Sommaire

Introduction au format Portable Executable

1. Bref historique
2. Vue globale
    1. Illustration

En-tête PE

1. Dos Stub / Compatibilité DOS
    1. En-tête DOS
    2. Exécuteur DOS
2. Image Nt Headers / En-têtes Nt
    1. Signature PE
    2. Image File Header / En-tête de fichier image
    3. Optional Header / En-tête optionnelle
3. Sections Headers / En-tête des sections

Section "Importations"

1. Présentation
2. Explication théorique
    1. Image Import Descriptor
        1. Pourquoi deux fois le même champ ?
    2. ImageThunkData / Import Lookup Table
        1. Types d'exportation
            1. Importation ordinale
            2. Importation par le nom
    3. Hint/Name Table
3. Illustration

Section "Exportations"

1. Présentation
2. Explication théorique
    1. Types d'exportation
        1. Exportation ordinale
        2. Exportation par le nom
        3. Application au contexte
    2. Tableaux pointés
3. Illustration

La section "Relocations"

1. Introduction
2. Explication théorique
    1. Fixup Block / Bloc de correctifs

Ajouter une section

1. Explication théorique
    1. Incrémentation du NumberOfSections
    2. Incrémentation du SizeOfImage
    3. Ajout de l'en-tête de section
    4. Ajout de la section
2. Illustrations

Les constantes des fichiers PE

1. Notations et valeurs
2. Le champ "Machine" / ImageFileHeader.Machine
3. Le champ "Characteristics" / ImageFileHeader.Characteristics
4. Le champ "SubSystem" / OptionalHeader.SubSystem
5. Le champ "DLL Characteristics" / OptionalHeader.DLLCharacteristics
6. Le champ "Characteristics" / ImageSectionHeader.Characteristics
7. Les types correctif de la section "Relocations"

Types d'adresses

1. Présentation
2. Définitions
3. Calculs
    1. Données exemples
    2. RVA -> Offset
    3. Offset -> RVA
    4. RVA -> VA -> RVA

Introduction au format Portable Executable

24/01/2006 yarocco

1. Bref historique

Le format PE (pour Portable Executable) est un format de fichier utilisé chez Microsoft pour les pilotes, les programmes, les DLL et autres fichiers executables. On comprend mieux le nom de ce format quand on remonte son histoire.En effet, Microsoft a voulu un fichier qui soit portable. Non pas qu'on puisse le passer d'un PC à un Mac sans modification mais plutôt qu'il puisse être porté sous les differents systemes que Windows NT supporte, car à la base Windows NT n'est pas seulement prévu pour les systemes à base de x86 sous Win32, il était destiné à

aussi supporter d'autres architectures tels que la serie des Motorola 68000, des Intel 64 ou encore même des Power PC. Il fallait donc élaborer des structures communes pour ces architectures.

Le PE est une de ces structures. Il est capable avec quelques modifications mineures d'être porté sous quelques architectures différentes. Ce format n'est pas né du néant mais dérive en partie du format COFF que l'on retrouve dans UNIX. Il y apporte quelques modifications dont je ne peux pas vous faire la liste ici, faute de spécification pour ce format de fichier (peut être lors d'une MAJ de cet article). Mais une chose est sûre, c'est que Microsoft désirait un nouveau fichier exécutable qui permettrait d'exploiter la machine à son maximum. En effet, l'ancêtre du PE posait d'énormes restrictions mémoire et pouvait poser problème lorsque les différentes sections étaient trop longue (par exemple les sections Ressources ou Code). Le format PE fut developpé pour combler ces lacunes et il est aujourd'hui encore utilisé (près de 13 ans après sa création) et risque encore de durer un certains temps vu que la technologie .Net utilise ce format et il a dejà subit quelques modifications pour s'adapter au 64 bits.

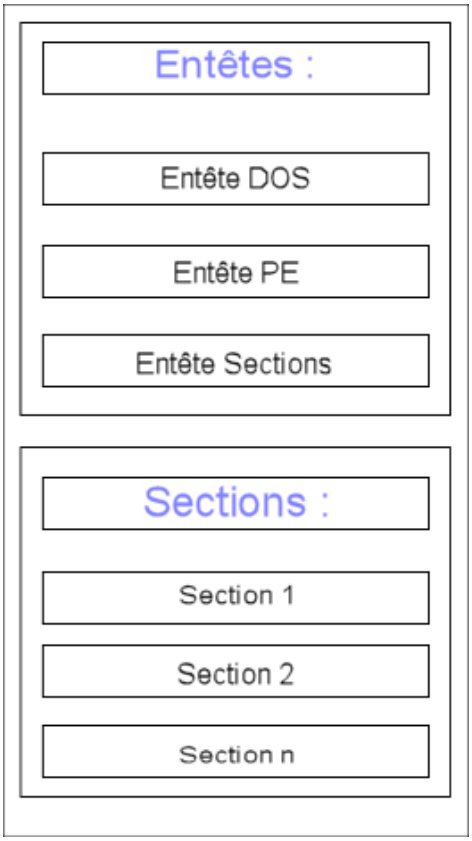
2. Vue globale

Les fichiers au format PE peuvent être divisé en plusieurs parties :

- Une partie pour assurer de fonctionner dans un environnement viable. Cette partie contient le kit necessaire pour charger un programme sous DOS et indiquer que le dit programme n'est pas fait pour être éxécuté sous ce système. Cette partie est là principalement pour indiquer aux utilisateurs qu'il faut changer de système s'ils désirent utiliser le programme.
- Une partie où l'on trouve l'en-tête d'un "Fichier Image Nt" (Image File Nt), aussi appellée en-tête PE, où sont stockées les informations nécessaires au loader pour charger le fichier en mémoire. En effet, il faut savoir que le fichier n'est pas chargé tel quel en mémoire, la structure sur le disque peut être totalement differente de celle une fois chargée en mémoire. Pour permettre un chargement correct du fichier, il faut donner des indications au loader :où se charge telle ou telle section, quelle taille faut t-il reserver et quelques autres parametres sont obligatoires pour la "mapping" (chargement en mémoire) du fichier et c'est dans cette partie que se trouvent les informatiosn. Après cette entête, il y a des informations sur les differentes sections que comporte le fichier. Ce sont donc les entêtes des sections qui comportent elles aussi des informations pour leur chargement en mémoire.
- Une autre partie où l'on trouve les sections. Les sections sont en fait des sortes de repertoires dans lesquels sont regroupées des données ayant la même fonctionnalité, du code ou des ressources par exemple, sous un même attribut, qui peut être en lecture seule et écriture par exemple. Il faut donc bien voir ce système de sections qui est présent tout le reste du fichier. Ces sections contiennent le code, quelques informations sur le chargement en mémoire du fichier ou encore des informations de débogage.

1. Illustration

Voici un fichier au format Portable Executable tel qu'on peut en trouver partout sur n'importe quelle machine étant équipée d'un Windows 95 au minimum.



En-tête PE

Explication de l'en-tête et des sous-en-têtes du format PE

24/01/2006 yarocco

1. Dos Stub / Compatibilité DOS

1. En-tête DOS

Il s'agit ici d'un en-tête qui permet au format PE de rester compatible avec DOS (dans une certaine limite…). Je passerai donc cet en-tête qui est pour nous pas du tout intéressant car exploitable que sous DOS.

Il faut juste savoir qu'il commence toujours par la série de caractères "MZ" et que son dernier champ "e_lfanew", à l'offset $3C (50), contient l'adresse de l'en-tête PE à proprement parler. Il n'y a pas de constante ou de champs qui référence l'offset de l'en-tête PE car le compilateur doit inclure dans l'image du code qui sera exécuté lors d'un chargement du programme sous DOS.

2. Exécuteur DOS

C'est ici qu'on retrouve le fameux « This program must be run under Win32 » (et parfois sa version française). Le compilateur peut donc mettre autant de code qu'il veut du moment que le champ "e_lfanew" pointe vers l'en-tête PE.

2. Image Nt Headers / En-têtes Nt

Dès que cet en-tête commence, on doit normalement trouvé une signature qui confirme qu'il s'agit bien d'un fichier PE.

1. Signature PE

Elle est de la forme "PE#0#0", et donne en hexadécimal : "45 50 00 00", c'est-à-dire un "P", un "E" suivit par 2 octets nuls. Si la signature n'est pas bonne, le loader refusera de charger l'exécutable.

2. Image File Header / En-tête de fichier image

Il s'agit ici du premier en-tête qui est intéressant, en effet on trouve ici le nombre de sections et quelques caractéristiques du fichier.

