Formatul fișierelor PE (Portable Executable Files)

CURS NR. 9 SI 10

Fișiere PE (Portable Executable)

Formatul fișierelor executabile

- **Portabil** se referă la faptul că formatul nu este specific unei arhitecturi anume (toate SO Windows, pe orice platformă folosesc PE)
- Se aplică asupra fișierelor cu extensiile: .exe, .dll, .ocx, .drv, și altele
- Suport pentru PE32 și PE32+ (pentru x64)

Fișiere imagine și Fișiere obiect

- **Fișier imagine** (image file) - un alt nume pentru fișier executabil
- **Fișier obiect** OBJ – fișier rezultat în urma compilării programului – au formatul COFF (Common Object File Format)

Formatul fișierelor PE este complex

- Cuprind mai multe header-e și secțiuni care indică linkeditorului dinamic cum să mapeze fișierul executabil în memorie

Fișierul executabil cuprinde mai multe regiuni

- Fiecare cu cerințe specifice pentru protecția memoriei
  - Ex. Segmentul de cod, denumit .text se mapează cu drepturi de citire/execuție
    Segmentul de date, denumit .data este mapat cu drepturi de citire-scriere/fără execuție
- La încărcare fiecare regiune trebuie aliniată cu începutul unei pagini de memorie

Conceptul de modul

- se referă aici la cod, date și alte resurse ale fișierului executabil sau DLL încărcat în memorie
- Un modul conține pe lângă cod și date o serie de structuri de date care permit gestiunea acelui modul de către SO
  - De ex. indică unde este modulul încărcat în memorie, etc.

Formatul fișierelor PE

Format definit în fișierul header WinNT.h

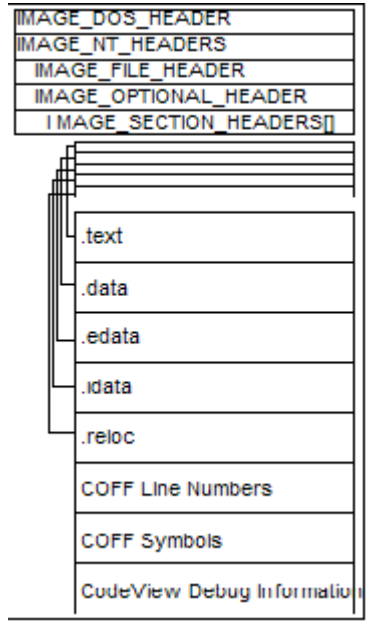
- Conține definițiile structurilor de date folosite în fișierele PE

Formatul fișierului executabil încărcat în memorie este foarte similar cu imaginea executabilului de pe disc

- Încărcătorul folosește mecanismul de mapare a fișierelor în memorie pentru încărcarea diferitelor părți ale fișierului executabil în spațiul de adrese al procesului

Formatul fișierelor PE este organizat într-un singur flux de date într-o zonă contiguă de memorie și include

- Cod, date, tabele de import-uri, tabela de exporturi, alte structuri ale modulului



- Structura este impusă de o colecție de câmpuri care sunt la locații bine definite (sau ușor de găsit) și care specifică organizarea fișierului (adresele de început ale diferitelor header-e și secțiuni)
  - Dacă se cunoaște adresa de bază unde s-a încărcat modulul, toate părține componente pot fi regăsite folosind pointerii stocați în fișierul PE care indică adresele de început ale diferitelor secțiuni

Concepte: secțiune, adresă de bază, adresă relativă

Secțiune

- Unitate de structură pentru cod și date
- Poate conține cod sau date
  - Declarate și utilizate direct de aplicație sau
  - Existente în biblioteci sau create de linkeditor pentru gestiunea executabilului
- Similar cu conceptul de segment, dar secțiunea este o zonă contiguă de memorie fără limite de dimensiune

- Fișierul PE include cel puțin 2 secțiuni – unul pentru date și altul pentru cod
- O aplicație Windows NT are 9 secțiuni predefinite
  - .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, .debug
  - Unele aplicații necesită mai puține secțiuni, altele pot defini mai multe (max 96)

