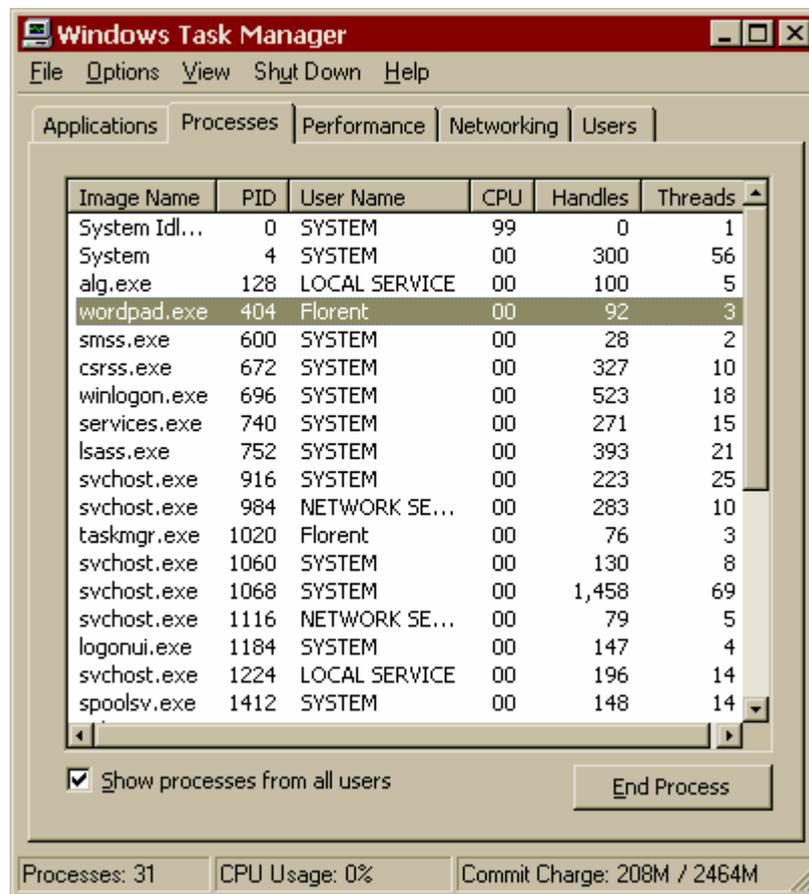


Image Name	PID	User Name	CPU	Handles	Threads
System Idl...	0	SYSTEM	99	0	1
System	4	SYSTEM	00	300	56
alg.exe	128	LOCAL SERVICE	00	100	5
wordpad.exe	404	Florent	00	92	3
smss.exe	600	SYSTEM	00	28	2
csrss.exe	672	SYSTEM	00	327	10
winlogon.exe	696	SYSTEM	00	523	18
services.exe	740	SYSTEM	00	271	15
lsass.exe	752	SYSTEM	00	393	21
svchost.exe	916	SYSTEM	00	223	25
svchost.exe	984	NETWORK SE...	00	283	10
taskmgr.exe	1020	Florent	00	76	3
svchost.exe	1060	SYSTEM	00	130	8
svchost.exe	1068	SYSTEM	00	1,458	69
svchost.exe	1116	NETWORK SE...	00	79	5
logonui.exe	1184	SYSTEM	00	147	4
svchost.exe	1224	LOCAL SERVICE	00	196	14
spoolsv.exe	1412	SYSTEM	00	148	14

Figure 19 : Liste des processus (Task Manager)

Présentons maintenant l'incontournable utilitaire *Process Explorer*¹⁸, abrégé *ProcExp*, qui donne accès à une mine d'informations sur les

¹⁸ Cet utilitaire, ainsi que d'autres, sont produits par SysInternals qui vient d'être racheté par Microsoft. ProcExp est téléchargeable depuis :

processus et leurs threads. Ces mêmes informations et bien d'autres sont présentées par d'autres programmes, sous d'autres formes. Pour simplifier, nous privilégierons ProcExp autant que possible.

Une des premières choses à connaître sur un processus est sa **généalogie**. Bien sûr, nous pouvons trouver son PID, le chemin complet de son image, ses statistiques, etc. Mais les ancêtres d'un processus sont déterminants pour expliquer ses droits.

En lançant ProcExp normalement, nous voyons l'arborescence habituelle de Windows dans la partie droite de la Figure 20. Le processus *System* (PID = 4) est l'ancêtre commun à tous les autres processus. Puis nous trouvons *smss.exe* qui gère les sessions et ensuite *winlogon.exe* qui opère l'ouverture de session (il y en a 2 dans notre cas, une pour la session locale et l'autre pour la session *Remote Desktop*).

En bas de l'arbre, nous apercevons une deuxième racine. Le processus *explorer.exe* semble être orphelin. En fait, cela signifie que son processus père s'est terminé, ce qui a causé une rupture de l'arbre car un processus ne connaît que son ancêtre immédiatement précédent. Pour découvrir qui a démarré *explorer.exe*, il suffit de lancer automatiquement ProcExp lors de l'ouverture de session (p.ex. en copiant son raccourci dans le dossier Startup du menu Start). La nouvelle capture d'écran indique le chaînon manquant entre *winlogon.exe* et *explorer.exe* : *userinit.exe*, dont le rôle est précisément de démarrer le *shell* de l'utilisateur¹⁹.

ProcExp peut être mis en pause en appuyant sur la barre espace. Il comporte un panneau inférieur (*lower pane*) qui, une fois affiché (Ctrl+L), indique soit les DLLs utilisées par le processus (Ctrl+D), soit les handles (Ctrl+H). Toutes les autres informations concernant un processus sont accessibles en ouvrant ses propriétés par double clic et en naviguant parmi les onglets.

Un processus, comme tout objet géré par le système, est représenté par une ou plusieurs structures de données. Nous pourrions explorer les entrailles de Windows et utiliser LiveKd²⁰ pour analyser ces structures de données. Mais ceci nous préparerait à des attaques en mode *kernel*, pas en mode *user*. Nous nous contenterons de renvoyer à [WI] p. 298 pour une liste des fonctions de l'API concernant les processus.

<http://www.sysinternals.com/Utilities/ProcessExplorer.html>

¹⁹ Cf. [WI] p. 272. Remarquons qu'il aurait pu y avoir plusieurs intermédiaires entre *winlogon.exe* et *explorer.exe*.

²⁰ Téléchargeable depuis <http://www.sysinternals.com/utilities/livekd.html>

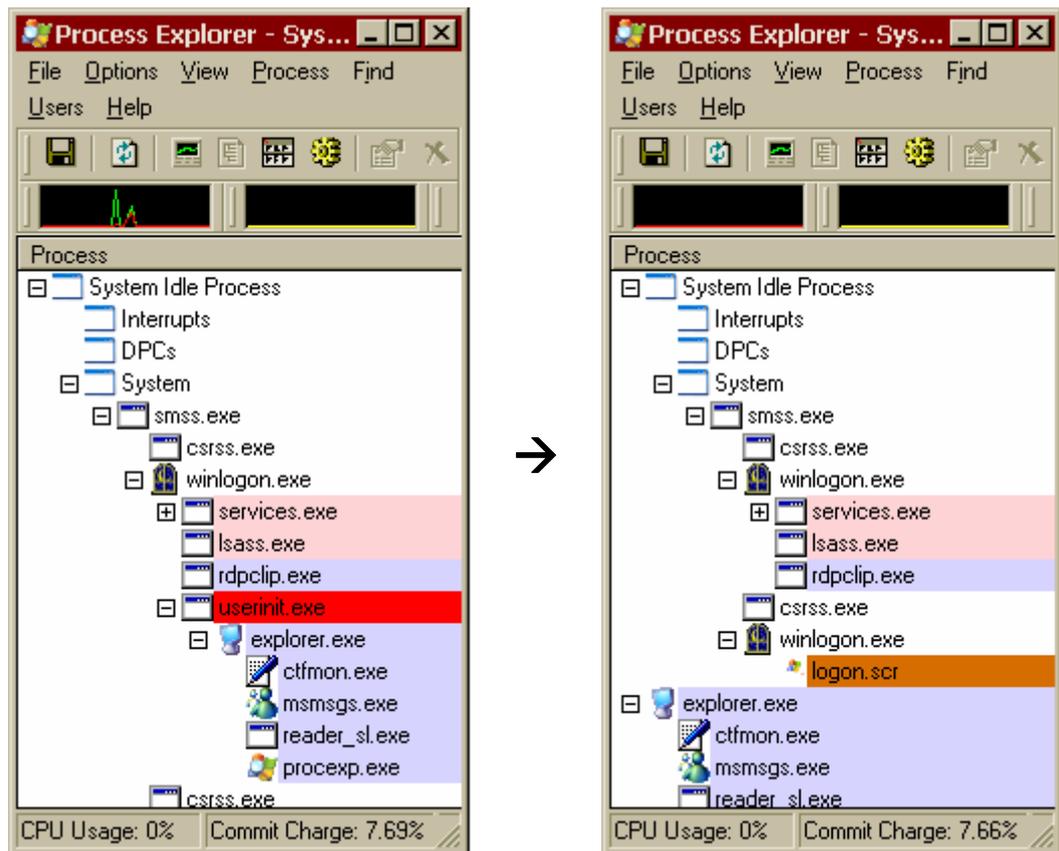


Figure 20 : Arbre des processus avec et sans userinit.exe (Process Explorer)

2.4.2. Threads

De nouveau, il y a énormément d'informations disponibles concernant les threads : statistiques d'utilisation de la mémoire et du temps CPU, priorités et changements de contexte, etc. Ce qui nous importe surtout est de savoir combien de threads s'exécutent et ce qu'ils font.

Lançons ProcExp, et affichons l'onglet *Threads* des propriétés du processus wordpad.exe. Nous apprenons que ce processus possède actuellement trois threads. Si nous ouvrons un fichier dans Wordpad, nous constatons que quatre threads supplémentaires sont créés. Comme pour les processus, ProcExp signale les créations de threads en vert et les destructions en rouge.

Avant de savoir ce que font les threads, il faut configurer correctement les **symboles**. Vaut-il mieux constater que le thread n° 1640 est en train d'exécuter la fonction à l'adresse 0x0101'9B4D? Ou bien savoir que ce thread exécute la fonction wWinMainStartup du module wordpad.exe?

Les symboles permettent à ProcExp et aux débogueurs de retrouver la correspondance entre les adresses particulières du module et les noms de fonctions, de variables ou de classes. Leur but est de donner du sens à ces adresses aux êtres humains que nous sommes.

En plus des *Debugging Tools*, nous avons besoin de télécharger et d'extraire le *Symbol Package*²¹ dans le dossier %WinDir%\Symbols. Puis il faut configurer ProcExp ainsi :

Dans la boîte de dialogue *Configure Symbols* du menu *Options*, entrer les valeurs suivantes :

- "*Dbghelp.dll path*" :
C:\Program Files\Debugging Tools for Windows\dbghelp.dll
- "*Symbols path*" :
srv*C:\Windows\Symbols*http://msdl.microsoft.com/download/symbols

Cela a pour effet de chercher les symboles dans l'entrepôt local, et de les télécharger depuis le serveur Microsoft au besoin. Bien sûr, ces symboles couvrent uniquement les logiciels fournis avec Windows, dont fait partie le Wordpad.

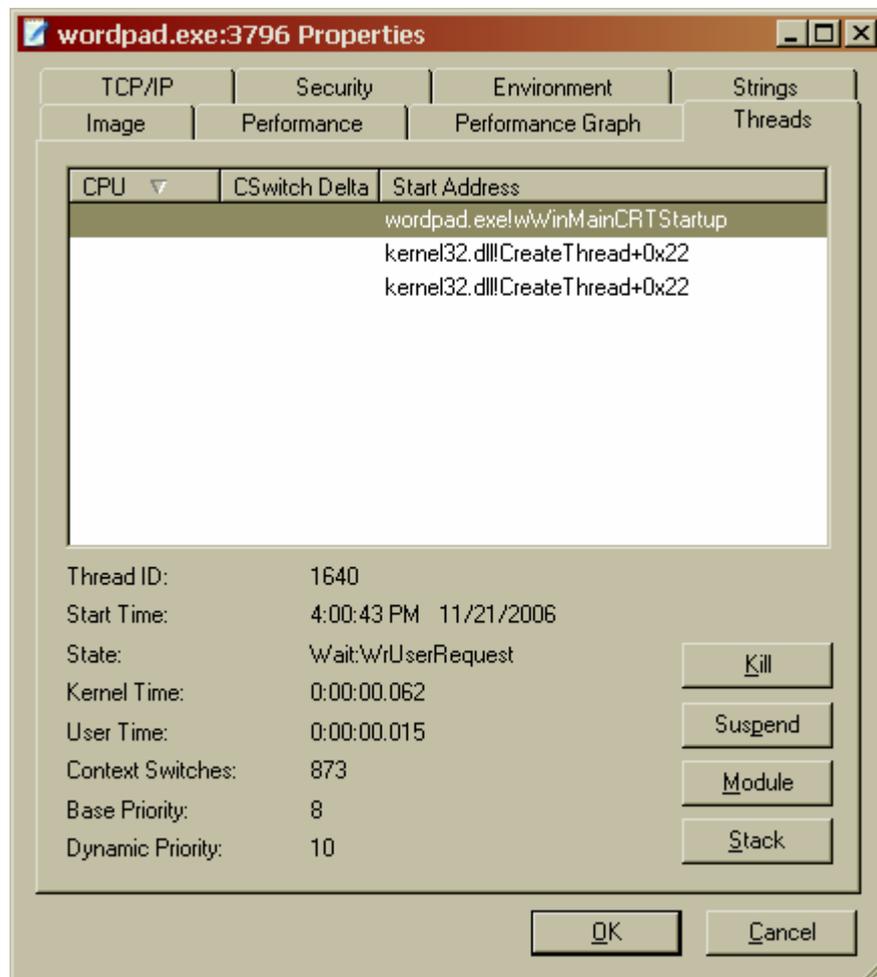


Figure 21 : Threads d'un processus (Process Explorer)

²¹ "Windows XP with Service Pack 2 x86 retail symbols, all languages" :
<http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.msp>

Voici, à la Figure 21, les threads du processus wordpad.exe. Les quatre boutons sur la droite permettent, après avoir sélectionné un thread, de le tuer (*kill*) ou le suspendre (*suspend*), d'afficher les propriétés de l'image (*module*) ou la pile d'appels (*stack*) du thread. Il faut savoir que si un thread reçoit l'ID X, alors aucun processus n'aura le même ID, et vice versa.

Nous prendrons la pile d'appel du thread principal du processus cmd.exe, qui est plus explicite. Nous voyons le flux de contrôle de bas en haut, les appels ayant circulé depuis la fonction mainCRTStartup, puis aux fonctions main, Parser, ParserStatement, GetToken, etc.

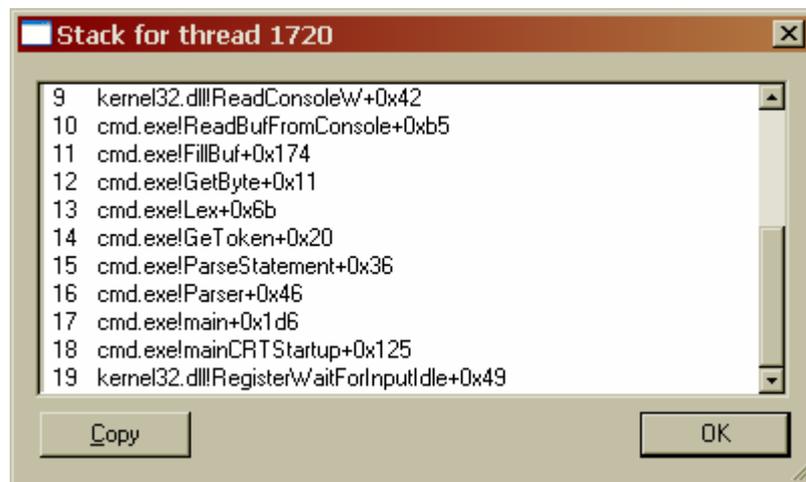


Figure 22 : Pile d'appels d'un thread (Process Explorer)

2.4.3. Handles

Les **handles** sont utilisés lorsqu'une application référence des blocs de mémoire ou des objets gérés par un autre système (le système d'exploitation dans notre cas). Contrairement au pointeur qui contient littéralement l'adresse de l'élément qu'il référence, un handle est une référence abstraite.

Ainsi, l'application ne peut pas accéder directement aux données d'un objet, mais elle doit toujours passer par le système d'exploitation, ce qui permet d'accroître le contrôle des ressources et la sécurité.

Supposons que nous voulions lire un fichier. Voici la marche à suivre :

1. Ouvrir un handle sur le fichier existant via CreateFile. A cette étape ont lieu les contrôles d'accès que nous aborderons plus loin.
2. Utiliser l'objet en passant son handle à d'autres fonctions, telles ReadFile ou ReadFileEx.
3. Fermer le handle via CloseHandle une fois son utilisation terminée.

Une application peut obtenir un handle sur toutes sortes d'objets du système. La plupart des handles ont une portée limitée au processus. Cela veut dire par exemple que si un processus X a reçu un handle sur un fichier valant 4, et que le processus Y cherche à fermer ce même fichier en passant une valeur de handle de 4, cela ne fonctionnera pas.

En effet, le handle 4 peut être invalide pour le processus Y, ou correspondre à un autre objet (une clé de registre, un sémaphore, etc.).

