

# Reverse Engineering III: PE Format

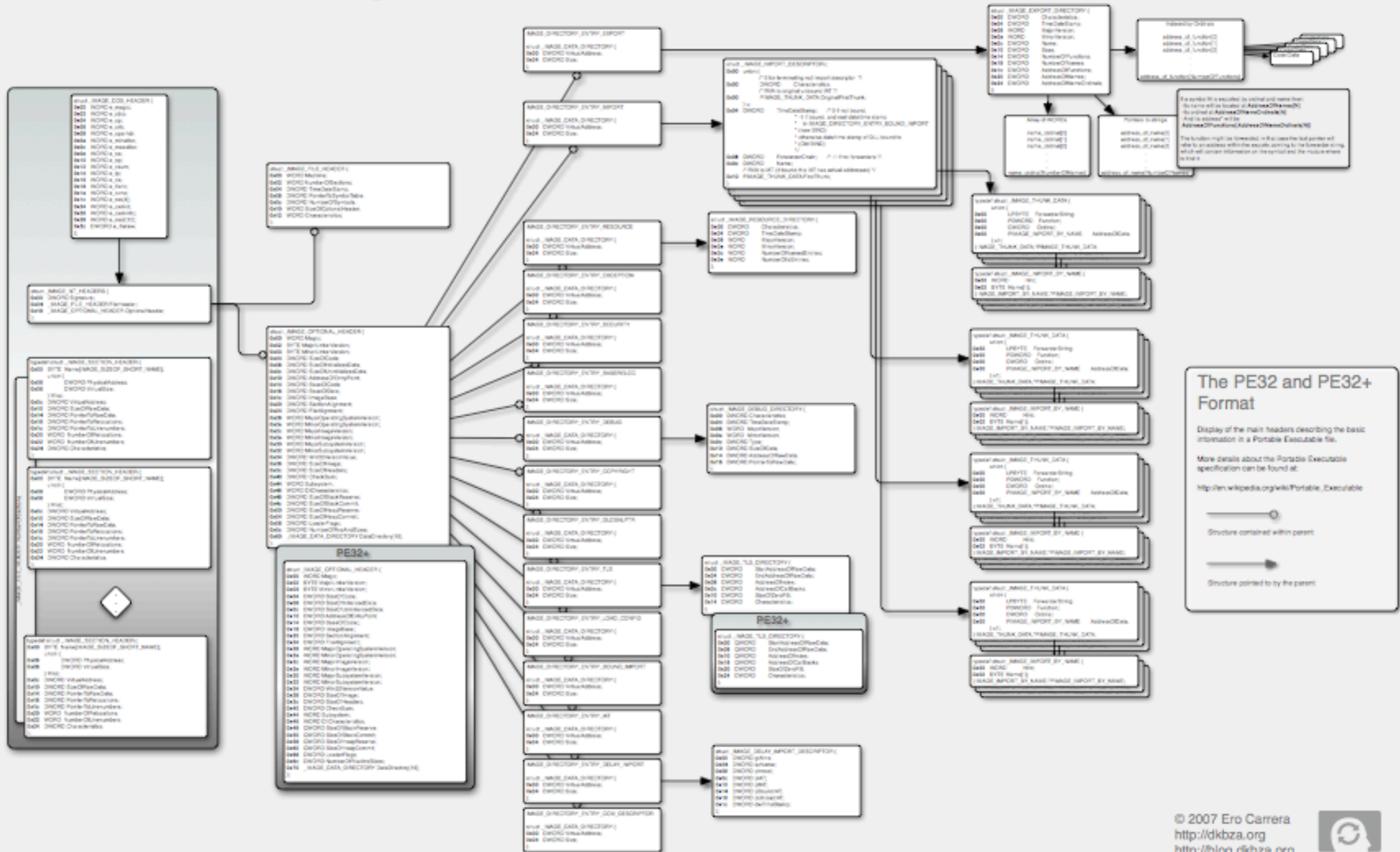
Gergely Erdélyi  
Research Manager



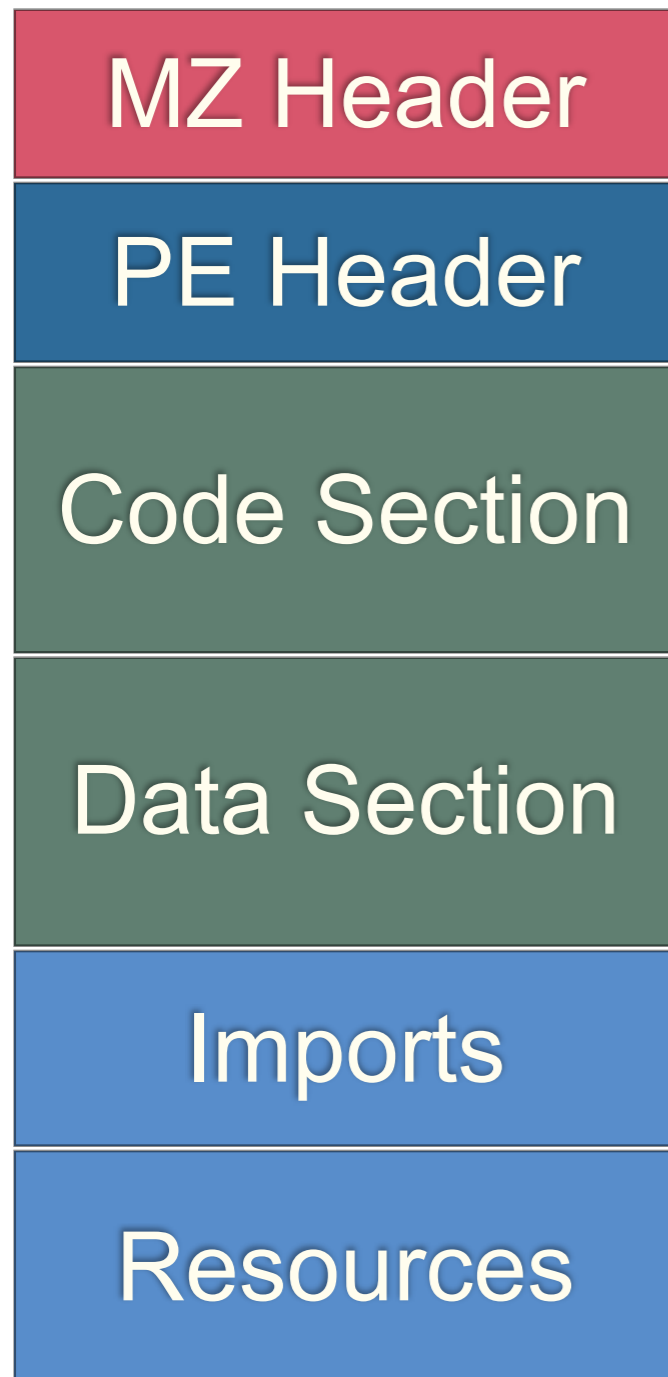
- PE stands for Portable Executable
- Microsoft introduced PE in Windows NT 3.1
- It originates from Unix COFF
- Features dynamic linking, symbol exporting/importing
- Can contain Intel, Alpha, MIPS and even .NET MSIL binary code
- 64-bit version is called PE32+

