Make your own packer

By Lilxam.

* PART I *

SOMMAIRE

1.Introduction.

2.Structure du projet.

3.Notions essentielles.

4.Manipulations préliminaires de l'exécutable, un premier package.

A. Ajout d'une section.
B. Écriture du loader.
C. Intégration du loader au sein de l'exécutable.
D. Redirection de l'Entry Point (EP).

5.Chiffrement du code.

1.Introduction

Vous savez surement tous ce qu'est un packer, il en existe déjà beaucoup, allant des plus simple comme UPX ou FSG à des plus complexes comme Themida, armadillo et bien d'autres encore.

Le but d'un packer est avant tout de compresser un exécutable (ce que font UPX et FSG) mais aussi de protéger celui-ci d'une éventuelle personne mal-intentionnée (ou non) voulant cracker votre programme.

Quand à la question du pourquoi réaliser mon propre packer alors qu'il en existe beaucoup d'autres et surement plus évolués que le miens, et bien je répondrai tout simplement que ce projet est très intéressant du fait qu'il nécessite une bonne connaissance des mécanismes mis en place par le système pour exécuter une tâche. Et ainsi j'ai apprit énormément de choses en le réalisant. Mon but est alors de vous en faire partager.

2. Structure de notre projet

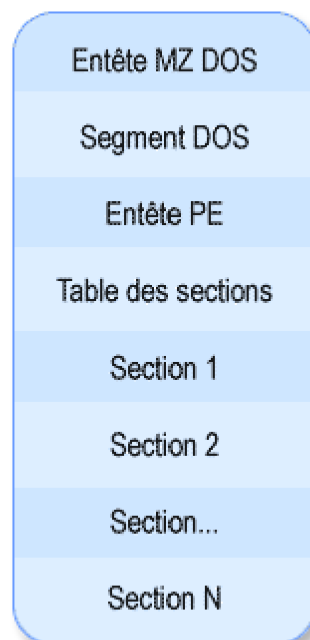
Bon alors avant de commencer je vais d'abord vous expliquer comment je compte m'y prendre pour réaliser ce packer. Je ferai donc un programme qui utilisera une librairie LP_LIB.dll, librairie qui sera en fait le cœur de notre projet. En fait j'ai appelé ce packer "Lilxam's Packer" :/. Rien de bien original mais bon... ^^

3. Notions Essentielles

Ce projet requiert quelques connaissances tout d'abord, et surtout, concernant le format PE.

Nous allons rapidement voir comment est constitué l'entête d'un exécutable et qu'elles sont les structures qui les composent. Ce sera très bref, car il existe énormément de doc sur le sujet.

Voici un petit schéma :



Nous voyons donc en premier l'**entête MZ DOS**. Cette structure contient notamment la signature 'MZ'. Voyons sa définition :

```
typedef struct _IMAGE_DOS_HEADER {
        WORD e_magic;
        WORD e_cblp;
        WORD e_cp;
        WORD e_crlc;
        WORD e_cparhdr;
        WORD e_minalloc;
        WORD e_maxalloc;
        WORD e_ss;
        WORD e_sp;
        WORD e_csum;
        WORD e_ip;
        WORD e_cs;
        WORD e_lfarlc;
        WORD e_ovno;
        WORD e_res[4];
        WORD e_oemid;
        WORD e_oeminfo;
        WORD e_res2[10];
        LONG e_lfanew;
} IMAGE_DOS_HEADER,*PIMAGE_DOS_HEADER;
```

Le champ **e_magic** correspond à la signature 'MZ', signature que tout exécutable, Dll ou driver doit avoir, et **e_lfanew** pointe vers l'entête PE.
Pour parcourir l'entête de l'exécutable j'ai codés plusieurs fonctions. Voici celle pour récupérer l'entête DOS :

```
PIMAGE_DOS_HEADER GetDOSHeader(HANDLE hBinary)
{
   PIMAGE_DOS_HEADER pDOSHeader = NULL;

   pDOSHeader = (PIMAGE_DOS_HEADER) hBinary; //Récupération de l'entête DOS

   return pDOSHeader;
}
```

Ensuite vient le **segment DOS**. C'est simplement un sous-programme qui est appelé en cas d'échec du *PE LOADER* et qui affiche un message du style «This program cannot be run in DOS mode. ». Rien de bien important pour nous.

