Étude d’un Loader Tout en Mémoire pour Packer

par *Homeostasie (Nicolas.D)*

15/06/2012

(trashomeo (at) gmail (dot) com)

Tables des matières

1.Introduction....3
2.Un loader tout en mémoire....4
2.1.Principe de fonctionnement....4
2.2.Description comportementale....5
2.2.1.Recopie du loader en mémoire....5
2.2.2.Copie en mémoire de l’exécutable....8
3.Analyse du loader tout en mémoire 13
3.1.L’évolution des entêtes PE....13
3.2.L’évolution des modules chargés....14
3.3.L’évolution des ressources....15
4.Quelques mots avant de conclure....16
5.Conclusion....16
6.Références....17

Tables des illustrations

Figure 1: Diagramme de séquences – Comportement du loader....5
Figure 2: Diagramme de séquences - Recopie du loader en mémoire....8
Figure 3: Table des sections d'un exécutable....9
Figure 4: Diagramme de séquences - Chargement de l'exécutable en mémoire....11
Figure 5: Entête PE de MyPELoader (droite) et Netcat (gauche)....13
Figure 6: Entête PE de MyPELoader après chargement de Netcat en mémoire....13
Figure 7: Liste des DLLs avant chargement de Netcat....14
Figure 8: Liste des DLLs après chargement de Netcat....14
Figure 9: Liste des Handles avant exécution de Netcat....15
Figure 10: Liste des Handles après exécution de Netcat....15

1. Introduction

Pouvoir exécuter un programme sans être détecté par un antivirus est un des enjeux primordiaux des développeurs de logiciels malveillants. D'un autre côté, les pentesteurs apprécieront, après avoir obtenu un accès sur une machine, pouvoir utiliser des outils normalement détectés par les antivirus et ainsi faciliter la prise de contrôle et la recherche d’informations. Cet article étudie une conception d'un loader « tout en mémoire » pour packer du point de vue d’un développeur et non par rétro-ingénierie d’un exécutable comme cela est le cas pour l’étude d’un programme malveillant dont on n’a pas le source.

Je précise que cet article est écrit dans un but essentiellement éducatif et informatif et n'engage en aucun cas ma responsabilité quant à l'usage que vous en feriez.

2. Un loader tout en mémoire

2.1. Principe de fonctionnement

Ce loader « fait-maison » a pour but de charger dans son espace mémoire un exécutable et de lui donner la main. De ce fait, le loader Windows ne sera pas sollicité.

Pour arriver à ces fins, le loader devra se recopier lui-même dans un emplacement libre de sa mémoire virtuelle pour éviter tout éventuel écrasement avec le programme à charger. En effet, il est très envisageable que l’image de base soit identique au deux ou que les tailles de sections fassent que finalement l’une empiète sur l’autre.

Une fois la recopie du loader effectué, celui-ci réalise un saut sur le point d'entrée de son propre code mais au nouvel emplacement mémoire. Nous verrons plus tard que le loader devra au préalable s’auto-patcher pour corriger les instructions assembleurs faisant appel à un adressage absolu.

Ensuite, le loader nouvellement recopié effectuera le chargement de l'exécutable souhaité. Pour réaliser ceci, il devra:

- Parcourir les différents champs du fichier PE à charger pour allouer les différentes sections
- Mapper les DLLs requises
- Construire l’IAT (« Import Address Table »)
- Fournir finalement l’exécution au programme injecté dans sa propre mémoire

Afin de résumer le comportement général de ce loader, ci-dessous un diagramme de séquences :