Machine : Word

On trouve tout d'abord ce champ qui indique le type de machine qui a servit à compiler le fichier qu'on lit.

Voir la correspondance des constantes PE.

NumberOfSection : Word

Il s'agit du nombre de sections qui sont dans l'exécutable. Ce champ est très important et ne peut pas être changé sans autre changement dans le fichier. La valeur "standard" de ce champ est $10 (16).

TimeDateStamp : DWord

Date et heure de création du fichier. Ce champ est presque inutile de nos jours car beaucoup de compilateurs mettent de fausses valeurs.

PointerToSymbolTable : DWord

Il s'agit de l'offset de la table des symboles. Les symboles servent au débogage d'un programme en général. Ils permettent de récupérer les noms des fonctions, leurs paramètres et autres détails qui sont effacés lors de la compilation.

NumberofSymbols : DWord

Il s'agit d'un champ qui donne des informations sur le fichier. Voir la correspondance des constantes PE.

3. Optional Header / En-tête optionnelle

Comme je l'ai lu souvent: "Cet en-tête n'a d'optionnel que le nom."

Je pense qu'elle a été nommée ainsi car c'est un rajout par rapport à la structure des fichiers COFF.

Magic : Word

Il s'agit d'un champ qui sert à différencier un fichier image 32 bits d'un fichier image 64bits.

Pour un fichier 32 bits, le champ doit être : $10B.

Pour un fichier 64 bits, le champ doit être : $20B

Si le fichier est pour les 64 bits, il y a quelques changements dont je ne parlerais pas pour le moment.

MajorLinkerVersion : Byte

Comme son nom l'indique, donne la version majeure du linker.

MinorLinkerVersion : Byte

Sans commentaire.

SizeOfCode : DWord

Donne la taille des sections qui ont du code mais peut être totalement faux si la constante "IMAGE_FILE_LINE_NUMS_STRIPPED" est dans les characteristics du fichier. En principe, c'est la taille de la section ".Code" ou ".Text".

SizeOfInitializedData : DWord

Donne la taille des sections qui contiennent des données initialisées, c'est-à-dire les variables globales, constantes etc... En principe, c'est la taille de la section ".Data".

SizeofUnInitializedData : DWord

Donne la taille des sections qui contiennent des données non initialisées.

En principe, c'est la taille de la section ".bss".

AddressOfEntryPoint : DWord

Premier champ qui devient réellement intéressant pour de futures modifications du fichier.

C'est le point d'entrée de l'application. C'est-à-dire que lorsque vous allez exécuter le programme, il va d'abord regarder l'intégrité du fichier puis va aller à cette adresse et va exécuter le code.

Attention, cette adresse, comme une grande partie des adresses dans l'en-tête est une RVA.

BaseofCode : DWord

Donne l'adresse RVA du début du code mais peut être faussée.

BaseofData : DWord

Donne l'adresse RVA du début des données mais peut être faussée.

ImageBase : DWord

Cette adresse est cruciale pour le loader. C'est à cette adresse qu'il va essayer de charger le fichier dans le cadre d'une exécution.

De plus, c'est l'adresse à laquelle les RVA doivent être relative. C'est-à-dire qu'une RVA doit être au moins supérieure ou égale à ce champ.

La plupart du temps, elle est de $400000.

SectionAlignment : DWord

Alignement des sections quand elles sont chargées en mémoire. C'est-à-dire que l'adresse des sections une fois chargée en mémoire doit être un multiple de ce champ. Ce champ doit être supérieur ou égal au FileAlignment.

FileAlignment : DWord

Alignement utilisé pour les sections sur le disque dur. C'est-à-dire que l'adresse et la taille des sections dans le fichier doivent être un multiple de ce champ. Cette valeur doit être une puissance de 2 entre 512 et 64 000.

La valeur par défaut est 512.

MajorOperatingSystemVersion : Word

Version majeure du système d'exploitation requis. N'a pas de réelle importance.

MinorOperatingSystemVersion : Word

Version mineure du système d'exploitation requis. N'a pas de réelle importance.

MajorImageVersion : Word

Version majeure de l'image mais n'est plus utilisée. Maintenant, la version est stockée dans les ressources.

MinorImageVersion : Word

Sans commentaire

MajorSubsystemVersion : Word

Version majeure du sous-système requis. N'a pas de réelle importance.

MinorSubsystemVersion : Word

Version mineure du sous-système requis. N'a pas de réelle importance.

Reserved : DWord

.....Euh je n'en sais pas plus.

SizeOfImage : DWord

C'est la taille du fichier et doit être un multiple de SectionAlignment.

SizeOfHeaders : DWord

C'est la taille des en-têtes, c'est à dire en-tête DOS + en-tête PE + en-têtes des sections, et doit être un multiple de FileAlignment.

Checksum : DWord

C'est un nombre qui résulte d'un algorithme propre à Windows (en fait, cet algorithme est disponible dans IMAGHELP.DLL).

Subsystem : Word

Le nom du sous-système requis pour exécuter le programme.

Voir plus bas le tableau des différentes valeurs.

DLL Characteristics : Word

Sert à définir quelques propriétés si le fichier est une DLL.

Voir les constantes PE pour plus d'informations.

SizeOfStackReserve : DWord

C'est la taille de la pile à réserver pour exécuter le programme.

SizeOfStackCommit : DWord

C'est la taille de la pile qui va être alloué. En principe, toute la pile n'est pas allouée tout de suite, c'est pour ça qu'il y a un champ pour donner la taille maximum (SizeOfStackReserve) et un champ pour donner la taille réellement allouée (SizeOfStackCommit).

SizeOfHeapReserve : DWord

C'est la taille du tas à réserver. Le tas est une immense mémoire qui peut servir au programme dans le cas ou la pile ne suffirait pas (la pile est assez limitée).

SizeOfHeapCommit : DWord

C'est la taille du tas réellement allouée.

LoaderFlags : DWord

N'est plus utilisé.

NumberOfRvaAndSizes : DWord

Donne le nombre d'entrées qu'il y a dans le tableau qui suit l'en-tête optionnelle mais de nos jours, il est souvent de $10 (16).

DataDirectory : Array[0..NumberOfRvaAndSizes-1] of TImageDataDirectory

C'est ici qu'on trouve le tableau des sections.

Ce tableau n'est là que pour donner un accès rapide aux sections. En effet, certaines lignes du tableau sont des entrées "standard" qui doivent pointer vers des sections particulières.

3. Sections Headers / En-tête des sections

L'en-tête des sections est un peu différent de ce que nous avons vu jusqu'ici car il n'a pas une taille constante. En effet, il y a autant d'en-tête de sections qu'il y a de sections dans le fichier. Le nombre de sections étant définis dans le champ "NumberOfSections" de l'en-tête du fichier image.

Name : Array[0..7] of Char

Il s'agit ici du nom de la section. Le nom comporte 8 caractères et est remplit de #0 si le nom en comporte moins, en clair il faut toujours 8 caractères et pas plus pas moins.

Ce nom est purement arbitraire car le loader de Windows n'y fait pas cas, il sert juste pour les programmeurs.

Pour les accès à des sections particulières, il y a le tableau des DataDirectory.

VirtualSize : DWord

Il s'agit de la taille à réserver par Windows lors du chargement en mémoire de la section.

La taille virtuelle, celle-ci, peut être plus grande que la taille sur le disque mais elle sera alors remplit de 0 sur la différence de taille. Il sera libre à vous alors d'utiliser cet espace à votre gré.

VirtualAddress : DWord

C'est l'adresse virtuelle de la section. C'est à dire celle qui sera utilisée lorsque la section sera chargée en mémoire.

Attention, cette section doit être alignée sur le champ "SectionAlignment" et c'est une RVA.

SizeOfRawData : DWord

Il s'agit de la taille réellement occupée dans le fichier par la section. Si cette section contient des données non initialisées, enfin est marquée comme contenant des données non initialisées, ce champ devrait être à 0.

Attention, ce champ doit être aligné sur le "FileAlignment".

PointerToRawData : DWord

C'est l'offset du début de la section. Si cette section est signalée comme contenant des données non initialisées, ce champ devrait être à 0.

Attention, ce champ doit être aligné sur le "FileAlignment" et doit être de préférence un multiple de 4 pour des raisons de performances.

PointerToRelocations : DWord

Il s'agit de l'adresse de l'entrée de la section dans la table de "relocations". Doit être à 0 si le fichier est un exécutable.

Les relocations servent lorsqu'un programme demande à chargé un fichier mais ce dernier ne peut pas être chargé à l'adresse donné par le champ "ImageBase".