- Secțiunile care apar cel mai frecvent:
  - Secțiunea de cod executabil: .text
  - Secțiuni de date: .data, .rdata, .bss
  - Secțiune de resurse: .rsrc
  - Secțiune de exporturi: .edata
  - Secțiune de importuri: .idata
  - Secțiune de informații debug: .debug

Concepte: secțiune, adresă de bază, adresă relativă

Adresă de bază

- Locația de memorie din cadrul spațiului de adrese virtual unde s-a încărcat executabilul

Adresă virtuală relativă (RVA – Relative Virtual Address)

- Referințele din cod - la cod și date – sunt calculate în funcție de adresa de bază
  - **poziția relativă față de adresa de bază unde s-a încărcat modulul**
- Fișierele PE de regulă conțin adrese relative la o adresă de bază preferată (adresa de bază preferată este hardcodată în executabil)

- Exemplu: dacă încărcătorul mapează fișierului PE la adresa de bază 0x10000 în spațiul de adrese virtual (al procesului), și avem o variabilă la adresa relativă 0x464 atunci locația variabilei în cadrul spațiului de adrese virtual este de fapt adresa 0x10464

```
(Virtual address 0x10464)-(base address 0x10000) = RVA 0x00464
```

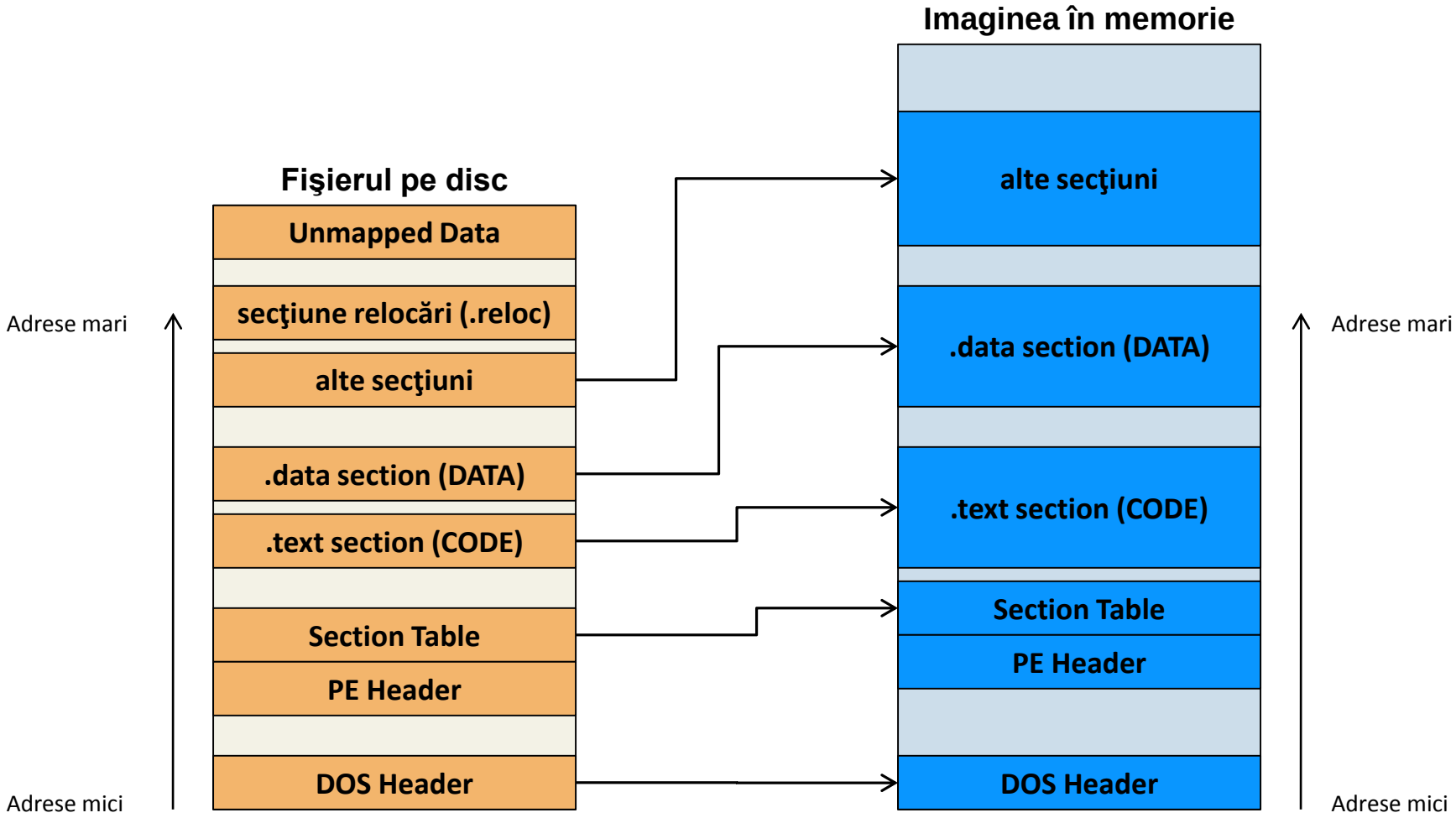
Pentru a converti RVA la un pointer utilizabil trebuie să-i adăugăm adresa de bază