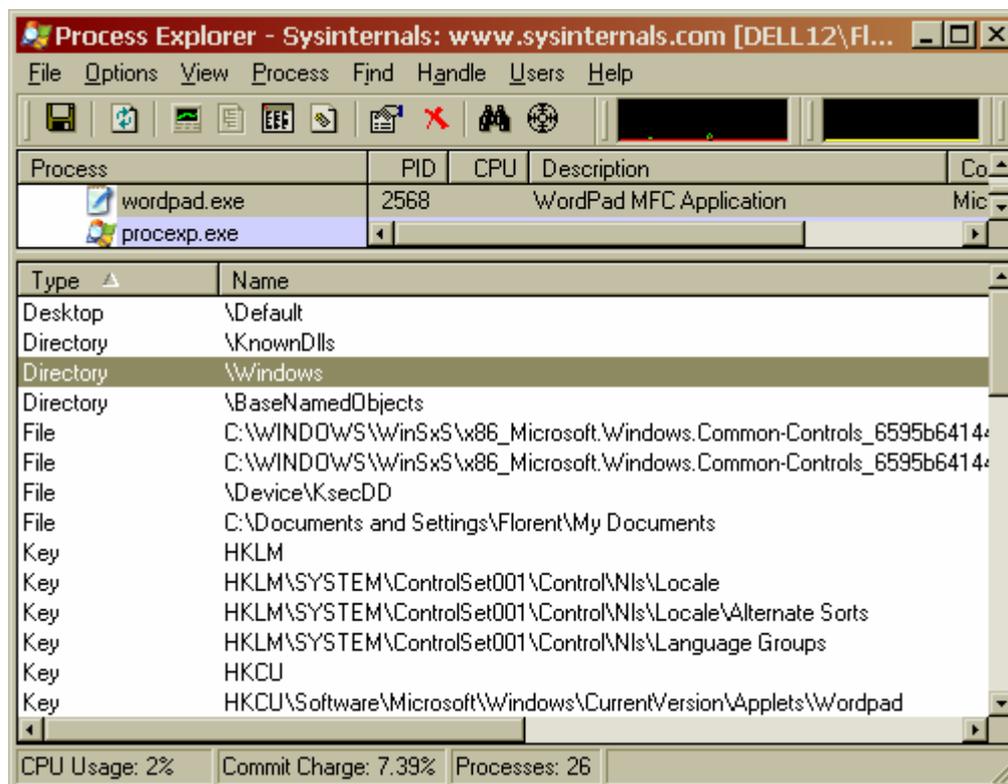


Figure 23 : Handles d'un processus (Process Explorer)

Une exception à cette règle est le handle de module (HMODULE) retourné par l'API `GetModuleHandle` qui est égal à l'adresse de base du module. De toute manière, le module étant situé dans l'espace utilisateur, cela ne pose pas de problème de sécurité d'y accéder. Certains numéros d'objets ont une portée globale à tout le système. C'est le cas notamment pour les identificateurs de processus (chaque processus porte un ID distinct), de threads et de fenêtres.

Nous pouvons lister les handles d'un processus dans ProcExp ou avec l'utilitaire en ligne de commande `handle`²². ProcExp (voir Figure 23) est bien plus convivial. Sélectionnons le processus `wordpad.exe`, puis affichons les handles (`Ctrl+L`, `Ctrl+H`). Pour chaque handle, nous pouvons afficher ses propriétés en double-cliquant dessus.

ProcExp donne des informations qui ne sont pas accessibles en mode utilisateur, grâce à son pilote en mode *kernel*. C'est ainsi que ProcExp peut donner l'adresse (dans l'espace système) de l'objet correspondant à un handle, et d'autres renseignements selon le type de handles.

²² Disponible sur <http://www.sysinternals.com/utilities/handle.html>

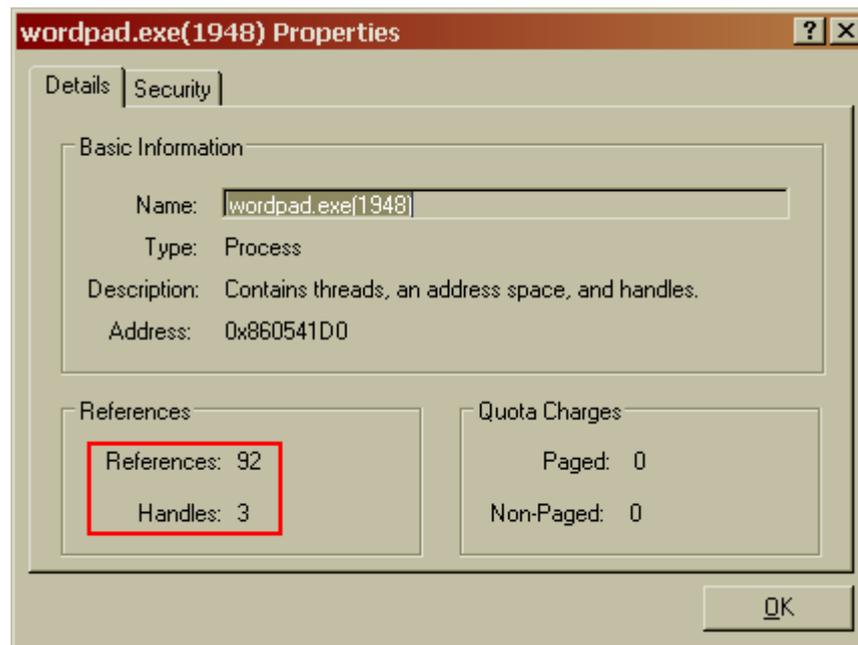


Figure 24 : Détails des propriétés d'un handle (Process Explorer)

Remarquons pour finir que le nombre de références (voir encadré, Figure 24) est le nombre de pointeurs en mode *kernel* qui pointent vers l'objet en question. Ces références sont employées pour savoir quand l'objet peut être. Le nombre de handles concerne le mode utilisateur et parle de lui-même. Le second onglet des propriétés d'un handle concerne la sécurité, que nous allons étudier au § 2.5.

2.4.4. Fichiers mappés et mémoire partagée

Les fichiers mappés en mémoire ou MMFs²³ (Memory-Mapped Files) offrent la possibilité d'accéder à des fichiers sur le disque aussi facilement qu'on accède à la mémoire dynamique : au moyen de pointeurs. Il suffit de mapper tout ou partie d'un fichier en mémoire, c'est-à-dire de faire correspondre les données du fichier à une page d'adresses virtuelles.

On retrouve le principe de mémoire virtuelle : à une page en RAM correspond une page sur le disque (nous en reparlerons au § 3.4.1). Les pages de codes proviennent de l'image²⁴ du fichier PE, tandis que les pages de données sont stockées dans le fichier d'échange du système (*system pagefile*). On peut créer, via l'API `CreateFileMapping`, un MMF correspondant à un fichier temporaire qui sera alors créé, à un fichier existant ou encore à rien du tout (i.e. au *system pagefile*).

²³ MMF : <http://msdn2.microsoft.com/en-us/library/ms810613.aspx>

²⁴ On peut se demander à juste titre ce qu'il se passe si on supprime le fichier image d'une application en exécution. La réponse est que c'est tout simplement impossible : les fichiers EXE et DLL sont verrouillés, donc impossible à effacer, tant qu'ils sont chargés par un processus.

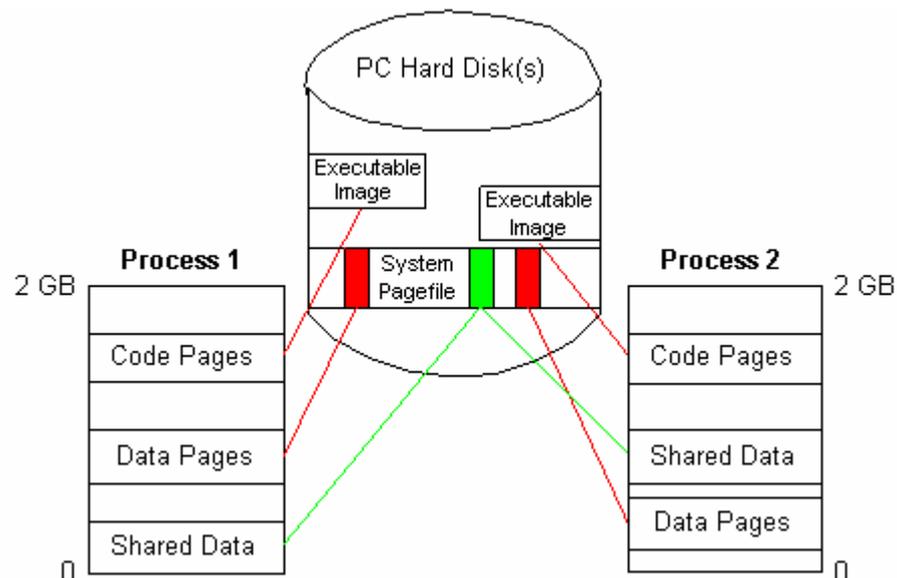


Figure 25 : Fichiers mappés en mémoire © Microsoft

Les MMFs permettent aussi de partager de la mémoire entre deux processus²⁵. Il faut pour cela que tous les processus utilisent le nom ou bien le handle du même MMF. Le handle peut être transmis par héritage ou par duplication. Se référer à MSDN pour un exemple de deux processus qui partagent de la mémoire nommée²⁶.

2.5. Modèle de sécurité

Le modèle de sécurité sous Windows est complexe. Nous essayerons de survoler les notions clé et de donner un aperçu global du contrôle d'accès (*authorization*). La problématique de la sécurité peut être résumée ainsi :

1. Faut-il permettre ou non à tel sujet d'effectuer telle opération sur tel objet?
2. Quels événements de succès ou d'échec de ces demandes d'accès faut-il auditer dans un fichier trace (*log*)?

2.5.1. Utilisateur et jeton d'accès

La sécurité de Windows NT est **centrée sur l'utilisateur**, c'est-à-dire que les autorisations d'accès ne concernent pas les applications mais bien les comptes (*accounts*). Il faut donc parler de l'authentification ou comment un utilisateur prouve son identité.

Windows identifie un utilisateur par son nom et le nom de son domaine (par exemple DELL12\Florent sur notre machine de test). Au lieu des noms sous forme de chaîne de caractères, ce sont des identificateurs de

²⁵ http://msdn.microsoft.com/library/en-us/memory/base/sharing_files_and_memory.asp

²⁶ *Creating Named Shared Memory* :
[http://msdn.microsoft.com/\(...\)/memory/base/creating_named_shared_memory.asp](http://msdn.microsoft.com/(...)/memory/base/creating_named_shared_memory.asp)

sécurité ou **SIDs** (*security identifiers*) qui sont utilisés pour identifier de manière univoque des entités. Chaque utilisateur, domaine, groupe local / de domaine, et membre de domaine possède son SID²⁷ unique.

L'utilitaire psgetsid²⁸ permet de traduire un SID en nom complet et le contraire.

```
C:\>psgetsid Administrator

PsGetSid v1.42 - Translates SIDs to names and vice versa
Copyright (C) 1999-2004 Mark Russinovich
Sysinternals - www.sysinternals.com

SID for DELL12\Administrator:
S-1-5-21-1454471165-963894560-725345543-500
```

Lors d'une ouverture de session (*logon*) réussie, le système crée un **jeton d'accès** (*access token*) qui décrit le contexte de sécurité de l'utilisateur. Ce jeton contient²⁹ :

- Le SID du compte utilisateur
- Les SIDs des groupes auxquels ce dernier appartient
- Eventuellement une liste de SIDs restreints
- Un tableau de privilèges (voir § 2.5.3)

Les ressources de l'ordinateur sont formalisées par des **objets**, gérés par le gestionnaire d'objet en mode *kernel*, tels : périphériques, fichiers, processus, threads, événements, ports d'E/S, éléments de synchronisation, sections de mémoire partagée, jetons d'accès, clés de la base de registre, etc.

Les sujets qui manipulent les objets sont les **threads**. Par défaut, un thread hérite de l'identité de son processus, mais il peut aussi usurper temporairement une autre identité³⁰. Pour simplifier, nous ferons l'hypothèse que tous les threads d'un processus possèdent le même contexte de sécurité, à savoir celui de leur processus. Nous parlerons donc uniquement des jetons primaires (*primary tokens*), par opposition aux *impersonation tokens*.

²⁷ Une description de la structure du SID ainsi que les valeurs particulières des SID standards (*well-known*) peuvent être consultées dans [WI] pp. 495-497.

²⁸ Téléchargeable sur <http://www.sysinternals.com/utilities/psgetsid.html>

²⁹ Cf. la structure de données dans [WI] p. 498.

³⁰ Ce mécanisme, appelé *impersonation* en anglais, est décrit dans [WI] pp. 502ss. Il permet à un thread d'une application serveur d'endosser l'identité d'un de ses clients.

Nous avons dit plus haut que la généalogie d'un processus est importante. En voici la raison : le processus explorer.exe, qui est l'ancêtre commun de tous les processus appartenant à l'utilisateur, reçoit le jeton d'accès créé par lsass.exe, et le propage à ses descendants.

2.5.2. Droits, handles et ACLs

Le contrôle d'accès, opéré par le composant **SRM** (*Security Reference Monitor*), concerne une certaine action sur un certain objet. Disons que le rôle du SRM est de résoudre une équation qui prend en entrée l'identité (le jeton) d'un thread, le type d'accès souhaité et les paramètres de sécurité de l'objet, et qui retourne une valeur booléenne.³¹

Cette vérification a lieu lors de l'ouverture d'un handle où l'utilisateur doit spécifier quelles opérations il désire effectuer sur l'objet. Si l'accès est accordé, ce handle-ci ne permettra de réaliser que ces opérations-là. Nous pouvons visualiser les handles dans ProcExp comme expliqué au § 2.4.3.

Les permissions d'accès à un objet sont décrites par la **DAACL** (*Discretionary Access Control List*), qui comporte plusieurs ACEs (*Access Control Entries*). Signalons que l'ordre des entrées dans une ACL est déterminant en raison de l'algorithme de contrôle d'accès. Enfin, les événements à auditer sont décrits dans la **SACL** (System ACL).

Le genre d'accès souhaité dépend du type d'objet. Il est décrit par un masque d'accès (*access control mask*). Tous ces composants et leur fonctionnement sont présentés en détails dans MSDN, ainsi que dans une série d'articles sur le site The Code Project³².

Nous pouvons observer dans ProcExp que Wordpad détient un handle sur la clé de la base de registre HKCU. En ouvrant les propriétés du handle (double-clic) puis en allant dans l'onglet Security, nous voyons la DAACL de cet objet. Nous ignorons quel accès a été accordé pour le handle en question, mais nous trouvons ici l'accès maximal à cet objet qui peut être accordé à tel compte (le groupe Administrators sur la capture d'écran).

³¹ Cf. [WI] au bas de la p. 494.

³² *The Windows Access Control Model* :
<http://www.codeproject.com/win32/accessctrl1.asp>

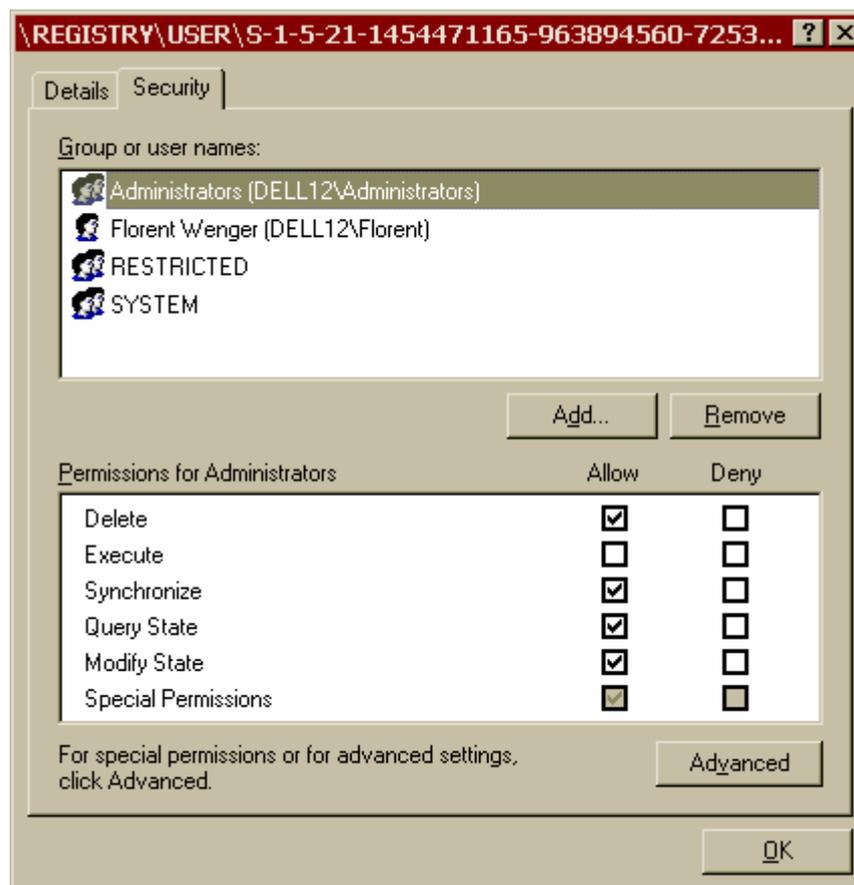


Figure 26 : Propriétés de sécurité d'un handle (Process Explorer)

2.5.3. Privilèges

Certaines opérations du système, comme la sauvegarde des fichiers, serait trop compliquées à implémenter avec les droits. Par exemple, il faudrait ajouter une entrée à la DACL de tous les fichiers pour que l'opérateur de sauvegarde puisse y accéder pour les sauvegarder. Il existe donc des **privilèges**³³ qui autorisent à outrepasser les mécanismes de sécurité usuels. Ces privilèges ne concernent pas des objets en particulier. Ils se rapportent à un utilisateur et font partie de son jeton.

Les privilèges suivants sont particulièrement puissants.

<i>Nom du privilège</i>	<i>Effet pour l'utilisateur</i>
SeCreateTokenPrivilege	Autorise à créer un jeton primaire
SeDebugPrivilege	Autorise à déboguer n'importe quel processus
SeLoadDriverPrivilege	Autorise à charger ou décharger un pilote (mode <i>kernel</i>)
SeRestorePrivilege	Autorise à restaurer (ajouter, supprimer) des fichiers et dossiers et à manipuler leurs autorisations NTFS

³³ Cf. [WI] pp. 516-523. Voir aussi la liste des "*privilege constants*" dans MSDN : http://msdn.microsoft.com/library/en-us/secauthz/security/authorization_constants.asp

SeSecurityPrivilege	Autorise à fixer la stratégie d'audit et à manipuler le journal d'événements
SeTcbPrivilege	Désigne comme faisant partie du système d'exploitation (TCB = Trusted Computer Base)
SeTakeOwnership	Autorise à s'approprier des fichiers et autres objets sans y avoir accès.

