

# Make your own packer

By [Lilxam](#).

**\* PART I \***

# SOMMAIRE

**1.Introduction.**

**2.Project's structure.**

**3.Required knowledges.**

**4.Basic manipulations of the application, a first packing.**

- A. Adding a section.
- B. Writing the loader.
- C. Integrating the loader into the program.
- D. Redirecting Entry Point (EP).

**5.Encryption of code.**

# 1.Introduction

You surely know what is a packer. There exist a lot of, from the simplest , like UPX or FSG, to more complexes ones like Themida, Armadillo and many others.

The packer purpose is pre-eminently to compress an executable (what do UPX and FSG) but also to protect it from eventual evil-minded (or not) persons wanting to crack you program.

Regarding the question “Why making my own packer whereas there exist many others and surely more sophisticated than mine ?”, I'll simply answer it's a very interesting project because it necessitates good knowledges of system's mechanisms used to launch a task. In this point I've learnt a tremendous amount of stuffs by realizing it. So my purpose is to share theses knowledges with you.

## 2. Project's structure

Well, before starting, I will explain you how I have organised my project. So I make a program that will use a library I named LP\_LIB, which is the kernel of our project. I'll probably use some others libraries, but I don't know exactly what for the moment.

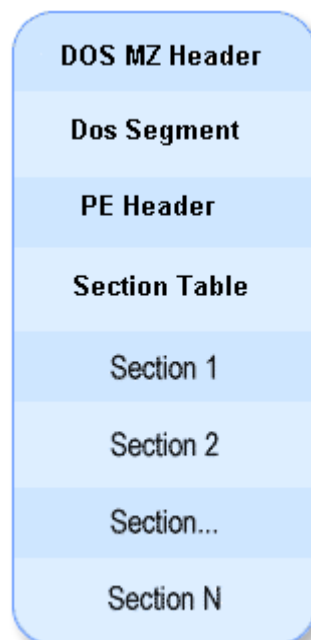
Actually I called my packer “Lilxam's Packer”... Nothing original but I wasn't inspired.

## 3. Required knowledges.

First you need to have some knowledges concerning the PE format.

We will rapidly see how the executable header is constituted and what are the structures composing it. It will be very short because there are à lot of documentations on the subject.

This is a schema showing the PE file format :



So we firstly see the **DOS Header**.

Let us take a look to its definition :

```
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;
```

The **e\_magic** field corresponds to the 'MZ' signature, signature that every executable on windows, like DLLs or drivers, need to have. And **e\_lfanew** points to the PE or NT Header.

To parse the executable header I coded some functions. Here is the one to obtain the DOS Header :

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

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

    return pDOSHeader;
}
```

Then comes the **DOS segment**. It is simply a subroutine which is called in the case of the *PE LOADER* fails and which displays a message looking like «This program cannot be run in DOS mode. ». Nothing really important for us.

Let us rather take an interest to the PE Header strictly said. It corresponds to the **IMAGE\_NT\_HEADERS** structure :

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

The first field is the signature certifying the file is a PE image (executable). This signature is 'PE\0\0' (that is 'PE' following by two null bytes).

The function to obtain this 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;
}
```

The second field is in point of fact a **IMAGE\_FILE\_HEADER** structure :

```
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;
```

It's the **COFF Header** ( Common Object File Format). Or the common object header on windows.

We will just consider that NumberOfSection corresponds to the number of sections that the application contains.

To get the COFF Header :

```
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;
}
```

Yet in the PE Header is the **IMAGE\_OPTIONAL\_HEADER** structure :

```
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;
```

I'll explain later, during the article, to what corresponds each field.  
My function to get it :

```
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;
}
```

After the PE Header follows the **Section Table**, that is an array of **PIMAGE\_SECTION\_HEADER** structure.

```
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;
```

I'll explain it when I'll show you how adding a section to the program.



I obtain this table like this :

```
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;
}
```

And this is a function showing how listing 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);

    }
}
```

Now let us start serious things.

## 4. Basic manipulations of the application, a first packing.

The idea is to add a new section to the program in which we will write our loader and then to redirect the Entry Point to our new section to execute the loader.

### A. Adding a section.

We will see here how adding a section to the program without disturbing its stability.

