The PE Format

Posted by Gaurav Mogre on December 8, 2008 · 8 Comments

Warning: This document is contains purely technical information. This can be considered as iron, out of which weapons can be made . Additionally, this is about 48 pages long and written by me

Introduction:

Windows uses the Portable Executable Format to store executable files, also known as an “image” of an executable. Although the PE file contains all the information required to “run” a program, the PE file must first be parsed, processed and loaded into memory. This process involves allocation of memory, relocations, imports, etc. Thus, the PE file is simply an “image” of the executable, the executable being referred to the program in memory.

The Portable Executable Format is a highly portable format, compatible for use with many different 32 machines, on the various versions of windows. The PE format can be used identically for 64 bit machines, with very minor modifications.

This document does not act as a standard for the portable executable format, since such manuals already exist. This document also does not cover all the aspects of a portable executable. This document dives head first into the PE format, by directly observing the PE format of a simple test program. It is in the process of understanding the program itself, that the implementation of the PE will also become clear. It concentrates on the most commonly used parts in a usual portable executable, and later an analysis on changing certain values from those parts.

Aims:

The aim of the project is:

- To develop an understanding on the structure of the Portable Executable
- Understanding the process involved in loading of a PE image file from the hard disk to the memory
- To emulate the working of the windows loader
- Ability to create and alter any portable executable file to suit needs and wants.

Preparations:

The Test Program:

The program chosen for investigation is the standard “Hello World” Program. Its chosen for its simplicity, as well as:

1. It uses library calls for outputting to a console
2. The main standard sections in the executable are used.

The program, written in C, is:

```
#include<stdio.h>

int main(void){

printf(“Hello World”);

return 0;

}
```

The code was compiled using Borland C++ compiler, free version 5.5.40.244. It was linked with the link32 linker also provided in the Borland compiler package.

<!– Insert console output of the compilation process –>

Thus, we get a 51.00 kb .exe file which will be used to investigate into the structure of portable executables

Preparing the Hex Viewer:

All the fields in the PE file are byte aligned and thus, a hex viewer is necessary to view the file. The following is the source code for the hex viewer which produces a hexadecimal representation of the entire file:

```
#include <iostream>

#include <iomanip>
```

```
#include <fstream>
/*
Name: Hex Viewer
Copyright: croSSArrow
Author: Gaurav Tushar Mogre
Date: 30/09/08 18:06
Description: A Basic Hex File Viewer.
*/
using namespace std;
int main(int argc,char* argv[])
{
if(argc<2){
cerr<<"Format: hexviewer.exe <filename>n";
return 1;
}
char fname[41];
strncpy(fname,argv[1],36);
fstream inpfile;
inpfile.open(fname,ios::in|ios::binary);
fstream outfile;
outfile.open(strcat(fname,".hex"),ios::out|ios::binary);
while(!inpfile.eof()){
unsigned char ch;
inpfile>>ch;
if(ch<16) outfile<<"0×0″<<std::hex<<setprecision(2)<<(int)ch<<" ";
else outfile<<"0x"<<std::hex<<setprecision(2)<<(int)ch<<" ";
}
inpfile.close();
outfile.close();
cout<<"Written successfully to file: "<<fname <<endl;
return 0;
}
```

Thus, a file: testprogram.exe.hex is produced on running the above program, which generates the hexadecimal representation of testprogram.exe which makes it easier to analyze the file.

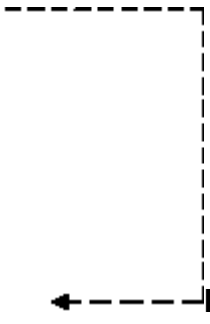
The PE format:

The overall format of a PE can be described by-

MS DOS Header

MS DOS Signature
Remaining Headers

MS DOS auxiliary header-



Pointer to PE Signature

MS DOS Stub Program (and MSDOS relocation table)

…

Portable Executable Signature

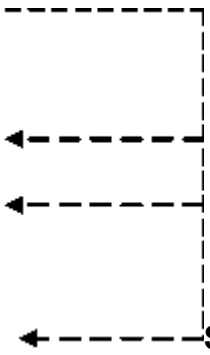
COFF File Header

Optional Headers

Standard Fields

Windows Specific Fields

Data Directories



Section Header

…

Section 1

Section 2

…

Section n

Core Terminologies and Concepts

RVA: Relative Virtual Address:

The Portable Executable format heavily uses relative virtual addressing for addressing contents within the file, and helps in making the executable relocatable.