Figure 27 : Tableau des privilèges puissants

Ces privilèges ou droits se retrouvent dans le jeton d'accès. Ils peuvent y être activés ou désactivés. ProcExp permet d'afficher le contexte de sécurité d'un processus dans l'onglet Security des propriétés d'un processus. Nous voyons à la Figure 28 le jeton d'accès du module wordpad.exe, lancé depuis un compte administrateur.

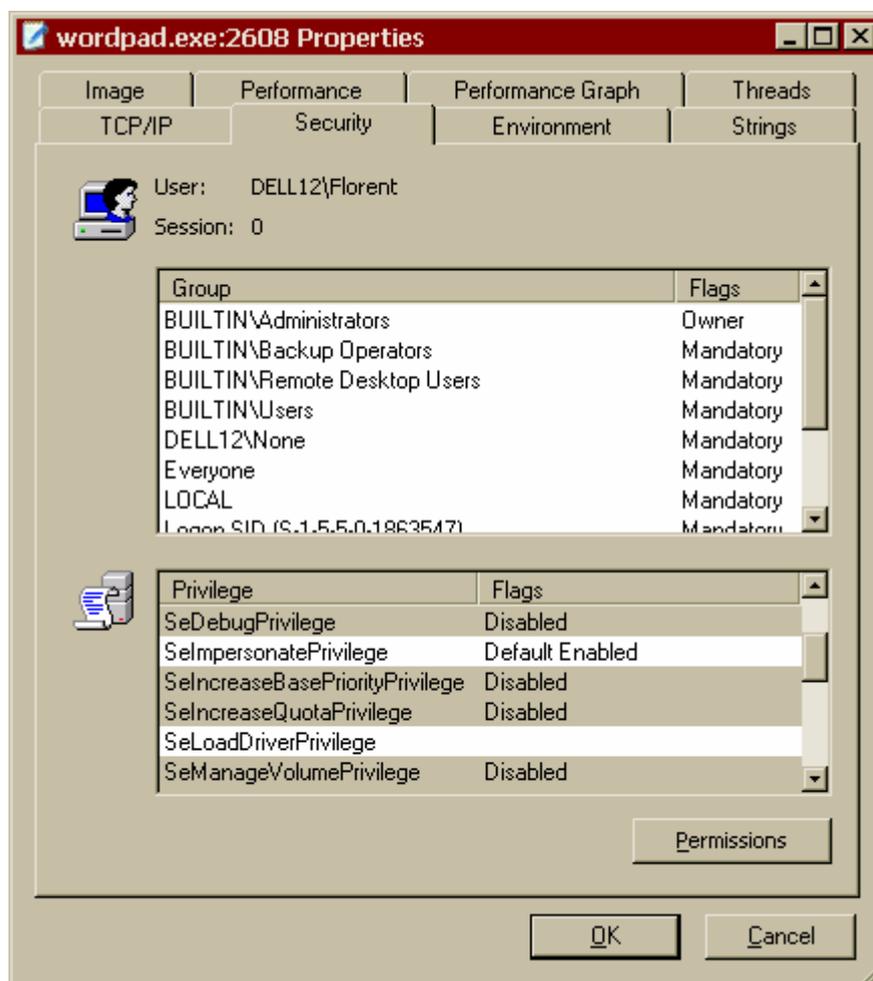


Figure 28 : Jeton d'un processus (Process Explorer)

Dans notre cas, l'utilisateur Florent est administrateur du système. Il dispose de nombreux privilèges même si certains d'entre eux sont

actuellement désactivés (en gris). Le nombre de privilèges d'un processus démarré depuis un compte limité est plus petit.

L'attribution des privilèges à des utilisateurs ou des groupes est déterminée par une stratégie locale. Nous trouvons cette dernière dans *Configuration Panel | Administrative Tools | Local Security Settings* (ou exécuter *secpol.msc*). En développant *Local Policies* puis *User Rights Assignments*, nous voyons par exemple quel seul l'utilisateur Florent détient le privilège *Debug* (au lieu du groupe Administrators par défaut).



Figure 29 : Stratégies locales de sécurité de Windows

De plus, un jeton peut être restreint³⁴ (*restricted token*), c'est-à-dire que des SIDs peuvent être désactivés ou restreints, et des privilèges supprimés. Pour observer un jeton restreint, créons un raccourci vers un exécutable quelconque, puis ouvrons les propriétés avancées (*Properties | Advanced*) et cochons "*Execute with different credentials*". Lançons maintenant le raccourci et validons la boîte de dialogue qui s'affiche. Le jeton du processus est restreint et ne présente plus qu'un seul privilège : *SeChangeNotifyPrivilege*.

En constatant que certains privilèges sont présents dans le jeton, mais sont désactivés, nous pouvons penser que ces privilèges peuvent être activés. Comme l'indique Mark Russinovitch, l'idée est que les privilèges ne doivent être activés qu'au moment où on en a besoin. Par exemple, nous pouvons voir le privilège *SeSystemtimePrivilege* basculer lors d'un changement d'heure³⁵.

³⁴ http://msdn.microsoft.com/library/en-us/secauthz/security/restricted_tokens.asp

³⁵ Cf. expérience dans [WI] p. 518s

Figure 30 : Boîte de dialogue *Run As*

Un privilège peut être activé grâce à la fonction `AdjustTokenPrivileges`. Voyons un exemple de code inspiré d'un tutoriel de CodeGuru³⁶. S'il n'y a pas d'erreur à l'exécution, nous avons activé le privilège `SeShutdownPrivilege` (à vérifier avec ProcExp).

```
HANDLE hToken;
DWORD dwTokenAccess = TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY;
TOKEN_PRIVILEGES tkp;

if ( !OpenProcessToken(GetCurrentProcess(), dwTokenAccess, &hToken) )
    FatalAppExitA(0, "Impossible d'ouvrir le handle!");

LookupPrivilegeValue( NULL, SE_SHUTDOWN_NAME,
    &tkp.Privileges[ 0 ].Luid );
tkp.PrivilegeCount = 1; // Un seul privilège à activer.
tkp.Privileges[ 0 ].Attributes = SE_PRIVILEGE_ENABLED;

if ( !AdjustTokenPrivileges( hToken, FALSE, &tkp, 0, NULL, 0 ) )
    FatalAppExitA( 0, "Erreur lors de l'activation du privilège!" );
CloseHandle( hToken );
```

2.5.4. Autres mécanismes : SRP, DEP

Abordons encore 2 mécanismes de Windows qui améliorent la sécurité. D'abord, les stratégies de restriction logicielle ou **SRP**³⁷ (*Software Restriction Policies*), qui sont configurées dans les *Local Security Setting* (`secpol.msc`), permettent de restreindre l'exécution à certaines images.

³⁶ http://www.codeguru.com/cpp/w-p/win32/tutorials/article.php/c8647_1

³⁷ SRP : [WI] p. 533ss, <http://support.microsoft.com/kb/310791>

Il faut spécifier d'abord un ensemble d'extensions (*Designated File Types*) qui correspondent aux fichiers exécutables, puis les critères d'application de la stratégie à une image en fonction de son chemin complet sur le disque, son empreinte (*hash*), sa zone Internet ou encore son certificat. Les règles peuvent être combinées : on pourrait par exemple limiter l'exécution des binaires à ceux qui se trouvent dans le dossier %WinDir%\System32 ou qui sont signés par tels éditeurs approuvés.

A priori, il est possible de configurer un système de manière à ce que les autorisations NTFS interdisent à un utilisateur limité d'écrire dans tous les répertoires où l'exécution est autorisée. Ainsi, cet utilisateur ne pourrait pas exécuter un programme malicieux, même s'il parvenait à le copier / télécharger sur le disque.

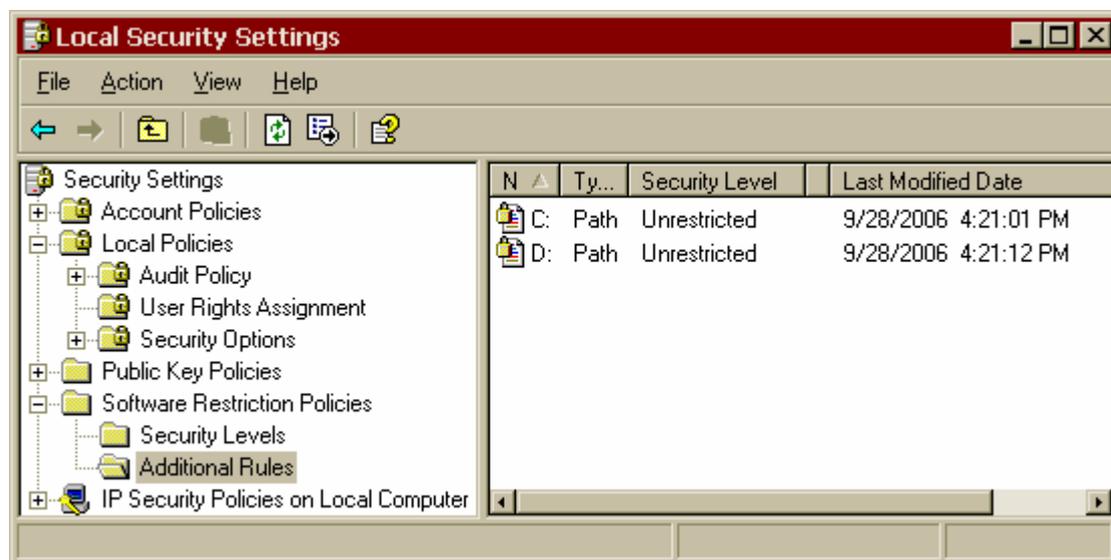


Figure 31 : Stratégies de restriction logicielle

Ensuite, la prévention de l'exécution des données ou **DEP**³⁸ (Data Execution Prevention) a pour but d'empêcher les attaques du type dépassement de tampon (*buffer overrun*) qui consistent à copier du code dans une page de données puis à l'exécuter.

DEP est appliquée de façon logicielle par Windows. Elle est configurée grâce à la valeur de l'option /noexecute dans le fichier de configuration \Boot.ini. Il est possible de l'activer toujours (*AlwaysOn*) ou jamais (*AlwaysOff*), ou bien pour les binaires du système plus les autres exécutables selon un principe de liste blanche (*OptIn*) ou de liste noire (*OptOut*).

³⁸ DEP : <http://support.microsoft.com/kb/875352>

Son application matérielle³⁹ est réalisée par un bit qui permet ou non l'exécution de code à l'intérieur d'une page. Cette technologie est appelée NX (*No Execute page-protection*) par AMD et XD (*Execute Disable bit*) par Intel. Tous les processeurs actuels n'implémentent pas cette technologie. Par exemple, c'est le cas pour le Pentium M 750 Dothan de mon portable, mais pas pour le Pentium 4 moins récent de ma machine de test.

A noter que certaines applications sont incompatibles avec DEP. Pour plus de précisions, utiliser la boîte à outils *Microsoft Application Compatibility Toolkit*. Enfin, ProcExp indique si DEP est activé pour un processus sous l'onglet Image de ses propriétés, comme le montre la Figure 32.

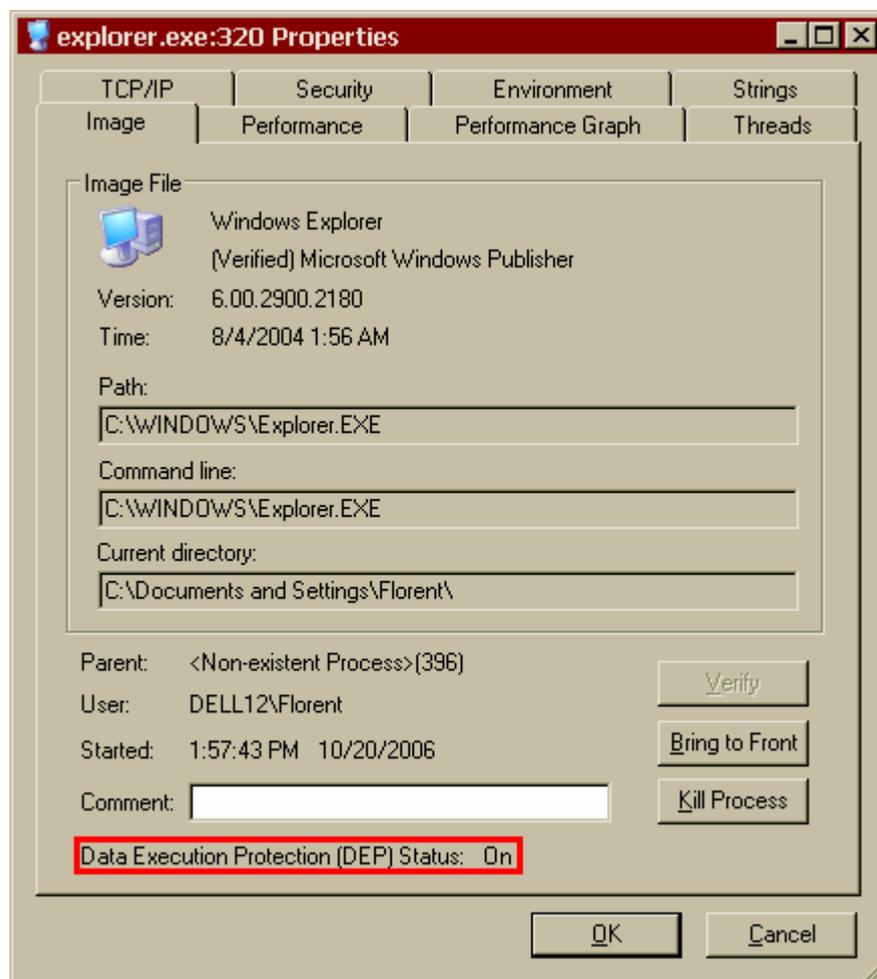


Figure 32 : Statut de DEP d'un processus (Process Explorer)

³⁹ Cf. l'article *Bypassing Windows Hardware-enforced DEP* : <http://uninformed.org/?v=2&a=4&t=sumry>

2.5.5. Sources

- "Windows NT Security" (article de fond) :
http://msdn.microsoft.com/library/en-us/dnsecure/html/msdn_ntprog.asp
- "Windows NT Security Theory and Practice" (autre article essentiel) :
http://msdn.microsoft.com/library/en-us/dnsecure/html/msdn_seccpp.asp
- "Access Control" (page d'accueil MSDN) :
http://msdn.microsoft.com/library/en-us/secauthz/security/access_control.asp
- Tout le chapitre 8 de [WI] intitulé... "Security".

2.6. Conclusion

Pour conclure cette première partie, disons que Windows recèle encore bien d'autres fonctionnalités. Le meilleur moyen de continuer son étude est de poursuivre la lecture de [WI] tout en multipliant les expériences pratiques. Mais nous devons nous arrêter ici pour passer à la deuxième partie, quitte à approfondir ultérieurement certains points selon nos besoins.

3. Techniques utilisées par les rootkits

3.1. Qu'est-ce qu'un rootkit?

3.1.1. Définition

D'une manière générale, un **rootkit** est un programme conçu pour contrôler le comportement d'une machine donnée. Voici une définition plus complète tirée de l'encyclopédie Wikipédia⁴⁰ :

« On nomme rootkit un programme ou ensemble de programmes permettant à un pirate de maintenir - dans le temps - un accès frauduleux à un système informatique. Le pré-requis du rootkit est une machine déjà compromise.

La fonction principale du rootkit est de camoufler la mise en place d'une ou plusieurs portes dérobées. Ces portes dérobées (utilisables en local ou à distance) permettent au pirate de s'introduire à nouveau au cœur de la machine sans pour autant exploiter une nouvelle fois la faille avec laquelle il a pu obtenir l'accès frauduleux initial, qui serait tôt ou tard comblée. (...)

A la différence d'un virus informatique ou un ver de nouvelle génération, un rootkit ne se réplique pas. (...)

Un rootkit ne permet pas en tant que tel de s'introduire de manière frauduleuse sur une machine saine. En revanche, certains rootkits permettent la collecte des mots de passe qui transitent par la machine corrompue. Ainsi, un rootkit peut indirectement donner l'accès à d'autres machines. (...)

Un rootkit a pour but principal la furtivité, il permet par exemple de cacher des processus, fichiers et autres clefs de la base de registres... Le rootkit est souvent couplé à d'autres programmes tels qu'un sniffeur de frappe (keylogger) ou de paquets (packet sniffer). »

3.1.2. Types de rootkits

Les rootkits peuvent se trouver à différents niveaux dans un ordinateur⁴¹ :

1. **Virtualisation** : en modifiant la séquence d'amorce (*boot*) afin d'être chargé en premier, le rootkit s'intercale entre le matériel et le système d'exploitation qui est alors exécuté sur une machine virtuelle.
2. **Noyau** (*kernel*) : le rootkit s'insère dans le noyau en modifiant son code ou en y ajoutant du code via le chargement d'un pilote de périphérique (*device driver*).

⁴⁰ <http://fr.wikipedia.org/wiki/Rootkit>

Voir aussi sur les articles suivants :

Rootkits: The "r00t" of Digital Evil : <http://www.omninerd.com/2005/11/22/articles/43>

Les rootkits :

<http://3psilon.info/Les-rootkits.html>

⁴¹ <http://en.wikipedia.org/wiki/Rootkit>

3. **Bibliothèque** (*library*) : le rootkit intercepte les appels au système et remplace les fonctions de l'API par des versions qui masquent les informations concernant l'attaquant.
4. **Application** : le rootkit remplace des binaires par des imitations contenant par exemple des chevaux de Troie, ou modifie le comportement d'applications existantes par divers moyens.

Rien n'empêche un rootkit de traverser plusieurs niveaux : les rootkits pour Windows utilisent souvent en combinaison les modes utilisateur (niveaux 3 et 4) et noyau. En effet, il est difficile d'écrire un pilote de périphérique car la moindre erreur peut entraîner le plantage du système. Par conséquent, tout ce qui peut être écrit en mode utilisateur l'est fait.

3.1.3. Le rootkit NtIllusion

Notre étude s'inspire de l'article de Kodmaker, "*Win32 portable userland rootkit*", paru dans le n°62 de Phrack⁴², et du code source de son rootkit *proof of concept* NtIllusion⁴³.