Intéressons nous plutôt à l'**entête PE** proprement dite. Il s'agit en fait de la structure **IMAGE_NT_HEADERS** :

```
typedef struct _IMAGE_NT_HEADERS{
 DWORD           Signature;
 IMAGE_FILE_HEADER    FileHeader;
 IMAGE_OPTIONAL_HEADER OptionalHeader;
}IMAGE_NT_HEADERS,*PIMAGE_NT_HEADERS;
```

Le premier champ est la signature certifiant que le fichier en question est bien une image PE (soit un exécutable). Cette signature est 'PE\0\0' (soit 'PE' suivit de deux octets nuls).
La fonction pour obtenir cette structure :

```
PIMAGE_NT_HEADERS GetPEHeader(HANDLE hBinary)
{
    PIMAGE_DOS_HEADER pDOSHeader = NULL;
    PIMAGE_NT_HEADERS pPEHeader = NULL;

    pDOSHeader = GetDOSHeader(hBinary);
    pPEHeader = (PIMAGE_NT_HEADERS) ((PUCHAR)pDOSHeader +
pDOSHeader->e_lfanew);//Récupération de l'entête PE

    return pPEHeader;
}
```

Le second champ est en fait une structure de type **IMAGE_FILE_HEADER** :

```
typedef struct _IMAGE_FILE_HEADER {
 WORD  Machine;
 WORD  NumberOfSections;
 DWORD TimeDateStamp;
 DWORD PointerToSymbolTable;
 DWORD NumberOfSymbols;
 WORD  SizeOfOptionalHeader;
 WORD  Characteristics;
}IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Il s'agit en fait de l'**entête COFF** (Common Object File Format). Soit l'entête commune aux objets sous windows.

On ne considèrera que NumberOfSections correspondant au nombre de sections que comporte notre exécutable. Nous reviendrons sur les sections plus tard.

Pour avoir l'entête COFF :

```
PIMAGE_FILE_HEADER GetCOFFHeader(HANDLE hBinary)
{
    PIMAGE_NT_HEADERS pPEHeader = NULL;
    PIMAGE_FILE_HEADER pCOFFHeader = NULL;

    pPEHeader = GetPEHeader(hBinary);

    pCOFFHeader = (PIMAGE_FILE_HEADER)&pPEHeader-
>FileHeader;//Récupération de l'entête COFF

    return pCOFFHeader;
}
```

Toujours dans l'entête PE se trouve la structure **IMAGE_OPTIONAL_HEADER** :

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD Magic;
    BYTE MajorLinkerVersion;
    BYTE MinorLinkerVersion;
    DWORD SizeOfCode;
    DWORD SizeOfInitializedData;
    DWORD SizeOfUninitializedData;
    DWORD AddressOfEntryPoint;
    DWORD BaseOfCode;
    DWORD BaseOfData;
    DWORD ImageBase;
    DWORD SectionAlignment;
    DWORD FileAlignment;
    WORD MajorOperatingSystemVersion;
    WORD MinorOperatingSystemVersion;
    WORD MajorImageVersion;
    WORD MinorImageVersion;
    WORD MajorSubsystemVersion;
    WORD MinorSubsystemVersion;
    DWORD Reserved1;
    DWORD SizeOfImage;
    DWORD SizeOfHeaders;
    DWORD CheckSum;
    WORD Subsystem;
    WORD DllCharacteristics;
    DWORD SizeOfStackReserve;
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;
    DWORD SizeOfHeapCommit;
```

```
    DWORD LoaderFlags;
    DWORD NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER,*PIMAGE_OPTIONAL_HEADER;
```

Je ne vais pas m'attarder sur cette structure, j'expliquerai au fur et à mesure à quoi correspondent les champs que j'utiliserai.
Ma fonction pour l'obtenir :

```
PIMAGE_OPTIONAL_HEADER GetOptionalHeader(HANDLE hBinary)
{
    PIMAGE_NT_HEADERS pPEHeader = NULL;
    PIMAGE_OPTIONAL_HEADER pOptionalHeader = NULL;

    pPEHeader = GetPEHeader(hBinary);

    pOptionalHeader = (PIMAGE_OPTIONAL_HEADER)&pPEHeader-
>OptionalHeader;//Récupération de l'entête Optional

    return pOptionalHeader;
}
```

Après l'entête PE suit la **table des sections**, ou plutôt un tableau de structures **PIMAGE_SECTION_HEADER** :

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

J'expliquerai par la suite cette structure quand je vous montrerez comment ajouter une section à un programme.