File Address (FA) sau File pointer

- Locație din cadrul fișierului de pe disc – nu din imaginea mapată
- De regulă FA și RVA sunt valori distincte
  - Adresa unui element din spațiul de adrese virtual este diferită de locația relativă a acestui element în cadrul fișierului executabil

Fișierul executabil – pe disc și încărcat în memorie

Observăm că dimensiunile secțiunilor pot diferi – datorită modului de aliniere la încărcare



Relocare

Relocarea unui modul

- Dacă PE nu poate fi încărcat la adresa preferată (aceasta fiind ocupată), atunci SO trebuie să îl relocheze prin asignarea unei alte adrese de bază
  - Relocarea implică recalcularea adreselor din cod
    - Se calculează diferența dintre adresa de bază preferată și cea alocată și diferența se adaugă la fiecare referință din cod
- Efectul secundar al relocării
  - Codul mapat prin relocare este propriu procesului – nu este partajabil
    - Se pierd beneficiile dll-urilor
  - Încetinește semnificativ procesul de încărcare a modulului

Relocare

- Evitarea relocării
  - Ex. dll-urile din pachetul Microsoft au adrese de bază precalculate astfel încât să nu existe suprapunere

- Facilitarea relocării
  - fișierele PE conțin informații de relocare – numite și informații de **fixup**

  - Exemplu:
    - Considerăm două procese P1 și P2
    - P1 are modulul M1 încărcat în spațiul său de adrese la adresa de bază implicită lui M1
    - P2 dorește să încarce modulul M1 dar adresa implicită al lui M1 este deja ocupată în spațiul de adrese al lui P2 de un alt modul
      - M1 trebuie relocat în spațiul lui P2 → necesită modificări asupra codului relocabil din M1
        - SO creează o copie a modulului M1 în care poate efectua fixup-urile și mapează acest modul în P2 (folosind copy-on-write)
    - Dezavantaje: necesită timp de procesare și spațiu fizic adițional și împiedică partajarea modulului