# Complete Structure of PE

## Portable Executable Format Layout



# File Headers: MZ header



```
struct _IMAGE_DOS_HEADER {  
0x00 WORD e_magic;  
0x02 WORD e_cblp;  
0x04 WORD e_cp;  
0x06 WORD e_crlc;  
0x08 WORD e_cparhdr;  
0x0a WORD e_minalloc;  
0x0c WORD e_maxalloc;  
0x0e WORD e_ss;  
0x10 WORD e_sp;  
0x12 WORD e_csum;  
0x14 WORD e_ip;  
0x16 WORD e_cs;  
0x18 WORD e_lfarlc;  
0x1a WORD e_ovno;  
0x1c WORD e_res[4];  
0x24 WORD e_oemid;  
0x26 WORD e_oeminfo;  
0x28 WORD e_res2[10];  
0x3c DWORD e_lfanew;  
};
```

# File Headers: PE Header

MZ Header

File Header

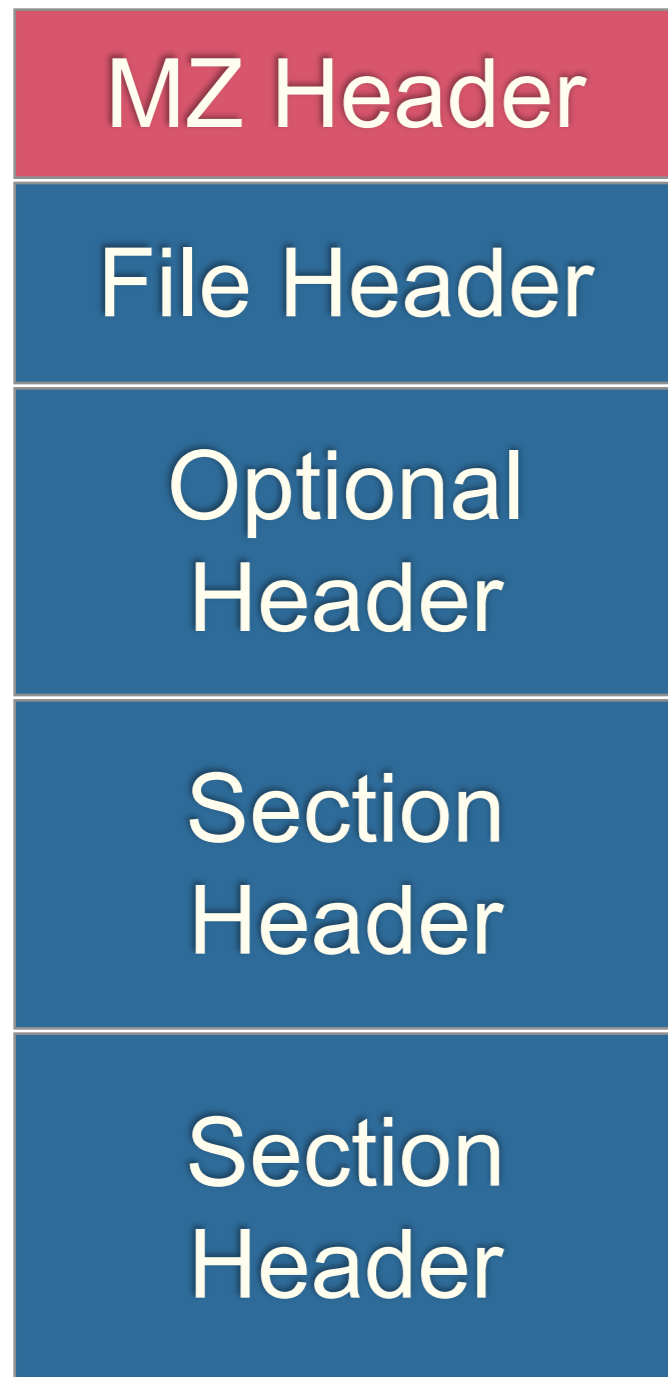
Optional  
Header

Section  
Header

Section  
Header

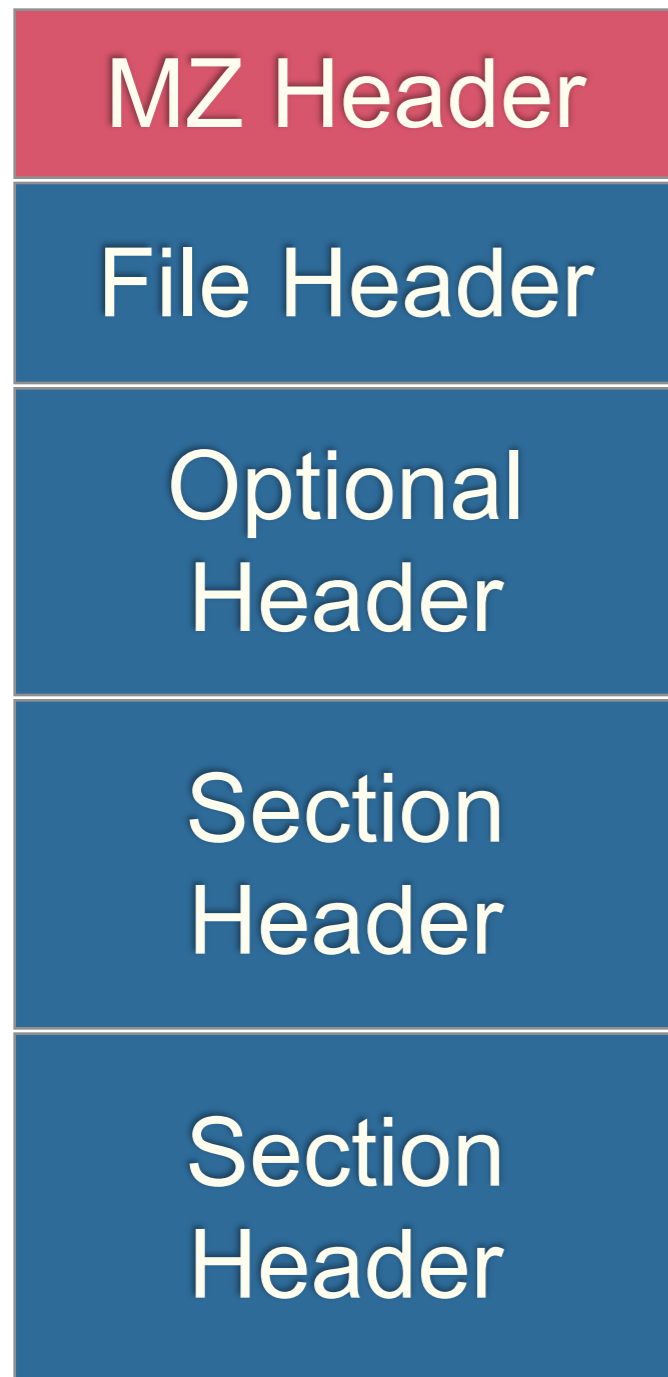
```
struct _IMAGE_NT_HEADERS {  
    0x00  DWORD Signature;  
    0x04  _IMAGE_FILE_HEADER FileHeader;  
    0x18  _IMAGE_OPTIONAL_HEADER OptionalHeader;  
};
```