PointerToLineNumbers : DWord

Il s'agit d'une adresse qui pointe vers le nombre de lignes de la section. Etant donné que je n'ai jamais rencontré ce champ avec un valeur autre que 0, je ne peux pas vous renseigner plus que ça.

NumberOfRelocations : Word

Il s'agit juste du nombre de "relocations" que l'on a vu plus haut et qui doit être, en général, à 0 si c'est un exécutable.

NumberOfLineNumbers : Word

Nombre de structure "Nombre de lignes" vu plus haut.

Characteristics : DWord

Il s'agit des différentes caractéristiques de la section. Elles vont définir le contenu de la section et ses attributs. C'est ici que l'on définit si il y a des lignes de codes ou pas, des données initialisées ou pas etc...

Voir la correspondance des constantes PE.

Section "Importations"

Explication de la section "Importations"

25/01/2006 yarocco

1. Présentation

La section "Importations" sert, comme son nom l'indique, à réfencer les differentes importations de fonctions dans le programme. En effet, chaque programme utilise des DLL (Dynamic Link Library = Bibliotheque de liens dynamiques) ou d'autres programmes qui exportent, c'est-à-dire mettent à disposition, des fonctions. Il y a par exemple les fonctions pour la GUI (Graphic User Interface = Interface graphique d'utilisateur) dans User32.dll ou d'autres très basiques dans kernel32.dll.

Seulement, pour pouvoir se servir de ces fonctions exportées il faut avoir leur adresse, et c'est à ça que sert la table d'importations. Elle contient donc les differentes fonctions que le programme pourra charger et, lors de son execution, les adresses des dites fonctions. Lorsque le fichier est sur le disque et n'est pas executé, il ne contient alors que des noms de fonctions mais si on regarde dans dans le code, lors d'un appel d'une fonction qui ne fait pas partie du programme, il y a un appel vers une adresse et non vers un nom. En fait, l'adresse référence un pointeur dans la table d'importation qui contient toujours pour l'instant le nom de la fonction. Mais lors du chargement ne mémoire du programme, le loader de Windows va chercher les DLL qui sont importées et va changer leur nom par leur adresses. En fait, Windows va d'abord devoir regarder les DLL appellées si elles exportent bien les fonctions, charger ces DLL dans l'espace mémoire du processus qui les appellent, et ensuite traduire les noms par les adresses. Ensuite, lorsque le programme va faire appel à une entrée de la table d'importation, celle ci pointera vers la fonction voulue.

2. Explication théorique

Le début de la section "Importation" commence par un tableau de *Image Import Descriptor* . La particularité de ce tableau est qu'on ne connait pas son nombre d'éléments mais la fin de celui-ci est signalée par un éléments dont tous ses champs sont nuls.

1. Image Import Descriptor

Regardons alors la structure de ses éléments, les *Image Import Descriptor* :

Characteristics / OriginalFirstThunk : DWORD

Il s'agit d'une VA qui pointe vers un tableau de ImageThunkData. Je décrirais ce champ un peu plus bas.

TimeDataStamp : DWORD

Ce champ est censé indiquer la date de création du fichier mais il est le plus souvent mis à 0.

ForwarderChain : DWORD

Index de la première réference "forwarder". J'avoue ne pas trop savoir ce que ce champ fait ici, de plus je l'ai toujours vu à 0 donc je ne peux pas donner plus d'explication. Si quelqu'un le sait : qu'il m'en fasse part.

Je vais tout de même vous expliquer le principe du "Forwarding", qui sera exploité dans la section "Exportations". Il s'agit en fait d'exporter une fonction qui n'est pas dans notre programme/DLL. C'est un peu une ruse pour dissimuler quelques fonctions internes de Windows dans des DLL qui sont peut être aménée à changer de nom ou autre modification. Par exemple, il y a pas mal de fonctions de Kernel32.dll qui sont en fait des "Forwarding" vers ntdll.dll. Microsoft peut ainsi changer le nom ou la méthode d'exportion de ntdll.dll et changer l'importation de Kernel32.dll sans dire quoi que ce soit car Kernel32.dll exportera toujours la même chose.

Name : DWORD

C'est une VA qui pointe vers le nom, à zéro terminal - qui se termine par un caractère nul-, de la DLL (Ex: Kernel32.dll).

FirstThunk : DWORD

Il s'agit exactement de la même chose que le champ Characteristics vu plus haut. Ce champ pointe vers un tableau de *ImageThunkData*.

1. Pourquoi deux fois le même champ ?

En effet, je pense que vous avez remarqué que le premier et le dernier champ pointent tous les deux vers une même structure : *ImageThunkData*.

On peut donc se demander s'il n'y a pas double emploi inutilement. La réponse est non. Déjà, les deux champs ne pointent pas vers la même adresse, ce qui élimine déjà le principe du double emploi.

Il faut se pencher un peu sur le fonctionnement du loader: comme je vous l'ai dis plus haut, lors du chargement du programme, le loader va charger les DLL et remplacer le nom des fonctions DLL dans la table des importations par leur adresse. Et bien en fait, c'est ici que cela se fait, le tableau pointé par le champ Characteristics contient et contiendra toujours le nom des fonctions des DLL alors que celui pointé par ce champ sera changé par le loader de Windows.

Petite exception pour les utilisateurs de Borland : il n'y a que le champ FirstThunk qui pointe vers un tableau qui est remplit correctement avec les noms des fonctions des DLL lorsque le fichier n'est pas chargé et Characteristics ne pointe pas vers un tableau valide alors que c'est normalement l'inverse (Characteristics : OK et FirstThunk : pas OK). Vive Borland :)

2. ImageThunkData / Import Lookup Table

Nous étudions ici la structure pointée par les champs Characteristics et FirstThunk. Cette structure est très simple puisqu'il s'agit d'un simple champ de type DWord.

Il faut ensuite vérifier une petite condition qui déterminera le type d'importation : on regarde le premier bit du champ. S'il est à 1, cela indique qu'il s'agit d'une importation ordinale, sinon il s'agit d'une importation par nom. Dans le cas d'un importation ordinale, le numéro ordinal est indiqué par les autres 31 bits.

Dans le cas d'une importation par nom, les 31 autres bits indique l'adresse VA, toujours pas relative à l'ImageBase, d'une nouvelle structure : la *Hint/Name Table* .

1. Types d'exportation

Je fais juste une petite aparté pour ceux qui ne connaissent pas encore les différents types d'importation.

1. Importation ordinale

Il s'agit en fait d'importer les fonctions d'après un numéro. Bien sur, il faut auparavant que la fonction importée appelle une fonction qui soit exportée ordinalement aussi ! Mais le problème avec ce genre d'exportation est qu'on est obligé d'ajouter toujours les fonctions exportées (on doit toujours incrementé le numéro d'exportation de la fonction) et qu'on ne peut pas supprimer une fonction exportée facilement sous peine de quoi on risque de chambouler tout les reste des numéro et les utilisateurs qui importaient la fonction x vont en fait importer la fonction x-1 par exemple ca qui n'est pas très sympa pour l'utilisateur de votre DLL.

2. Importation par le nom

Comme son nom l'indique, on va importer les fonctions par leur nom, ce qui est beaucoup plus pratique est donc beaucoup plus répendu de nos jours.

3. Hint/Name Table

Cette structure est encore une fois très simple.Elle décrit le nom et le numéro d'une fonction importée par son nom.

Hint : Word

Ce champ indique le numéro d'importation de la fonction. Mais il n'est pas obligatoirement valide.Il est juste là à titre indicatif pour la recherche de la fonction dans la DLL pour le loader.