Linkeditare

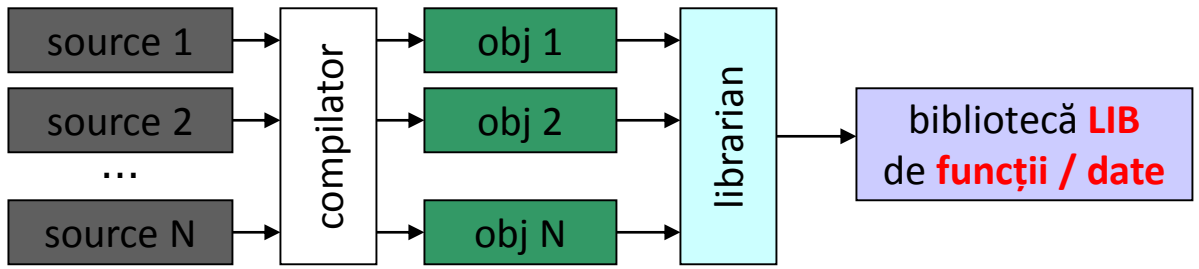
Linkeditorul

- Pregătește programul pentru a fi încărcat în memorie și lansat în execuție
  - inserează cod (sau mapează biblioteci partajate) pentru rezolvarea referințelor externe (către biblioteci) și / sau
  - combină codurile obiect (rezultate în urma etapei de compilare) într-o imagine executabilă

- Editarea legăturilor se poate efectua la compilare, la încărcare sau în timpul rulării

Linkeditare statică

- Linkeditorul **copiază toate modulele de bibliotecă folosite de program în imaginea executabilă** – imediat după etapa de compilare – o singură dată
- Cel mai frecvent mod de linkeditare
  - Dezavantaje
    - Necesită mai **mult spațiu** pe disc și în memorie (conține atât programul apelant cât și modulele apelate)
    - Dacă referințele externe se modifică este necesară **recompilarea și re-editarea legăturilor**
  - Avantaje:
    - Necesită timp constant pentru încărcare și este mai **rapidă**
    - **Portabilitate** mai bună – deoarece nu necesită prezența bibliotecii folosite pe sistemul pe care se rulează aplicația



- Funcțiile și datele exportate sunt incluse într-un LIB
- Biblioteca poate fi distribuită fără cod sursă – doar fișiere header + LIB
- Adresa relativă dintre două funcții este determinată în timpul linkeditării - și rămâne aceeași pentru fiecare rulare

Linkeditare

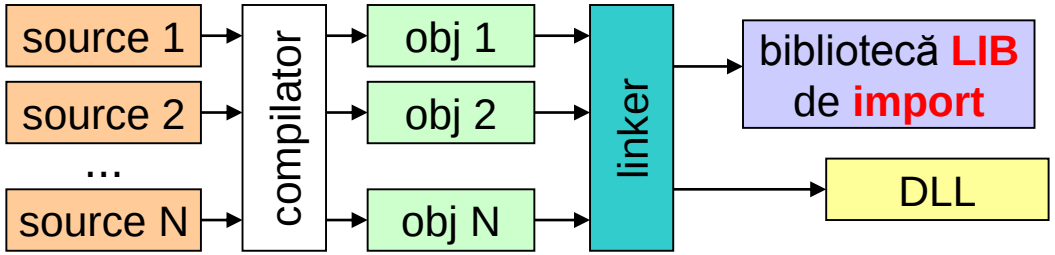
Linkeditare dinamică

- Plasează numele bibliotecilor referite (biblioteci partajabile) în imaginea executabilă
- **Legarea efectivă** cu rutinele de bibliotecă se face **doar la încărcare sau rulare** – când atât executabilul cât și biblioteca sunt încărcate în memorie
  - Dezavantaje
    - Programele care folosesc biblioteci partajate sunt de regulă **mai lente** decât cele care folosesc bilioteci legate static
    - Dependența de existența biliotecii necesare pe sistemul pe care se rulează aplicația
  - Avantaje
    - mai multe programe pot **partaja** o singură copie a bibliotecii încărcată în memorie
    - Dacă referințele externe se modifică, **doar modulul** extern partajat trebuie **recompilat**
    - **Timpul de încărcare variază** – poate fi redus dacă biblioteca referită este deja încărcată în memorie