1. Puisque l'installation d'un rootkit présuppose que la machine cible a déjà été compromise, on ne se préoccupe pas de savoir comment le code malicieux est mis en place.
2. Le rootkit est employé précisément lorsque l'assaillant n'a pas accès à un compte administrateur. Il doit s'exécuter dans la session d'un **simple utilisateur** qui ne dispose pas de privilèges puissants (p.ex. Debug).
3. Il contrôle de près tous les programmes en mode *user* qui pourraient être utilisés pour énumérer les **fichiers**, les **processus**, les **clés de la base de registre** (BDR) et les **ports TCP/IP** actifs, de sorte qu'ils ne révèlent pas des choses indésirables.
4. Le rootkit cherche à intercepter un mot de passe administrateur, qui serait entré par exemple lors de l'appel à la commande "*Run as*"⁴⁴.

3.2. Généralités

3.2.1. Motivation

On peut mettre en doute la valeur éthique de notre démarche. Est-ce mal de dévoiler des techniques utilisées par des *malwares*? Cela ne va-t-il pas faciliter la création de nouveaux rootkits et constituer une menace pour la sécurité informatique?

⁴² http://www.phrack.org/archives/62/p62-0x0c_Win32_Portable_Userland_Rootkit.txt

⁴³ <https://www.rootkit.com/vault/kdm/NTIllusion.rar>

(Il faut être enregistré sur le site Rootkit pour le télécharger).

⁴⁴ Lancée via l'item "Exécuter en tant que" dans le menu contextuel d'un raccourci ou d'un exécutable, ou via la commande *runas* dans l'invite de commandes.

Tout d'abord, la plupart des techniques qui suivent sont décrites publiquement sur Internet. N'importe qui peut donc y avoir accès à condition de savoir chercher et de faire suffisamment d'efforts⁴⁵.

Ensuite, ces techniques se contentent dans la plupart des cas d'utiliser l'API de Windows⁴⁶, ce qui veut dire qu'il s'agit de mécanismes conçus, mis à disposition et documentés par Microsoft. Aucune d'elles n'exploitent de failles.

Enfin, ces mêmes techniques sont employées par des logiciels légitimes tels anti-virus, pare-feux, etc. Ce n'est pas la technique en soi qui est mauvaise, c'est l'utilisation qu'on peut en faire. Mieux comprendre le fonctionnement des rootkits permet de s'en protéger et de les détecter.

3.2.2. Evolution de la technique

Il est évident que la technique est en perpétuelle évolution. Dans le domaine qui nous intéresse, les *hackers* et les éditeurs de logiciels de sécurité se livrent à une véritable course aux armements. Les premiers doivent constamment inventer des nouvelles ruses pour échapper aux seconds. C'est le jeu du chat et de la souris.

Ceci amène deux remarques :

1. Il est illusoire de penser que les techniques présentées ici (et décrites sur le Web) sont les plus récentes qui soient. En général, les techniques postées sur le site Rootkit datent d'au moins 2 ans car les *hackers* ne révèlent pas leurs derniers secrets⁴⁷.
2. Il est difficile de garantir dans quelles circonstances ou jusqu'à quand telles techniques fonctionneront, puisque Microsoft et d'autres sociétés s'efforcent d'améliorer sans cesse la sécurité⁴⁸.

Nous n'essayerons pas de donner des solutions prêtes à l'emploi, mais de donner les principes et des exemples de mise en œuvre. Il restera, comme toujours, à l'ingénieur de choisir et d'adapter les méthodes à sa situation particulière selon ses besoins spécifiques.

3.2.3. Méthode statique ou dynamique?

L'attaque de logiciels peut être :

- **Statique** : modification des fichiers image sur le disque dur.
- **Dynamique** : altération du module en mémoire (RAM).

⁴⁵ Je ne crois pas que l'ignorance favorise la sécurité. Mais bien sûr, je n'approuverais pas la publication d'un rootkit fonctionnel à 100% que tous les *scripts kiddies* du monde pourraient déployer à leur guise...

⁴⁶ Rappelons-nous que notre étude se limite à Windows XP Pro SP2. Nous parlerons plus loin des différences avec Windows Vista.

⁴⁷ Selon le principe : « *Share Your Old Stuff, Keep Your Good Stuff* ».

⁴⁸ Cependant, pour des raisons de compatibilité, Windows ne peut pas renier tout son héritage d'un coup et doit donc conserver certaines fonctionnalités critiques en termes de sécurité.

Parmi elles les techniques d'attaques des *malwares*, nous ignorerons celles qui consistent à modifier les fichiers sur le disque au profit des méthodes dynamiques qui agissent en mémoire.

En effet, comme nous l'avons vu dans la première partie, il est aisé de modifier l'IAT, d'insérer ou d'altérer du code, de changer le point d'entrée ou encore d'ajouter une section dans un fichier image. Toutefois, un simple contrôle d'intégrité (MD5, SHA-1 ou autre) suffirait à déceler ces modifications, de même qu'une recherche de signature (*pattern*) sur le disque démasquerait un *malware* connu.

Les rootkits sont plus discrets que cela et agissent **en mémoire** exclusivement. L'installation sur le disque de NtIllusion revient à copier quelque part un programme (kNtiLoader.exe) et une DLL (kNtIllusion.dll). Les images de toutes les applications contrôlées par le rootkit demeurent strictement inchangées.

Notons au passage l'arrivée d'une nouvelle génération de *malwares* qui n'écrivent absolument rien sur le disque dur, ne modifient que quelques mots de données de l'OS et sont donc extrêmement difficiles à détecter. Bien qu'ils ne survivent pas au *reboot*, ils exploitent les serveurs qui tournent en permanence pour infecter tous leurs clients à chaque démarrage⁴⁹.

3.2.4. Piratage de l'API

Enfin, voyons le principe de fonctionnement du rootkit. Ses techniques de dissimulation (de fichiers, répertoires, processus, clés de la BDR) nécessitent de modifier le comportement de certaines fonctions de l'API de Windows.

L'idée est simple : prenons un programme qui souhaite lister les processus existants (p.ex. Task Manager). Ce programme va demander à l'OS via l'API une liste de tous les processus. Voici la circulation des informations en temps normal :

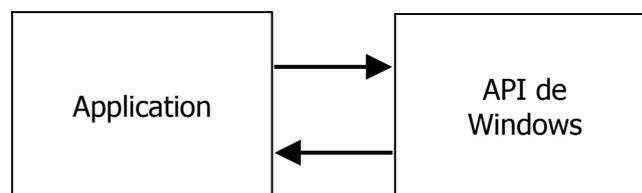


Figure 33 : Flux normal des données de l'API

⁴⁹ Voir les 2 présentations de BlackHat suivantes :

- "Hide'n'Seek : Anatomy of Stealth Malware" par Gergely Erdélyi de F-Secure [http://www.blackhat.com/\(...\)/bh-eu-04-erdelyi/bh-eu-04-erdelyi.pdf](http://www.blackhat.com/(...)/bh-eu-04-erdelyi/bh-eu-04-erdelyi.pdf)
- "Rootkits vs. Stealth, by Design Malware" par Joanna Rutkowska www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Rutkowska.pdf

Supposons maintenant qu'un rootkit est installé et qu'il dissimule un processus X qui est en fait une *backdoor*. Le rootkit doit absolument supprimer l'entrée du processus X de la liste des processus retournée par l'API. Le rootkit s'accroche (*hook*) entre l'application et l'API, afin de d'intercepter les appels et de filtrer leurs résultats.

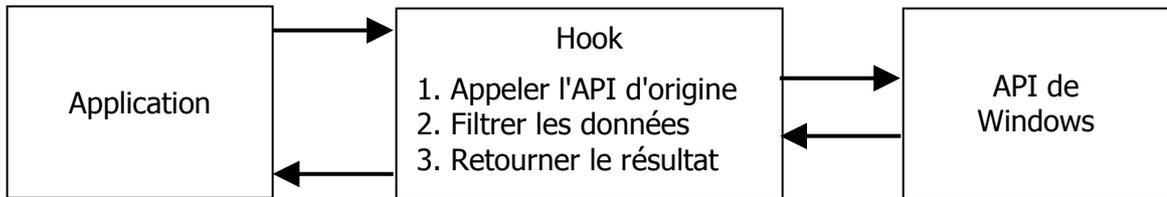


Figure 34 : Flux détourné des données de l'API

En bref, pirater l'API revient à :

1. Injecter nos fonctions de remplacement dans un processus (§ 3.4)
2. Rediriger les appels à certaines fonctions de certaines DLLs (§ 3.5)

3.3. Techniques fondamentales

3.3.1. Localisation des processus

Mais tout d'abord, voyons comment localiser un processus. Nous avons dit que la sécurité de Windows est orientée utilisateur. Par conséquent, un processus lancé par Eve peut manipuler tous les processus appartenant à Eve.

Le but visé est ici de rechercher les instances du Wordpad en exécution et de les fermer. Il s'agit d'abord de prendre un instantané des processus existants grâce à l'API `CreateToolhelp32Snapshot`, puis de parcourir la liste des processus obtenue (`Process32First`, `Process32Next`). Enfin, nous tentons d'ouvrir chaque processus (`OpenProcess`). Si l'opération aboutit et que le nom de l'exécutable est "wordpad.exe", nous transmettons le handle à une fonction de callback.

```

typedef void (HandleProc)( HANDLE ); // Type de fonction de callback.

BOOL FindProcess( PWSTR exeFile, DWORD desiredAccess, HandleProc
  callbackFunction ) {

  HANDLE processSnap, currentProcess;
  PROCESSENTRY32 pe32;

  // Crée un instantané des processus du système.
  processSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
  if ( !processSnap ) return FALSE;
  pe32.dwSize = sizeof PROCESSENTRY32; // Initialisation.

  // Parcourt la liste des processus.
  if ( !Process32First( processSnap, &pe32 ) ) return FALSE;
  do {

```

```

// Tente d'ouvrir le processus avec les droits désirés.
currentProcess = OpenProcess( desiredAccess, FALSE,
    pe32.th32ProcessID );
if ( !currentProcess ) continue; // Handle pas accordé!
// Traite l'exécutable si son nom d'image correspond.
if ( _wcsicmp( pe32.szExeFile, exeFile ) == 0 )
    callbackFunction( currentProcess ); // Traitement.
CloseHandle( currentProcess );
} while ( Process32Next( processSnap, &pe32 ) );

CloseHandle( processSnap );
return TRUE;
}

```

Créons maintenant la fonction de callback qui doit avoir une signature⁵⁰ identique à `HandleProc`. Les opérations autorisées sur le processus dépendent des droits d'accès obtenus avec le handle.

```

// Fonction de callback pour la recherche de processus.
void KillProcess( HANDLE process ) {
    TerminateProcess( process, 0 );
}

```

Il ne reste qu'à appeler `FindProcess` en spécifiant le bon droit d'accès⁵¹.

```

// Termine toutes les instances du Wordpad.
FindProcess( L"wordpad.exe", PROCESS_TERMINATE, KillProcess );

```

L'exécution de ce programme par l'utilisateur limité Eve ferme bien toutes les instances du Wordpad lancées par ce même utilisateur, mais pas les autres. A noter qu'un administrateur peut tuer aussi les processus des utilisateurs limités.

Cette technique telle quelle a peu de chances d'être rencontrée dans un vrai rootkit, puisque la fermeture brutale d'un programme éveillerait la suspicion de l'utilisateur et nuirait à la furtivité du rootkit. Mais elle illustre bien les possibilités de l'API de Windows.

3.3.2. Vue d'ensemble des hooks

Une autre fonctionnalité intéressante de l'API est l'accrochage (*hooking*). Par définition, un **hook**⁵² est un mécanisme par lequel une fonction peut intercepter des événements (messages, actions de la souris, frappes du clavier) avant qu'ils atteignent une application⁵³.

⁵⁰ La signature d'une fonction spécifie ses arguments (leur nombre, leur type et leur ordre) et son résultat.

⁵¹ cf. "Process Security and Access Rights" : <http://msdn2.microsoft.com/en-us/library/ms684880.aspx>

⁵² Voir l'API `SetWindowsHookEx` dans MSDN et l'article "Win32 Hooks" : <http://msdn2.microsoft.com/en-us/library/ms997537.aspx>

⁵³ Pour chaque type de message, une application peut choisir de conserver le comportement par défaut en transmettant le message à l'API `DefWindowProc`, ou

La fonction `SetWindowsHookEx` installe une procédure définie par l'utilisateur, appelé la fonction de filtre, afin de surveiller certains événements concernant :

- soit un thread particulier, i.e. une application ;
- soit tous les threads d'un même bureau (*desktop*). On parle alors de crochet global (*system-wide hook*).

La Figure 35 donne le principe d'une fonction de filtre qui traite les frappes de clavier. Lorsque survient un événement signalant qu'une touche a été pressée, la fonction de filtre peut passer le message tel quel (en vert), modifier l'événement (en bleu) ou même le supprimer (en rouge). Il est ainsi très facile de réaliser un *keylogger* qui enregistre toutes les frappes de touches⁵⁴.

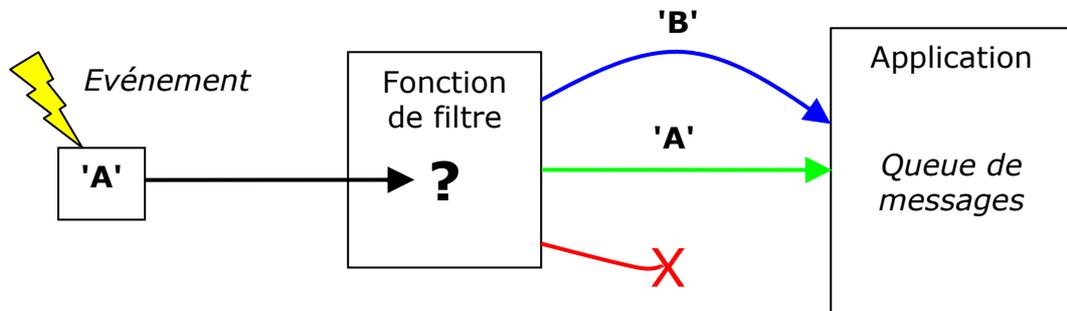


Figure 35 : Hook et fonction filtre

Notons encore que plusieurs crochets peuvent être installés en chaîne. Par conséquent, la procédure de traitement devrait appeler la fonction `CallNextHookEx` afin de transmettre l'événement au crochet suivant. Je ne donne pas d'exemple de code car il existe de très nombreux exemples de mise en œuvre sur le Web. Il suffit de chercher le mot-clé "hook" sur les sites CodeGuru ou CodeProject.

Dans le cas d'un crochet global, la fonction de filtre doit se trouver dans une DLL. Pourquoi? Lorsque l'événement surveillé survient, c'est le thread courant qui exécute la procédure du crochet. Il faut donc que le code à exécuter se trouve dans l'espace d'adresses virtuelles du processus auquel appartient ce thread. Pour ce faire, Windows injecte automatiquement la DLL contenant la procédure dans l'EAV des processus concernés.

Cela nous amène naturellement à la section suivante.

bien de définir un traitement spécifique. Dans une application Win32 typique, la gestion des événements est implémentée dans la fonction `WndProc`.

⁵⁴ cf. "Keystroke Logger and More..." par Zhefu Zhang

http://www.codeguru.com/cpp/w-p/system/security/article.php/c5761_1

3.4. Techniques d'injection

L'**injection** est tout simplement le "parasitage" de l'espace d'adresses d'un processus, dans lequel nous plaçons du code qui n'y a rien à faire en temps normal.

3.4.1. Problématique du *mapping*

Puisque chaque processus vit dans son propre espace d'adresses virtuelles, le rootkit doit parvenir à s'introduire ou s'injecter dans l'EAV de tous les processus qu'il vise. Cela revient à copier ou plutôt mapper tout son code et ses données. Avant d'aller plus loin, nous devons mieux comprendre comment la gestion de la mémoire s'opère dans l'**espace utilisateur**⁵⁵.

Pour rappel, la mémoire est gérée par pages. A un moment donné, chaque page peut se trouver soit en RAM, soit sur le disque si le gestionnaire de mémoire a décidé de la paginer pour libérer de la mémoire physique.

Les modules (EXEs et DLLs) sont constitués de code et de données. Les pages de code, qui ne sont pas modifiées, sont lues depuis l'image de l'exécutable. Comme nous l'avons dit au § 2.4.4 (voir la note 24 p. 32), le fichier PE est verrouillé pendant son exécution et ne peut pas être supprimé. Les pages de données, elles, sont stockées temporairement dans le fichier d'échange du système (*system pagefile*) puisqu'elles sont censées être volatiles (voir la Figure 25).

Prenons maintenant plusieurs processus qui se servent de la même DLL. Afin d'économiser la **mémoire physique**, Windows ne charge qu'une copie du code de la DLL en RAM. Cela se défend puisque par défaut ces pages sont marquées EXECUTE_ONLY, donc elles ne seront jamais écrites. Inutile donc de stocker plusieurs fois exactement les mêmes informations en RAM!

Mais les processus ont chacun leur propre vue logique de la mémoire physique qui est appelée EAV. En **mémoire virtuelle**, une même DLL peut être mappée à (débuter à) des adresses différentes selon les processus, en raison d'éventuels relogements dans certains processus et pas dans d'autres⁵⁶. La Figure 36 montre 3 processus qui partagent une même page de code. Même si chaque processus accède à cette page via des adresses virtuelles distinctes, il atteint en réalité toujours la même plage d'adresses physiques.