A base address is the memory address of the beginning of the executable when it is loaded into memory. The RVA of certain data in memory can then be calculated as:

RVA = Absolute Address in memory – Base address

Let us consider now the reason for using of RVA: Consider data in the file references some other data in the file using an absolute address. This address would then mean that the executable MUST be loaded into the same memory address space every time the file is executed. This is however not possible, since that memory address space may not be completely free when the file wishes to execute. Thus, the file would cease to run.
If RVA's are used instead, the file could become relocatable. The file could be loaded into any area of memory. When the loader loads the file, it looks up the base address to which the file has been loaded to. When a particular RVA needs to be addressed, the loader simply adds the Base address to the existing RVA to get the absolute location of the data.

Thus, as long as the entire file is loaded logically contiguously, it can be completely addressed. Note that the loader does not "convert" the RVA's to absolute address when loading the executable into memory, i.e. the RVA's are copied to the memory. It is when one RVA is referenced (and it will be referenced by the loader) that the loader calculates the absolute address at that point.

Alignment:

All the data in the Portable executable is broken down into equal length parts. Alignment is the process of putting data into these parts such that no data is broken between parts. For example, if a particular field is word aligned, then that field must comprise of 2 bytes. It is incapable of handling more bytes, and padding must be applied if data is smaller than the field.

Portable Executables have 2 types of alignment: File Alignment and Memory alignment (also called section alignment). The File Alignment specifies the alignment of data within a file. When the data is loaded from the file into the memory, its alignment changes to the section alignment. The section alignment is a multiple of the File Alignment, thus the boundaries are maintained. However, when the loading takes place, more bytes may be set aside for padding.

We see that Section Alignment must be a multiple of File alignment to maintain boundaries within the parts of a file. The File Alignment and the Section Alignment can have the same value, in which case the image is loaded identically into the memory. However, there are cases when Section Alignment can be purposely made larger than the file alignment. When such a situation arises, each section contains more padding bytes when initially loaded, and these bytes can be overwritten with data. Thus, data produced/derived during execution can be stored in the extension provided by the extra padding.
If the File Alignment was instead made as big as the Section Alignment, the File would be filled with a lot of padding zeroes, thus unnecessarily increasing the file size.

A potential ambiguity which arises when using 2 alignments is the value of the RVA. The RVA is the difference between the absolute loaded address and the base address loaded into memory. Thus, the RVA is calculated and set using the section alignment for the image.

In order to calculate the data to which the RVA points to in a file, we must therefore follow the following steps:

1. Find the RVA for the data
2. From the RVA, derive the section to which the data referenced belongs. This is trivial, since sections don't overlap. The starting addresses of the various sections are available in the file header
3. Find the difference between the RVA and the starting address of the section to find the data offset, ie, offset of that data within a section.
4. From the file header, for the same section, find the location of the same section in the file.
5. Add the data offset to the location of section in the file, to find the address of the data in the file.

Further discussions on the File Alignment and Section Alignment will be taken up later. However, it is important to understand how to convert an RVA from one alignment value to another.

Endian-ness:

All the data in a PE file is stored in the little endian format. This means that the least significant byte is stored at the lower address. Thus, when we interpret a value in a PE file, we must first convert the number from little endian to human readable.

For eg, the field "pointer to PE" in the MS DOS optional header is a dword length (4 bytes)… Thus, when the value of the 4 bytes (starting at 0×3c location) is 0×00 0×04 0×00 0×00 , it should be interpreted as 0×00000400 , i.e. 0×400 location.

The MS DOS Stub:

A PE image file begins with a MS DOS stub. The PE file format has been around since the beginning of windows, and hasn't changed to maintain backward compatibility. The MS DOS stub is simply a small program which precedes the actual windows executable. It was used so that when a program is run in MS DOS, it would give a good error message (that the program must be run in windows) instead of crashing MS DOS.