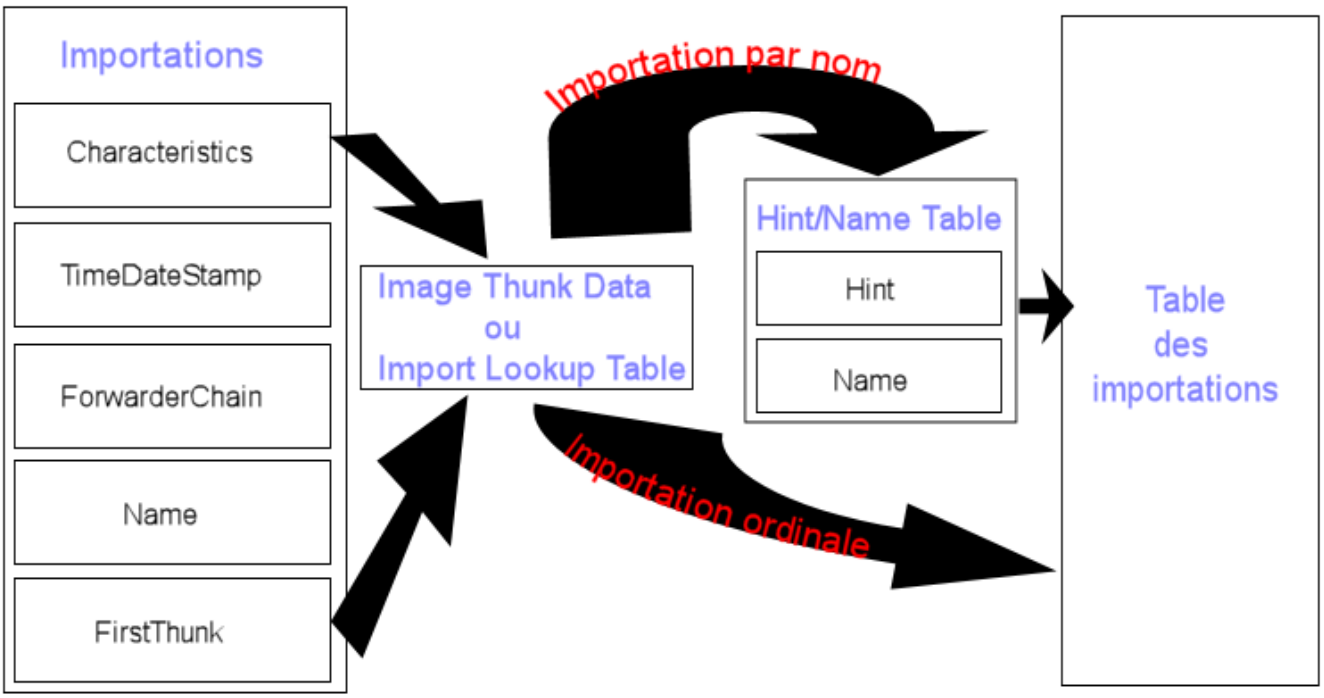
Name : Array[0..???] of Byte

Ce champ indique le nom de la fonction importée. La taille est inconnu car la chaine est à zéro terminal, comprendre que la chaine se termine par un caractère nul.

3. Illustration

Pour essayer d'illustrer le concept des importations dans un petit dessin.

Bien sur, il n'y a pas de quoi s'extasier devant mon art mais je pense qu'il peut aider à comprendre rapidement l'implémentation de cette section.



Section "Exportations"

Explication de la section "Exportations"

25/01/2006 yarocco

1. Présentation

La section « Exportations » (EAT en anglais pour Export Address Table) sert, comme son nom l'indique, à référencer les différentes fonctions qui doivent être exportées.

Exporter une fonction, cela veut dire la rendre accessible par tous les programmes/DLL qui le désire. C'est donc l'opposé des importations, qui s'en serait douté :). Dans la plupart des cas, les programmes n'exportent pas de fonctions alors que les DLL si. Donc, en principe, vous trouverez une section .edata (pour Export Data) dans les DLL et non dans les EXE.

Cependant, rien n'interdit les EXE standard d'exporter des fonctions. En fait, une des principales raisons qui pousse à éviter l'exportation dans les programmes standard (EXE), c'est qu'ils sont assez lourds. Et comme le programme qui désire importer les fonctions d'un fichier doit d'abord le charger en mémoire, cela alourdit considérablement la taille occupée par son process. Ce qui n'est, en général, pas très bon pour le système.

2. Explication théorique

Au début de la section Exportations, nous avons une structure de type ImageExportDirectory qui donne les quelques informations qu'il y a à savoir sur cette section. En effet, alors que d'autres sections font une multitude d'appels à d'autres structures, VA et autres, cette section est plutôt simple à explorer.

Commençons donc par analyser ce ImageExportDirectory :

Characteristics : DWord

Ce champ n'est pas utilisé, il est toujours à 0.

TimeDateStamp : DWord

Indique la date de création du fichier, enfin est censé l'indiquer car la plupart du temps, on trouve 0.

MajorVersion : Word

Ce champ ne sert à rien et est à 0.

MinorVersion : Word

Encore un champ qui ne sert à rien et qui est à 0.

Name : DWord

Il s'agit de type l'adresse virtuelle d'une chaîne, à zéro terminal, qui indique le nom du fichier. Ainsi, si le fichier est renommé, on pourra toujours retrouver le nom original mais cela sert surtout au loader en interne.

Base : DWORD

Il s'agit d'un nombre par rapport auquel il faudra se baser.

En fait, le programmeur qui exporte ses fonctions dans ses DLL n'est pas obligé de commencer son tableau ordinal à 1, il peut très bien le commencer à 20. Il lui faudra juste le signaler dans le champ Base en disant que la base est de 19 (début-1). Ainsi, on saura que lorsqu'on voudra lire un éléments dans le tableau pointé par AddressOfFunctions, il faudra retirer autant d'éléments que Base l'indique. Il n'y a que ce tableau qui "joue" avec le paramètre base, les autres tels que le tableau pointé par AddressOfNameOrdinals partent réellement de 0 et non de Base.

En résumé, pour le tableau qui contient toutes les fonctions :

IndexReel = Index - Base

NumberOfFunctions : DWORD

C'est le nombre total de fonctions exportées.

NumberOfNames : DWORD

C'est le nombre total de fonctions exportées par leur nom.

AddressOfFonctions : DWORD

C'est l'adresse du tableau qui contient toutes les adresses des fonctions à exporter.

AddressOfNames : DWORD

C'est l'adresse du tableau qui contient les adresses des noms des fonctions exportées par leur nom.

AddressOfNameOrdinals : DWORD

C'est l'adresse du tableau qui contient les numéros des fonctions exportées.

1. Types d'exportation

Comme vous avez pu le constater dans les specifications des champs, il existe plusieurs types d'exportation possible.

1. Exportation ordinale

Les fonctions qui sont exportées comme tel sont repérées par un numéro ordinal. Il y a un certain désavantage à exporter ses fonctions ainsi : si on ne peut pas supprimer ou remanier l'exportation des fonctions sans grand changement sans quoi l'utilisateur qui utilisait cette DLL appelait la fonction qui avait le numéro x, mais si vous changez les numéro, l'utilisateur appellera la fonction numéro x-1 par exemple, il lui faudra alors reprendre son code. Ce qui peut être assez perturbant. C'est pour cela qu'aujourd'hui, la plupart des exportations se font par le nom des fonctions.

2. Exportation par le nom

Les fonctions qui sont exportées comme tel sont donc exportées par leur nom. C'est le type d'exportation qui est majoritairement employé de nos jours.

3. Application au contexte

Vous avez sûrement remarqué qu'il n'y a qu'un seul type d'exportation (Exportation par nom) qui était comptabilisé dans les champs vus plus haut. Cela est dû simplement au fait que le nombre de fonctions exportées par leur numéro est implicite. En effet, on a le nombre total de fonctions et le nombre de fonctions exportées par leur nom, sachant qu'il n'y a que 2 possibilités pour exporter une fonction, le nombre résultant de la différence entre le nombre de fonction total et le nombre de fonction exportées par leur nom sera le nombre de fonctions exportées par un numéro ordinal.

En résumé :

NombreDeFonctionsOrdinales = NombresDeFonctions - NombresDeFonctionsNoms

Les fonctions exportées par un numéro ordinal sont directement dans le tableau pointé par AddressOfFunctions.