⁵⁵ Voir l'article de MSDN "*Managing Memory-mapped files in Win32*" :
<http://msdn2.microsoft.com/en-us/library/ms810613.aspx>

Et aussi "*Memory Protection*" :

<http://msdn2.microsoft.com/en-us/library/aa366785.aspx>

⁵⁶ Remarque : certaines DLLs de Windows ne peuvent être relogées en aucun cas. Selon Robert Kuster, il s'agit de kernel32.dll, user32.dll et ntdll.dll :

<http://www.codeproject.com/threads/winspy.asp#appendixes>

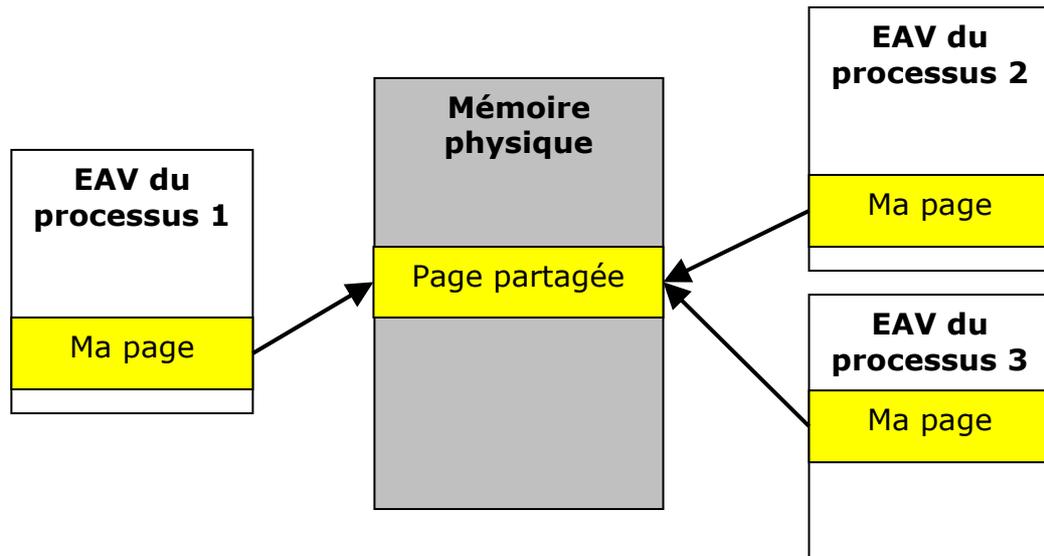


Figure 36 : Pages partagés avant modification

Maintenant, cette DLL maintient probablement dans ses variables des informations qui sont propres à chaque processus. Le contenu de ses pages de données est donc susceptible de varier selon les processus. Mais là encore, Windows s'efforce d'économiser au maximum la mémoire.

Ces pages sont marquées `COPY_ON_WRITE`. Cela signifie que tant que le contenu d'une page est le même pour tous les processus, il n'en existe qu'une seule copie en mémoire. Dans tous les processus qui utilisent cette DLL, cette page virtuelle pointe vers la même page physique, comme à la Figure 36.

Mais lorsqu'un processus écrit dans une page `COPY_ON_WRITE`, le gestionnaire de mémoire crée une copie privée de cette page pour ce processus. Les autres processus référencent toujours la même page inchangée⁵⁷. Le résultat se trouve à la Figure 37.

Tout cela pour dire que si un *malware* veut altérer le code d'une DLL partagée par plusieurs processus, il doit d'abord changer les attributs des pages concernées pour s'accorder le droit d'y écrire. Ensuite, il faut qu'il effectue la modification à l'intérieur de l'EAV de chaque processus cible. Pourquoi? Parce cette modification s'effectue en mémoire virtuelle et que le gestionnaire de mémoire applique alors le même mécanisme `COPY_ON_WRITE` aux pages de code.

⁵⁷ cf. [WI] pp. "Shared memory & mapped files" 386-388 & "Copy-On-Write" 392-394.

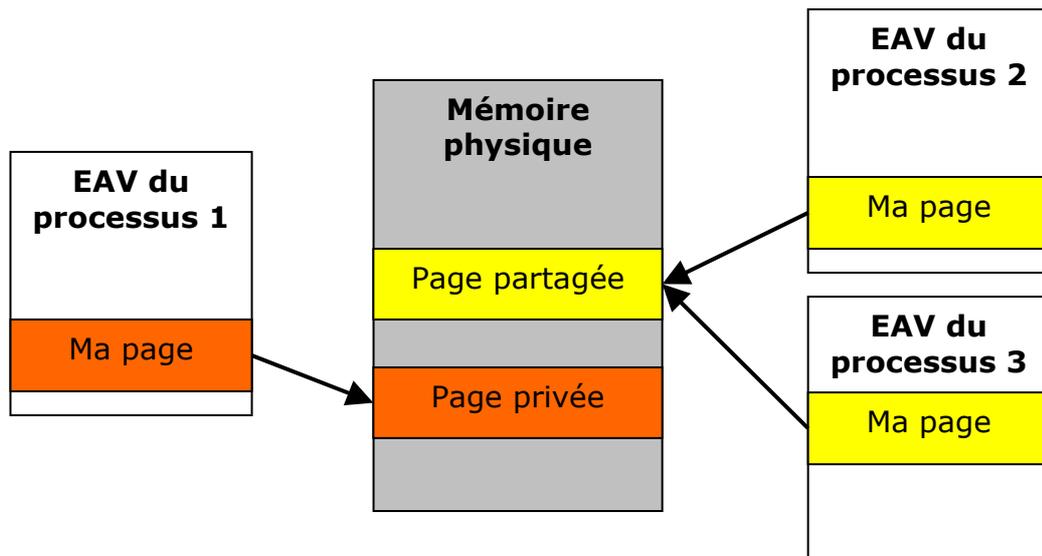


Figure 37 : Pages partagées après modification

Pour clore cette section, disons encore que si un *malware* parvient à modifier directement la mémoire physique, cela aura un impact sur tous les processus sans exception. La manipulation de la mémoire physique est possible via l'objet `\Device\PhysicalMemory`, dont l'accès en mode utilisateur est restreint au compte SYSTEM⁵⁸. Bien sûr, le mode *kernel* autorise de manipuler la mémoire physique (données SMBIOS, RAM, etc.).

3.4.2. Injection de DLL via la base de registre

Il existe plusieurs techniques d'injection que nous allons voir maintenant. La façon la plus simple (et la moins discrète) d'injecter une DLL dans tous les processus consiste à placer son chemin dans la clé de la BDR suivante :

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows**AppInit_DLLs**

Cette technique est très facilement testable en modifiant la clé `AppInit_DLLs`. Après redémarrage de la machine⁵⁹, nous pouvons vérifier que notre DLL de test (`InjectedDLL.dll`) a bien été injectée dans tous les processus, y compris Explorer, Winlogon et Lsass, comme le montre la Figure 38. J'ai utilisé la fonctionnalité très pratique de recherche de handle ou de DLL de ProcExp (Ctrl+F).

⁵⁸ Un tel accès a d'ailleurs été supprimé à partir de Windows 2003 Server SP1 : [http://technet2.microsoft.com/WindowsServer/en/Library/\(...\)](http://technet2.microsoft.com/WindowsServer/en/Library/(...))

⁵⁹ A noter qu'il n'est pas nécessaire de redémarrer pour que le changement soit effectif. En effet, Windows lit le contenu de cette clé à chaque création de processus.

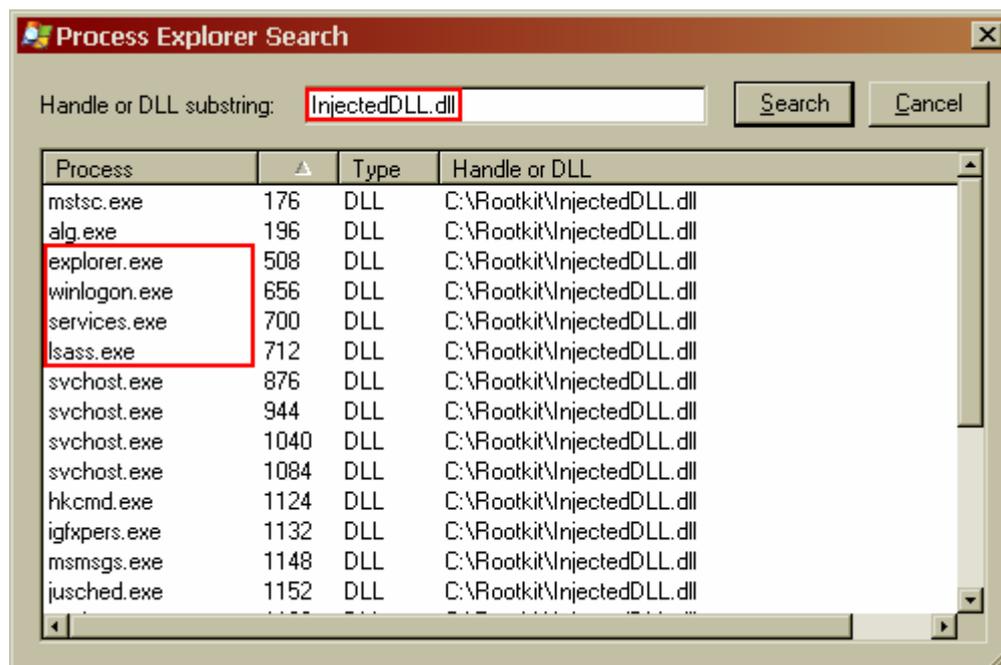


Figure 38 : Recherche de DLL ou de handle (Process Explorer)

Observons que, parmi tous les processus existants sur la machine, seuls les processus System, sms.exe et csrss.exe échappent à cette injection. Heureusement, un utilisateur limité ne peut pas écrire dans cette clé! Ce n'est donc pas une technique utilisable avec nos contraintes, mais elle aide à comprendre les suivantes.

3.4.3. Injection de DLL grâce aux hooks

Comme c'est toute la DLL qui est injectée et pas seulement la procédure du crochet (voir § 3.3.2), certains rootkits se servent des hooks pour s'injecter dans tous les **processus graphiques**, c'est-à-dire qui importent user32.dll. Le rootkit installe un hook, pas pour intercepter un événement, mais pour que Windows injecte sa DLL ; dès que cela est fait, le rootkit supprime le hook.

Remarquons que :

- La DLL n'est chargée qu'au moment où l'événement arrive. Il faut donc bien choisir l'événement à surveiller⁶⁰.
- Les applications en ligne de commande (CLI) telles que cmd et netstat échappent à cette méthode. Nous verrons plus loin comment les infiltrer.

3.4.4. Injection de DLL avec CreateRemoteThread & LoadLibrary

D'abord, il faut se rappeler que notre DLL n'a habituellement rien à faire avec l'exécutable du processus et n'est donc référencée d'aucune

⁶⁰ NtIllusion utilise le type de crochet WH_CBT (Computer Based Training).
Voir "CBTProc Function" : <http://msdn2.microsoft.com/en-us/library/ms644977.aspx>

manière dans ce dernier. Heureusement pour nous, l'API LoadLibrary⁶¹ offre précisément de charger un module (DLL ou EXE) dans l'espace d'adresses du processus appelant.

Cela signifie que nous ne pouvons pas appeler LoadLibrary depuis un thread du processus attaquant. Nous devons créer un thread distant⁶² (*remote thread*) à l'intérieur du processus cible. La difficulté vient du fait que la valeur de l'argument de LoadLibrary doit se trouver dans l'EAV du processus cible.

Il faut d'abord obtenir un handle (OpenProcess) avec les droits suivants, nécessaires aux opérations sur le processus :

- PROCESS_QUERY_INFORMATION
- PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_VM_OPERATION
- PROCESS_CREATE_THREAD :

Voici la marche détaillée à suivre ensuite. Les vérifications d'erreurs ont été supprimées pour alléger le code.

1. Stocker dans un tampon le chemin complet de la DLL à injecter :

```
#define INJECTED_DLL "InjectedDLL.dll"
// (...)
char szDllPath[ MAX_PATH ];
// Récupère le chemin complet de la DLL.
HMODULE hInjected = GetModuleHandleA( INJECTED_DLL );
if ( hInjected == NULL ) { // Si la DLL n'est pas déjà mappée...
    LoadLibraryA( injectedDll ); // ... effectue son chargement.
    hInjected = GetModuleHandleA( INJECTED_DLL );
}
GetModuleFileNameA( hInjected, szDllPath, MAX_PATH );
```

2. Allouer un bloc mémoire de taille suffisante et y copier le chemin de la DLL :

```
PVOID pMem = VirtualAllocEx( hProcess, NULL, MAX_PATH, MEM_COMMIT,
    PAGE_EXECUTE_READWRITE );
WriteProcessMemory( hProcess, pMem, (LPVOID) szDllPath, MAX_PATH,
    NULL );
```

⁶¹ Comme pour de nombreuses fonctions de Windows, il existe 2 versions de l'API LoadLibrary : une version ANSI (caractères sur 1 octet) LoadLibraryA et une version Unicode (caractères sur 2 octets) LoadLibraryW.

⁶² Il est aussi possible de parasiter un thread existant du processus. Le thread peut être suspendu (SuspendThread) et repris (ResumeThread). Son contexte (ensemble des registres dont le pointeur d'instruction EIP) peut être manipulé en lecture (GetThreadContext) et en écriture (SetThreadContext). Je n'ai pas mis en œuvre cette technique intrusive qui nécessite probablement le privilège Debug.

- Créer un thread distant qui va exécuter l'API LoadLibraryA, exportée par le module kernel32.dll, dont nous aurons préalablement obtenu l'adresse :

```
HMODULE hKernel32 = GetModuleHandleA( "kernel32.dll" );
LPTHREAD_START_ROUTINE ThreadProc = (LPTHREAD_START_ROUTINE)
    GetProcAddress( hKernel32, "LoadLibraryA" );
HANDLE hThread = CreateRemoteThread( hProcess, NULL, 0, ThreadProc,
    pMem, 0, NULL );
```

- Attendre la fin du thread et récupérer son code de retour qui contient l'adresse de base du module injecté :

```
DWORD hLibModule; // Adresse de base de la DLL injectée.
WaitForSingleObject( hThread, INFINITE ); // Synchronisation.
GetExitCodeThread( hThread, &hLibModule );
```

- Remplir avec des zéros le bloc de mémoire (facultatif) puis le libérer, et fermer le handle :

```
SecureZeroMemory( szDllPath, MAX_PATH );
WriteProcessMemory( hProcess, pMem, (LPVOID) szDllPath, MAX_PATH,
    NULL );
VirtualFreeEx( hProcess, pMem, MAX_PATH, MEM_RELEASE );
CloseHandle( hThread );
```

L'injection de notre DLL dans le processus Wordpad n'échappe pas à OllyDbg, comme en témoigne la capture d'écran à la Figure 39. Nous voyons la création et la fin du thread distant n° 0x588, qui retourne (*exit code*) l'adresse de base de notre DLL (0x1000'0000).

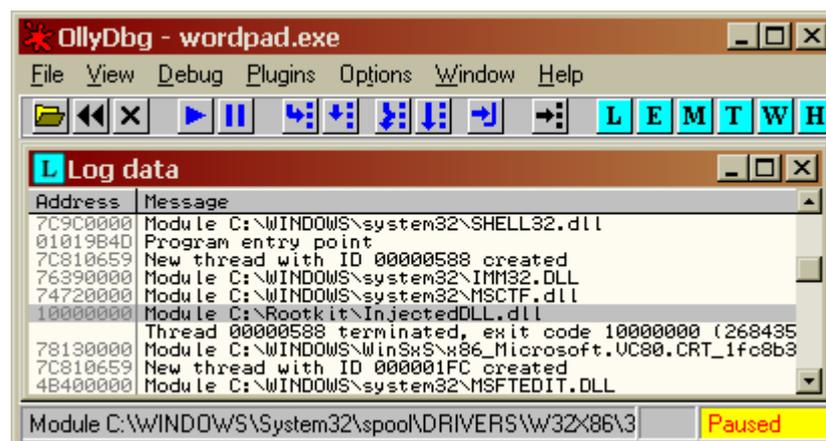


Figure 39 : Injection de DLL avec CreateRemoteThread (OllyDbg)

Voilà, notre DLL est injectée. Si elle dépend d'autres DLLs, ces dernières seront également chargées par Windows le cas échéant.

3.4.5. Injection de code avec WriteProcessMemory

La 4^{ème} façon d'injecter du code est de copier un bloc de mémoire dans l'EAV du processus cible, contenant les instructions et les données voulues. Cette technique est vraiment pointue! En plus de nécessiter une compréhension approfondie du langage machine, de l'architecture de la machine cible et du processus de compilation, elle impose de nombreuses contraintes sur le code à injecter.

C'est pourquoi nous ne l'implémenterons pas, mais nous demanderons au lecteur de se reporter à l'excellent article suivant qui présente également les deux techniques précédentes (*hooking* et injection de DLL) :

"3 ways to inject your code into another process" par Robert Kuster
<http://www.codeproject.com/threads/winspy.asp>

Voici les grandes étapes de cette technique :

1. Obtenir un handle sur le processus distant (OpenProces).
2. Allouer deux blocs de mémoire dans l'espace d'adresses du processus, l'un pour les données, l'autre pour le code à injecter (VirtualAllocEx).
3. Copier la fonction et les données injectées dans la mémoire récemment allouée (WriteProcessMemory).
4. Démarrer le thread distant qui exécute la fonction injectée (CreateRemoteThread).
5. Attendre que le thread distant se termine (WaitForSingleObject).
6. Récupérer le résultat du processus distant (ReadProcessMemory ou GetExitCodeThread).
7. Libérer la mémoire allouée et fermer les handles ouverts (VirtualFreeEx, CloseHandle).

Au vu de la difficulté de cette méthode, il est recommandé de ne l'utiliser que lorsqu'on a seulement quelques instructions à injecter...

3.5. Techniques d'interception

Intercepter une fonction revient à rediriger les appels à cette fonction vers une fonction de remplacement. Là encore, il y a plusieurs manières de le faire.

3.5.1. A quel niveau?

Il faut garder à l'esprit que, lors de l'appel à une API, le flux de contrôle traverse plusieurs couches. Nous pouvons l'observer avec un débogueur tel que LiveKD⁶³. Par exemple, un appel à l'API CreateFile circule ainsi (format : module!fonction)⁶⁴ :

⁶³ Téléchargeable ici (KD vient de Kernel Debugger) :

<http://www.microsoft.com/technet/sysinternals/Utilities/LiveKd.msp>

⁶⁴ Pour les amateurs de *reversing* : l'API kernel32!CreateFileW appelle la fonction native ntdll!NtCreateFile (qui pour une raison que j'ignore possède un alias nommé ntdll!ZwCreateFile), qui à son tour fait appel au service 0x25 du système. Le passage en mode *kernel* s'effectue par l'instruction machine SYSENTER, à la suite de quoi le

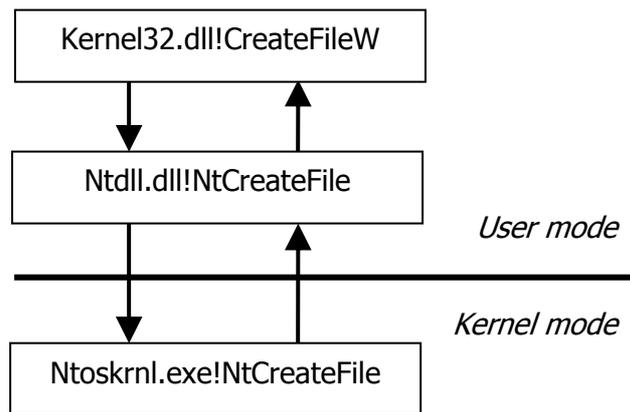


Figure 40 : Flux de contrôle d'un appel système

Quelle fonction faut-il intercepter? Cela dépend du but recherché. En général, seule celle de la couche supérieure (CreateFileW dans notre cas) est documentée et garantie de subsister telle quelle d'une version ou d'un Service Pack de Windows à l'autre.