Basically, a MS DOS program is a self contained program, and can run independently as well on a DOS system. When DOS executes a PE, it starts executing from the beginning, and since the file format is the DOS format (the MZ format), it executes the stub. The stub usually just prints out a message and interrupts back to the OS. When a windows loader tries to execute a PE, on the other hand, it will simply verify whether the DOS stub is valid, then directly jump to the PE part of the file, and start executing from there.

The DOS header is of the format:

Offset	Length	Name	Comments
0	2	identifier	The first 2 bytes must respectively contain the MZ signature. They must be respectively set to the value 0×4d and 0×5a
2	2	last page size	The number of bytes in the last page of the stub
4	2	number of pages	The total number of pages in the file, including the last page (and header)
6	2	relocations size	The number of relocation entries in relocation table (similar to relocation table in PE)
8	2	header size	Header size in paragraphs
10	2	minimum	The minimum number of paragraphs that must be available in addition to

		extra para	the code size of the MZ
12	2	maximum extra para	The maximum number of paragraphs that can be allocated. Usually set to 0xffff
14	2	Initial SS	Initial SS with respect to the base address of loading
16	2	Initial SP	Initial SP with respect to the initial SS
18	2	Checksum	Checksum of executable (usually 0)
20	2	Initial CS	Initial CS with respect to the base address of loading
22	2	Initial IP	Initial IP with respect to the initial CS
24	2	relocation table	Offset to the relocation table (0×40 for PE)
26	2	Overlay number	Unused

In MS DOS, the file is loaded in terms of pages, each page consisting of many paragraphs. A paragraph has a fixed length of 16 bytes. On the other hand, a page can have a maximum length of 512 bytes. All pages in a MZ file are of 512 bytes, except the last file, which can vary, and its size is specified in the header.

Thus, to calculate the size of the MZ stub (without padding): number of page * 512 – (512 – last page size)

The MZ file format also contains a reference to the relocation table. The Actual executable part (as well as data) begins right after the relocation table. Following the mandatory MZ header is an optional header. The optional header is not used by MS DOS is any way, and is program dependent. PE file stores a pointer to the actual windows executable in this header. The length of the optional header is=relocation table pointer – 0×1c .

We will restrict our discussion of MZ format to that. A windows loader does not even actually check all the fields in the MZ file. It simply checks the first 2 bytes of the file, i.e. the identifiers 0×4d 0×5a (which btw is ‘MZ’ in ascii). Then it jumps to a fixed location, which is part of the optional header. That location is **0×3c**. Then, the 4 bytes, starting from (and including) 0×3c are read, and the value is a 4 byte pointer to the location of the PE code.

The pointer is a 4 byte offset, from the beginning of the file. It is not an RVA, it is from the beginning of the file.

For eg, a hex dump of the Hello World program gave the following MS DOS Stub:

0×4d 0×5a 0×50 0×00 0×02 0×00 0×00 0×00 0×04 0×00 0×0f 0×00 0xff 0xff 0×00 0×00 0xb8 0×00 0×00 0×00 0×00 0×00 0×00 0×00
0×40 0×00 0×1a 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00
0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 **0×00 0×02 0×00 0×00** 0xba 0×10 0×00 0×0e 0×1f 0xb4 0xcd 0×21
0xb8 0×01 0×4c 0xcd 0×21 0×90 0×90 0×54 0×68 0×69 0×73 0×70 0×72 0×6f 0×67 0×72 0×61 0×6d 0×6d 0×75 0×73 0×74 0×62 0×65
0×72 0×75 0×6e 0×75 0×6e 0×64 0×65 0×72 0×57 0×69 0×6e 0×33 0×32 0×24 0×37 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00
0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 ……

We see that the starting 2 bytes are the correct identifiers. We also see at location 0×3c, which is made bold, the pointer to the PE header exists, which is set to 0×200

(The padding null bytes are required for file alignment to sections, since the PE header can begin on a new section file. This value is specified in the PE header)

PE Signature:

At the starting of the actual PE, we find an identifier, or called the PE signature. It identifies that the executable following is at least supposed to be in PE format. The PE signature is 4 bytes long, and has the value: 0×50 0×45 0×00 0×00 . This is character data, which can be translated to “PE” string (The letters ‘P’ and ‘E’ followed by 2 null bytes).