# File Headers: File Header



```
struct _IMAGE_FILE_HEADER {  
0x00 WORD Machine;  
0x02 WORD NumberOfSections;  
0x04 DWORD TimeDateStamp;  
0x08 DWORD PointerToSymbolTable;  
0x0c DWORD NumberOfSymbols;  
0x10 WORD SizeOfOptionalHeader;  
0x12 WORD Characteristics;  
};
```

# File Headers: Optional Header

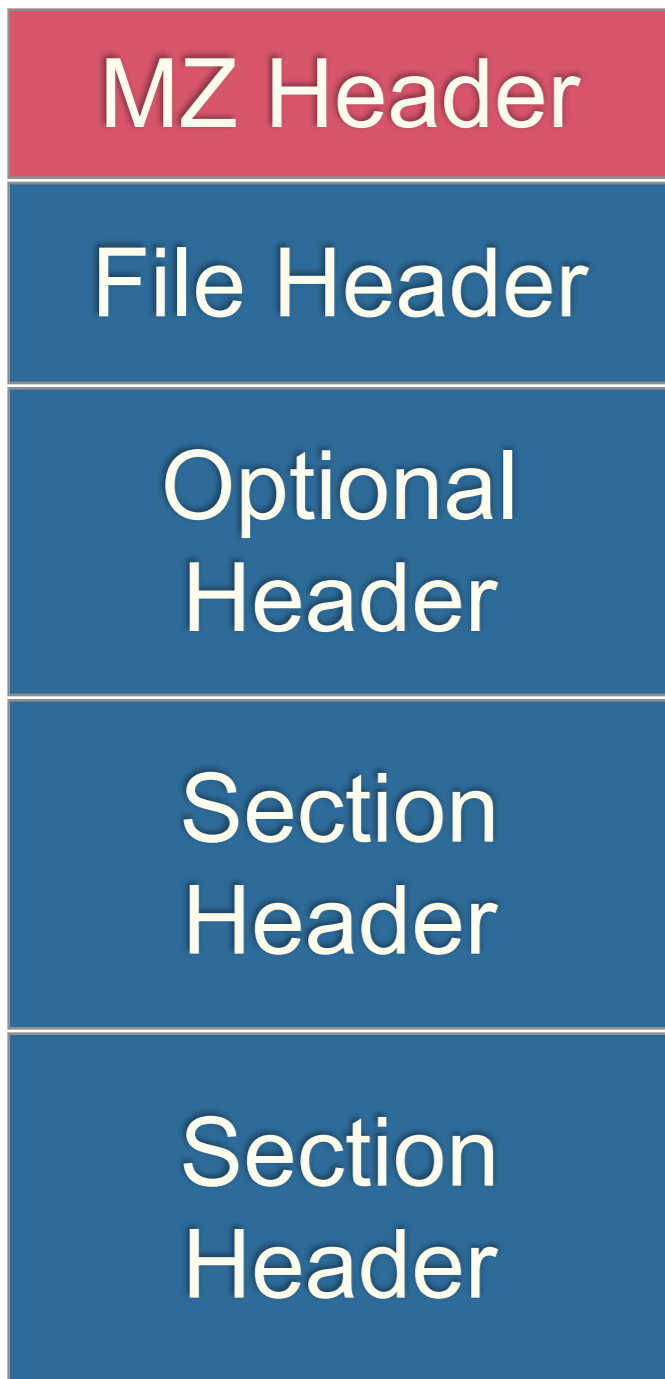


```

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;
0x28 WORD MajorOperatingSystemVersion;
0x2a WORD MinorOperatingSystemVersion;
0x2c WORD MajorImageVersion;
0x2e WORD MinorImageVersion;
0x30 WORD MajorSubsystemVersion;
0x32 WORD MinorSubsystemVersion;
0x34 DWORD Win32VersionValue;
0x38 DWORD SizeOfImage;
0x3c DWORD SizeOfHeaders;
0x40 DWORD CheckSum;
0x44 WORD Subsystem;
0x46 WORD DllCharacteristics;
0x48 DWORD SizeOfStackReserve;
0x4c DWORD SizeOfStackCommit;
0x50 DWORD SizeOfHeapReserve;
0x54 DWORD SizeOfHeapCommit;
0x58 DWORD LoaderFlags;
0x5c DWORD NumberOfRvaAndSizes;
0x60 _IMAGE_DATA_DIRECTORY DataDirectory[16];
};