2. Tableaux pointés

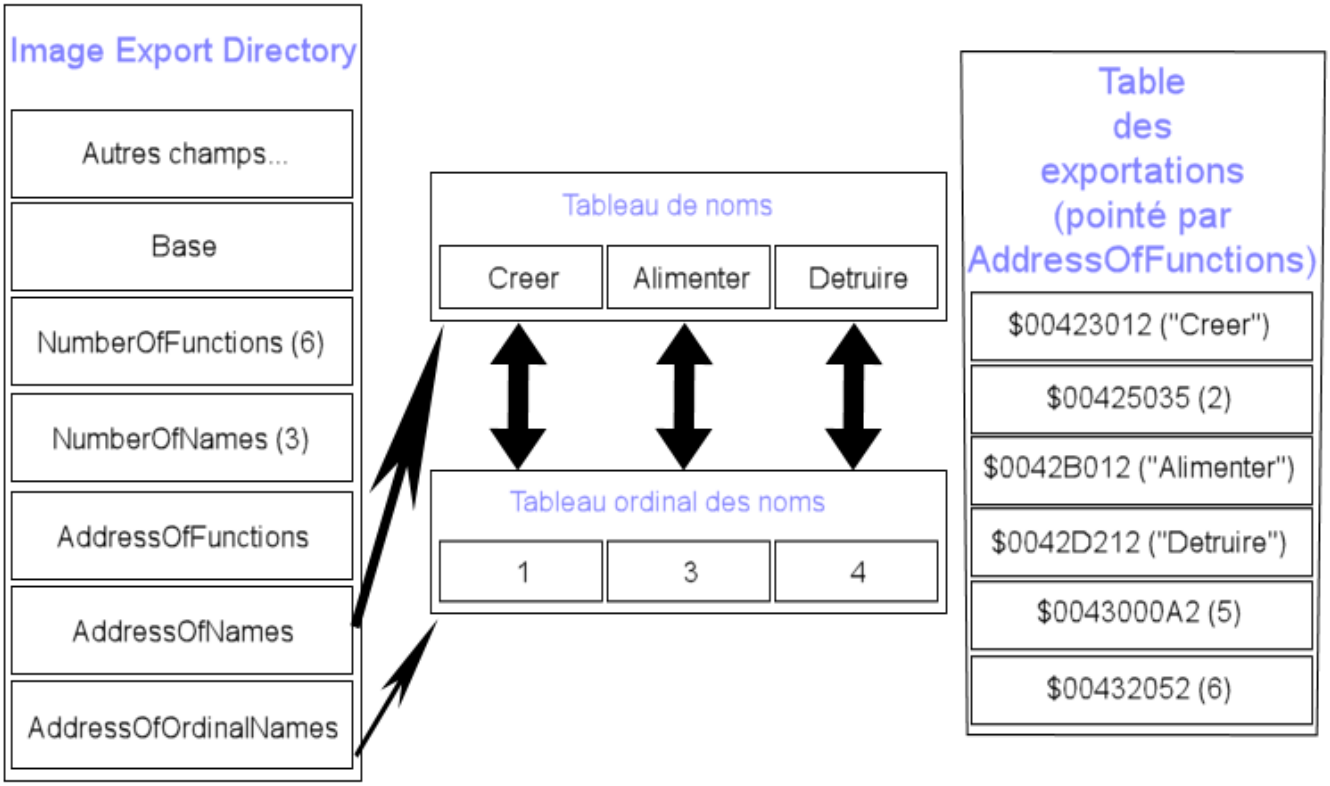
Présentation rapide des différents tableaux pointés par les champs expliqués plus haut.

- Celui pointé par AddressOfFunctions: Il s'agit d'un tableau de DWORD et le nombre d'éléments est égal au NumberOfFunctions
- Celui pointé par AddressOfNames: Il s'agit d'un tableau de DWORD qui sont en fait les adresses des noms. Il faut donc lire chaque éléments (adresse), puis aller à cette adresse pour lire le nom de la fonction qui est une chaîne à zéro terminal.
- Celui pointé par AddressOfNameOrdinals: Il s'agit d'un tableau simple de WORD, il y a autant d'éléments que dans le tableau pointé par AddressOfNames.

3. Illustration

Pour essayer d'illustrer le concept des exportations dans un petit dessin.

Bien sur, il n'y a pas de quoi s'extasier devant mon art mais je pense qu'il peut aider à comprendre rapidement l'implémentation de cette section.



La section "Relocations"

Explication de la section "Relocations"

04/02/2006 yarocco

1. Introduction

Pour comprendre l'utilité de cette section, il faut d'abord comprendre quleques concepts.

Tout d'abord, il faut savoir que lors d'une compilation d'un programme standard, les adresses sont calculées par le compilateur et sont ainsi écrites dans le programme en "dur". C'est pour cette raison qu'on trouve souvent des instructions comme : PUSH 0040100000 ou MOV EAX, 0042000000

Ces adresses sont des RVA (Relative Virtual Address) et sont par leur définition calculées par rapport au champ

ImageBase de l'en-tête du programme. Ce champ est une adresse qui indique au loader de Windows que le programme a été compilé pour être chargé à cette adresse. Aussi, lorsque le loader chargera le programme de base en mémoire, il va créer une espace mémoire pour le programme et le charger à partir de l'adresse indiquée par le champ *ImageBase*.

Jusqu'ici, tout va bien mais les choses se compliquent si ce programme veut charger un autre programme dans son espace mémoire (pour appeller ses fonctions par exemple). En effet, la plupart des programmes sont compilés avec un champ *ImageBase* à $0040000000. Donc, si nous avons déjà un programme chargé à cette adresse et qu'un autre programme demande aussi à être chargé à cette adresse, il risque d'y avoir un écrasement. Pour éviter ce genre de chose, le loader va charger le second programme à une autre adresse. Mais voilà que survient le problème des adresses dans le code du second programme.

En effet, j'ai signlé plus haut que les adresses étaient relatives à l'ImageBase. Or l'*ImageBase* est censé représenter l'adresse de chargement du programme et dans le cas du second programme chargé, cette adresse n'est pas bonne car le programme a été chargé à une adresse arbitraire. Les références vers des adresses constantes dans le code du second programme sont donc faussées.

Pour remettre un peu d'ordre, le linker doit prévoir ce genre de comportement et intégrer les adresses de toutes les références à des adresses constantes dans le code. Ainsi, une fois que le programme aura été chargé, il suffit ensuite de lire où se situent les adresses du code qui peuvent être faussée et les modifier par rapport au delta du champ *ImageBase* et de l'adresse de chargement réelle.

Bien sur, toutes les adresses ne sont pas prévues pour être relocalisées excepté si l'utilisateur le demande lors de la compilation.

J'espere avoir été clair dans mon explication.

2. Explication théorique

Cette section est divisée en plusieurs parties. Chaque partie représente un bloc de 4Ko appelé page. Cette représentation est en partie liée aux processeurs x86 qui disposent d'un mécanisme de pages de 4Ko avec la pagination.

1. Fixup Block / Bloc de correctifs

Au début de chaque bloc, on trouve cette structure :

Page RVA : DWord

C'est un offset qui est à ajouté au champ ImageBase pour obtenir la page où le loader doit appliquer les correctifs.

Bloc Size : DWord

C'est le nombre d'octets qu'il y a dans ce bloc en comptant cet en-tête. Il y a donc en fait Bloc Size - 8 octets.

Après cet en-tête, on trouve un tableau d'élèments qui indique le type de correction à effectuer. Pour connaitre le nombre d'élèments, il suffit de faire : **NbrElements = (BlocSize - 8) / 2** car chaque élément fait 2 octets.

Type : 4 bits

Ce sont les 4 bits de poids fort de la structure (2 octets). Ils indiquent le type de correctif qu'il faut appliquer.

Voir les constantes PE.

Offset : 12 bits

C'est l'offset qu'il faut ajouter à l'adresse pointée par : ImageBase + PageRVA pour trouver l'adresse à modifier.

Ajouter une section

Ajouter une nouvelle section dans un fichier au format Portable Executable

25/01/2006 yarocco

1. Explication théorique

L'ajout d'une section dans un fichier n'est pas une chose très difficile à faire. D'une part, elle ne demande pas beaucoup d'opérations une fois que l'en-tête est lu, et d'autre part, il n'y pas pas de connaissance très technique à connaitre à part la notion d'alignement mais qui n'est vraiment pas dure à comprendre.

Nous allons donc voir les 4 étapes qu'il faut pour ajouter une nouvelle section.

1. Incrémentation du NumberOfSections

Tout d'abord, il faut le situer. Ce champ est le 2eme dans l'en-tête d'un fichier image (*ImageFileHeader*) à l'offset **$6** par rapport au début de l'en-tête PE (*PE Header*). Nous avons donc juste à lire cette valeur, l'incrémenter et la remplacer. Ce n'est vraiment pas une chose difficile et ce quelque soit le langage utilisé.

2. Incrémentation du SizeOfImage

Ce champ est 20ième dans l'en-tête optionnel (*Optional Header*) à l'offset **$50** par rapport au début de l'en-tête PE (*PE Header*). Il faut donc faire comme au-dessus :rechercher, incrémenter de la taille de notre section, remplacer.

3. Ajout de l'en-tête de section

Une fois que nous avons dis qu'il y avait un nombre de sections plus élévé qu'à l'origine, il faut bien représenter le nombre de sections qui différe physiquement sinon le loader cherchant à lire un nombre NumberOfSections+1 de sections ne va pas comprendre. Il faut donc tout d'abord mettre une en-tête à notre section.

Il faut donc localiser le dernier en-tête de section puis rajouter le notre juste derrière, en faisant bien attention de ne pas déborder de l'en-tête. Pour cela, on verifie que l'offset de notre en-tête de section reste en dessous de la valeur du champ *SizeOfHeaders* dans l'en-tête optionnel.