Je récupère cette table ainsi :

```
PIMAGE_SECTION_HEADER GetSectionHeader(HANDLE hBinary)
{
    PIMAGE_NT_HEADERS pPEHeader = NULL;
    PIMAGE_SECTION_HEADER pSectionHeader = NULL;

    pPEHeader = GetPEHeader(hBinary);
    pSectionHeader =
(PIMAGE_SECTION_HEADER)IMAGE_FIRST_SECTION(pPEHeader);

    return pSectionHeader;
}
```

Et voici une fonction permettant de lister les sections :

```
VOID ListSections(HANDLE hBinary)
{
   PIMAGE_SECTION_HEADER pSectionHeader = NULL;
   PIMAGE_FILE_HEADER pCOFFHeader = NULL;
   size_t i;

   pSectionHeader = (PIMAGE_SECTION_HEADER)GetSectionHeader(hBinary);
   pCOFFHeader = (PIMAGE_FILE_HEADER)GetCOFFHeader(hBinary);

   printf("\n[+]Sections");

   for(i = 0; i < pCOFFHeader->NumberOfSections; i++)
   {
       printf("\n    %s", pSectionHeader[i].Name);
       //printf("\n        VirtualSize : 0x%x", pSectionHeader[i].Misc.VirtualSize);
       //printf("\n        Virtual Address: 0x%x", pSectionHeader[i].VirtualAddress);
       //printf("\n        SizeOfRawData : 0x%x", pSectionHeader[i].SizeOfRawData);
       //printf("\n        PointerToRawData : 0x%x", pSectionHeader[i].PointerToRawData);
       //printf("\n        PointerToRelocations : 0x%x",
pSectionHeader[i].PointerToRelocations);
       //printf("\n        PointerToLinenumbers : 0x%x",
pSectionHeader[i].PointerToLinenumbers);
       //printf("\n        NumberOfRelocations : 0x%x",
pSectionHeader[i].NumberOfRelocations);
       //printf("\n        NumberOfLinenumbers : 0x%x",
pSectionHeader[i].NumberOfLinenumbers);
       //printf("\n        Attributes : 0x%x", pSectionHeader[i].Characteristics);

   }
}
```

Bien maintenant passons aux choses sérieuses.

4. Manipulations préliminaires de l'exécutable, un premier package.

L'idée va être d'ajouter une section au programme dans laquelle on écrira notre loader puis de rediriger le point d'entrée du programme vers notre section afin d'exécuter le loader.

A. Ajout d'une section.

Nous allons voir ici comment ajouter une section à notre programme sans pour autant perturber sa stabilité.

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

Voici un récapitulatif des champs qui vont nous intéresser :

-**Name** : le nom de la section, il n'a pas vraiment d'utilité, il ne doit pas dépasser 8 octets(**IMAGE_SIZEOF_SHORT_NAME** = 8).
-**VitualSize** : la taille de la section arrondie au prochain multiple de **SectionAlignment** de la structure **IMAGE_OPTIONNAL_HEADER**.
-**SizeOfRawData** : la taille de la section arrondie au prochain multiple de **FileAlignment** de la structure **IMAGE_OPTIONNAL_HEADER**.
-**PointerToRawData** : l'offset auquel débute la section.

-**Characteristics** : ce champ contient les drapeaux qui caractérisent la section. Les drapeux que nous allons utiliser sont :
***IMAGE_SCN_MEM_EXECUTE** : la section est exécutable.
***IMAGE_SCN_MEM_READ** : la section est lisible.
***IMAGE_SCN_MEM_WRITE** : la section est éditable.
***IMAGE_SCN_CNT_CODE** : la section contient le ou une partie du code du programme.

Pour ajouter notre section il nous faudra donc renseigner ces champs. Mais pas seulement ceux-ci, en effet il nous faut également incrémenter le champ **NumberOfSections** de la structure **IMAGE_FILE_HEADER** ainsi que le champ **SizeOfImage**, de la structure **IMAGE_OPTIONAL_HEADER**, de la taille de notre section.