Mais rien n'empêche une application utilisateur d'appeler directement d'autres fonctions en mode *user* (p.ex. NtCreateFile). En effet, les couches inférieures peuvent être explorées par *reverse engineering*. Il existe d'ailleurs des sites qui documentent l'API native fournie par Ntdll⁶⁵.

Afin d'assurer la furtivité du rootkit, nous devons intercepter tous les appels qui, par exemple, listent les processus existants. Choisir la fonction supérieure (**Windows API**) nous ferait peut-être manquer certains appels, tandis qu'utiliser fonction inférieure (**Native API**) diminue la portabilité du code.

De plus il est possible soit de **filtrer les données**, soit de **contrôler leur affichage**. Lorsqu'il est irréalisable d'altérer les données, on peut toujours les empêcher de s'afficher, même si c'est moins élégant et que cela s'applique uniquement à une application donnée.

Voici la liste des fonctions remplacées par NtIllusion (les fonctions précédées d'un signe "+" visent l'affichage en mode GUI ou CLI d'un utilitaire précis) :

flux de contrôle saute à l'adresse stockée dans une entrée de la SSDT (*System Service Descriptor Table*). Cette entrée se situe à l'emplacement `nt!KiServiceTable + 0x25*4` (une entrée fait 4 octets et nous allons à l'offset 0x25). Nous arrivons à la fonction `nt!NtCreateFile`, qui elle-même appelle `nt!IoCreateFile` et... nous nous arrêtons-là (`nt` est un raccourci pour `ntoskrnl.exe`). Curieusement, la fonction en mode *kernel* `nt!ZwCreateFile` permet aussi d'appeler `nt!NtCreateFile` via la SSDT.

⁶⁵ Visiter <http://undocumented.ntinternals.net/>

Objectif visé	Fonctions à remplacer
Contrôle des processus	CreateProcessW (kernel32.dll) LoadLibraryA/W/Ex (idem)
Dissimulation de processus	NtQuerySystemInformation (ntdll.dll) + SendMessageW (user32.dll) pour Task Manager
Dissimulation de fichiers	FindFirstFileA/W (kernel32.dll) FindNextFileA/W (idem)
Dissimulation de clés	RegEnumValueA/W (advapi32.dll)
Dissimulation de ports	GetTcpTable (iphlpapi.dll) AllocateAndGetTcpExTableFromStack (idem) + CharToOemBuffA (user32.dll) pour netstat + WriteFile (kernel32.dll) pour fport

Figure 41 : Liste des fonctions remplacées par NtIllusion

A part une seule fonction native dont le nom débute par "Nt" (en bleu), toutes les fonctions interceptées par NtIllusion font partie de l'API de Windows. C'est un choix que son concepteur a fait pour atteindre une portabilité maximale.

3.5.2. Modification de l'IAT

Voyons maintenant comment intercepter des fonctions de l'API. Nous savons que les appels à une fonction importée transitent via l'**IAT**, pour faciliter le travail du chargeur de programmes (voir § 2.3.2).

Nous n'aurons, pour rediriger tout appel à une certaine fonction, qu'à modifier l'adresse contenue dans l'entrée de l'IAT, soit à y écrire 4 octets. Cette modification peut être statique (dans l'image sur le disque) ou dynamique (en mémoire). En voici le schéma de principe à la Figure 42.

Comme on peut s'y attendre, cette technique présente aussi des subtilités. Premièrement, à quel moment faut-il opérer la modification dynamique de l'IAT? Il faut le faire évidemment avant tout appel à la fonction à intercepter.

M'inspirant du rootkit NtIllusion⁶⁶, j'ai écrit une application qui injecte une DLL contenant la fonction de remplacement, puis écrase l'entrée correspondante dans l'IAT du processus Wordpad. L'API cible est shell32.dll!ShellAboutW (version Unicode).

⁶⁶ Cf. la fonction HijackApi du fichier kHijackEng.c à modifier comme suit : accorder les droits en écriture dans l'IAT (VirtualProtect).

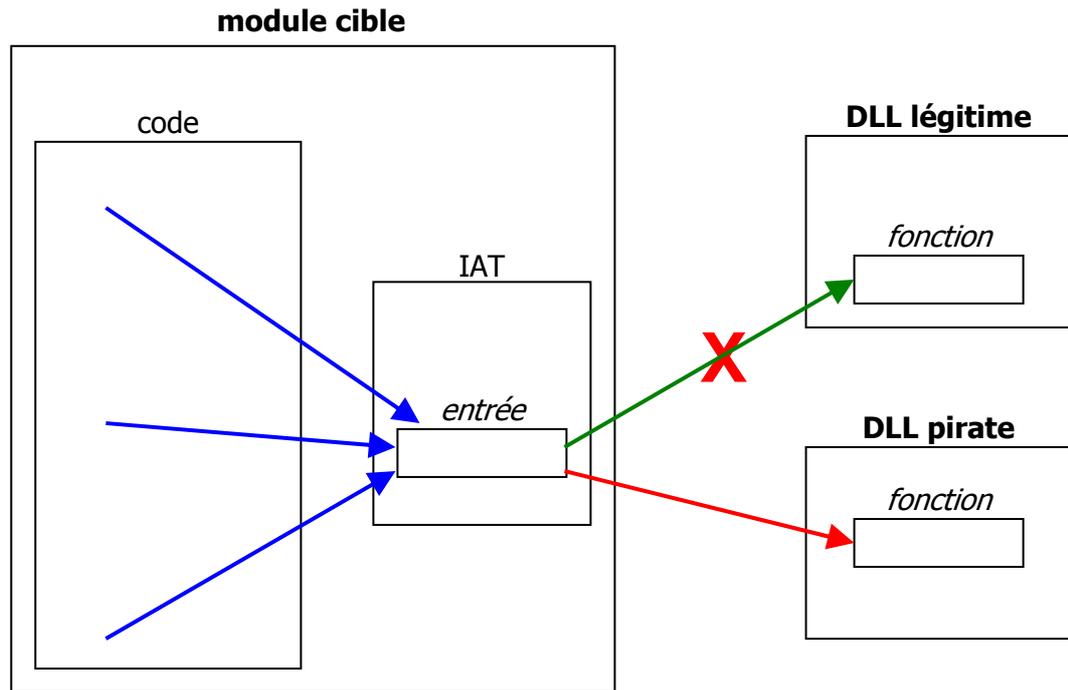


Figure 42 : Modification de l'IAT

La fonction de remplacement doit posséder une **signature**⁶⁷ identique à la fonction d'origine, mais aussi respecter la même **convention d'appel**⁶⁸. Ici la convention utilisée est `__stdcall` qui est défini par le symbole `WINAPI`.

```
// Déclaration dans windef.h : #define WINAPI __stdcall
int WINAPI ShellAboutW( HWND, LPCSTR, LPCSTR, HICON );
```

La boîte de dialogue "A propos de" d'origine est montrée à la Figure 43 (aller dans *Help | About WordPad*). Après modification, elle a changé. Bien sûr, à ce moment-là tout est possible : j'aurais pu conserver le même affichage et déclencher un traitement en arrière-plan, ou altérer le texte (p.ex. transformer la chaîne "WordPad" en "HijackedSoft"), etc.

Quant au *delay load import*, il emploie une autre table appelée "Delay Import Address Table" (**DIAT**) définie par le format PE. J'ai déduit de mes observations que l'adresse de chaque fonction est écrite dans l'entrée correspondante juste avant le premier appel.

⁶⁷ Pour rappel : nombre, ordre, type des arguments et du résultat.

⁶⁸ La convention d'appel détermine la façon dont les arguments sont passés (dans tels registres ou sur la pile, dans tel ordre), et qui de l'appelant ou de l'appelé sauvegarde les registres ou nettoie la pile.

Cf. "Calling conventions" <http://msdn2.microsoft.com/en-us/library/k2b2ssfz.aspx>



Figure 43 : Boîte de dialogue "A propos de" (Wordpad)

N'ayant pas trouvé d'exécutable de Windows qui possède une DIAT, j'ai choisi un *freeware* : HDD Free Hex Software⁶⁹. En étudiant son fichier PE à l'aide de PEBrowse Pro, nous trouvons l'adresse relative de la DIAT (0x6C8B0), ainsi qu'une fonction à chargement retardé `GetOpenFileNameA`, importée depuis `ComDlg32.dll` (voir Figure 44). Sachant que l'adresse de base de l'exécutable vaut 0x0040'0000, nous obtenons par simple addition l'adresse virtuelle de la DIAT : 0x0046'C8B0.

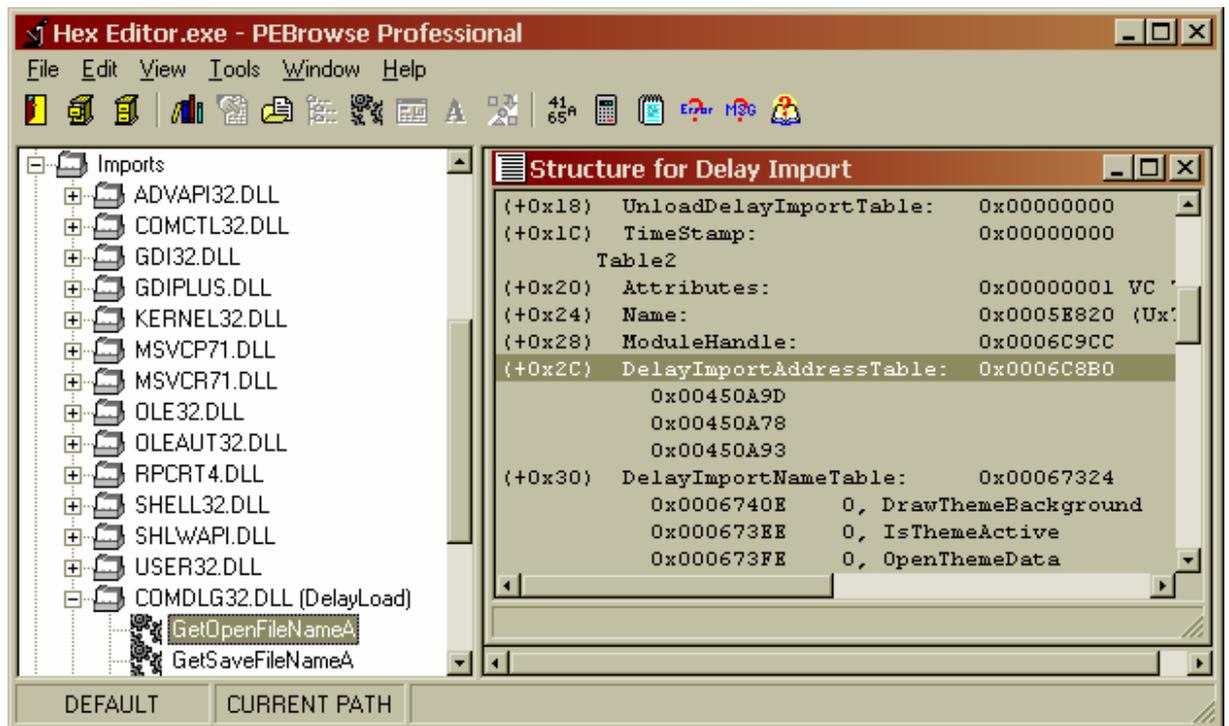


Figure 44 : Import à chargement retardé (PEBrowse Pro)

⁶⁹ <http://www.hhdsoftware.com/Download/hex-editor.exe>

Déboquons maintenant HexEditor avec OllyDbg. Ouvrons un *dump* du contenu de la DIAT, puis lançons l'exécution (F9). Lorsque nous demandons l'ouverture d'un fichier dans HexEditor, nous voyons qu'une entrée de la DIAT se remplit à l'adresse 0x0046'C91C. Cette dernière pointe sur `cmdlg32.GetOpenFileNameA` (0x763B'311E).

Nous allons maintenant écrire une adresse dans cette entrée avant qu'elle soit remplie, pour voir si notre adresse sera écrasée. Pour ce faire, redémarrons l'application (Ctrl+F2 puis F9) et écrivons la valeur 0x7C81'CDDA dans l'entrée de la DIAT (0x0046'C91C) comme à la Figure 45. Lorsque nous voulons maintenant ouvrir un fichier, le processus se termine⁷⁰!

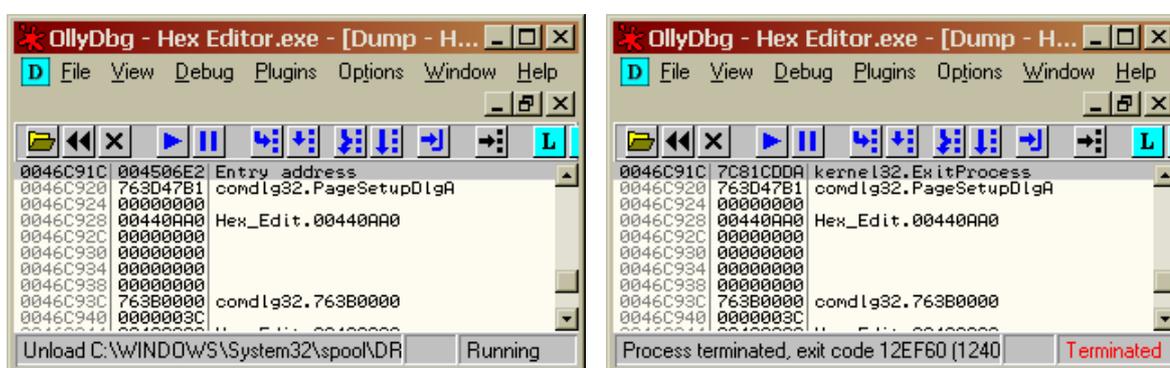


Figure 45 : Entrée de la DIAT avant et après initialisation (OllyDbg)

On peut deviner que le *runtime*, détectant que l'entrée contient déjà une adresse valide, ne recherche pas celle de la fonction `GetOpenFileNameA`. Ce comportement est facilement exploitable, comme nous en avons donné la démonstration.

3.5.3. Inline patching

La meilleure façon d'intercepter tous les appels à une fonction est de se placer à son point d'entrée (i.e. à l'emplacement de sa 1^{ère} instruction). Les premières instructions machine d'une fonction sont appelées son **prologue**.

Afin de rediriger le flux de contrôle, il suffit de placer un saut inconditionnel (Jump) vers notre fonction de remplacement. Mais pour corser les choses, les instructions de l'architecture Intel 32 bits (IA-32) ont une longueur variable. L'instruction `JMP` tient sur 5 octets, tandis que la taille des autres instructions varie de 1 à plus de 6 octets.

Seulement voilà, le rootkit a besoin d'appeler la fonction d'origine. Il existe 2 méthodes qui le permettent. La 1^{ère} méthode (Figure 46) consiste à stocker le prologue de l'API dans un tampon. Quand le rootkit veut utiliser la fonction d'origine, il restaure le prologue ce qui a pour

⁷⁰ En effet, 0x7C81'CDDA est l'adresse de l'API `kernel32!ExitProcess` sous Windows XP SP2.

effet de supprimer l'instruction JMP en rouge. Une fois l'appel terminé, le saut est réinstallé.

Cette méthode n'est pas fiable à 100% : en raison des aléas du multithreading, il y a un faible risque de ne pas intercepter tous les appels à une fonction si au moins deux threads du même processus l'appellent.

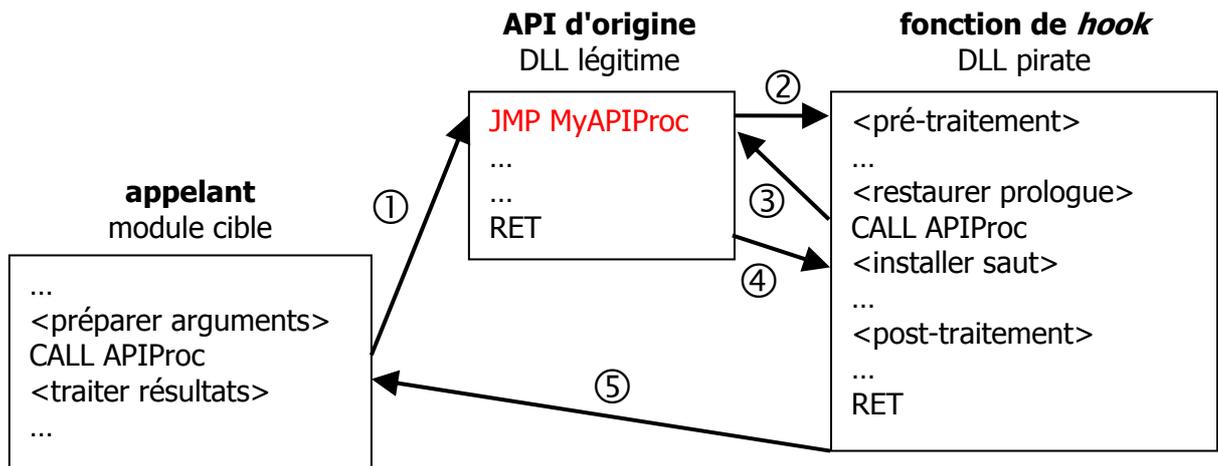


Figure 46 : 1^{ère} méthode d'*inline patching*

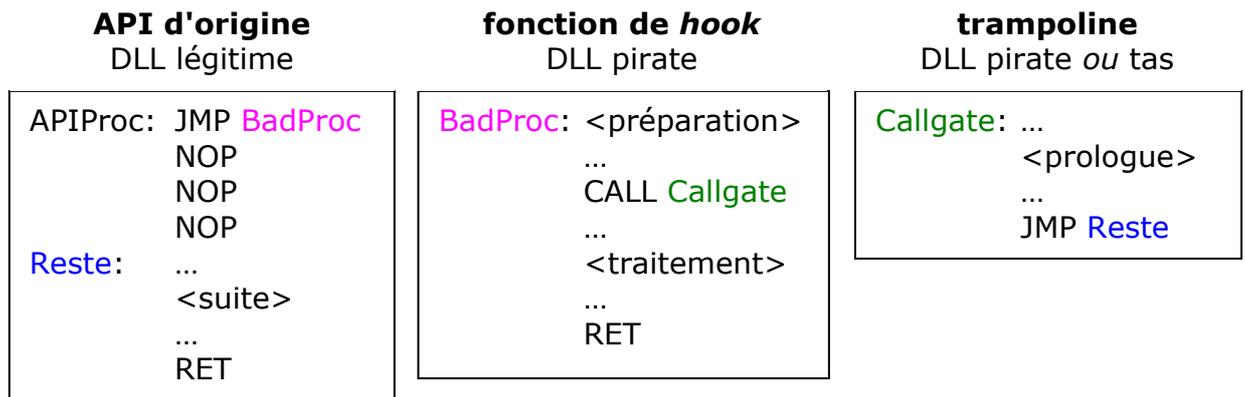
L'idée de la 2^{ème} méthode est d'ajouter un saut supplémentaire. Cette technique est mise en œuvre dans NtIllusion⁷¹ et dans la bibliothèque Detours, publiée par Microsoft Research⁷². Elle implique de déterminer combien d'instructions entières il faudra déplacer pour dégager un espace d'au moins 5 octets. Bien sûr, la fonction doit faire plus que 5 octets. Ensuite, nous créons un **trampoline** (voir Figure 47).