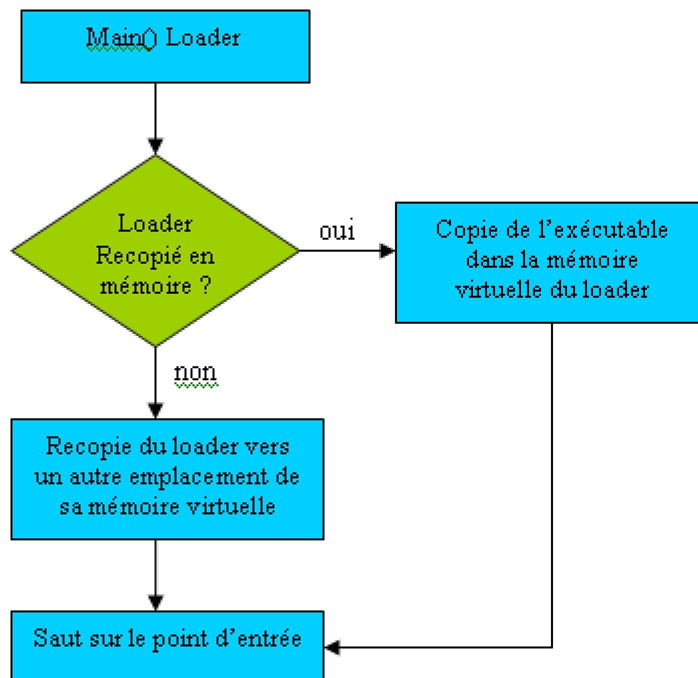

Figure 1: Diagramme de séquences – Comportement du loader

2.2. Description comportementale

2.2.1. Recopie du loader en mémoire

Avant de charger le programme souhaité dans son espace mémoire, le loader devra être capable de se recopier lui-même dans un autre espace de sa propre mémoire virtuelle et de déterminer si cette action a déjà été effectuée. Pour effectuer son chargement, le loader viendra lire son propre fichier exécutable pour récupérer les valeurs contenues dans les différents champs PE [1] comme la base de l'image, la base et la taille des sections « code » et « data », le point d'entrée etc...

Afin de vérifier si sa propre recopie est effective, le loader s'appuie sur l'utilisation d'un objet nommé, ici l'utilisation d'une mutuelle exclusion (mutex) [2]. D'autres moyens plus ou moins discrets auraient pu être utilisé.

Dans un premier temps, le loader tente d'ouvrir la mutex nommée. Si l'appel échoue, cela signifie que la mutex n'a pas encore été crée et que le loader ne s'est donc pas recopié.

Un exemple de code d'utilisation est présenté ci-dessous:

```
HANDLE hMutex = NULL;

hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, "MutexFor1337");
if(hMutex == NULL)
{
    /* First execution, so let's create a named mutex */
    hMutex = CreateMutex(NULL, TRUE, "MutexFor1337");

    /* Copy of the loader itself into virtual memory and jump on */
    …
}
…
/* Mutex is already created, so we can load the exe */
…
```

Ainsi, une fois que le loader se sera recopié et qu'il se sera exécuté à son nouvel emplacement, la mutex étant dorénavant créée, la partie du code nécessaire au chargement du binaire souhaité s'effectuera.

La figure 4 à la page suivante illustre le diagramme de séquences de recopie du loader.

On constate sur cette figure, un bloc du diagramme mis volontairement en évidence en orange. En effet, une fois que le loader s'est chargé à partir de l'image sur le disque dans un nouvel emplacement de son espace de mémoire virtuelle, il lui sera nécessaire de s'auto-patcher pour corriger toutes les instructions faisant appel à une adresse absolue.

Par exemple l'instruction suivante appelle la fonction contenue à l'adresse 0x0040E204:

```
0040128D   FF15 04E24000  CALL DWORD PTR DS:[40E204]
```

En supposant que le loader se recopie à l'adresse 0x01000000 et sans s'auto-patcher, l'instruction précédente deviendra :

```
0100128D   FF15 04E24000    CALL DWORD PTR DS:[40E204]
```

Et en s'auto-patchant, l'instruction obtenue sera :

```
0100128D   FF15 01E20004    CALL DWORD PTR DS:[100E204]
```

Si cela n'était pas fait, le risque serait d'appeler le code du loader original alors qu'il a été supprimé, voire remplacé par l'exécutable à charger en mémoire. Vous vous doutez bien du résultat catastrophique.