En effet, nous pouvons dans la grande majorité des cas rajouter une section car la taille de l'en-tête complète doit être alignée sur le *FileAlignment*, ce qui signifie qu'il y aura un remplissage avec des "0" pour aligner l'en-tête. Et c'est ici que nous pouvons profiter de ce "gaspillage" pour ajouter notre en-tête.

Pour connaitre l'adresse d'un en-tête de section, il faut faire :

DébutPEHeader + $78 + (NumberOfRvaAndSizes * $8) + (NumberOfSections * $28) = FinDeLaDernièreEntêteDeSection

Explications du calcul :

- *DébutPEHeader* : en principe, on se référence toujours par rapport au début du PEHeader
- *$78* : C'est la taille du PEHeader sans les DataDirectory
- *NumberOfRvaAndSizes * $8* : Taille des DataDirectory
- *(NumberOfSections+1) * $28* : Nous allons à la fin de l'en-tête de la dernière section et nous ajoutons la taille d'une en-tête de section. Nous sommes donc pret à écrire le nouvel en-tête.

Une fois que nous avons trouvée l'offset pour écrire, il nous faut encore les valeurs des champs. C'est peut être la chose la plus compliquée de cet article.

Pour cela, nous allons reprendre les champs d'un en-tête de section :

Name : Array[0..7] of Byte

Nom de la section.

VirtualSize : DWORD

Taille virtuelle de la section (peut être plus élevée que la taille réelle).

Valeur : La même que la taille réele

VirtualAddress : DWORD

Adresse virtuelle de la section (Aligné sur **SectionAlignment**)

Les sections virtuelles doivent se suivre.

Soit x = (DernièreSection.VirtualAddress + DernièreSection.VirtualSize)

Valeur : x + SectionAlignment - (x mod SectionAlignment)

SizeOfRawData : DWORD

Taille physique de la section (Aligné sur **FileAlignment**).

Valeur : Taille que nous allons rajouter (à vous de voir)

PointerToRawData : DWORD

Pointe vers le début de la section physique et peut être n'importe où dans le fichier (Aligné sur **FileAlignment**). Comme nous allons rajouter la section à la fin du fichier :

Valeur : Taille du fichier avant modification

PointerToRelocations : DWORD

En principe ne vous servira pas mais consulter l'explication de la section Relocations si vous voulez en savoir plus.

Valeur : 0

PointerToLinenumbers : DWORD

Ne sert à rien si le Flag *IMAGE_FILE_LINE_NUMS_STRIPPED* est à 1 dans les **Characteristics** du **ImageFileHeader**.

Valeur : 0

NumberOfRelocations : Word

De même que PointerOfRelocations au dessus.

Valeur : 0

NumberOfLinenumbers : Word

Ne sert à rien si le Flag *IMAGE_FILE_LINE_NUMS_STRIPPED* est à 1 dans les **Characteristics** du **ImageFileHeader**.

Valeur : 0

Characteristics : DWORD

Voir les explications dans les tableaux des constantes PE.

Peut-être mit à 0.

Toutes ces valeurs sont à indiquer en octet !

Il ne vous reste plus qu'à écrire ces valeurs à l'offset trouvé plus haut.

4. Ajout de la section

Maintenant, il faut encore écrire la section elle même.

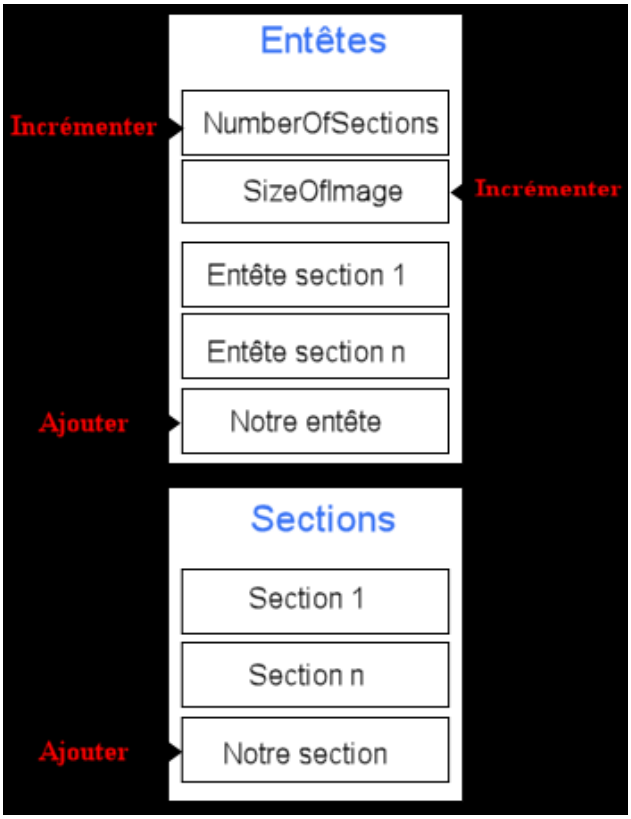
Chose pas bien difficile puisqu'il suffit d'écrire autant de "0" qu'il faut pour que le fichier ait une taille égale à son ancienne taille plus la taille de votre section.

NouvelleTaille = AncienneTaille + SizeOfRawData

Voilà vous savez, en théorie ajouter une section à un fichier PE existant. Pour tester, il suffit d'executer le fichier modifié et s'il démarre normalement, c'est que vous avez réussi.

2. Illustrations

Comme à mon habitude, j'ai décidé de vous aider par une petite illustration qui résume les différentes modificatiosn à effectuer pour ajouter une section.



Les constantes des fichiers PE

Les constantes des fichiers PE et leur description

11/02/2006 yarocco

1. Notations et valeurs

Tout d'abord, je dois signaler que pour un soucis d'homogénéité avec les autres articles, celui ci signale une valeur

héxadecimale par le caractère "$" (que l'on trouve en Pascal/Delphi) et non le "0x" (comme en C/C++). L'absence de signe indique une valeur au format décimal.

Ensuite, il faut bien regarder les tableaux. En effet, certaines tableaux rassemblent des constantes. Cela signifie que la valeur du tableau correspond à la constante. Cepedant, d'autres tableaux rassemblent des attributs. Cela signifie qu'il peut y en avoir plusieurs pour le champ concerné et qu'il faut tous les tester à l'aide de masques.

Les tableaux des constantes auront une colone "Constante" alors que les tableaux des attributs auront une constante "Drapeau".

Les constantes que vous trouverez ci dessous sont en principe déjà implémentées dans les bibliothéques des différents langages (winnt.h pour C/C++, windows.pas pour Delphi etc...), Vous ne devriez donc pas les réecrire si vous décidez de vous en servir.

2. Le champ "Machine" / ImageFileHeader.Machine

Le champ "Machine"

Constante	Valeur	Description
IMAGE_FILE_MACHINE_UNKNOWN	$0	Machine inconnue
IMAGE_FILE_MACHINE_ALPHA	$184	Alpha AXP™.
IMAGE_FILE_MACHINE_ARM	$1c0	
IMAGE_FILE_MACHINE_ALPHA64	$284	Alpha AXP™ 64-bit.
IMAGE_FILE_MACHINE_I386	$14c	Intel 386 ou processeurs compatibles et supérieurs.
IMAGE_FILE_MACHINE_IA64	$200	Intel IA64™
IMAGE_FILE_MACHINE_M68K	$268	Motorola 68000 series.
IMAGE_FILE_MACHINE_MIPS16	$266	
IMAGE_FILE_MACHINE_MIPSFPU	$366	MIPS with FPU
IMAGE_FILE_MACHINE_MIPSFPU16	$466	MIPS16 with FPU
IMAGE_FILE_MACHINE_POWERPC	$1f0	Power PC, little endian.
IMAGE_FILE_MACHINE_R3000	$162	
IMAGE_FILE_MACHINE_R4000	$166	MIPS® little endian.
IMAGE_FILE_MACHINE_R10000	$168	
IMAGE_FILE_MACHINE_SH3	$1a2	Hitachi SH3
IMAGE_FILE_MACHINE_SH4	$1a6	Hitachi SH4
IMAGE_FILE_MACHINE_THUMB	$1c2	

3. Le champ "Characteristics" / ImageFileHeader.Characteristics

Le champ "Characteristics"