```
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;
```

Here is an explanation of the interesting fields for us :

-**Name** : the section's name, it can't be longer than 8 bytes (**IMAGE\_SIZEOF\_SHORT\_NAME** = 8).

-**VirtualSize** : the section's (virtual) size rounded to the next multiple of **SectionAlignment** to the **IMAGE\_OPTIONNAL\_HEADER** structure.

-**SizeOfRawData** : the section's (raw) size rounded to the next multiple of **FileAlignment** to the **IMAGE\_OPTIONNAL\_HEADER** structure.

-**PointerToRawData** : the offset where start the section.

-**Characteristics** : this field contains the flags characterizing the section. We will use these ones :

\***IMAGE\_SCN\_MEM\_EXECUTE** : the section is executable.

\***IMAGE\_SCN\_MEM\_READ** : the section is readable.

\***IMAGE\_SCN\_MEM\_WRITE** : the section is writable.

\***IMAGE\_SCN\_CNT\_CODE** : the section contains the or a part of the code of the program.

To add our section we will have to fill-in these fields. But not only these ones, as a part of it we also have to increment the **NumberOfSections**, to **IMAGE\_FILE\_HEADER** structure, and **SizeOfImage**, belonging to **IMAGE\_OPTIONAL\_HEADER**, to the size of our section.

Without further delay I give you the function aiming at doing this task :

```
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 address 0x%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. Writing the loader.**

Now we have to elaborate our loader, that is to say the subroutine we will integrate in the new section. And then we will redirect the Entry Point of the program to the loader in order to make it be executed. Once the execution done, the flow will have to be repassed to the main thread.

We will begin with a simple loader which will just jump on the main program. Well we have :

```

MOV EAX, 0
JMP EAX

```

And after we replace the '0' by the Original Entry Point address (OEP). Then I build the shellcode with nasm/masm or others compilers, it depends of your preferences. I obtain :

```

char loader[] = {
    0xB8,0x00,0x00,0x00,0x00,    // offset : 0
    MOV EAX,0
    0xFF,0xE0,    // offset : 5          JMP EAX
};

```

## C. Integrating the loader into the program.

At this point we will integrate our loader into the application by writing it in the section we have just created.

To achieve this task, I wrote a function aiming at writing in a section :

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

    pSectionAddress= GetSectionAddress(hBinary, pSectionName);

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

    memcpy(pSectionAddress,pBuf, size);

    return 0x1;
}
```

Function using another which returns the starting address of a section by passing its name :

```
PDWORD GetSectionAddress(HANDLE hBinary, PCHAR 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)((PCHAR)hBinary +
pSectionHeader[i].PointerToRawData);
    }

    return 0x0;
}
```

## D. Redirecting the Entry Point(EP).

Now let us redirect the **Entry Point**. We just have to take care to fill the **virtual address**, and not the raw one, where the section will be situated once the program loaded in memory.

Here is a function to get a section's virtual address :

```
PDWORD GetSectionVirtualAddress(HANDLE hBinary, PCHAR 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;
}
```

And here is how I redirect the **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;
}
```

At present we have a rough-shape of what will be our packer.

## 5. Encryption of code.

The compression won't be a subject of this article, it's not easy, but it will probably be the topic of my next paper.

Nevertheless we will see how encrypting some parts of the application. I said only some parts because in fact for example we can't affect the **Import Table**.

Only the code will be encrypted.

What for ? Simply because the disassembling of the program will be impossible. For example with **IDA** we won't see anything of the code while the application isn't unpacked.

To do this the **IMAGE\_OPTIONAL\_HEADER** structure will be useful thanks to the **BaseOfCode** and **SizeOfCode** fields.

The problem is that **BaseOfCode** is an **ImageBase** (from **IMAGE\_OPTIONAL\_HEADER**) relatif pointer. Therefore we will obtain a virtual address in combining the two, because the **ImageBase** is the virtual address where is loaded the program. We need the raw address.

To obtain it, I will parse all the sections checking if the **BaseOfCode** is situated in one by comparing the virtual addresses. Commonly the code is located in the '.text' section but I prefer consider other possibilities. Moreover it's almost sure the code starts at the begin of the section but this time again I prefer anticipate the case of it wouldn't be the case.

Good, now I can show you how getting the base of 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 débiterait pas au début de la section, on
ajoute la différence.
    }

    return 0x0;
}

```

Then I proceed like this :

```

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;
}

```

Now the code is encrypted, we have to code the decryption procedure :

```

Decrypt PROC

    MOV EAX, 00h
    MOV EBX, 00h

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

    RET
Decrypt ENDP

```

Therefore we continue in completing the loader with the required addresses.

So these addresses are the virtual address where starts the code and the virtual address of its end. To obtain this last address, we just have to get and add the size of code to the start address.

Good, that is the end of this article. I have shown you how making packer bases. Subsequently we will make it evolve to give it more specific functions.

I think my next article will deal with the compression of the program.

Here are the sources and binaries :

[Library LP\\_LIB](#)

[Online consultable sources LP\\_LIB](#)

[The loader](#)

[The entire project](#)

And here is a packed binary :

[UnpackMe](#)