A titre d'illustration, voici un extrait de code pour corriger certaines instructions :

```
char ModifyCodeMemory(DWORD *pAddress, DWORD dwNewValue)
{
   char bRet;
   DWORD dwOldProtect;

   bRet = VirtualProtect(pAddress, sizeof(DWORD), PAGE_EXECUTE_READWRITE, &dwOldProtect);
   if(bRet != 0)
   {
      /* Modify the call address */
      *pAddress = dwNewValue;

      /* Restore orignal protection */
      VirtualProtect(pAddress, sizeof(DWORD), dwOldProtect, NULL);
   }

   return bRet;
}


char PatchInstructionWithDirectAddress(LPVOID ptrLoc)
{
   …
   if( (*pStart == 0xFF && *(pStart+1) == 0x15) || (*pStart == 0xFF && *(pStart+1) == 0x25) ||
        (*pStart == 0x83 && *(pStart+1) == 0x0D) ||(*pStart == 0x81 && *(pStart+1) == 0x0D) ||
        (*pStart == 0x89 && *(pStart+1) == 0x0D) ||(*pStart == 0x83 && *(pStart+1) == 0x3D) ||
        (*pStart == 0x8B && *(pStart+1) == 0x3D) || (*pStart == 0x89 && *(pStart+1) == 0x1D) )
   {
      /* CALL DWORD PTR DS:[XXXXXXXX]     => FF15 XXXXXXXX */
      /* JMP DWORD PTR DS:[XXXXXXXX]      => FF25 XXXXXXXX */
     /* OR DWORD PTR DS:[XXXXXXXX],1      => 830D XXXXXXXX 01 */
     /* OR DWORD PTR DS:[XXXXXXXX],80     => 810D XXXXXXXX 80 */
     /* MOV DWORD PTR DS:[XXXXXXXX],ECX  => 890D XXXXXXXX 80 */
     /* CMP DWORD PTR DS:[XXXXXXXX],0     => 833D XXXXXXXX 00 */
     /* MOV EDI,DWORD PTR DS:[XXXXXXXX]=> 8B3D XXXXXXXX*/
     /* … */

      pAddress = (DWORD*)(pStart+2);
      dwNewAddress = ((DWORD)*pAddress - (DWORD)dwOriginaBaseAddr) + (DWORD)ptrNewLoc;

      pStart += 5;
      ModifyCodeMemory(pAddress, dwNewAddress);
   }
…
}
```

Je passerais l'explication sur les autres blocs du diagramme, ceux-ci seront en partie expliqués dans la section suivante «*Copie en mémoire de l'exécutable* ».

Finalement il ne restera plus qu'à faire un saut sur le point d'entrée de notre loader nouvellement chargé, ceci en ajoutant à notre nouvelle adresse de base (par exemple 0x01000000), la RVA (Relative Virtual Address) de l'EP (Entry Point).

Ci-dessous, le diagramme de séquences de recopie du loader :

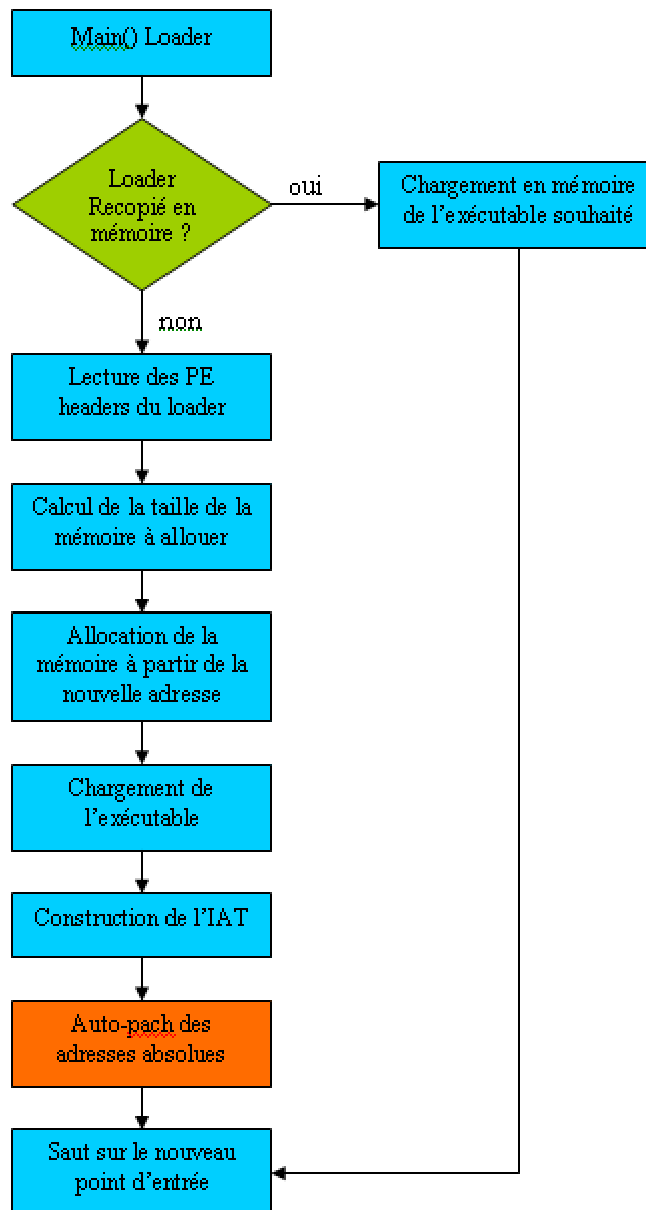

Figure 2: Diagramme de séquences - Recopie du loader en mémoire

2.2.2. Copie en mémoire de l'exécutable

Une fois que le loader s'est dupliqué dans son espace mémoire virtuelle, il sera libre de charger l'exécutable à son image de base par défaut.