```

# File Headers: Optional Header

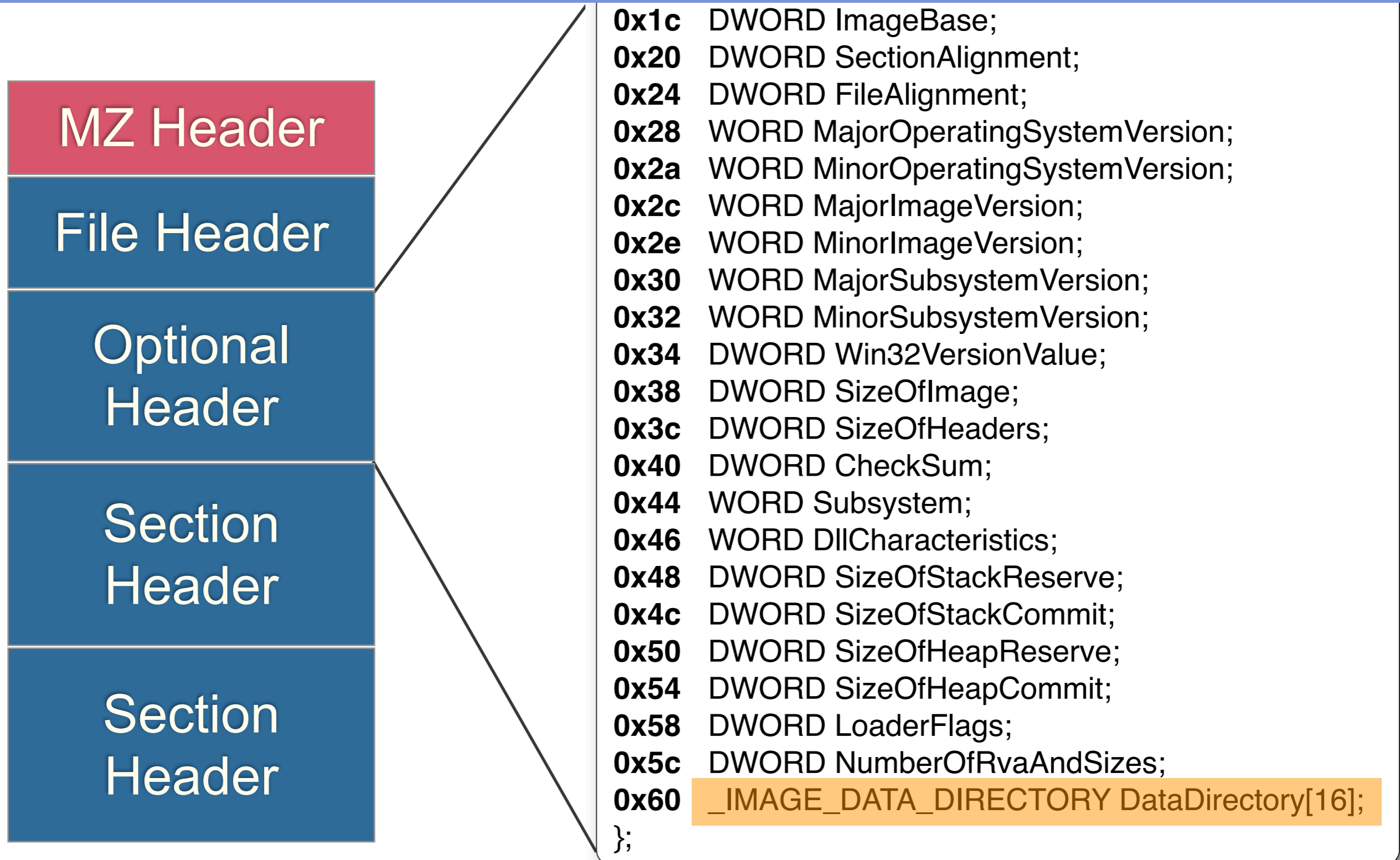


```

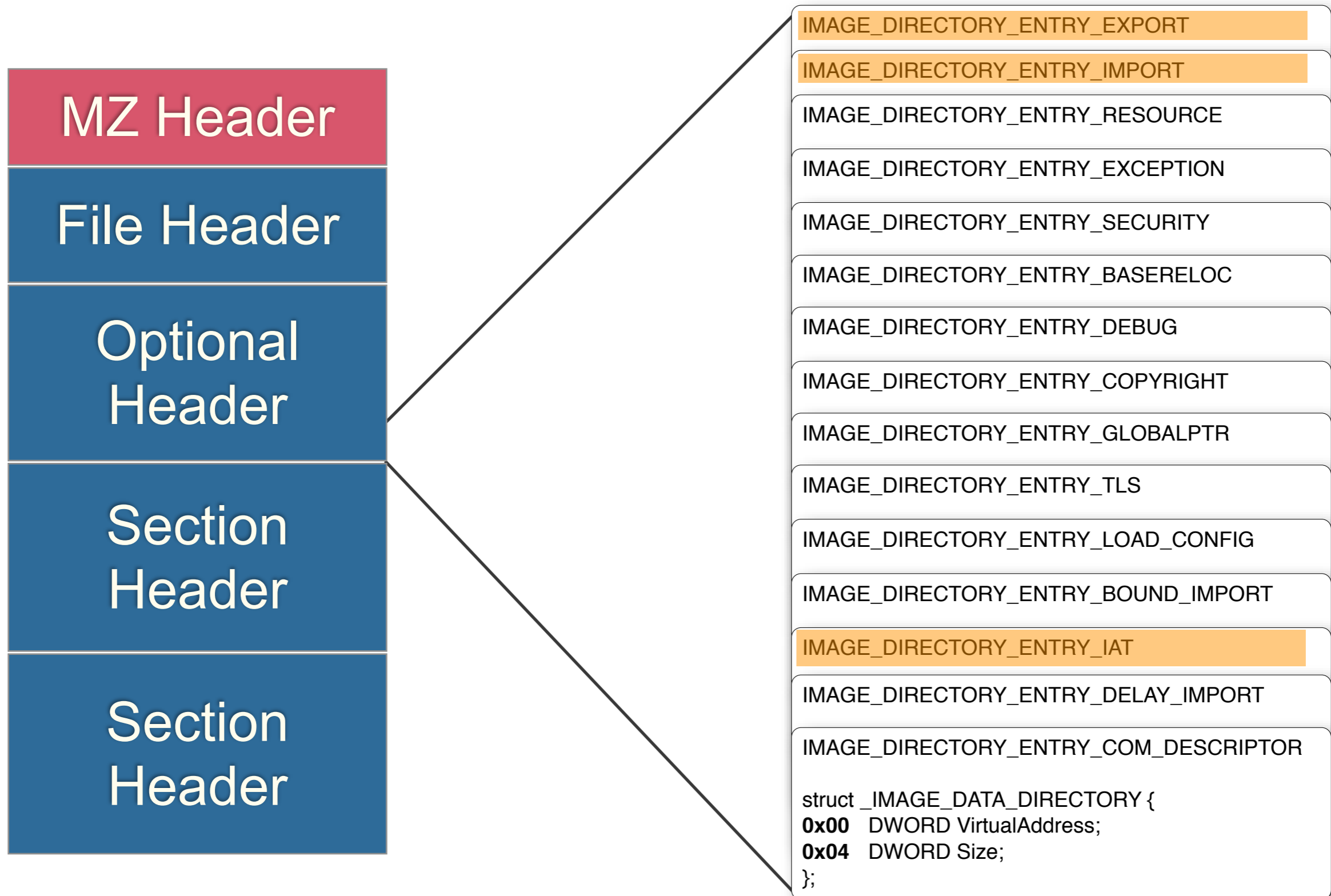
struct _IMAGE_OPTIONAL_HEADER {
0x00  WORD Magic;
0x02  BYTE MajorLinkerVersion;
0x03  BYTE MinorLinkerVersion;
0x04  DWORD SizeOfCode;
0x08  DWORD SizeOfInitializedData;
0x0c  DWORD SizeOfUninitializedData;
0x10  DWORD AddressOfEntryPoint;
0x14  DWORD BaseOfCode;
0x18  DWORD BaseOfData;
0x1c  DWORD ImageBase;
0x20  DWORD SectionAlignment;
0x24  DWORD FileAlignment;
0x28  WORD MajorOperatingSystemVersion;
0x2a  WORD MinorOperatingSystemVersion;
0x2c  WORD MajorImageVersion;
0x2e  WORD MinorImageVersion;
0x30  WORD MajorSubsystemVersion;
0x32  WORD MinorSubsystemVersion;
0x34  DWORD Win32VersionValue;
0x38  DWORD SizeOfImage;
0x3c  DWORD SizeOfHeaders;
0x40  DWORD CheckSum;
0x44  WORD Subsystem;