Implicite linking

Linkeditare dinamică în timpul rulării

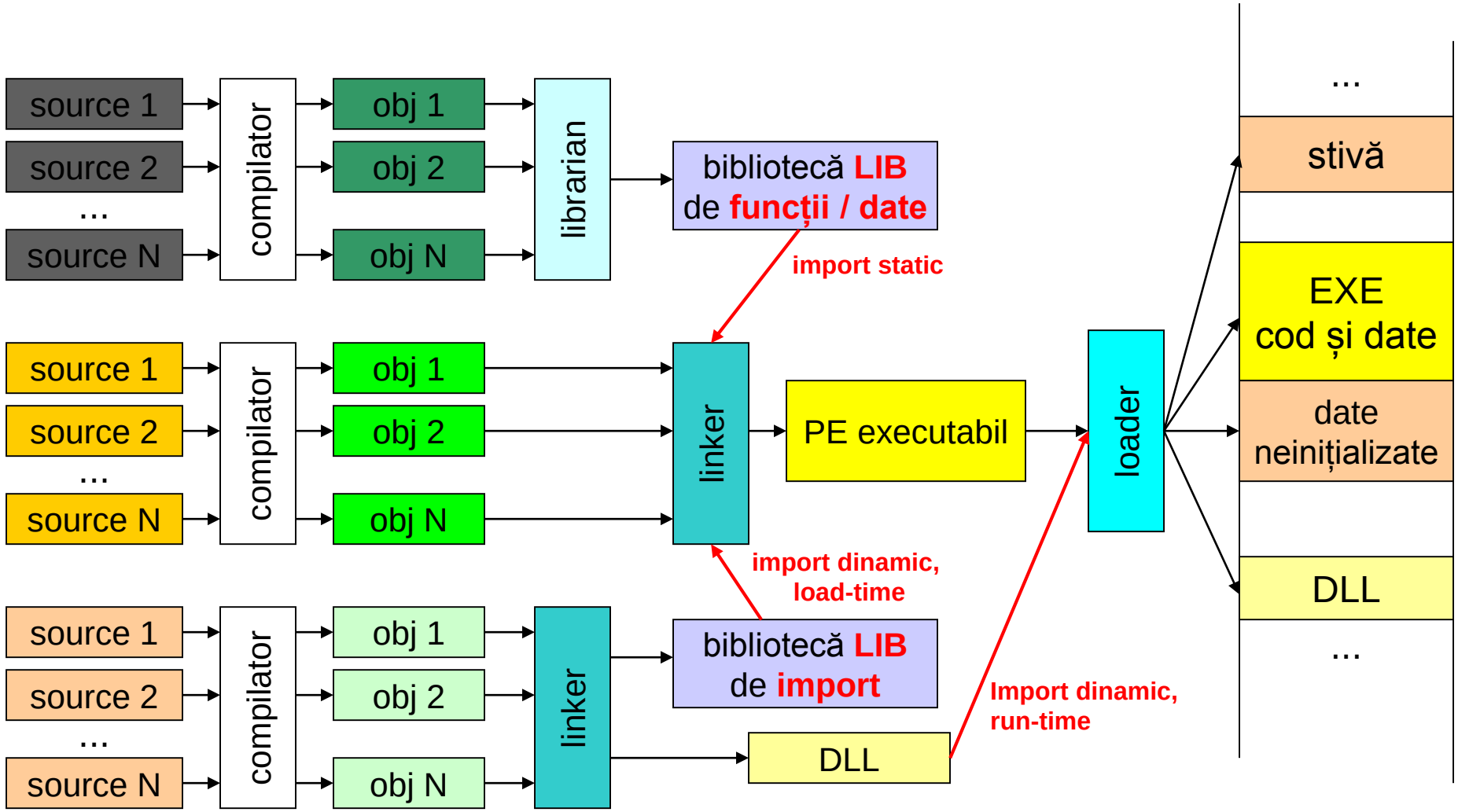
- Funcțiile și datele exportate sunt incluse într-un DLL
- Nu este necesar **import LIB**
- Este necesară încărcarea explicită a DLL-ului prin LoadLibrary
- Adresa relativă dintre două funcții este determinată în timpul rulării - și este variabil în timpul aceleiași rulări