Drapeau	Valeur	Description
IMAGE_FILE_RELOCS_STRIPPED	$0001	Pour les images seulement sur Windows CE, Windows NT et supérieurs. Indique que ce fichier ne contient pas de relocations et doit être chargé à l'adresse indiquée par le champ ImageBase. Si cette adresse n'est pas valable, le loader créer une erreur. Par défaut, le linker ne créer pas de relocations dans un fichier EXE.
IMAGE_FILE_EXECUTABLE_IMAGE	$0002	Pour les images seulement. Indique ce fichier est valide et qu'il peut être executé. Si ce drapeau n'est pas armé, cela signifie généralement qu'il y a eu une erreur du linker.
IMAGE_FILE_LINE_NUMS_STRIPPED	$0004	Les "COFF line numbers" ont été supprimées.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	$0008	Les entrées des symboles locaux de la table "COFF symbol" ont été supprimées.
IMAGE_FILE_AGGRESSIVE_WS_TRIM	$0010	Aggressively trim working set.
IMAGE_FILE_LARGE_ADDRESS_AWARE	$0020	L'application peut adresser sur plus de 2Gb
IMAGE_FILE_16BIT_MACHINE	$0040	Réservé pour le futur
IMAGE_FILE_BYTES_REVERSED_LO	$0080	Little endian.
IMAGE_FILE_32BIT_MACHINE	$0100	L'architecture de la machine est en 32 bits.
IMAGE_FILE_DEBUG_STRIPPED	$0200	Les informations de debogage ont été supprimées.
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	$0400	Si cette image est sur un support amovible, elle doit être copiée et executé depuis cette copie.
IMAGE_FILE_SYSTEM	$1000	Ce fichier image est un fichier système, pas un programme utilisateur.
IMAGE_FILE_DLL	$2000	Ce fichier image est une DLL (Dynamic Link Library). Ces fichiers sont considérés comme des executables excepté qu'ils ne peuvent pas executé directement (sans programme hôte).
IMAGE_FILE_UP_SYSTEM_ONLY	$4000	File should be run only on a UP machine.
IMAGE_FILE_BYTES_REVERSED_HI	$8000	Big endian.

4. Le champ "SubSystem" / OptionalHeader.SubSystem

Le champ "Subsystem"

Constante	Valeur	Description
IMAGE_SUBSYSTEM_UNKNOWN	0	Sous-système inconnu.
IMAGE_SUBSYSTEM_NATIVE	1	Utilisé pour les drivers de périphérique et process natifs de Windows NT.
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	L'image doit être éxecuté sous le système graphique Windows™.
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	L'image doit être éxecuté sous le système de caractère Windows (comprendre le mode console).
IMAGE_SUBSYSTEM_POSIX_CUI	7	L'image doit être éxecuté sous le système POSIX.
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	L'image doit être éxecuté sous Windows CE.
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	L'image est une application EFI.
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	11	Image is an EFI driver that provides boot services.
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	12	Image is an EFI driver that provides runtime services.

5. Le champ "DLL Characteristics" / OptionalHeader.DLLCharacteristics

Le champ "DLL Characteristics"

Constante	Valeur	Description
	$0001	Réservé.
	$0002	Réservé.
	$0004	Réservé.
	$0008	Réservé.
IMAGE_DLLCHARACTERISTICS_NO_BIND	$0800	Ne pas lier cette image.
IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	$2000	Le pilote est un un pilote de type WDM.
IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	$8000	L'image est de type Terminal Server.

6. Le champ "Characteristics" / ImageSectionHeader.Characteristics

Le champ "Characteristics"

Drapeau	Valeur	Description
IMAGE_SCN_TYPE_REG	$00000000	Réservé pour le futur
IMAGE_SCN_TYPE_DSECT	$00000001	Réservé pour le futur
IMAGE_SCN_TYPE_NOLOAD	$00000002	Réservé pour le futur
IMAGE_SCN_TYPE_GROUP	$00000004	Réservé pour le futur
IMAGE_SCN_TYPE_NO_PAD	$00000008	La section ne doit pas être remplie jusqu'au prochain alignement. Cet attribut est obsolète et a été remplacé par IMAGE_SCN_ALIGN_1BYTES. N'est valide que pour les fichiers objets.
IMAGE_SCN_TYPE_COPY	$00000010	Réservé pour le futur
IMAGE_SCN_CNT_CODE	$00000020	La section contient du code éxecutable.
IMAGE_SCN_CNT_INITIALIZED_DATA	$00000040	La section contient des données initialisées.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	$00000080	La section contient des données non initilaisées.
IMAGE_SCN_LNK_OTHER	$00000100	Réservé pour le futur

IMAGE_SCN_LNK_INFO	$00000200	La section contient des commentaires ou autres informations. La section .drectve est de ce type. N'est valide que pour les fichiers objets.
IMAGE_SCN_TYPE_OVER	$00000400	Réservé pour le futur
IMAGE_SCN_LNK_REMOVE	$00000800	La section ne doit pas être inclus dans un fichier image. N'est valide que pour les fichiers objets.
IMAGE_SCN_LNK_COMDAT	$00001000	La section contient des données COMDAT. N'est valide que pour les fichiers objets.
IMAGE_SCN_MEM_FARDATA	$00008000	Réservé pour le futur
IMAGE_SCN_MEM_PURGEABLE	$00020000	Réservé pour le futur
IMAGE_SCN_MEM_16BIT	$00020000	Réservé pour le futur
IMAGE_SCN_MEM_LOCKED	$00040000	Réservé pour le futur
IMAGE_SCN_MEM_PRELOAD	$00080000	Réservé pour le futur
IMAGE_SCN_ALIGN_1BYTES	$00100000	Alignement des données sur 1 octet. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_2BYTES	$00200000	Alignement des données sur 2 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_4BYTES	$00300000	Alignement des données sur 4 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_8BYTES	$00400000	Alignement des données sur 8 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_16BYTES	$00500000	Alignement des données sur 16 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_32BYTES	$00600000	Alignement des données sur 32 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_64BYTES	$00700000	Alignement des données sur 64 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_128BYTES	$00800000	Alignement des données sur 128 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_256BYTES	$00900000	Alignement des données sur 256 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_512BYTES	$00A00000	Alignement des données sur 512 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_1024BYTES	$00B00000	Alignement des données sur 1024 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_2048BYTES	$00C00000	Alignement des données sur 2048 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_4096BYTES	$00D00000	Alignement des données sur 4096 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_ALIGN_8192BYTES	$00E00000	Alignement des données sur 8192 octets. N'est valide que pour les fichiers objets.
IMAGE_SCN_LNK_NRELOC_OVFL	$01000000	La section contient des relocations étendues.
IMAGE_SCN_MEM_DISCARDABLE	$02000000	La section peut être détachée du fichier.
IMAGE_SCN_MEM_NOT_CACHED	$04000000	LA section ne peut pas être mise en cache.
IMAGE_SCN_MEM_NOT_PAGED	$08000000	La section ne peut pas être paginée.
IMAGE_SCN_MEM_SHARED	$10000000	La section peut être partagée en mémoire.
IMAGE_SCN_MEM_EXECUTE	$20000000	La section peut éxecuté du code.
IMAGE_SCN_MEM_READ	$40000000	La section peut être lu.
IMAGE_SCN_MEM_WRITE	$80000000	La section peut être écrite.

7. Les types correctif de la section "Relocations"

Les types de correctif

Constante	Valeur	Description
IMAGE_REL_BASED_ABSOLUTE	0	Ce type doit être sauté. Il n'est présent que pour remplir la structure jusqu'à l'alignement.
IMAGE_REL_BASED_HIGH	1	The fixup adds the high 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the high value of a 32-bit word.
IMAGE_REL_BASED_LOW	2	The fixup adds the low 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the low half of a 32-bit word.
IMAGE_REL_BASED_HIGHLOW	3	The fixup applies the delta to the 32-bit field at Offset.
IMAGE_REL_BASED_HIGHADJ	4	The fixup adds the high 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the high value of a 32-bit word. The low 16 bits of the 32-bit value are stored in the 16-bit word that follows this base relocation. This means that this base relocation occupies two slots.
IMAGE_REL_BASED_MIPS_JMPADDR	5	Fixup applies to a MIPS jump instruction.
IMAGE_REL_BASED_SECTION	6	Reserved for future use
IMAGE_REL_BASED_REL32	7	Reserved for future use
IMAGE_REL_BASED_MIPS_JMPADDR16	9	Fixup applies to a MIPS16 jump instruction.
IMAGE_REL_BASED_DIR64	10	This fixup applies the delta to the 64-bit field at Offset
IMAGE_REL_BASED_HIGH3ADJ	11	The fixup adds the high 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the high value of a 48-bit word. The low 32 bits of the 48-bit value are stored in the 32-bit word that follows this base relocation. This means that this base relocation occupies three slots.