```

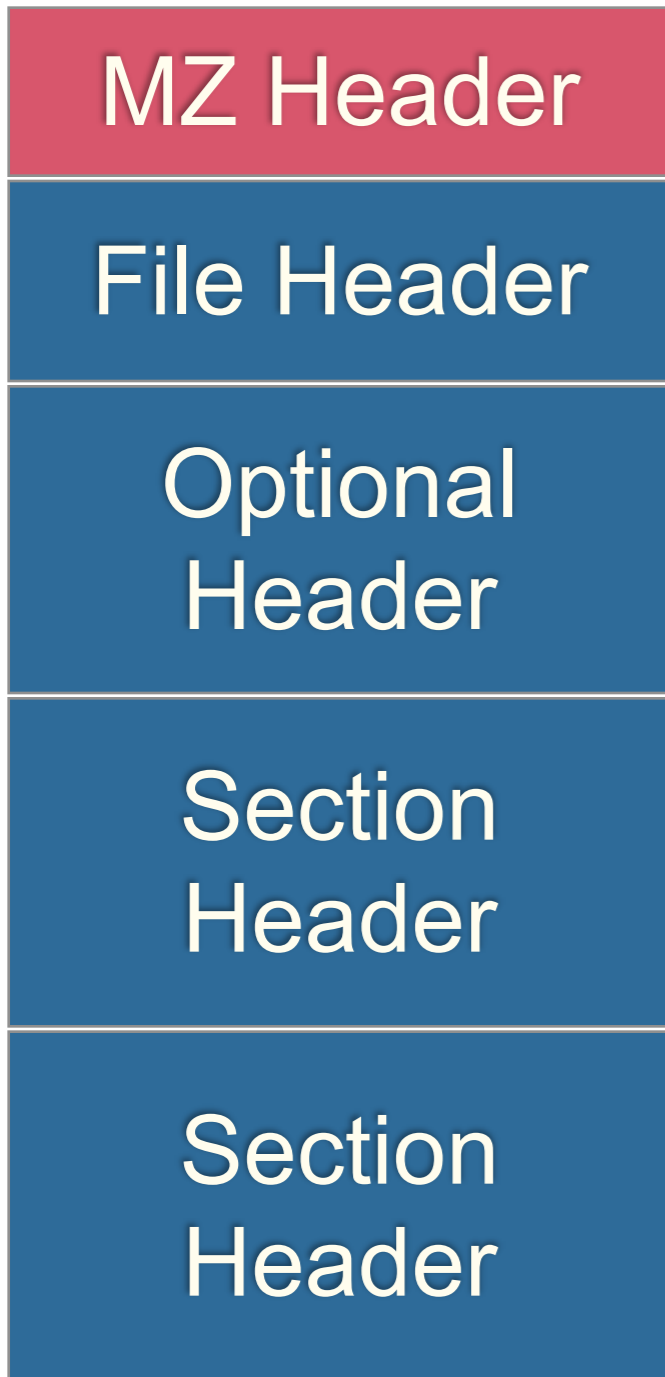
# File Headers: Optional Header



# File Headers: Image Directory

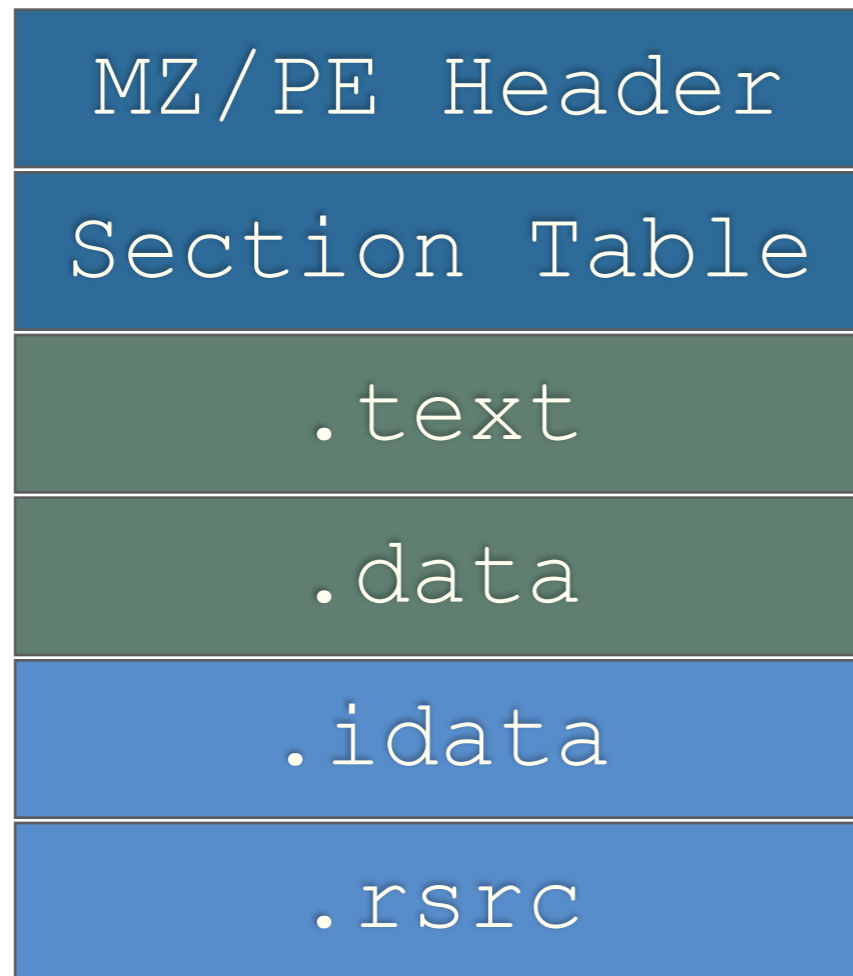


# File Headers: Section Header

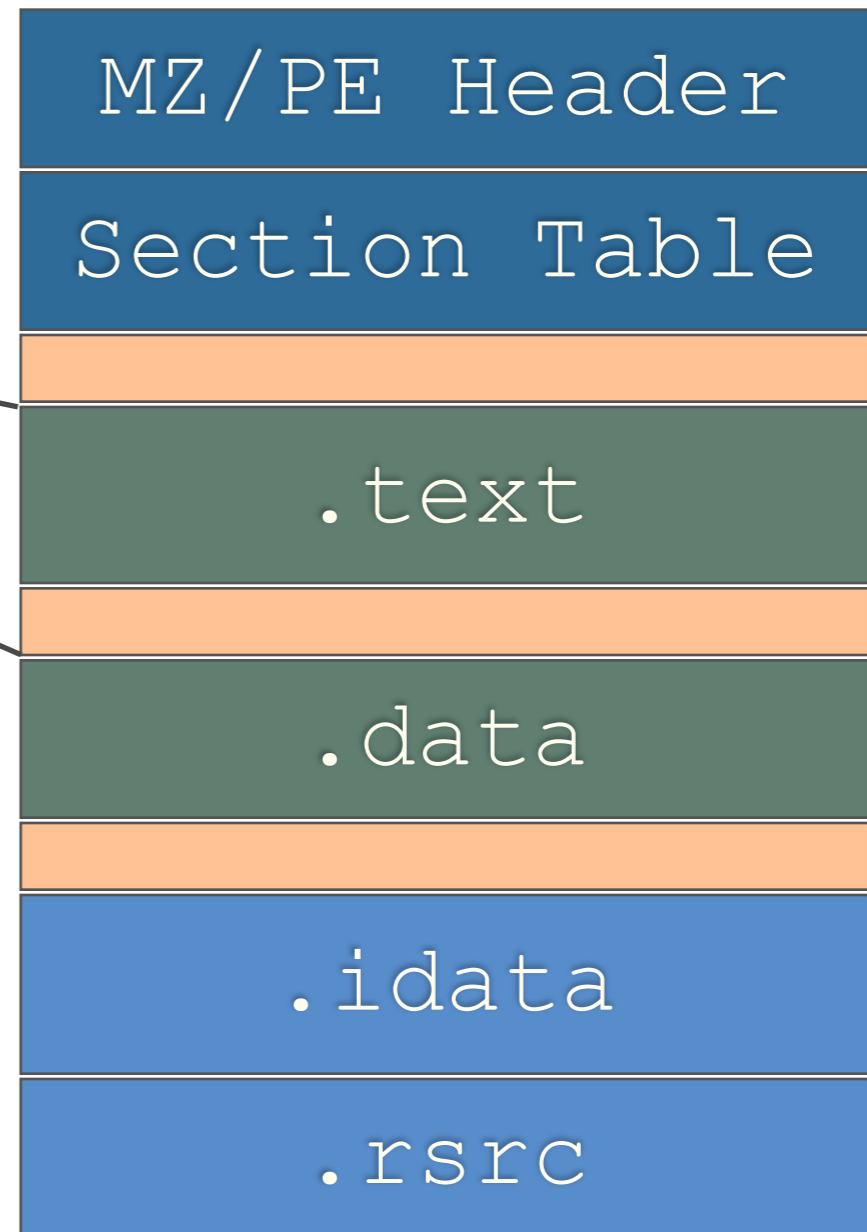


```
typedef struct _IMAGE_SECTION_HEADER {
0x00 BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
0x08     DWORD PhysicalAddress;
0x08     DWORD VirtualSize;
    } Misc;
0x0c     DWORD VirtualAddress;
0x10     DWORD SizeOfRawData;
0x14     DWORD PointerToRawData;
0x18     DWORD PointerToRelocations;