In the hello world program, we find the same signature at 0×200 location in file:

0×50 0×45 0×00 0×00

COFF Header:

The COFF header immediately follows the PE signature. It is essential in both PE as well as COFF files (In COFF file, the COFF header is the beginning of the file, since there is no stub or PE signature). It consists of values essential for describing the nature of the program. It comprised of the following details:

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. The most common values of the Machine are:

Value	Machine Type

			0×0 Machine independent 0×8664 AMD 64 bit 0×14c Intel 386 + processors 0×200 Intel Itanium processors
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created. Mainly used while exporting functions or checking resources
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. It is 0 for PE image files, since debugging information is depreciated.
12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value is zero for an image because debugging information is deprecated.
16	2	SizeOfOptionalHeader	The size of the optional header
18	2	Characteristics	The flags that indicate the attributes of the file. It's a bit field value, which indicates the properties of the executable. **Bit Description** 2 File is working executable 6 File can handle very large addresses (>2GB) 9 Machine is 32 bit 10 Debugging info is stripped from file and stored separately 14 The file is a DLL The other bits also have significance, but are seldom used, and are generally 0.

When we look at the COFF header for the Hello World executable, we see:

0×4c 0×01 0×08 0×00 0×82 0×25 0xe2 0×48 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0xe0 0×00 0×0e 0×03

We find that the machine required is an intel 386+ based machine. There are 8 sections in sections table. We also find that size of the optional header is 0xe0 . The characteristics state that the file is a working executable, it is 32 bit, not a dll, and the debugging information is stripped from the file.

Optional Header:

Optional header is truly optional only for object COFF files. However, the "optional" header is mandatory for a PE file, and gives very important information to the windows program loader. The Optional header differs slightly between the PE32 (32 bit executable) and PE32+ (64 bit executable). The type of PE is determined by looking at the "magic number" bytes in the optional header, which tells the format for the rest of the header.

The Optional header can be further divided into 3 parts (each following the next):

Standard fields: Fields defined for all versions of COFF, including unix

Windows specific fields: Fields which tell windows how to handle the file

Data Directories: Fields which tell windows the location of certain data/code in the executable

Standard Fields:

Offset	Size	Field	Description
0	2	Magic	The unsigned integer that identifies the state of the image file. The common numbers are:

			0×10b : PE32 executable 0×107 : ROM image 0×20b : PE32+ (64 bit) executable
2	1	MajorLinkerVersion	The linker major version number. This and the minor linker version are set by the linker and are optional (can be set to 0)
3	1	MinorLinkerVersion	The linker minor version number.
4	4	SizeOfCode	The size of the code (text) section or the sum of all code sections if there are multiple sections.
8	4	SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections.
16	4	AddressOfEntryPoint	The address of the entry point relative to the image base (beginning of the PE file) when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs (since they are a set of functions). When no entry point is present, this field must be zero.
20	4	BaseOfCode	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory. The value at that location in file is loaded into memory, and that memory location is loaded as the CS
24	4	BaseOfData	The address that is relative to the image base of the beginning-of-data section when it is loaded into memory. When the data at the section is loaded into memory, that memory location is loaded into the DS

The value of the standard optional header for the Hello World Program is:

0×0b 0×01 0×05 0×00 0×00 0×90 0×00 0×00 0×00 0×30 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×10 0×00 0×00 0×00 0×10 0×00 0×00 0×00 0xa0 0×00 0×00

We find that the image type is a normal PE32 executable. The size of code is 0×9000 bytes. We find size of initialized data is 0×3000 . The program does not require any uninitialized data, thus the size of uninitialized data is 0×0000. The address of the entry point is 0×1000 , which is also the address to the base of code (i.e., the code starts execution from the beginning of the code section, since both the values are respective to the top of the image). Finally, we see that the data section is located at 0xA000 in the file.

We shall see later that these values are mapped to sections in the file in the sections table.