L'instruction CALL stocke l'adresse de la prochaine instruction (pointée par le registre EIP) sur la pile, puis saute vers l'endroit précisé. L'instruction RET récupère l'adresse de retour sur le sommet de la pile (pointé par ESP) puis retourne à l'appelant. Les instructions NOP (No Operate = ne rien faire) ne sont jamais exécutées, mais elles permettent vraisemblablement au désassembleur d'interpréter correctement les octets qui composent la suite du code⁷³.

⁷¹ Voir la fonction ForgeHook du fichier source kEPhook.c. NtIllusion se sert de la fonction GetInstLenght, déclarée dans ZDisasm.c, pour déterminer la longueur d'une instruction.

⁷² Téléchargeable gratuitement sur <http://research.microsoft.com/sn/detours/> (voir les publications).

⁷³ Autrement, il risque d'y avoir un décalage d'interprétation si le débogueur considère l'octet immédiatement après le JMP comme le commencement de la prochaine instruction. L'instruction NOP, qui tient sur un octet, fait ici office de bourrage jusqu'au début de l'instruction suivante d'origine.

Figure 47 : 2^{ème} méthode d'*inline patching*

Là encore, des complications peuvent survenir en fonction des instructions machine du code de la fonction d'origine. Il faudrait entre autres corriger les *offsets* d'éventuels sauts relatifs à destination de / vers les instructions du prologue.

Heureusement, c'est rarement le cas. Sous Windows XP SP2, Microsoft a fait en sorte que le prologue d'un grand nombre de fonctions de l'API soit toujours le même et qui plus est, fasse exactement de 5 octets.

```
// Prologue standard (code machine | instruction assembleur)
8bff  |  mov   edi,edi    // Instruction sans effet!
55    |  push  ebp
8bec  |  mov   ebp,esp
```

On peut se demander pourquoi on trouve l'instruction "mov edi,edi" au lieu de deux instructions "nop". Mes hypothèses sont que cela n'occupe qu'une ligne au lieu de deux dans le code assembleur, et que ça permet peut-être d'avoir un marqueur sur le début des fonctions (l'instruction "mov edi,edi" a peu de chances de se trouver ailleurs, tandis qu'il est fréquent de rencontrer une suite de "nop").

3.5.4. Expérience avec Detours et WinDbg

Kamal Shankar a publié un article intéressant sur CodeProject, qui exploite la bibliothèque Detours pour contourner les stratégies locales de Windows. Le code source de ce projet est un bon point de départ pour mettre en œuvre Detours. Avant d'aller plus loin, téléchargeons sa démonstration (Detours01Demo.zip)⁷⁴.

"Circumventing Windows Group Policies using Detours"
<http://www.codeproject.com/system/KamalDetours01.asp>

⁷⁴ Pour décompresser les archives ZIP des sources, utiliser WinZip v9+ ou 7-zip.

La restriction à contourner est la désactivation de l'interpréteur de commandes cmd.exe et de l'exécution des fichiers batch (.CMD ou .BAT). Elle est définie en créant la clé suivante de type DWORD :

HKCU\Software\Policies\Microsoft\Windows\System\DisableCMD = 1

Vérifions maintenant que la stratégie est bien appliquée. Lorsqu'on lance le *Command Prompt*, nous obtenons le message suivant :



Figure 48 : Invite de commande désactivée

Prenons pour acquis que c'est le processus cmd.exe qui vérifie la valeur de la clé DisableCMD, en appelant l'API **RegQueryValueExW**, exportée par la DLL advapi32.dll.⁷⁵ Ce qui nous intéresse maintenant, c'est d'observer le code machine de cette fonction. En gardant le *Command Prompt* ouvert (i.e. sans appuyer sur une touche), lançons WinDbg⁷⁶ et attachons le processus cmd.exe (F6). Cela à pour effet de déboguer le processus et de l'interrompre.

Entrons la commande suivante (en bleu), qui recherche parmi les symboles exportés par advapi32.dll, ceux qui commencent par RegQueryValue. Nous trouvons 4 versions de la fonction, dont celle qui nous intéresse à l'adresse 0x77DD'6FC8.

```
0:001> x advapi32!RegQueryValue*
77dd6fc8 ADVAPI32!RegQueryValueExW = <no type information>
77dd7883 ADVAPI32!RegQueryValueExA = <no type information>
77ddd8e2 ADVAPI32!RegQueryValueW = <no type information>
77dfcc10 ADVAPI32!RegQueryValueA = <no type information>
```

Affichons maintenant le prologue de la fonction qui nous intéresse, en désassemblant le code à l'adresse que nous venons de trouver. Nous constatons que les trois premières instructions machines (prologue standard) font respectivement 2, 1 et 2 octets, ce qui nous donne un

⁷⁵ Nous l'avons déterminé avec le puissant outil Process Monitor de SysInternals, qui permet entre autres de surveiller les activités sur la BDR :

<http://www.microsoft.com/technet/sysinternals/processesandthreads/processmonitor.mspx>

⁷⁶ L'article suivant explique comment installer WinDbg et les symboles de Windows : "Introduction to Reverse Engineering..." <http://uninformed.org/?v=1&a=7&t=pdf>

total de 5 octets. C'est juste ce qu'il faut pour placer un saut inconditionnel.

```
0:001> u 77dd6fc8
ADVAPI32!RegQueryValueExW:
77dd6fc8 8bff          mov     edi,edi
77dd6fca 55           push   ebp
77dd6fcb 8bec        mov     ebp,esp
77dd6fcd 83ec0c      sub     esp,0Ch
(sortie tronquée)
```

Maintenant, exécutons la démonstration de Kamal Shankar. Il faudrait pour cela lancer le script TestCMD.bat. Or, les fichiers batch sont désactivés. Pour contourner cela, créer quelque part un raccourci qui pointe vers l'exécutable withdll.exe avec les arguments suivants :

```
withdll.exe -d:DetGPCircumvent.dll CMD.EXE
```

Cette ligne demande au programme withdll.exe de créer une instance de *Command Prompt* et d'injecter la DLL spécifiée dans le processus créé. Lorsque DetGPCircumvent.dll est chargée (appel à son point d'entrée DllMain avec la raison DLL_PROCESS_ATTACH), Detours est invoqué pour installer un *hook* sur l'API RegQueryValueExW. Comme la fonction de remplacement signale que la clé DisableCMD n'existe pas (la clé est masquée), *Command Prompt* démarre normalement.

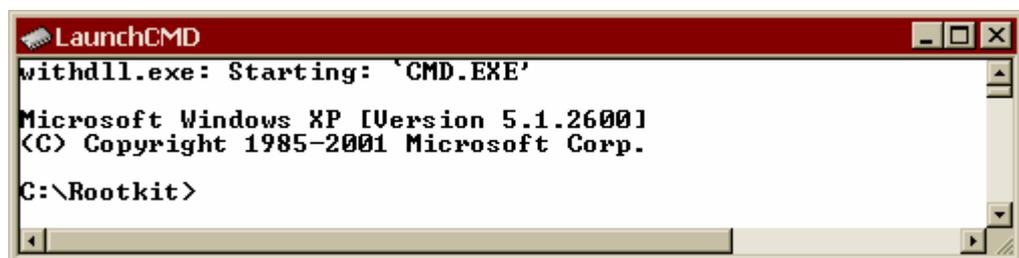


Figure 49 : Invite de commande réactivée

De la même manière, déboguons ce nouveau processus cmd.exe avec WinDbg et retournons voir le prologue de l'API cible :

```
0:001> u advapi32!RegQueryValueExW
ADVAPI32!RegQueryValueExW:
*** WARNING: Unable to verify checksum for
C:\Detours\DetGPCircumvent.dll
*** ERROR: Module load completed but symbols could not be loaded for
C:\Detours\DetGPCircumvent.dll
77dd6fc8 e963a02298   jmp     DetGPCircumvent+0x1030 (10001030)
77dd6fcd 83ec0c      sub     esp,0Ch
(sortie tronquée)
```

(L'erreur de chargement des symboles est normale : on ne dispose généralement pas des informations de débogage pour un module inconnu!). Les 5 premiers octets ont été remplacés par une instruction JMP à destination de la DLL injecté, puis nous retrouvons la suite de la fonction (SUB ESP, 0Ch).

La prochaine étape est bien sûr d'analyser la fonction de remplacement qui se trouve à l'adresse 0x1000'1030. Cette fois-ci, nous allons désassembler sur une plage plus grande en ajoutant l'option L :

```

0:001> u DetGPCircumvent+0x1030 L20
DetGPCircumvent+0x1030:
10001030 53          push      ebx
10001031 8b1d80600010  mov     ebx,dword ptr
[DetGPCircumvent+0x6080
(10006080)]
10001037 56          push      esi
10001038 57          push      edi
10001039 8b7c2414   mov     edi,dword ptr [esp+14h]
1000103d be30800010  mov     esi,offset DetGPCircumvent+0x8030
(10008030)
10001042 56          push      esi
10001043 57          push      edi
10001044 ffd3       call     ebx
(suite de la fonction)
1000106d 57          push      edi
1000106e 51          push      ecx
1000106f e88cffffff  call    DetGPCircumvent+0x1000 (10001000)
(sortie tronquée)

```

Voici les 2 premières d'instructions CALL rencontrées dans le code :

- 0x1000'1004 : il s'agit d'un appel via le registre EBX qui est chargé un peu plus haut (0x1000'1031). OllyDbg nous apprend que l'adresse 0x1000'6080 est l'entrée de l'IAT qui pointe vers la fonction kernel32.lstrcmpiW (0x7C80'A996), une fonction de comparaison de chaînes de caractères.
- 0x1000'106F : il est vraisemblablement fait appel à une fonction interne de la DLL. Voyons le code de cette sous-routine :

```

0:001> u DetGPCircumvent+0x1000
DetGPCircumvent+0x1000:
10001000 8bff       mov     edi,edi
10001002 55          push    ebp
10001003 8bec       mov     ebp,esp
10001005 e9c35fdd67  jmp     ADVAPI32!RegQueryValueExW+0x5
(77dd6fcd)
1000100a 90          nop
(sortie tronquée)

```

Comme par surprise, nous redécouvrons les 3 instructions du prologue de RegQueryValueExW, suivie d'un saut vers la suite de la fonction

d'origine. (On saute à l'*offset* 5 car le prologue occupe les 5 octets se trouvant aux indices 0 à 4). Il s'agit bel et bien du trampoline qui a été créé statiquement par Detours.

L'expérience est concluante et nous avons mis en avant un risque important. En effet, cette astuce fonctionne depuis un compte limité, donc un simple utilisateur peut contourner aisément ainsi les stratégies locales.

Nous pouvons aussi vérifier que l'interception de l'API (la modification du prologue de la fonction) ne concerne que l'instance dans laquelle la DLL a été injectée. Il suffit de lancer simultanément le *Command Prompt* via le raccourci et via la méthode standard. La première instance est autorisée à s'exécuter, tandis que la seconde ne l'est pas. CQFD.

3.5.5. Redirection de DLL

Cette liste ne serait pas complète sans la mention de la technique "*DLL redirection*". Bien qu'à ma connaissance aucun rootkit ne l'exploite, il vaut la peine de connaître cette technique qui illustre une fois de plus les grandes possibilités de *hacking* offertes par Windows. Elle consiste à écrire une DLL de remplacement et faire en sorte qu'elle soit chargée à la place de la DLL d'origine (qui est inchangée sur le disque), comme l'explique Craig Heffner dans son article :

"*API Interception via DLL Redirection*"

<http://milw0rm.com/papers/105>

Le mécanisme de "DLL redirection" est décrit dans MSDN, ainsi la liste des dossiers et l'ordre de recherche des DLLs.

"*Dynamic-Link Library Redirection*"

<http://msdn2.microsoft.com/en-us/library/ms682600.aspx>

"*Dynamic-Link Library Search Order*"

<http://msdn2.microsoft.com/en-us/library/ms682586.aspx>

Windows offre 2 méthodes pour rediriger une DLL. La première s'appelle DotLocal et consiste à créer un fichier ou un dossier portant le nom de l'exécutable suivi de ".local". Si le chargeur de programmes trouve un tel fichier / dossier, il va chercher en priorités les DLLs dans le répertoire de l'exécutable / le répertoire ".local" (voir l'article de Craig Heffner pour plus de précisions). Il semble que cette méthode ne fonctionne plus sous Windows XP SP2.

La deuxième s'appuie sur un **manifeste** (*manifest*) au format XML. Ce dernier peut être soit intégrée à l'image comme une ressource, soit stocké dans un fichier séparé. Si Windows trouve un manifeste dans la section .rsrc et un autre dans un fichier .manifest, c'est le second qui l'emporte. Nous pouvons voir le manifeste du WordPad grâce à CFF Explorer⁷⁷.

⁷⁷ Explorer Suite, créée par Daniel Pistelli : <http://www.ntcore.com/exsuite.php>

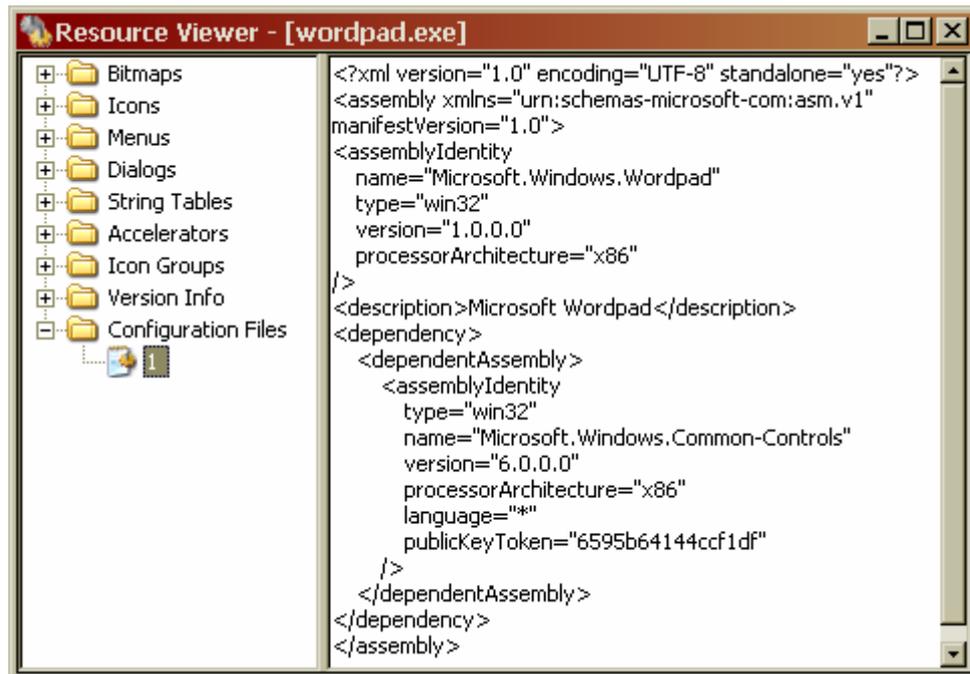


Figure 50 : Manifeste intégrée à une image (CFF Explorer)

A condition que les permissions NTFS l'autorisent, nous pouvons créer un fichier `wordpad.exe.manifest` dans le même dossier que `wordpad.exe`. Nous pourrions ainsi rediriger une DLL dont dépend l'application sans toucher à ni à l'exécutable, ni à l'image de la DLL sur le disque.⁷⁸ (Bien sûr, une DLL de remplacement doit exporter tous les symboles que l'application importe pour que les choses se passent bien).

3.6. Techniques additionnelles

En supplément à toutes les techniques précédentes, un rootkit peut chercher à disparaître ou encore à se protéger du *reverse engineering*.