Seules les relocations de type *IMAGE_REL_BASED_ABSOLUTE* et *IMAGE_REL_BASED_HIGHLOW* concernent les architectures de type x86.

Types d'adresses

Explication des diffèrents types d'adresses

25/01/2006 yarocco

1. Présentation

Cette page est consacrée à l'explication des différents types d'adresses que l'on peut trouver dans les fichiers au format PE.

Comme je n'ai pas la science infuse, je ne peux pas affirmer que les définitions que je vais donner soit universelle et il se peut très bien que vous puissiez tomber au détour d'une page sur un des termes que je vais aborder avec une tout autre définition. je peux juste vous affirmer que les définitions que je vais vous donner sont valables pour les fichiers PE.

2. Définitions

- RVA (Relative Virtual Address) : La traduction littérale donne "Adresse Virtuelle Relative". Elle est utilisée pour désigner une adresse chargée en mémoire. En effet, lorsque qu'un programme est chargé en mémoire, les adresses qu'il utilise sont relative à une adresse de base (Voir l'entête du PE) et sont virtuelles. Il ne faut donc

pas se fier à ce genre d'adresse si on veut regarder une valeur ou en changer une avec un éditeur hexadécimal.
- VA (Virtual Address) : La traduction littérale donne "Adresse Virtuelle". Il s'agit presque d'une RVA, exceptée qu'elle n'est pas relative. Malgré le fait qu'une adresse soit toujours relative - car tout est relatif - on dira que celle-ci ne l'est pas car elle est relative à 0.
- Offset : Pour mes articles, ça sera une adresse « physique ». C'est-à-dire qu'elle est simplement relative au début du fichier.

3. Calculs

Nous allons maintenant nous intéresser à la manière de passer de l'un à l'autre de ces formats d'adresses.

1. Données exemples

```
ImageBase (Cardinal) : $00400000

*** Section n°1 ***
Name (8 bytes) : CODE
VirtualSize (Cardinal) : $00069158
VirtualAddress (Cardinal) : $00001000
SizeOfRawData (Cardinal) : $00069200
PtrToRawData (Cardinal) : $00000400

*** Section n°2 ***
Name (8 bytes) : DATA
VirtualSize (Cardinal) : $000013C8
VirtualAddress (Cardinal) : $0006B000
SizeOfRawData (Cardinal) : $00001400
PtrToRawData (Cardinal) : $00069600

*** Section n°3 ***
Name (8 bytes) : BSS
VirtualSize (Cardinal) : $00000BE9
VirtualAddress (Cardinal) : $0006D000
SizeOfRawData (Cardinal) : $00000000
PtrToRawData (Cardinal) : $0006AA00
```

2. RVA -> Offset

Les RVA sont relatives à l'ImageBase, c'est-à-dire qu'elles sont le plus souvent sous la forme $004xxxxx et supérieures à $00400000 (en supposant que $00400000 soit l'ImageBase).

Prenons par exemple l'adresse **$00402030**, la première opération pour avoir son équivalent physique (offset) est de soustraire l'ImageBase à l'adresse.

Dans notre cas, nous avons donc : $0046C030 - $00400000 = $6C030

Maintenant, il faut regarder à quelle section du fichier elle fait référence.

Nous partons donc de la première section (ici CODE), et nous regardons avec VirtualAddress et VirtualSize:

Est ce que **$6C030** est entre le début et la fin de la section ? <=> $1000 < $6C030 < $69258 ($1000 + $69158) ?

Ici, ce n'est pas la cas, nous passons donc à la seconde section : $6B000 < $6C030 < $6C3C8 ($6B000 + $13C8) ?

On trouve que c'est juste. Cela signifie que notre adresse appartient à la seconde section. Il ne nous reste plus maintenant qu'a faire de simples additions et soustractions.

Récapitulatif

	Notre adresse	La section
VA :	**$6C030**	**$6B000**
Offset :	CeQueNousCherchons	**$69600**

 Il ne faut pas faire de produit en croix, les adresses ne sont pas proportionnelles !!!

Il nous faut d'abord avoir la différence entre notre adresse et celle de la section : $6C030 - $6B000 = $1030

Nous avons maintenant une adresse relative au début de la section, si nous voulons une adresse relative au début du fichier, il nous suffit d'ajouter l'offset de la section, nous obtenons alors : $1030 + $69600 = $6A630

Voila, nous avons maintenant un offset absolu (enfin relatif au début du fichier).

En résumé : nous avons une adresse X, de type RVA

- Nous devons d'abord soustraire l'ImageBase : X <= X - ImageBase
- Nous recherchons à quelle section l'adresse appartient, c'est-à-dire que l'on regarder si : AdresseRVASection < X < (AdresseRVASection + TailleVirtuelle) jusqu'à qu'on trouve la bonne section
- Nous faisons la différence entre notre adresse et l'adresse RVA du début de la section trouvée : X <= X - AdresseRVASection
- Nous ajoutons notre adresse à l'offset de la section trouvée : X <= X + OffsetSection

Nous pouvons aussi faire la même chose avec les VA. Sachant que ces dernières sont des RVA sans ImageBase, il suffit alors de sauter la première étape.

En résumé : nous avons une adresse X, de type RVA

- Nous recherchons à quelle section l'adresse appartient, c'est-à-dire que l'on regarder si : AdresseRVASection < X < (AdresseRVASection + TailleVirtuelle) jusqu'à qu'on trouve la bonne section
- Nous faisons la différence entre notre adresse et l'adresse RVA du début de la section trouvée : X <= X - AdresseRVASection
- Nous ajoutons notre adresse à l'offset de la section trouvée : X <= X + OffsetSection

Nous avons donc vu comment passer d'une adresse RVA ou VA à son équivalent physique (offset).

3. Offset -> RVA

Pour avoir cette équivalence, il suffit presque de faire l'inverse ce ce que nous venons de voir juste au dessus, il y a juste le nom des champs qui changent.

Reprenons donc nos données de départ, et prenons par exemple l'adresse **$6A630**, nous pourrons ainsi vérifier que notre exemple du début était bon (à moins que je ne me trompe partout....).

Tout d'abord, nous devons faire les sections une à une et regarder si notre adresse appartient à l'une d'elle :

Nous partons donc de la première section (ici CODE), et nous regardons avec PtrToRawData et SizeOfRawData:

Est ce que **$6C030** est entre le début et la fin de la section ? <=> $400 < $6A630 < $69600 ($400 + $69200) ?

Ici, ce n'est pas la cas, nous passons donc à la seconde section : $69600 < $6A630< $6AA00 ($69600 + $1400) ?

On retrouve bien la seconde section. Nous avons donc :

Récapitulatif

	Notre adresse	La section
Offset:	**$6A630**	**$69600**
VA:	CeQueNousCherchons	**$6B000**

 Je le répète, il ne faut pas faire de produit en croix, les adresses ne

sont pas proportionnelles !!!

Nous faisons donc comme plus haut.

C'est-à-dire que nous allons faire la différence entre notre adresse et celle de la section trouvée : $6A630 - $69600 = $1030

Il nous faut encore ajouter la VA de la section à notre adresse : $1030 + $6B000 = $6C030.

Nous avons maintenant une adresse VA. Mais comme nous voulons avoir une RVA, il nous manque une opération.

Il suffit d'ajouter l'ImageBase à notre adresse pour obtenir la RVA : $6C030 + $400000 = $46C030 = $0046C030.

On retrouve bien notre adresse de départ.

En résumé : nous avons une adresse X, de type offset

- Nous recherchons à quelle section l'adresse appartient, c'est-à-dire que l'on regarde si : OffsetSection < X < (Offset + TailleRéelle) jusqu'à qu'on trouve la bonne section
- Nous faisons la différence entre notre adresse et l'offset du début de la section trouvée : X <= X - OffsetSection
- Nous ajoutons notre adresse à l'adresse RVA de la section trouvée : X <= X + AdresseRVA
- Nous ajoutons l'Imagebase si nous voulons une adresse qui pointe sur la mémoire : X <= X + ImageBase

Vous savez maintenant passer d'un offset à son équivalent RVA ou VA.

4. RVA -> VA -> RVA

Je pense que nous attaquons maintenant la partie la plus facile, d'ailleurs si vous avez lu l'article jusqu'ici vous avez peut-être même déjà compris.

En effet, pour passer d'une adresse RVA à son équivalent VA, il suffit de soustraire l'ImageBase. Donc : VA = RVA - ImageBase

Et par opposition pour passer d'une VA à son équivalent RVA : RVA = VA + ImageBase

Ce n'est pas plus compliqué.