Windows Specific Optional Headers:

Offset	Size	Field	Description
28	4/8	ImageBase	The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is 0×10000000, the default for normal executables is 0×00400000.
32	4	SectionAlignment	The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.
36	4	FileAlignment	The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the

			architecture's page size, then FileAlignment must match SectionAlignment.
40	2	MajorOperatingSystemVersion	The major version number of the required operating system.
42	2	MinorOperatingSystemVersion	The minor version number of the required operating system.
44	2	MajorImageVersion	The major version number of the image.
46	2	MinorImageVersion	The minor version number of the image.
48	2	MajorSubsystemVersion	The major version number of the subsystem.
50	2	MinorSubsystemVersion	The minor version number of the subsystem.
52	4	Win32VersionValue	Reserved, must be zero.
56	4	SizeOfImage	The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment.
60	4	SizeOfHeaders	The combined size of an MSDOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
64	4	CheckSum	The image file checksum. It involves not only the file, but also its dynamic imports to calculate the checksum. It can safely be kept as 0.
68	2	Subsystem	The subsystem that is required to run this image. The various subsections are: **Value Subsection** 1 Device Drivers/Native 2 Windows GUI 3 Windows Console 7 POSIX character CLI 10 EFI (Extensible Firmware Interface) Applications
70	2	DllCharacteristics	DLL characteristics… Set to 0 for normal executables
72	4	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
76	4	SizeOfStackCommit	The size of the stack to commit.
80	4	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
84	4	SizeOfHeapCommit	The size of the local heap space to commit.
88	4	LoaderFlags	Reserved, must be zero.
92	4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

It is in this header that we define the SectionAlignment and FileAlignment. The SectionAlignment is the number of bytes per section loaded in memory. The file can also logically be divided into sections (which are mapped directly to sections in memory), and the FileAlignment gives the size of each section in the File. Since every section in memory is mapped from a file, SectionAlignment must be greater than or equal to the FileAlignment. However, for RVA calculations, it's easier if SectionAlignment is always a multiple of FileAlignment (and windows enforces it). The entire PE file is infact divided into FileAlignment sized sections. This includes the sections for the MS DOS stub, for the image header, etc.

The image base is the preferred address for loading the executable. If this location is available, then the executable will be loaded at that location. If it is unavailable, then the executable will still get loaded, but would have to undergo numerous relocation of its references.

For our Hello World program, the windows specific optional header is:

0×00 0×00 0×40 0×00 0×00 0×10 0×00 0×00 0×00 0×02 0×00 0×00 0×04 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×04 0×00 0×00 0×00
0×00 0×00 0×00 0×00 0×00 0×00 0×30 0×01 0×00 0×00 0×06 0×00 0×00 0×00 0×00 0×00 0×00 0×03 0×00 0×00 0×00 0×00 0×00 0×10 0×00
0×00 0×00 0×00 0×00 0×00 0×10 0×00 0×00 0×10 0×00 0×00 0×00 0×00 0×00 0×00 0×10 0×00 0×00 0×00

We find that the recommended image base is 0×400000 . Now we see that the SectionAlignment is 0×1000 where as the FileAlignment is 0×200 . We must hence be way of back-calculating the RVA's within the PE. The Size of the image is 0×13000 , which is 77284 bytes. This is the size required in memory, with SectionAlignment alignment. Since the FileAlignment is $1/5^{th}$ that, we find that the size of the image in the file is much lesser. However, calculation of FileAlignment from this value is not an easy task, since many sections present in the file are never actually loaded into the memory.

The size of the headers is totally 0×600 bytes. This is rounded off to the nearest multiple of file alignment. We find that the Borland linker failed to calculate the checksum for the executable and left it blank.

The value for the windows subsystem is 0×03, which means that the program is a windows console applications, which is what it is. Since the program is not a DLL (as indicated in the characteristics of the standard COFF header), it's DLL characteristics are all 0 (note: that means this is a normal executable. However, it is possible for executables to have export functions as well, in which case DLL characteristics may be defined, especially when the subsystem is 0×01, i.e., windows native).

We find the size of stackReserve is 0×00100000 , though actually committed at the run time is 0×2000 . This means that the stack will actually have size 0×2000 , but can achieve a maximum size of 0×100000 . Similarly, the heap reserve is 0×00100000 but the actually committed run time heap is 0×1000 .

Finally, we see that the number of RVA's and sizes in the following table is 0×10 .

Optional Header: Data Directories:

The data directories section consists of an array of 2 entries: The virtual address and the size. The virtual address refers to the RVA of a particular section, while the size refers to the size of a section. The index of an entry determine the significance of that entry. For e.g., the first entry refers to the export directory RVA.