Pour cela, dans un premier temps il va effectuer une lecture du « PE Header » pour vérifier la validité du champ « e_magic » de l'entête DOS ainsi que le champ « signature » de l'entête NT.

Ensuite, il est indispensable de déterminer la taille à allouer pour charger ce nouvel exécutable à un emplacement de sa mémoire virtuelle. Il faudra tenir compte du fait que la taille du binaire en mémoire ne corresponds pas à celle stockée sur le disque dur. En effet, chaque section doit être alignée sur la valeur contenue dans le champ « SectionAlignment » de l' « Optional Header » de l'entête NT. Sur un environnement 32 bits, cette valeur est fixé à 0x1000 soit des pages de 4Ko. Pour observer ceci, nous pouvons visualiser sur la figure 3 la taille réelle (« Raw Size ») et la taille virtuelle (« Virtual Size ») des différentes sections grâce à l'outil LordPE [3].

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	000057E4	00000400	00005800	60500060
.data	00007000	00000060	00005C00	00000200	C0300040
.rdata	00008000	00000628	00005E00	00000800	40300040
.bss	00009000	00004120	00000000	00000000	C0300080
.idata	0000E000	00000748	00006600	00000800	C0300040

Figure 3: Table des sections d'un exécutable

Pour comprendre en partie le calcul de la taille pour l'ensemble des sections, le code pourrait avoir la forme suivante :

```
…
for(int i = 0; i < numSections; i++)
{
   if(SecHdr[i].virtualSize)
   {
       if(SecHdr[i].virtualSize % alignment == 0)
       {
          totalSize += SecHdr[i].virtualSize;
       }
       else
       {
           int val = SecHdr[i].virtualSize / alignment;
           val++;
           totalSize += (val * alignment);
        }
   }
}
```

Une fois la taille finale connue, il sera nécessaire d'allouer cet espace en mémoire virtuelle avec pour adresse de départ, l'adresse de base de l'exécutable à charger. L'API Windows utilisée est tout simplement « VirtualAlloc() » [4] dont le prototype est le suivant :

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

Nous connaissons donc la valeur des paramètres « lpAddress » et « dwSize », il reste donc à spécifier « flAllocationType » à « MEM_RESERVE|MEM_COMMIT » pour réserver et engager la mémoire, et à fixer les droits des pages à « PAGE_EXECUTE_READWRITE » grâce au paramètre « flProtect ».

Toutefois cet appel à « VirtualAlloc() » a potentiellement des chances d'échouer. En effet, l'espace mémoire peut-être déjà réservé, voire même engagé, notamment par le loader original. N'oublions pas que ce dernier a été recopié mais les ressources mémoires allouées n'ont pas été libérées. De ce fait, il pourra être nécessaire de libérer les différents espaces mémoires précédemment utilisés.

Pour illustrer une méthode, je vous fournis un extrait du code permettant de libérer une région en fonction de son type d'allocation:

```
VirtualQuery(pAddrRegion, &MemBasicInfo, sizeof(MemBasicInfo));
if(MemBasicInfo.Type == MEM_MAPPED || MemBasicInfo.Type == MEM_IMAGE)
{
   if(UnmapViewOfFile(pAddrRegion) != 0)
   {
      VirtualQuery(pAddrRegion, &MemBasicInfo, sizeof(MemBasicInfo));
   }
}

if(MemBasicInfo.State == MEM_COMMIT)
{
   VirtualFree((void*)MemBasicInfo.BaseAddress, MemBasicInfo.RegionSize, MEM_DECOMMIT);
}
if(MemBasicInfo.State == MEM_RESERVE)
{
   VirtualFree((void*)MemBasicInfo.BaseAddress, 0, MEM_RELEASE);
}
```

Après avoir libéré et alloué l'espace mémoire nécessaire pour accueillir l'exécutable, l'étape suivante consiste à le charger en mémoire. Pour cela, chaque section incluse dans l'entête PE est parcourue afin de récupérer l'adresse de début et la taille des données brutes (« Raw Size ») à copier en mémoire virtuelle. Il faudra prendre soin à la gestion de l'alignement durant cette phase.

