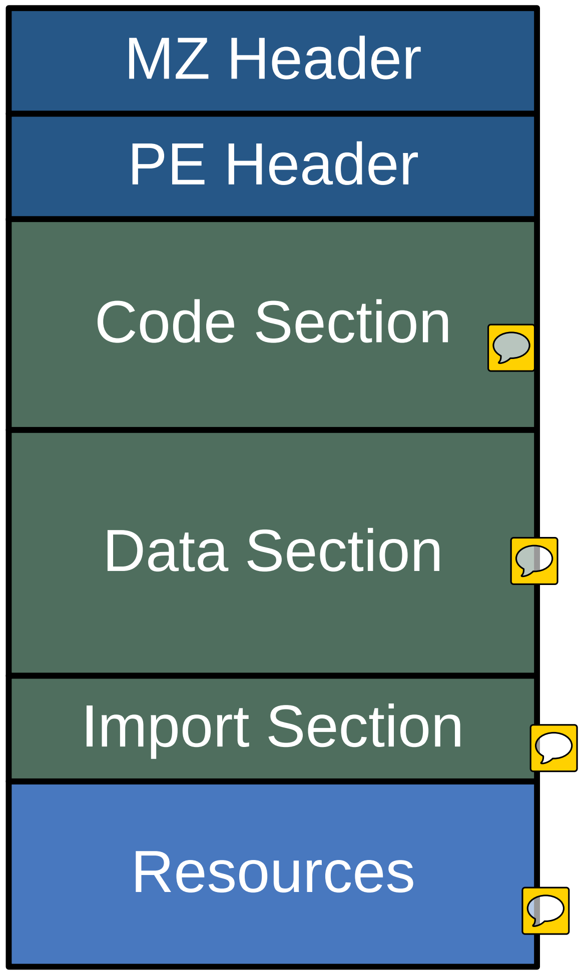
Portable Executable

PE 101

- PE = Portable Executable
- Introduced in Windows NT 3.1
- Originates from Unix COFF
- Dynamic loading, symbol importing and exporting
- Can contain Intel, MIPS, ARM, even .NET MSIL binary code
- 64-bit known as PE32+

PE File

How it looks on disk?



MZ / PE Headers: containing elementary data, including data directory and section table

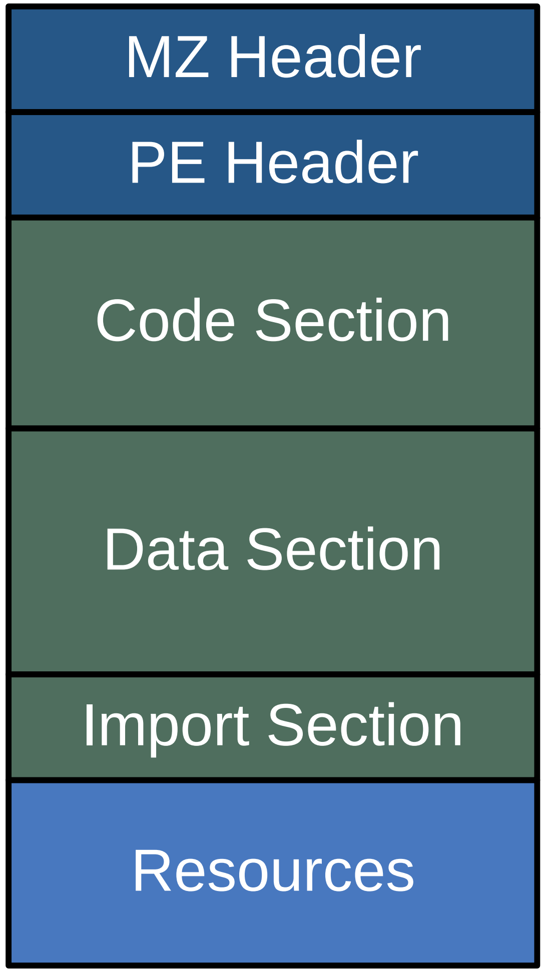
Code Section (.text): executable code

Data Section (.data): program data

Import Section (.idata): DLLs / Symbols

Resources (.rsrcs): optional icons, version, etc.

Relative Virtual Addressing



0x00400000 – Image Base

RVAs are relative to Image Base

Example:

RVA 0x1000 becomes
0x00401000

Image Base value might change, for common DLLs and current Operating Systems.

Image Loading

- PE image on disk and in memory is similar but different
- Section are usually aligned both in file and memory
  - In file the alignment is typically 512 bytes but not needed at all
  - In memory the alignment is either 4096 (IA-32) or 8192 (IA-64)
- Needed parts mapped by kernel from file to memory
- DLLs are not always loaded to the preferred address

Optional Header

- Optional header is a structure that is last member of IMAGE_NT_HEADERS
- Contains information about the logical layout in PE file
- Some are important some are not