0x1c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

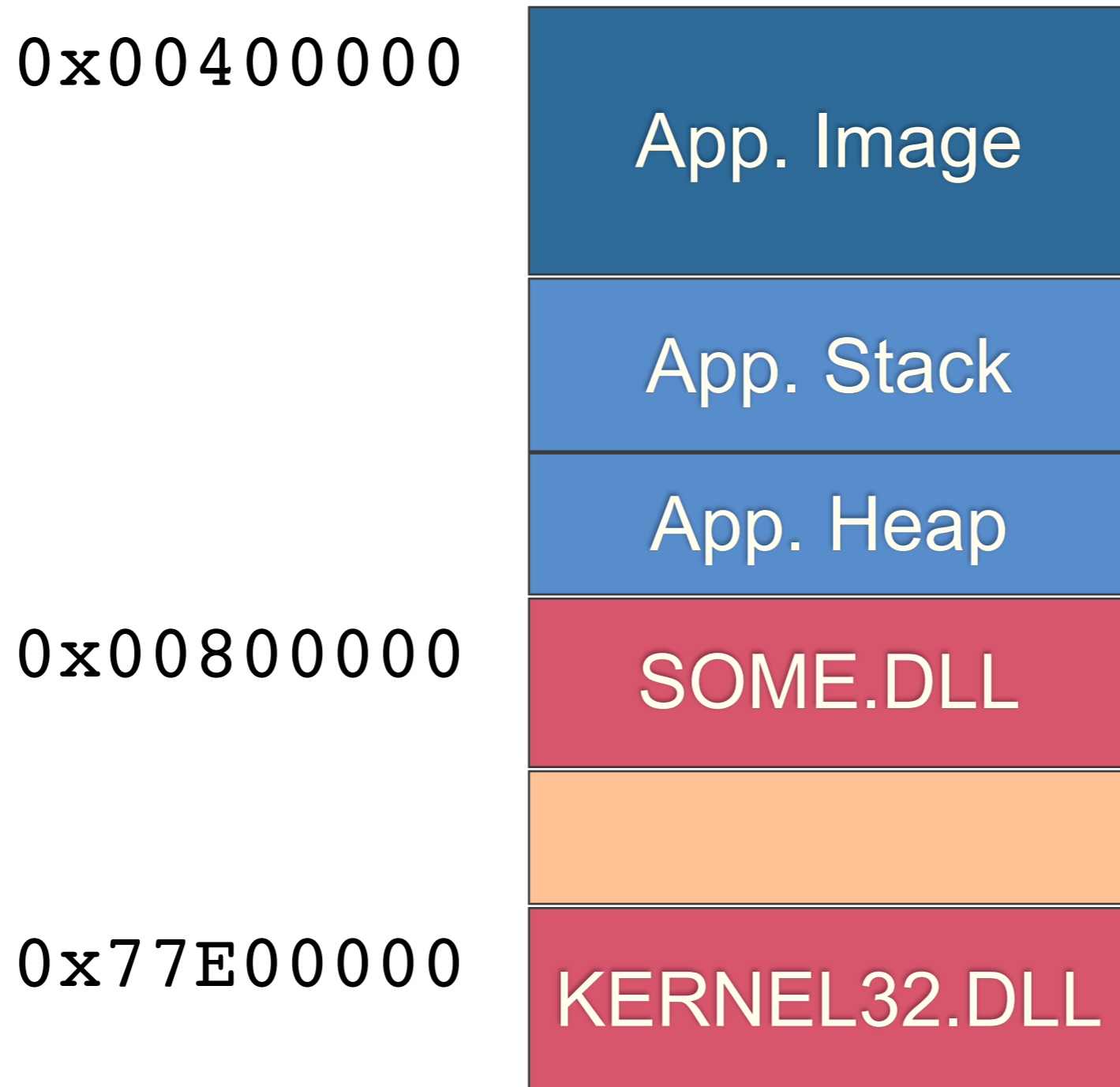
## File image



## Memory Image

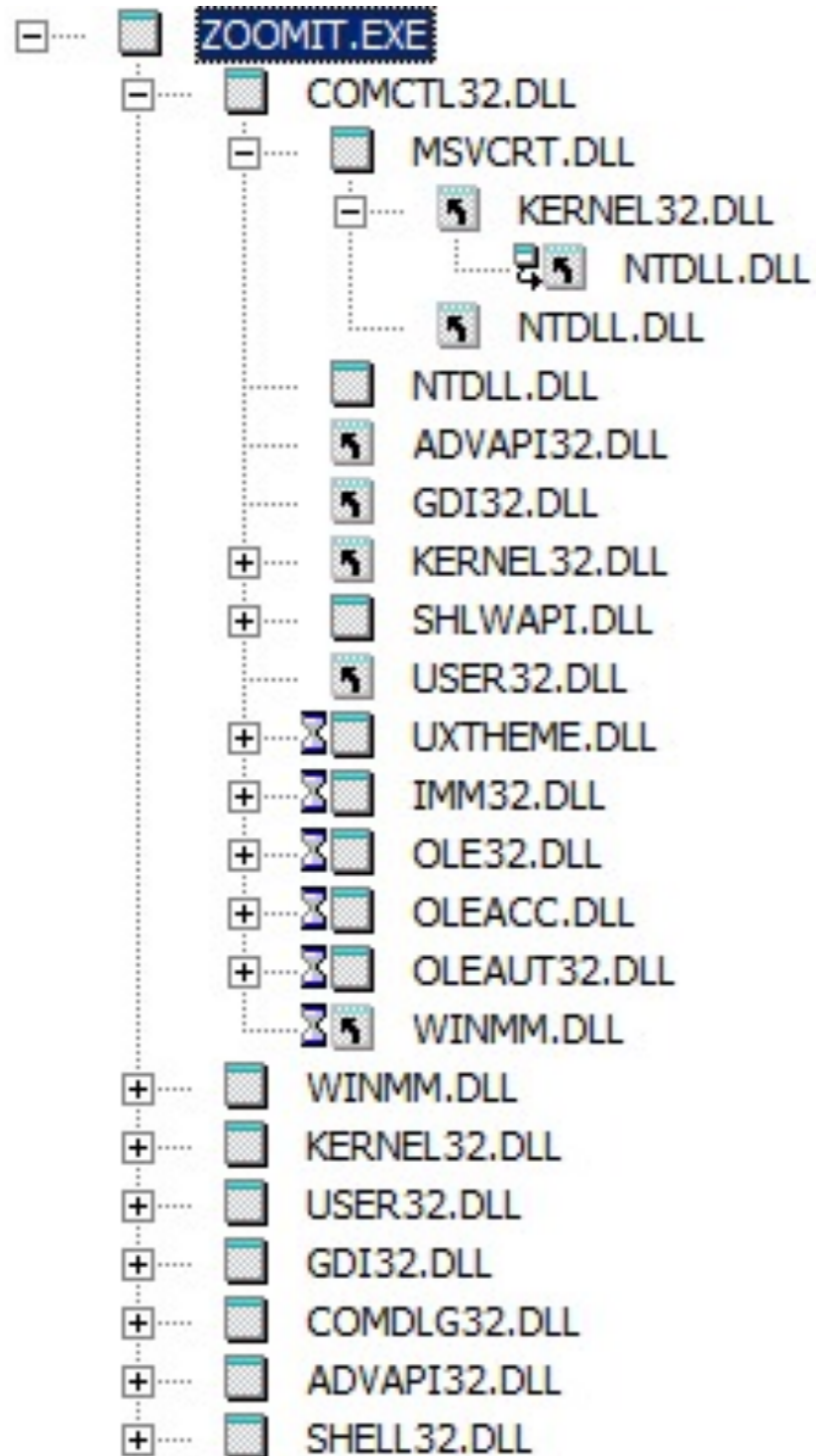


# Memory Layout



- Symbols (functions/data) can be imported from external DLLs
- The loader will load external DLLs automatically
- All the dependencies are loaded as well
- 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

# DLL Dependency Tree



Depends tool shows all dependencies

<http://www.dependencywalker.com/>

- Each DLL has one `IMAGE_IMPORT_DESCRIPTOR`
- The descriptor points to two parallel lists of symbols to import
  - Import Address Table (IAT)
  - Import Name Table (INT)
- The primary list is overwritten by the loader, the second one is not
- Executables can be pre-bound to DLLs to speed up loading
- Symbols can be imported by ASCII name or ordinal

## .idata section

```
IMPORT_DESCRIPTOR:
```

```
USER32.DLL
```

```
ADVAPI32.DLL
```

```
Import Name Table (INT):
```

```
`IMPORT1`
```

```
`IMPORT2`
```

```
Import Address Table (IAT):
```

```
DWORD IMPORT1
```

```
DWORD IMPORT2
```

Calling:

```
CALL [IMPORT1]
```

...