Une fois que l'exécutable est correctement copié, notre loader embarqué doit résoudre sa table d'importation. Il existe sur la toile des exemples de code effectuant ce travail. De ce fait, je ne détaillerais pas cette partie mais je préciserais que certaines fois, il sera nécessaire d'effectuer une importation de fonction exportée par ordinal et non par nom comme on peut souvent le constater.

Ci-dessous le diagramme de séquences de copie de l'exécutable en mémoire à partir de notre loader nouvellement recopié, un comportement quasi-identique à la recopie du loader :

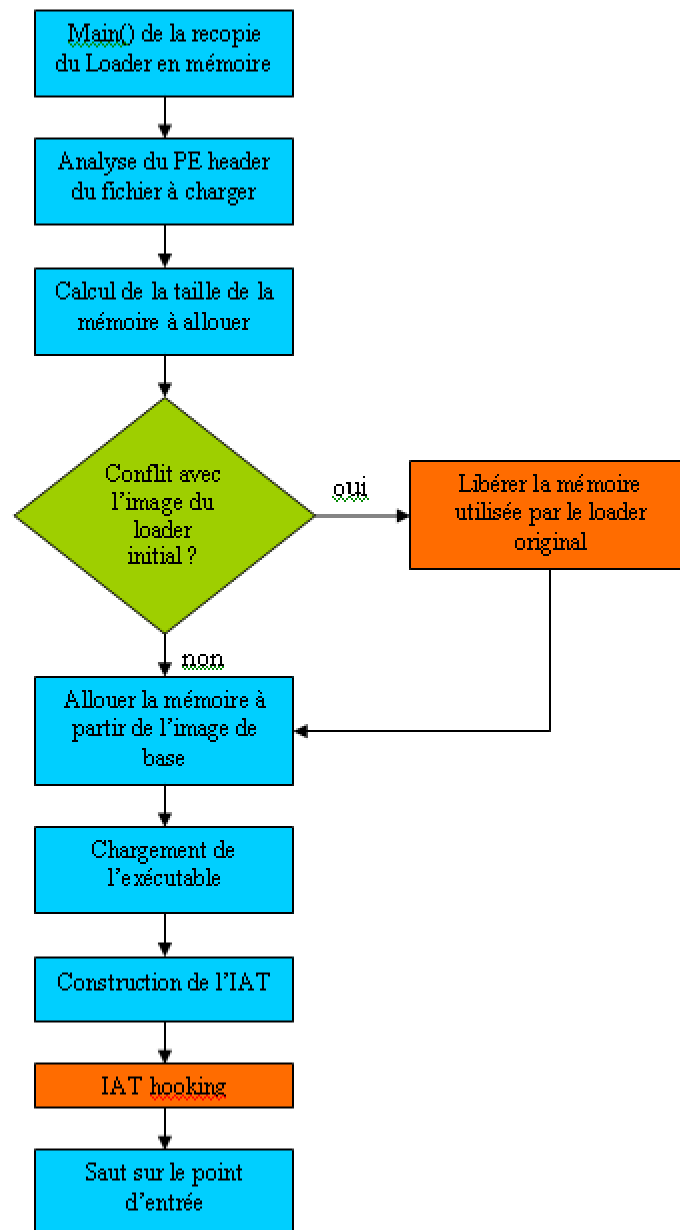

Figure 4: Diagramme de séquences - Chargement de l'exécutable en mémoire

A la vue de ce diagramme, on peut constater que je n'ai pas discuté du bloc « IAT hooking ». Il s'avère parfois nécessaire de hooker les APIs du nouvel exécutable car certaines valeurs sont stockées dans des sections spécifiques au chargement de notre loader « fait-maison » par le loader Windows.

C'est notamment le cas pour les APIs « GetCommandLineA » et « GetModuleHandleA() » [5] dont le code assembleur de cette dernière ressemblera partiellement à cela:

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOV EAX,DWORD PTR DS:[EAX+8]
POP EBP
RETN 4
```

Ce code charge tout d'abord dans le registre EAX l'adresse linéaire du TIB (Thread Information Block). Puis vient récupérer à l'offset 0x30 l'adresse du PEB. Finalement, à l'offset 0x08 de la structure du PEB, on retrouve l' « Image Base Address ».

Ci-dessus, un extrait de la structure du PEB :