Sans plus attendre je vous donne la fonction me permettant de faire ceci :

```
VOID AddLPSection(HANDLE hBinary)
{
   PIMAGE_SECTION_HEADER pNewSectionHeader = NULL;
   PIMAGE_SECTION_HEADER pSectionHeader = NULL;
   PIMAGE_FILE_HEADER pCOFFHeader = NULL;
   PIMAGE_OPTIONAL_HEADER pOptionalHeader = NULL;

   DWORD dwSectionSize, dwSectionAlignment, dwFileAlignment;

   pCOFFHeader = GetCOFFHeader(hBinary);
   pSectionHeader = GetSectionHeader(hBinary);
   pOptionalHeader = GetOptionalHeader(hBinary);

   dwSectionSize = sizeof(loader)+sizeof(DWORD)+1;
   dwSectionAlignment = pOptionalHeader->SectionAlignment;
   dwFileAlignment = pOptionalHeader->FileAlignment;


   pNewSectionHeader = (PIMAGE_SECTION_HEADER) ((PUCHAR)
(&pSectionHeader[pCOFFHeader->NumberOfSections-1].Characteristics) + 0x4);

   printf("\n[+]Adding section .lilxam at address0x%x...", pNewSectionHeader);

   memcpy(*(&pNewSectionHeader->Name), ".lilxam", 7);

   *(&pNewSectionHeader->VirtualAddress) =
GetAlignment(pSectionHeader[pCOFFHeader->NumberOfSections-1].VirtualAddress+
pSectionHeader[pCOFFHeader->NumberOfSections-1].Misc.VirtualSize,
dwSectionAlignment);
   *(&pNewSectionHeader->Misc.VirtualSize) = GetAlignment(dwSectionSize,
dwSectionAlignment);
```

```
    *(&pNewSectionHeader->SizeOfRawData) = GetAlignment(dwSectionSize,
dwFileAlignment);;
    *(&pNewSectionHeader->PointerToRawData) =
GetAlignment(pSectionHeader[pCOFFHeader-
>NumberOfSections-1].PointerToRawData+ pSectionHeader[pCOFFHeader-
>NumberOfSections-1].SizeOfRawData, dwFileAlignment);
    *(&pNewSectionHeader->Characteristics) = IMAGE_SCN_MEM_EXECUTE |
IMAGE_SCN_MEM_READ | IMAGE_SCN_CNT_CODE;
    *(&pNewSectionHeader->PointerToRelocations) = 0x0;
    *(&pNewSectionHeader->PointerToLinenumbers) = 0x0;
    *(&pNewSectionHeader->NumberOfRelocations) = 0x0;
    *(&pNewSectionHeader->NumberOfLinenumbers) = 0x0;

    /*Champs à ne pas oublier*/
    *(&pCOFFHeader->NumberOfSections) += 0x1;
    *(&pOptionalHeader->SizeOfImage) = GetAlignment(pOptionalHeader-
>SizeOfImage+dwSectionSize, dwSectionAlignment);
    *(&pOptionalHeader->SizeOfHeaders) = GetAlignment(pOptionalHeader-
>SizeOfHeaders+ sizeof(IMAGE_SECTION_HEADER), dwFileAlignment);

    printf("OK");

    return;
}
```

B. Écriture du loader.

Il faut maintenant mettre au point notre loader, c'est-à-dire le sous programme que nous intégrerons dans notre nouvelle section et vers lequel on redirigera le point d'entré du programme lui-même afin que notre loader soit exécuté. Une fois le loader exécuté il faudra que celui-ci repasse la main au programme.

Nous allons commencer avec un loader simple qui va seulement sauter sur le programme principal. On a donc :

```
MOV EAX, 0
JMP EAX
```

Et on remplace ensuite le '0' par l'adresse du point d'entré d'origine de l'application (**OEP**). Après établit notre shellcode avec nasm/masm ou autres, suivant vos préférences. Moi j'obtiens :

```
char loader[] = {
    0xB8,0x00,0x00,0x00,0x00,      // offset : 0
```

```
MOV EAX,0
  0xFF,0xE0,      // offset : 5             JMP EAX

        };
```

C. Intégration du loader au sein de notre exécutable.

Ensuite nous allons intégrer notre loader dans l'exécutable en l'écrivant dans la section que nous venons de créer.