```
CALL [IMPORT2]
```

# Import Descriptors

IMAGE\_DIRECTORY\_ENTRY\_IMPORT

```
struct _IMAGE_DATA_DIRECTORY {  
0x00 DWORD VirtualAddress;  
0x04 DWORD Size;  
};
```

```
struct _IMAGE_IMPORT_DESCRIPTOR {  
0x00 union {  
/* 0 for terminating null import descriptor */  
0x00 DWORD Characteristics;  
/* RVA to original unbound IAT */  
0x00 PIMAGE_THUNK_DATA OriginalFirstThunk;  
} u;  
0x04 DWORD TimeDateStamp; /* 0 if not bound,  
0x08 DWORD ForwarderChain; /* -1 if no forwarders */  
0x0c DWORD Name;  
/* RVA to IAT (if bound this IAT has actual addresses) */  
0x10 PIMAGE_THUNK_DATA FirstThunk;  
};
```

```
typedef struct _IMAGE_THUNK_DATA {  
union {  
0x00 LPBYTE ForwarderString;  
0x00 PDWORD Function;  
0x00 DWORD Ordinal;  
0x00 PIMAGE_IMPORT_BY_NAME AddressOfData;  
} u1;  
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```

```
typedef struct _IMAGE_IMPORT_BY_NAME {  
0x00 WORD Hint;  
0x02 BYTE Name[1];  
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

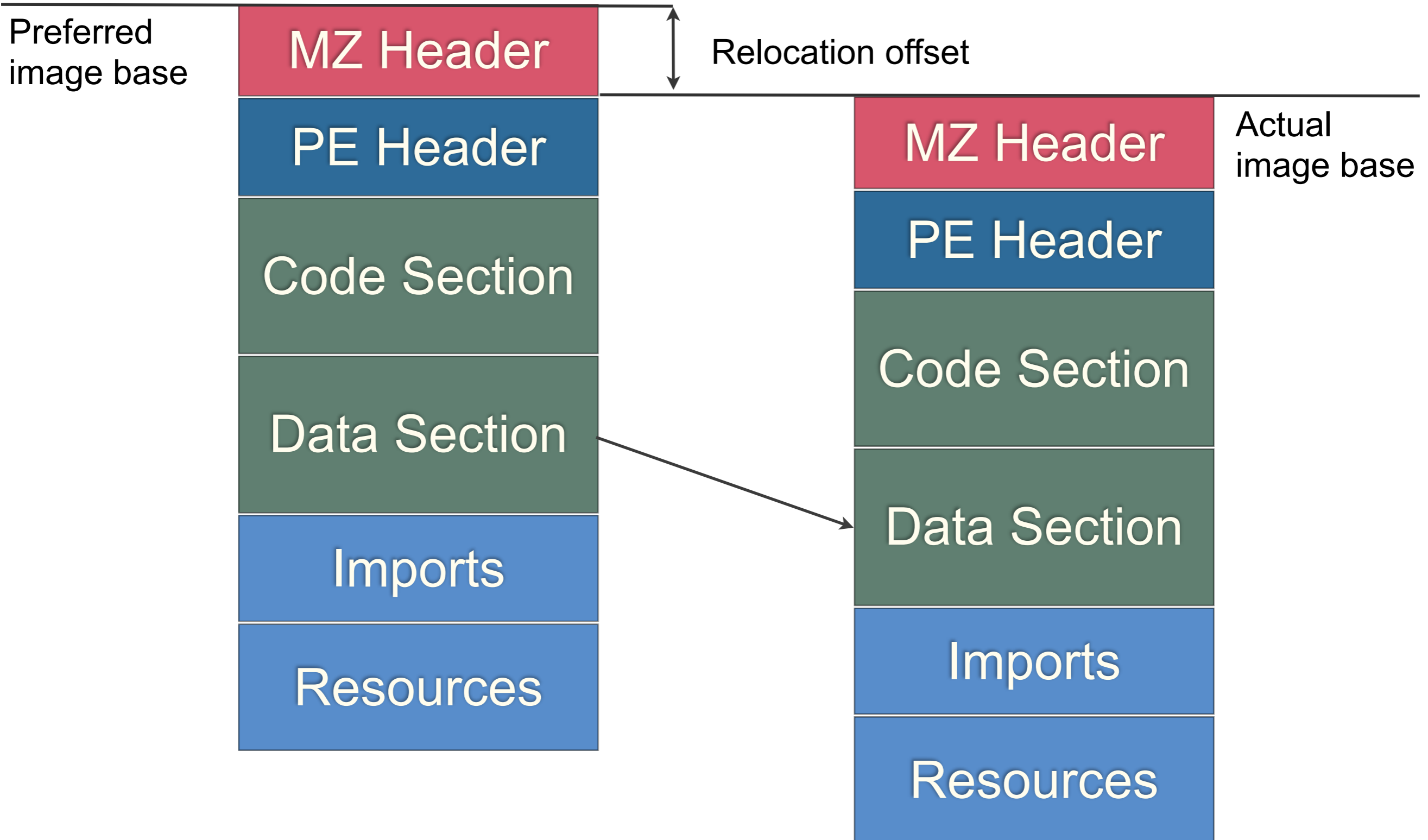
# Exporting Symbols



- Symbols can be exported with ordinals, names or both
- Ordinals 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
  - Example: `NTDLL.RtlAllocHeap`

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

# Base Relocation



- Sometimes a DLL can not be loaded to its preferred address
- When rebasing, the loader has to adjust all hardcoded addresses
- 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 offsets are adjusted according to the difference

## Hex Editors

- HT Editor at <http://hte.sourceforge.net/>
- BIEW at <http://biew.sourceforge.net/en/biew.html>
- Hiew at <http://www.hiew.ru/>

## Programmatic Access

- pefile – python module at <http://dkbza.org/pefile.html>

Matt Pietrek: An In-Depth Look into PE:

<http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx>

<http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx>

Microsoft Portable Executable and Common Object File Format  
Specification

<http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>

Special thanks to Ero Carrera for his beautiful PE diagrams and the permission to use them in this presentation!

[https://www.openrce.org/reference\\_library/files/reference/PE%20Format.pdf](https://www.openrce.org/reference_library/files/reference/PE%20Format.pdf)

<http://blog.dkbza.org/2007/03/tiny-and-crazy-pe.html>