```
typedef struct _PEB {

  BOOLEAN           InheritedAddressSpace;
  BOOLEAN           ReadImageFileExecOptions;
  BOOLEAN           BeingDebugged;
  BOOLEAN           Spare;
  HANDLE            Mutant;
  PVOID             ImageBaseAddress;
  PPEB_LDR_DATA     LoaderData;
  …
}
```

Or le segments FS pointe sur le bloc de donnée privée du thread principal et se situe sur ma plateforme à l'adresse 0x7FFDF000.

Lorsque le programme nouvellement chargé par notre loader fera appel à GetModuleHandle() alors le handle initial du programme originellement exécuté sera retourné, c'est-à-dire celui de notre loader et non de l'exécutable chargé. Évidemment, si l'adresse de base est identique pour les deux programmes, il n'y aura pas de soucis mais dans le cas contraire, préparez vous à un plantage!

J'ai opté personnellement pour l'utilisation du hooking d'IAT mais il aurait pu être aussi envisageable de modifier directement la valeur dans la zone pointée par le segment FS.

Finalement pour donner la main à ce nouvel exécutable chargé en mémoire, il suffit simplement de faire un saut sur son point d'entrée, comme il a été précédemment fait avec le loader nouvellement chargé.

3. Analyse du loader tout en mémoire

3.1. L’évolution des entêtes PE

Cette partie a pour but de se projeter à la place d’une personne pratiquant une première analyse de programmes malveillants. On s’attardera sur la vérification de caractéristiques durant l'exécution de l'exécutable, notamment l’évolution du mapping mémoire durant les différentes phases d’exécution ainsi que les différentes ressources utilisées.

Dans le cadre de cette étude, le loader se nommera tout simplement « MyPEloader.exe » et le fichier à charger en mémoire sera le logiciel « Netcat » [6] nommé « nc.exe ».

Tout d’abord, faisons une analyse des entêtes PE de notre loader, du logiciel Netcat et du résultat après avoir chargé « nc.exe » dans l’espace mémoire virtuelle de « MyPEloader.exe ». Pour cela, utilisons l’outil bien connu « LordPE » [3].

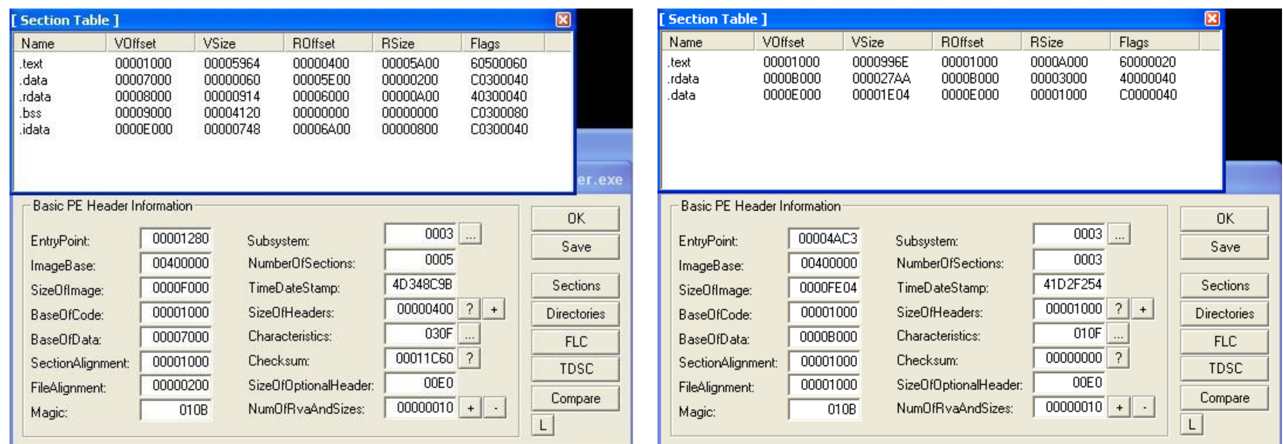

Figure 5: Entête PE de MyPELoader (droite) et Netcat (gauche)

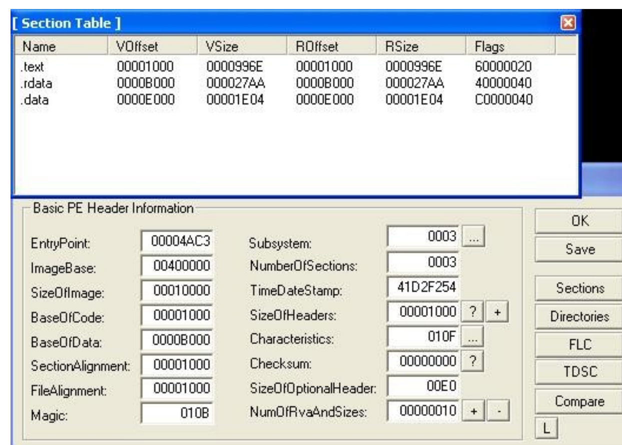

Figure 6: Entête PE de MyPELoader après chargement de Netcat en mémoire

Côté suspicion, rien d'anormal pour ces deux exécutables, leurs points d'entrée respectifs et les flags de section sont corrects.

Nous constatons que « MyPELoader.exe » présentent deux sections de plus, les points d'entrée sont différents ainsi que la taille de l'image, la base de la section « data » et la valeur d'alignement du fichier.