The data directories are used to signify sections which have certain special significance. They are declared to the windows loader, so that the loader may use that information for certain tasks, such as dynamic linking or relocations or local thread management. We shall discuss some of the sections in brief details later.

As mentioned earlier, each entry in the data directories list consists of a node of 2 elements:

DWORD virtualAddress
DWORD size

Thus, each entry in the data directories table is 8 bytes long.

The various indices and their significance in the data directories table are:

Offset	Size	Field	Description
96	8	Export Table	The export table address and size. Discussed later
104	8	Import Table	The import table address and size. Discussed later
112	8	Resource Table	The resource table address and size. Discussed later
120	8	Exception Table	The exception table address and size. Contained usually in the .pdata section , it is used for structured exception handling
128	8	Certificate Table	The attribute certificate table address and size. Discussed briefly later
136	8	Base Relocation Table	The base relocation table address and size. Discussed briefly later.
144	8	Debug	The debug data starting address and size. It is usually found at the end of the PE
152	8	Architecture	Reserved, must be 0
160	8	Global Ptr	The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero.

168	8	TLS Table	The thread local storage (TLS) table address and size.
176	8	Load Config Table	The load configuration table address and size.
184	8	Bound Import	The bound import table address and size. Discussed along with imports section
192	8	IAT	The import address table address and size.
200	8	Delay Import Descriptor	The delay import descriptor address and size.
208	8	CLR Runtime Header	The CLR runtime header address and size. (only for object files.. for PE, set to 0)
216	8	Reserved, must be zero	

For our Hello World program, we see the hex dump as:

0×00 0×00 0×01 0×00 0×6f 0×00 0×00 0×00 0×00 0xf0 0×00 0×00 0×80 0×04 0×00 0×00 0×00 0×10 0×01 0×00 0×00 0×02 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×20 0×01 0×00 0xf8 0×07 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0xe0 0×00 0×00 0×18 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00

Thus, from the above hex dump, we get the following sections table

Directory	Relative Virtual Address	Size (in memory) (to nearest value of section alignment)	Hex Code of record
Export Table	0×10000	0×6f	0×00 0×00 0×01 0×00 0×6f 0×00 0×00 0×00
Import Table	0xf000	0×480	0×00 0xf0 0×00 0×00 0×80 0×04 0×00 0×00
Resource Table	0×11000	0×200	0×00 0×10 0×01 0×00 0×00 0×02 0×00 0×00
Exception Table	0×00 (Not exist)	0×00	0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00
Certificate Table	0×00 (not exist)	0×00	0×00 0×00 0×00 0×00 0×00 0×00 0×00 0×00
Base Relocations	0×12000	0×7f8	0×00 0×20 0×01 0×00 0xf8 0×07 0×00 0×00

We see that the rest of the entries are 0, i.e. the data directories do not exist. Thus, when discussing data directories, we will be concerned with the export table, import table, resource table, and base relocations table.

We also note that the addresses for the table are an RVA when the image is loaded into memory. Thus, looking up the offset 0×12000 in the file would not yield the relocations table. Since we are concerned with these tables, we would like to edit them directly on the file, i.e. we would like to map the RVA's to the offsets in the file. In order to do this, we must have the address that each table belongs to and then offset of each table within that section (and coincidently, we find that since all tables lie at the memory alignment edge, i.e. sectionAlignment and thus, all data directory tables are lying in a separate section at the beginning of each section). This leaves us with the question of the mapping of sections in file to the sections in memory.

Sections Table:

The sections table contains a mapping between the sections present in a file to those that must be loaded into memory. The sections table is located right after the optional header. This position is necessary, because there is no pointer in the header pointing to the sections table. Instead, the position is calculated by calculating position of the first byte after the optional header.

Each entry in the sections table effectively corresponds to one section, i.e. each entry in the sections table can be considered as a section header. The number of entries in the sections table is dictated by the numberOfSections field in the COFF header. Each entry in the sections table has the following format:

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. Strings in executable

			images cannot be more than 8 byte long, since executable images cannot contain string table. Long names in object files are truncated if they are emitted to an executable file.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. (The RVA)
16	4	SizeOfRawData	The size of the section in the file. For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the file. This must be a multiple of FileAlignment from the optional header. When a section contains only uninitialized data, this field should be zero.