Linkeditare dinamică în timpul încărcării

- Funcțiile și datele exportate sunt incluse într-un DLL
- În timpul linkeditării executabilul se creează folosind un **import LIB**
  - Apeluri de funcții prin tabela de importuri
- Adresa relativă dintre două funcții este determinată la încărcare - și rămâne aceeași doar pentru o rulare

Compilarea, linkeditarea și încărcarea executabilelor



Structura fișierelor PE

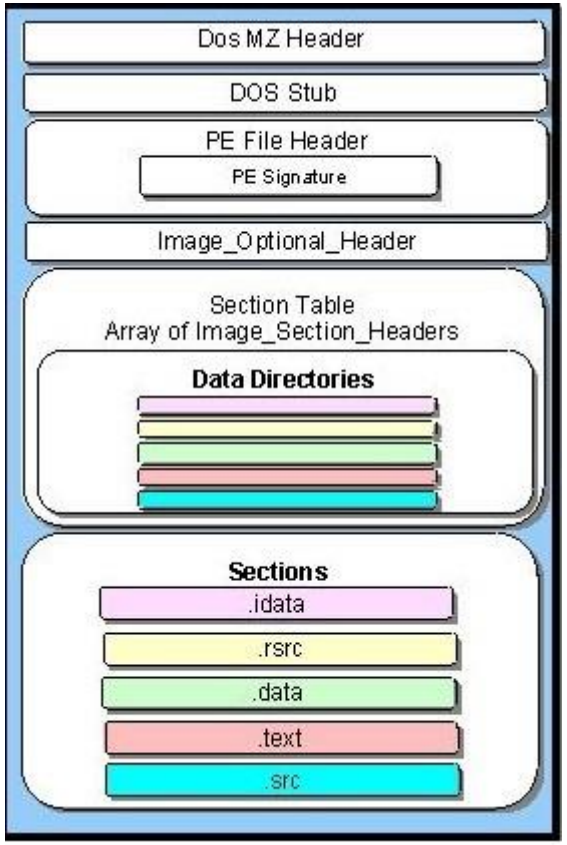
- **MS-DOS Header**
  - Pentru ca fișierul să fie citibil și de SO Windows mai vechi - și înțeles că nu sunt compatibile
    - Ex. Un executabil Win32 rulat sub MS-DOS v.6 afișează “This program cannot be run in DOS mode.”
  - Structura de 64 de octeți:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
   USHORT e_magic;         // Magic number
   USHORT e_cblp;          // Bytes on last page of file
   USHORT e_cp;            // Pages in file
   USHORT e_crlc;          // Relocations
   USHORT e_cparhdr;       // Size of header in paragraphs
   USHORT e_minalloc;      // Minimum extra paragraphs needed
   USHORT e_maxalloc;      // Maximum extra paragraphs needed
   USHORT e_ss;            // Initial (relative) SS value
   USHORT e_sp;            // Initial SP value
   USHORT e_csum;          // Checksum
   USHORT e_ip;            // Initial IP value
   USHORT e_cs;            // Initial (relative) CS value
   USHORT e_lfarlc;        // File address of relocation table
   USHORT e_ovno;          // Overlay number
   USHORT e_res[4];        // Reserved words
   USHORT e_oemid;         // OEM identifier (for e_oeminfo)
   USHORT e_oeminfo;       // OEM information; e_oemid specific
   USHORT e_res2[10];      // Reserved words
   LONG   e_lfanew;        // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Număr magic – identifică tipurile de fișiere compatibile MS-DOS
- Executabilele MS-DOS au setată valoarea 0x5A4D, adică caracterele ASCII MZ – de aici și numele

e_lfanew este offsetul (FA – file address) unde începe header-ul PE

```
// Ignoring typecasts and pointer conversion issues for clarity...
pNTHeader = dosHeader + dosHeader->e_lfanew;
```



- **Real-mode program stub**
  - Programul rulat de MS-DOS la încărcarea executabilului
    - O simplă afișare a unui mesaj: "This program requires Microsoft Windows v3.1 or greater.„
    - Programatorul poate schimba stub-ul să afișeze altceva – mai sugestiv sau specific pt aplicație