Maintenant analysons le résultat sur la Figure 6 du chargement de « nc.exe » dans l'espace virtuelle de notre loader. L'outil « LordPE » montre que notre loader a pris toutes les caractéristiques de « Netcat » excepté la taille de l'image qui ne corresponds ni à celle de notre loader ni à « Netcat ». J'avoue ne pas avoir cherché à comprendre exactement l'origine mais cela est probablement dû à l'algorithme de calcul de la taille virtuelle du programme à charger.

3.2. L'évolution des modules chargés

Vérifions maintenant ce qu'il en est au niveau des DLLs chargées avant et après que le programme Netcat soit injecté dans la mémoire de « MyPELoader ». Je précise qu'à ce moment, le flot d'exécution n'est pas encore redirigé vers le programme « Netcat ».

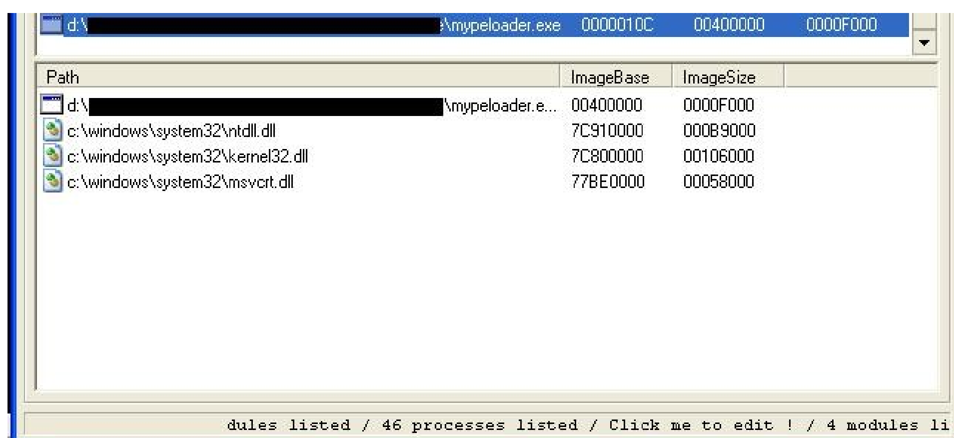

Figure 7: Liste des DLLs avant chargement de Netcat

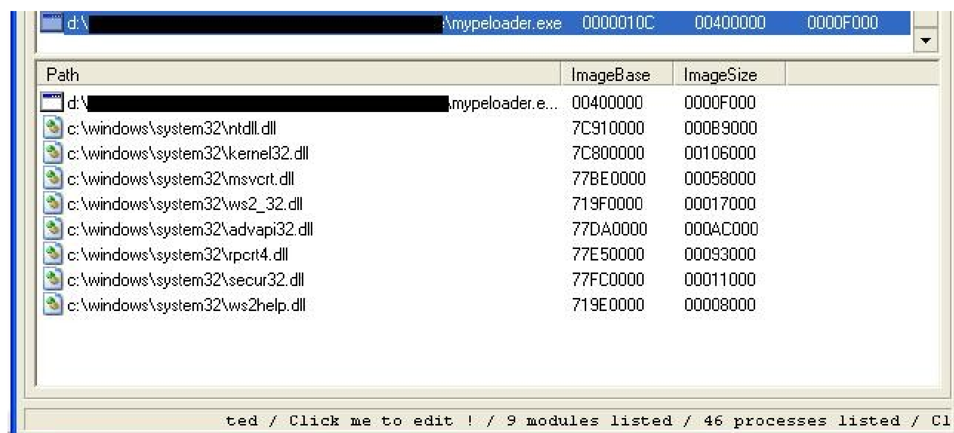

Figure 8: Liste des DLLs après chargement de Netcat

La différence est flagrante, notre loader a bien fait son travail en chargeant les DLLs nécessaires pour le bon fonctionnement du logiciel Netcat.

3.3. L'évolution des ressources

Qu'en est-il maintenant de l'évolution des différents « handles » ouvert par « MyPEloader.exe »? Pour cela, utilisons le logiciel « Process Explorer » [7] :

cmd.exe	2560	Interpréteur de commandes Windows	30 460 K
MyPELoader.exe	3712		7 424 K

Type	Name
Directory	\Windows
Directory	\KnownDlls
File	D:\Programmes\[redacted]\Release
KeyedEvent	\KernelObjects\CritSecOutOfMemoryEvent
WindowStation	\Windows\WindowStations\WinSta0

Figure 9: Liste des Handles avant exécution de Netcat

cmd.exe	2560	Interpréteur de commandes Windows	30 460 K
MyPELoader.exe	3712		9 276 K

Type	Name
Directory	\BaseNamedObjects
Directory	\Windows
Directory	\KnownDlls
File	D:\Programmes\[redacted]\Release
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_Catalog5
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\Protocol_Catalog9
Key	HKLM
KeyedEvent	\KernelObjects\CritSecOutOfMemoryEvent
Mutant	\BaseNamedObjects\MutexFor1337
Mutant	\BaseNamedObjects\MutexFor1337
Thread	MyPELoader.exe(3712): 2384
WindowStation	\Windows\WindowStations\WinSta0

Figure 10: Liste des Handles après exécution de Netcat

Comme expliqué dans le précédent chapitre, le loader s'appuie sur une mutex nommée pour déterminer si celui-ci s'est déjà recopié ou non en mémoire. Comme prévu, on voit bien apparaître l'objet nommé « MutexFor1337 ». Il aurait été évidemment possible de fermer ce « handle » avant de donner la main au programme « Netcat ».