3.6.1. Module fantôme

Il faut savoir que Windows maintient les informations sur les modules d'un processus dans l'espace utilisateur. Chaque module est décrit par une structure de données `_LDR_MODULE`. Il y a 3 listes de modules doublement chaînées qui contiennent les mêmes éléments triés selon :

- L'ordre d'initialisation
- L'ordre de chargement
- La position dans la mémoire

⁷⁸ Il est aussi possible d'injecter un manifeste dans une image qui en est dépourvue : <http://www.codeproject.com/dotnet/ManifestInjection.asp>

Ces informations peuvent être observées dans WinDbg par exemple (à noter que toutes les adresses sont inférieures à 0x8000'0000, donc accessibles en mode *user*).

```

0:002> !peb
PEB at 7ffdf000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: Yes
  ImageBaseAddress: 01000000
  Ldr: 00191e90
  Ldr.Initialized: Yes
// Pointeurs sur l'élément de la liste: Précédent. Suivant
  Ldr.InInitializationOrderModuleList: 00191f28 . 00193228
  Ldr.InLoadOrderModuleList: 00191ec0 . 00193218
  Ldr.InMemoryOrderModuleList: 00191ec8 . 00193220
Base      TimeStamp      Module
01000000  41107cde Aug 04 08:06:22 2004 C:\Program F(...)\wordpad.exe
7c900000  411096b4 Aug 04 09:56:36 2004 C:\(...)\system32\ntdll.dll
(sortie tronquée)

```

L'API de Windows offre d'énumérer les modules d'un processus avec les fonctions `EnumProcessModules` ou `CreateToolhelp32Snapshot`⁷⁹. Pour y parer, le rootkit `NtIllusion` parvient à extraire sa DLL de ces 3 listes après son chargement (cf. le fichier source `KDIIHideEng.c`), comme l'explique Kodmaker au § 4.7 de son article. Le résultat est un module fantôme, indétectable même par les fonctions de l'API inchangées.

3.6.2. Contre le *reversing*

En fin de compte, un rootkit sera toujours détectable par sa simple présence dans l'EAV du processus cible. En effet, il est possible de traquer un *malware* connu en effectuant une recherche de motif dans l'espace utilisateur (en mémoire donc).

Supposons que nous trouvions du code suspect, que ce soit sur le disque ou en mémoire. Il reste encore à déterminer son comportement pour savoir ce que ce code réalise vraiment. Or, il existe de nombreuses techniques qui ont pour but de compliquer le *reverse engineering* d'un programme. Sans entrer dans les détails puisque ces techniques sont à la limite du cadre de notre étude, mentionnons :

- Le *packing* : compression du fichier image pour empêcher son désassemblage.
- L'obfuscation : brouillage du code ou des identificateurs pour augmenter la difficulté de la décompilation (concerne surtout Java ou .Net).

Il y a encore mille et une astuces de programmation pour perturber le désassemblage ou le débogage d'un binaire, ou rendre le code machine

⁷⁹ cf. "Module walking" : <http://msdn2.microsoft.com/en-us/library/ms684236.aspx>

très difficile à comprendre. Cela nous amène aux remarques de la section suivante.

3.6.3. Mesures et contre-mesures

Je tiens encore à dire quelques mots sur l'évolution des rootkits et autres *malwares*. En plus de ce que nous avons dit à la section précédente, il est évident qu'un *malware* en mode utilisateur (*userland*) sera toujours détectable par un programme en mode *kernel*. Il existe deux grandes familles de méthodes de détection :

1. **X-VIEW** (*cross view*) : elle consiste à appeler des fonctions de niveaux différents et à comparer leur résultat. L'existence de différences peut indiquer la présence d'un rootkit. Par exemple, on compare une liste des processus obtenue via l'API de Windows avec celle obtenue via l'API native (ou d'autres fonctions en mode *kernel*).
2. **ECD** (*Explicit Compromise Detection*) : il s'agit de rechercher des traces anormales à des endroits précis. On peut par exemple détecter des modifications de tables telles que l'IAT (plus la SSDT & l'IDT en mode *kernel*).

Quoiqu'il en soit, les *malwares* n'ont pas fini de jouer à cache-cache avec les logiciels de sécurité. Prenons pour simplifier le cas de l'*inline patching* (§ 3.5.3). On peut imaginer qu'un anti-virus contrôle la première instruction de toutes les fonctions de l'API en vérifiant l'absence de JMP. Alors le *hacker* va placer son saut un peu plus loin dans la fonction. Mais le prochain logiciel de sécurité vérifie maintenant que tous les JMPs à l'intérieur d'une fonction aboutissent dans le même module.

Au fur et à mesure, le pirate pourra imaginer toutes sortes de ruses pour échapper à la détection : insérer plusieurs sauts successifs, utiliser des sauts conditionnels dont la condition sera toujours vraie, etc. Le jour où les chercheurs en sécurité mettent la main sur son *malware*, ils découvriront ces ruses par *reverse engineering* et proposeront des contre-mesures. Mais le pirate aura déjà imaginé de nouvelles attaques... et ainsi de suite!

3.7. Et sous Windows Vista?

Il est légitime de se demander si les techniques présentées sont toujours valables sous Windows Vista. N'ayant pas le temps d'analyser en profondeur la sécurité de cette nouvelle version de Windows, je me contenterai de tester quelques exemples de code pour comparer avec Windows XP SP2.

Tous les tests qui suivent ont été effectués sous **Windows Vista RTM** (*Release To Manufacturing*), téléchargeable sur MSDN depuis le 17 novembre 2006. J'ai installé la version Ultimate sur un Pentium 4 HT à 2,8 Ghz avec 1,5 Go de RAM (le PC de test obtient un *Windows Experience Index* de 4.0). La configuration est inchangée (*out of the box*), hormis la création d'un simple utilisateur que j'ai nommé Eve.

3.7.1. 1^{er} test : terminaison des processus

But : Terminer toutes les instances possibles du Wordpad (voir § 3.3.1 pour le code).

Scénario : Eve, qui est une utilisatrice standard, ouvre une session et démarre une instance normale du Wordpad et une instance lancée en tant que Administrator. Puis elle ouvre le programme de test.

Résultat : Sous Windows Vista, les deux instances sont tuées tandis que sous Windows XP, l'instance lancée avec un jeton d'administrateur survit.

Commentaire : Ce comportement est pour le moins surprenant, puisque le processus d'une simple utilisatrice peut maintenant tuer un processus ayant des droits supérieurs, mais qui appartenant au même bureau.

3.7.2. 2^{ème} test : modification de l'IAT

But : Pirater la boîte de dialogue "About" du Wordpad en injectant une DLL et en modifiant l'IAT de l'application (voir § 3.5.2).

Scénario : Identique au test précédent (§ 3.7.1).

Résultat : Le comportement est le même sous les deux versions de Windows (Vista et XP SP2). Seule l'instance lancée par l'utilisateur standard est piratée avec succès.



Figure 51 : Boîte "A propos de" du Wordpad (Vista)

Commentaire : Un processus de niveau inférieur ne peut pas créer de thread distant dans un processus de niveau supérieur. Par conséquent, l'appel à `CreateRemoteThread` ciblant le processus privilégié échoue.

3.7.3. 3^{ème} test : inline patching avec Detours

But : Contourner la stratégie locale désactivant l'invite de commandes en masquant la clé de la BDR concernée par *inline patching* avec Detours (voir § 3.5.3 et 3.5.4).



Figure 52 : Invite de commandes désactivée (Vista)

Scénario : L'administrateur met en place la stratégie locale⁸⁰. Puis Eve tente de démarrer le *Command Prompt*. Ensuite, elle exécute le programme de test qui essaie de lancer cmd.exe à son tour.

Résultat : Là encore, le test réussit sous les 2 versions de Windows.

Figure 53 : Invite de commandes réactivée⁸¹ (Vista)

Commentaire : Le contournement de stratégie locale fonctionne toujours. Mais que peut-on vraiment faire de cette invite? Le *Command Prompt* s'exécute au même niveau d'intégrité que l'appelant (*asInvoker*). De plus, il est impossible d'élever le niveau d'un processus existant⁸².

3.7.4. 4^{ème} test : interception de mots de passe

But : Intercepter un mot de passe administrateur entré dans la session d'un utilisateur standard (commande de type "runas").

⁸⁰ Voici la marche à suivre : la session d'Eve étant ouverte, lancer l'éditeur du registre en tant qu'administrateur (appuyer sur Win+R, entrer "regedit" et valider avec Ctrl+Alt+Entrée pour demander l'élévation des privilèges) et créer la clé comme indiqué au § 3.5.4.

Remarque : la clé HKCU d'un utilisateur est en fait un lien vers la clé HKEY_USERS\{SID_de_l'utilisateur}, qui peut donc être modifiée aussi depuis la session d'un administrateur.

⁸¹ La commande "color f0" sert juste à économiser l'encre noire de l'imprimante...

⁸² Contrairement à un *shell* de type UNIX où on peut temporairement élever les privilèges grâce aux commandes "su" ou "sudo".

Scénario : Depuis la session d'Eve, démarrer un programme avec élévation des privilèges (clic-droit, puis choisir "Run as administrator"). Après avoir entré le mot de passe, mais avant de valider, lancer une application telle que WinSpy⁸³ pour intercepter le contenu du champ.

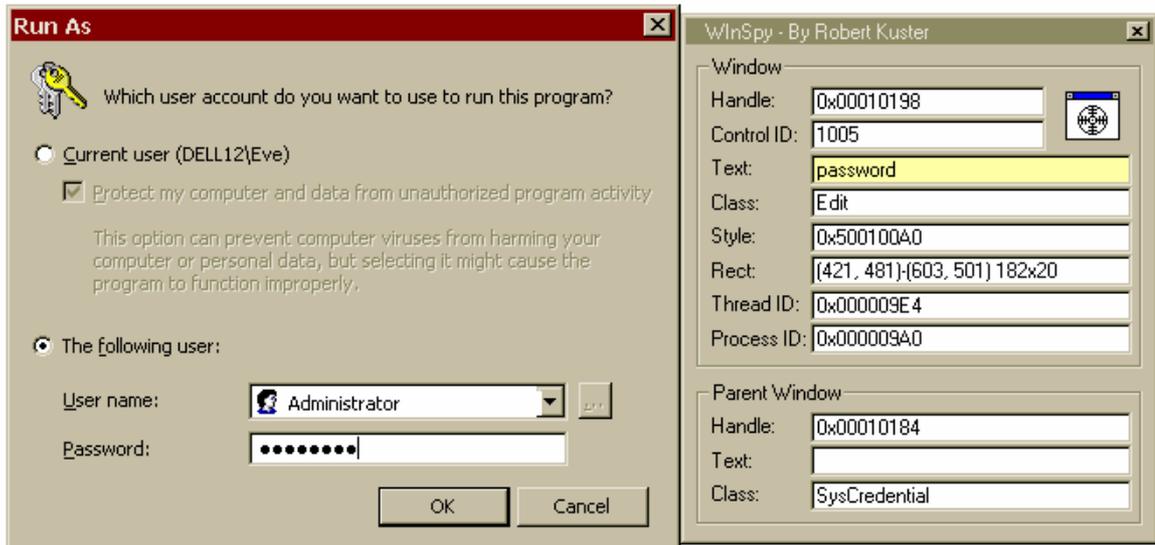


Figure 54 : Interception de mot de passe avec WinSpy (XP SP2)

Résultat : L'interception de mot de passe fonctionne bien sous Windows XP SP2 (voir Figure 54). Par contre, Windows Vista effectue un changement de bureau avant d'afficher la boîte de dialogue. L'écran se noircit et aucune application tournant sur le bureau de l'utilisateur ne peut alors intervenir avec l'interface graphique.

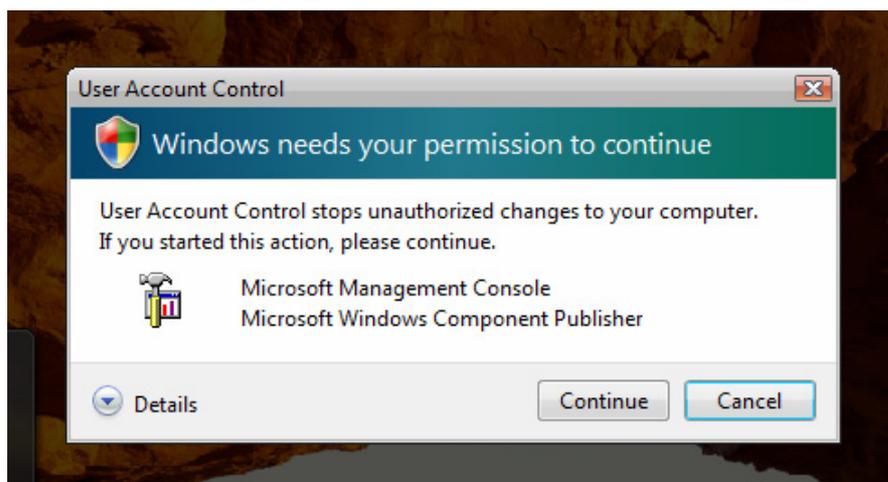


Figure 55 : Boîte de dialogue UAC (Vista)

⁸³ WinSpy est développé par Robert Kuster et est disponible à l'URL suivante : http://www.codeproject.com/threads/winspy/winspy_app.zip

Commentaire : L'isolation d'interface graphique qui a été introduite dans Windows Vista améliore sensiblement la sécurité. Puisqu'un bureau regroupe les entrées du clavier et de la souris et la sortie de l'écran, il semble impossible d'intercepter les frappes de touches ou de prendre un instantané de l'affichage. Ainsi, il faut recourir à un appareil photo numérique ou exécuter Vista sur une machine virtuelle pour capturer un *screenshot* tel que celui à la Figure 55.

3.7.5. Bilan

La sécurité de Windows Vista a été accrue selon les principes de **défense en profondeur** et du **moindre privilège**. En bref, Microsoft a implémenté des contrôles d'intégrité sur les fichiers (binaires et données) et a réduit les droits accordés aux comptes administrateur (notion de *Protected Administrator*), aux services (répartis sous divers comptes restreints) et aux pilotes (*User Mode Drivers*). De plus, Vista introduit les concepts de niveau d'intégrité et de protection de l'interface utilisateur, dans le but d'isoler les processus entre eux. Toutes ces améliorations vont certainement compliquer la tâche des *malwares*.

Cependant, l'isolation interprocessus ne concerne pas les processus de même niveau d'intégrité, appartenant au même utilisateur standard. En outre, les attaques que j'ai présentées agissent en mémoire, donc sans modifier les fichiers images sur le disque. C'est la raison pour laquelle ces techniques sont toujours d'actualité.

Attention tout de même à ne pas conclure que Vista soit implicitement sûr! Encore une fois, la sécurité dépend beaucoup de la configuration et il existe toutes sortes de progiciels spécialisés qui mettent en échec les techniques de *malwares*.

Sources :

"Vista Security for Developers" :

[http://download.microsoft.com/download/\(...\)/VistaSecurityForDevelopers.ppt](http://download.microsoft.com/download/(...)/VistaSecurityForDevelopers.ppt)

"Developer Best Practices and Guidelines..." :

<http://msdn.microsoft.com/library/en-us/dnlong/html/AccProtVista.asp>

4. Postface

4.1. Conclusion

Ce travail de diplôme m'a permis de découvrir le fonctionnement interne du système d'exploitation le plus utilisé au monde, et de m'initier à la programmation système Win32. Face à la taille et la complexité de Windows, personne n'a le temps de tout comprendre dans les détails, mais doit se concentrer sur ce qu'il a besoin et faire abstraction du reste.

J'estime avoir donné un aperçu des mécanismes fondamentaux de Windows et de son modèle de sécurité, et donné des références suffisantes pour approfondir tel ou tel aspect de l'OS. Je me suis aussi efforcé de présenter une série d'outils qui permettent d'expérimenter ce qu'il se passe dans le PC.

En deuxième partie, j'ai étudié et exposé les principales techniques des rootkits en mode *user* qui n'exploitent pas de failles, mais tirent parti des choix de conception de Microsoft. Par ces quelques exemples concrets, j'espère avoir montré combien il est facile de modifier le comportement d'applications ou du système d'exploitation et, suivant comment ces techniques sont mises en œuvre, de dissimuler des informations sur le disque et des processus en exécution.

Sans avoir développé moi-même un rootkit complet, je pense en avoir dit assez pour y parvenir. D'ailleurs, le code source du rootkit NtIllusion est un bon point de départ. De plus, j'ai constaté que Windows Vista ne résolvait pas tous ces problèmes de *design*. Le modèle de sécurité reste centré sur le compte, ce qui autorise toutes sortes d'interaction entre les processus d'un utilisateur. Il semble aussi que les *malwares* ont toujours au moins une longueur d'avance sur les logiciels de sécurité...

On peut alors se demander si la technologie pourra un jour assurer la sécurité à 100 %. Au-delà de telle ou telle technique, nous voyons bien que la faille se situe dans l'être humain. Windows, à l'image des hommes qui l'ont produit, a quelque chose de grandiose tout en restant faillible. Son insécurité provient de l'échec des uns à concevoir et à implémenter parfaitement, et de la malice des autres à exploiter les erreurs des premiers.

En quoi ou plutôt en qui nous plaçons-nous notre confiance?

Genève, le 27 novembre 2006
Florent Wenger

4.2. Remerciements

Je tiens à remercier particulièrement les personnes suivantes :

- M. Litzistorf pour m'avoir proposé ce sujet et m'avoir suivi
- Laurent Guinnard et Mario Pasquali pour leur accueil chez Ellisys et pour leur collaboration
- Nicolas Sadeg et Sébastien Contreras pour leur appui technique
- Vincent Zumbo pour la mise en page et la relecture du mémoire
- Thomas Perez pour ses recherches sur Windows Vista
- Yann Souchon pour m'avoir indiqué l'article "DLL redirection"
- Ma famille pour son soutien logistique et psychologique sans faille

4.3. Références

- [WI] "Microsoft Windows Internals, Fourth Edition",
Mark E. Russinovich & David Solomon, Microsoft Press, 2005"
- [WSP] "Windows System Programming, Third Edition"
Johnson M. Hart, Addison-Wesley, 2005
- [REV] "Reversing (Secrets of Reverse Engineering)"
Eldad Eilam, Wiley, 2005

Quelques liens intéressants :

- <http://www.rootkit.com/> Le site de référence (voir la section *Bookmarks*)
- <http://www.phrack.org/> Underground ezine
- <http://www.antirootkit.com/> La contre-attaque au 1^{er} site
- <http://www.uninformed.org/> Publications sur la SSI
- <http://msdn.microsoft.com/> Documentation officielle MSDN
- <http://www.codeproject.com/> Ressources pour les développeurs
- <http://www.tuts4you.com/> Ressources pour le *reverse engineering*
- http://www.sans.org/reading_room/ Plus de 1600 articles sur la SSI
- <http://www.microsoft.com/technet/sysinternals/> Outils de SysInternals
- <http://www.miscmag.com/> Excellent magazine papier sur la SSI

-*- EOF -*-