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  0x00 WORD Magic;
  0x02 BYTE MajorLinkerVersion;
  0x03 BYTE MinorLinkerVersion;
  0x04 DWORD SizeOfCode;
  0x08 DWORD SizeOfInitializedData;
  0x0c DWORD SizeOfUninitializedData;
  0x10 DWORD AddressOfEntryPoint;
  0x14 DWORD BaseOfCode;
  0x18 DWORD BaseOfData;
  0x1c DWORD ImageBase;
  0x20 DWORD SectionAlignment;
  0x24 DWORD FileAlignment;
  ..
  0x46 WORD DllCharacteristics;
};
```

Source: https://github.com/deptofdefense/SalSA/wiki/PE-File-Format#optional-header

Field	Meanings
AddressOfEntryPoint	It's the RVA of the first instruction that will be executed when the PE loader is ready to run the PE file. If you want to divert the flow of execution right from the start, you need to change the value in this field to a new RVA and the instruction at the new RVA will be executed first.
ImageBase	It's the preferred load address for the PE file. For example, if the value in this field is 400000h, the PE loader will try to load the file into the virtual address space starting at 400000h. The word "preferred" means that the PE loader may not load the file at that address if some other module already occupied that address range.
SectionAlignment	The granularity of the alignment of the sections in memory. For example, if the value in this field is 4096 (1000h), each section must start at multiples of 4096 bytes. If the first section is at 401000h and its size is 10 bytes, the next section must be at 402000h even if the address space between 401000h and 402000h will be mostly unused.
FileAlignment	The granularity of the alignment of the sections in the file. For example, if the value in this field is 512 (200h), each section must start at multiples of 512 bytes. If the first section is at file offset 200h and the size is 10 bytes, the next section must be located at file offset 400h: the space between file offsets 522 and 1024 is unused/undefined.
MajorSubsystemVersion MinorSubsystemVersion	The win32 subsystem version. If the PE file is designed for Win32, the subsystem version must be 4.0 else the dialog won't have 3-D look.
SizeOfImage	The overall size of the PE image in memory. It's the sum of all headers and sections aligned to SectionAlignment.
SizeOfHeaders	The size of all headers+section table. In short, this value is equal to the file size minus the combined size of all sections in the file. You can also use this value as the file offset of the first section in the PE file.
Subsystem	Tell in which of the NT subsystem the PE file is intended for. For most win32 progs, only two values are used: Windows GUI and Windows CUI (console).
DataDirectory	An array of IMAGE_DATA_DIRECTORY structures. Each structure gives the RVA of an important data structure in the PE file such as the import address table.

DllCharacteristics

- Data Prevention Execution (DEP) enable

  IMAGE_DLLCHARACTERISTICS_NX_COMPAT

- Address Space Layout Randomization (ASLR) enable

  IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE

Exercise

Data Directory

- Certain special data in PE is listed in Data Directory
  - Imports, Exports, Resources, etc.
- These are important to have direct pointers in every Data Directory Header
- There are different 15 data directory entries

Section Headers

```
typedef struct _IMAGE_SECTION_HEADER {
  0x00  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
  0x08    DWORD PhysicalAddress;
  0x08    DWORD VirtualSize;
        } Misc;
  0x0c  DWORD VirtualAddress;
  0x10  DWORD SizeOfRawData;
  0x14  DWORD PointerToRawData;
  0x18  DWORD PointerToRelocations;
  0x1c  DWORD PointerToLinenumbers;
  0x20  WORD  NumberOfRelocations;
  0x22  WORD  NumberOfLinenumbers;
  0x24  DWORD Characteristics;
};
```

Field	Meanings
Name	Actually the name of this field is "name" but the word "name" is an MASM keyword so we have to use "Name1" instead. This member contains the name of the section. Note that the maximum length is 8 bytes. The name is just a label, nothing more. You can use any name or even leave this field blank. Note that there is no mention of the terminating null. The name is not an ASCIIZ string so don't expect it to be terminated with a null.
VirtualAddress	The RVA of the section. The PE loader examines and uses the value in this field when it's mapping the section into memory. Thus if the value in this field is 1000h and the PE file is loaded at 400000h, the section will be loaded at 401000h.
SizeOfRawData	The size of the section's data rounded up to the next multiple of file alignment. The PE loader examines the value in this field so it knows how many bytes in the section it should map into memory.
PointerToRawData	The file offset of the beginning of the section. The PE loader uses the value in this field to find where the data in the section is in the file.
Characteristics	Contains flags such as whether this section contains executable code, initialized data, uninitialized data, can it be written to or read from.

Importing Symbols

- Symbols (functions/data) can be imported from external DLLs
- The loader will automatically load external DLLs
- All the dependencies are loaded as well
- All DLLs will be loaded only once
- External addresses are written to the Import Address Table (IAT)
- IAT is most often located in the .idata section

Example 1: Calling external function

```
CALL ShellExecute
```

Code Section

```
ShellExecute:
        JMP ShellExecute_In_Shell32.DLL
```

.idata Section

Example 2: Calling external function (faster call)

CALL DWORD PTR [ShellExecute]

Code Section

ShellExecute:
    offset of ShellExecute_In_Shell32.DLL

.idata Section

```
__declspec(dllimport) void SomeFunction();
```

Exporting Symbols

- Exporting can be done with ordinals, name or both
- Oridinals are simple index numbers of symbols
- Name is a full ASCII name of the exported symbol
- Exports are described by three simple lists
  - List of ordinals
  - List of symbol addresses
  - List of symbol names
- Exports can be forwarded to another DLL
  - For example: NTDLL.RtlAllocHeap
- Forwarded symbol’s address points to a name in the exports section (.edata)

Resource Directory

- Resources in PE are similar to an archive
- Resource files can be organized into directory trees
- The data structure is quite complex but there are tools to handle it
- Most common resources area
  - Icons
  - Version information
  - GUI resources

Base Relocations

- Sometimes a PE file can not be loaded to its preferred address
- When rebasing, the loader has to adjust all hardcoded addresses
- Base relocations are stored in .reloc section
- Relocations are done in 4KiB blocks (page size on x86)
- Each relocation entry gives a type and points to a location
- The loader calculates the base address difference
- The new address of the memory location is adjusted according to the difference