Un autre aspect qui ressort est l'augmentation de la taille virtuelle utilisée par « MyPELoader.exe » qui augmente de quasiment de 2000 Ko.

4. Quelques mots avant de conclure

Cette méthode que j'ai développé et dont je partage la conception reste à être optimisée et est toutefois contraignante à implémenter pour plusieurs raisons.

Premièrement, en fonction de l'exécutable final à charger, il sera éventuellement nécessaire de hooker d'autres APIs. L'investigation pour connaître la cause du plantage requiert du temps et des connaissances en rétro-ingénierie.

De même, il sera fortement probable qu'ils soient nécessaires d'ajouter le patching d'instructions d'adressage absolu non pris en compte pour un précédent binaire.

Concernant l'exécutable à charger, celui-ci peut-être sous forme de fichier texte chiffré ou préalablement embarqué et chiffré sous forme de ressources dans notre loader.

La force de réaliser son propre loader, c'est d'augmenter l'aspect non détectable. Ce dernier est en effet personnalisable à souhait, comme la modification des entêtes PE du binaire chargé, et permet donc d'utiliser un binaire se présentant sous la forme de son choix indépendamment du loader Windows.

5. Conclusion

Nous avons pu étudier une méthode parmi d’autres de concevoir un loader « tout en mémoire ». Loin de là l’idée d’inciter à en développer un pour du profit, mais cela permet d‘appréhender le mécanisme du point vue d’un développeur, ce qui pourrait être d’une aide non négligeable durant une phase de rétro-ingénierie de programmes malveillants.

Finalement, ne perdons toujours pas de vue qu’il ne faut pas se fier à un antivirus, même si celui-ci est une protection nécessaire, le facteur humain déterminera le plus souvent la sécurité de son système en faisant le choix ou non d’exécuter un binaire provenant d’une source ou d'un moyen douteux.

Pour d’éventuelles questions, remarques, ou précisions sur le contenu de ce document, vous pouvez me contacter à l’adresse «trashomeo (at) gmail (dot) com».

6. Références

[1] Le PE Header http://assembly.ifrance.com/pehader.htm#Caract%C3%A9ristiques
[2] CreateMutex function http://msdn.microsoft.com/en-us/library/ms682411%28v=vs.85%29.aspx
[3] LordPE tool http://www.woodmann.com/collaborative/tools/index.php/LordPE
[4] VirtualAlloc function http://msdn.microsoft.com/en-us/library/aa366887%28v=vs.85%29.aspx
[5] GetModuleHandle function http://msdn.microsoft.com/en-us/library/ms683199%28v=vs.85%29.aspx
[6] The GNU Netcat project http://netcat.sourceforge.net/
[7] Process Explorer Tool http://technet.microsoft.com/fr-fr/sysinternals/bb896653