J'ai pour cela codé une fonction permettant d'écrire dans une section :

```
BYTE WriteInSection(HANDLE hBinary, PUCHAR pSectionName, PUCHAR pBuf,
UINT size)
{
   PDWORD pSectionAddress= NULL;

   pSectionAddress= GetSectionAddress(hBinary, pSectionName);

   if(pSectionAddress== 0x0)
            return 0x0;

   memcpy(pSectionAddress,pBuf, size);

   return 0x1;
}
```

Fonction elle-même utilisant une autre servant à récupérer l'adresse à laquelle débute une section :

```
PDWORD GetSectionAddress(HANDLE hBinary, PUCHAR pSectionName)
{
   PIMAGE_SECTION_HEADER pSectionHeader = NULL;
   PIMAGE_FILE_HEADER pCOFFHeader = NULL;
   size_t i;

   pSectionHeader = (PIMAGE_SECTION_HEADER)GetSectionHeader(hBinary);
   pCOFFHeader = (PIMAGE_FILE_HEADER)GetCOFFHeader(hBinary);


   for(i = 0; i < pCOFFHeader->NumberOfSections; i++)
   {
      if(!strcmp(pSectionHeader[i].Name, pSectionName))
                   return (PDWORD)((PUCHAR)hBinary +
pSectionHeader[i].PointerToRawData);

   }
```

```
        return 0x0;
    }
```

Rien de bien compliqué pour l'instant.

D.Redirection du point d'entré (EP).

A présent redirigeons l'**Entry Point** de notre programme vers notre loader. Il faut alors faire attention de bien spécifier l'**adresse virtuelle** à laquelle se situera la section une fois le programme chargé en mémoire.

Voici une fonction permettant de récupérer cette adresse virtuelle d'une section :

```
PDWORD GetSectionVirtualAddress(HANDLE hBinary, PUCHAR pSectionName)
{
    PIMAGE_SECTION_HEADER pSectionHeader = NULL;
    PIMAGE_FILE_HEADER pCOFFHeader = NULL;
    size_t i;

    pSectionHeader = (PIMAGE_SECTION_HEADER)GetSectionHeader(hBinary);
    pCOFFHeader = (PIMAGE_FILE_HEADER)GetCOFFHeader(hBinary);


    for(i = 0; i < pCOFFHeader->NumberOfSections; i++)
    {
        if(!strcmp(pSectionHeader[i].Name, pSectionName))
                        return (PDWORD)pSectionHeader[i].VirtualAddress;

    }

    return 0x0;
}
```

Et voici comment je redirige l'**EP** :

```
VOID RedirectEntryPoint(HANDLE hBinary)
{
    PDWORD pSectionVirtualAddress= NULL;
    PIMAGE_OPTIONAL_HEADER pOptionalHeader = NULL;

    printf("\n[+]Redirecting Entry Point...");

    pSectionVirtualAddress= GetSectionVirtualAddress(hBinary, ".lilxam");

    pOptionalHeader = GetOptionalHeader(hBinary);

    *(&pOptionalHeader->AddressOfEntryPoint) = pSectionVirtualAddress;
```

```
        printf("OK");

        return;
    }
```

Nous avons maintenant une première ébauche de ce que sera notre packer.

5.Chiffrement du code.

Je ne vous parlerai pas de compression dans cet article, ce n'est pas simple, mais ce sera probablement le sujet du prochain.

Nous allons quand même voir comment crypter certaines parties de notre programme. Je dis certaines parties en effet car on ne va par exemple pas toucher à l'**Import Address Table** (**IAT**).

On chiffrera seulement le code.

Pourquoi chiffrer le code ? Et bien tout simplement parce que le désassemblage de l'application se trouve alors complètement falsifié. Par exemple avec **IDA** on ne verra rien du code, il faudra d'abord unpacker l'exécutable.

Pour faire cela la structure **IMAGE_OPTIONAL_HEADER** va nous être utile avec les champs **BaseOfCode** et **SizeOfCode**.

Le problème est que **BaseOfCode** est un pointeur relatif à l'**ImageBase** (toujours de la structure **IMAGE_OPTIONAL_HEADER**). En combinant les deux on obtiendra donc une **adresse virtuelle**, puisque l'**ImageBase** est l'adresse virtuelle à laquelle est chargée notre programme. Nous nous avons besoin de l'adresse ***raw***, en dur.

Pour l'obtenir je vais en fait parcourir toutes les sections et voir si le **BaseOfCode** est située dans l'une d'entre elles en comparant les adresses virtuelles. Dans la majorité des cas il s'agira de la section .text mais bon je préfère envisager d'autres possibilités et faire comme si je ne le savais pas. De plus il est presque certain que le code commence au début de la section mais j'ai là aussi préféré prévoir le cas où ce ne serait pas le cas.

Alors voilà la fonction me permettant de récupérer l'adresse du début du code :

```
PDWORD GetRawBaseOfCode(HANDLE hBinary)
{
   PIMAGE_OPTIONAL_HEADER pOptionalHeader = NULL;
   PIMAGE_SECTION_HEADER pSectionHeader = NULL;
   PIMAGE_FILE_HEADER pCOFFHeader = NULL;
```

```
    size_t i;

    pOptionalHeader = GetOptionalHeader(hBinary);
    pCOFFHeader = GetCOFFHeader(hBinary);
    pSectionHeader = GetSectionHeader(hBinary);

    for(i = 0; i <= pCOFFHeader->NumberOfSections; i++)
    {
        //On regarde si le code est bien dans cette section
        if((PDWORD)pOptionalHeader->BaseOfCode >=
(PDWORD)pSectionHeader[i].VirtualAddress && (PDWORD)pOptionalHeader-
>BaseOfCode <= (PDWORD)((PUCHAR)pSectionHeader[i].VirtualAddress +
pSectionHeader[i].Misc.VirtualSize))
                        return (PDWORD)
((PUCHAR)pSectionHeader[i].PointerToRawData + pOptionalHeader->BaseOfCode -
pSectionHeader[i].VirtualAddress);
                        //Au cas (très rare) où le code ne débuterait pas au début de la section, on
ajoute la différence.
    }

    return 0x0;
}
```

Ensuite je procède ainsi :

```
VOID CryptCode(HANDLE hBinary)
{
    PIMAGE_OPTIONAL_HEADER pOptionalHeader = NULL;
    PUCHAR pSectionName = NULL;
    PDWORD pBeginOfCode = NULL;
    PDWORD pEndOfCode = NULL;
    DWORD dwFlag;

    pOptionalHeader = GetOptionalHeader(hBinary);

    pBeginOfCode = (PDWORD)GetRawBaseOfCode(hBinary);
    pEndOfCode = (PDWORD)((PUCHAR)pBeginOfCode+pOptionalHeader-
>SizeOfCode);

    printf("\n[+]Encryption of code (from 0x%x to 0x%x", pBeginOfCode, pEndOfCode);

    pSectionName = (PUCHAR)calloc(IMAGE_SIZEOF_SHORT_NAME,
sizeof(CHAR));
    pSectionName = GetRawAddrSectionOwner(hBinary, (DWORD)pBeginOfCode);
    printf(". Section : %s)...", pSectionName);

    /* On chiffre le code */
    Crypt(hBinary, pBeginOfCode, pEndOfCode);
```

```
        /* On oublie pas de changer le flag de la section pour pouvoir déchiffrer le code par la suite */
        dwFlag = GetSectionFlag(hBinary, pSectionName);
        dwFlag |= IMAGE_SCN_MEM_WRITE;
        SetSectionFlag(hBinary, pSectionName, dwFlag);

        return;
    }
    VOID Crypt(HANDLE hBinary, PDWORD pStart, PDWORD pEnd)
    {
        size_t i;
        (PUCHAR)pStart += (ULONG)hBinary;
        (PUCHAR)pEnd += (ULONG)hBinary;

        for(i = 0; ((PUCHAR)pStart+i) <= pEnd; i++)
            *((PUCHAR)pStart+i) ^= 0x7;

        return;
    }
```

Maintenant que nous avons chiffré le code, codons une procédure permettant de le déchiffrer :

```
Decrypt PROC

   MOV EAX, 00h
   MOV EBX, 00h

   .while(EAX &lt; EBX)
      XOR BYTE PTR DS:[EAX], 07h
      INC EAX
   .endw

   RET
Decrypt ENDP
```

Dès lors il faut qu'on complète le loader avec les adresses nécessaires.

Ainsi on récupère l'adresse virtuelle à laquelle débutera le code et la taille de celui-ci pour établir l'adresse de fin.

Bon et bien ce sera tout pour ce premier article. Je vous ai montré comment établir les bases d'un packer, nous verrons par la suite comment le faire évoluer pour lui donner des fonctions précises.

Je pense que mon prochain article sera porté sur la compression de l'exécutable.

En attendant voici les sources et binaires :

La librairie LP_LIB
Sources consultables en ligne de LP_LIB
Le loader
Le Projet tout entier

Et voici un binaire packé :
UnpackMe