Structura fișierelor PE

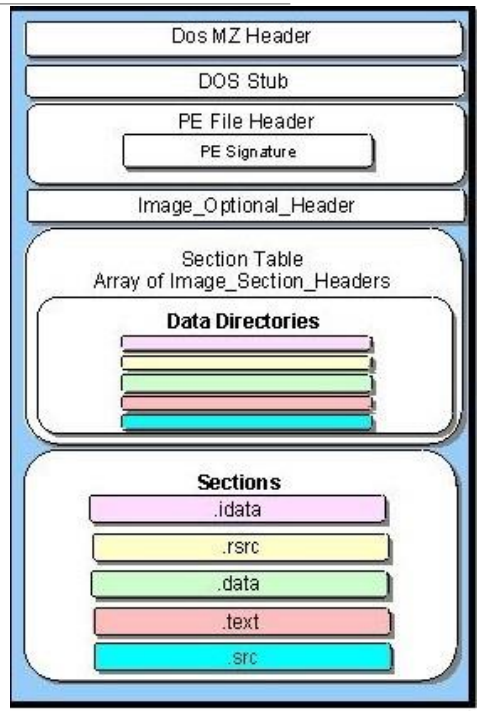
- **PE file header & PE file signature**
  - Regăsirea header-ului: offsetul (FA) se găsește în componenta `e_lfanew` al headerului DOS
  - Structura header-ului PE:

```
typedef struct _IMAGE_NT_HEADERS {
   DWORD Signature; /* "PE"\0\0 */    /* 0x00 */
   IMAGE_FILE_HEADER FileHeader;      /* 0x04 */
   IMAGE_OPTIONAL_HEADER OptionalHeader;   /* 0x18 */
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Semnătura indentifică tipul fișierului

```
#define IMAGE_DOS_SIGNATURE     0x5A4D      /* MZ   */
#define IMAGE_OS2_SIGNATURE     0x454E      /* NE   */
#define IMAGE_OS2_SIGNATURE_LE  0x454C      /* LE   */
#define IMAGE_OS2_SIGNATURE_LX  0x584C      /* LX */
#define IMAGE_VXD_SIGNATURE     0x454C      /* LE   */
#define IMAGE_NT_SIGNATURE      0x00004550 /* PE00 */
```

Pentru Win32 semnătura este “PE\0\0”



  - Structura IMAGE_FILE_HEADER
    - Aceeași structură apare la începutul fișierelor OBJ în formatul COFF – rezultate după compilare

```
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;
```

Procesorul pe care poate rula aplicația

Numărul de secțiuni

Timpul la care a fost produs fișierul – de linkeditor (sau compilator pt. COFF)

Adresa relativă (offsetul) la care se află tabela de simboluri COFF

Numărul de simboluri din tabela de simboluri COFF

Dimensiunea header-ului opțional (pt. PE nu este opțional, pt COFF este 0)

Flaguri cu informații despre fișier

  - Exemple de id-uri definite pentru procesoare (Machine):
    - 0x014c – Intel i386
    - 0x8664 – AMD64, x86-64
    - 0x0200 – Intel IA64
    - .....
  - Câteva flaguri importante (Characteristics):
    - 0x0001 – nu sunt info de relocare
    - 0x0002 – fișier executabil – nu OBJ sau DLL
    - 0x2000 – este fișier DLL

Exemplu

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
   USHORT e_magic;         // Magic number
   // .. ..
   LONG   e_lfanew;        // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Număr magic – identifică tipurile de fișiere compatibile MS-DOS

- Executabilele MS-DOS au setată valoarea 0x5a4D, adică caraterele ASCII MZ – de aici și numele

e_lfanew este la adresa 0x3C și conține adresa relativă (FA) unde începe header-ul PE

- 0x000000F8 în exemplu

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
- 14C machine (x86)
- 9 number of sections
- 57D17271 time date stamp Thu Sep 8 16:15:13 2016
- 0 file pointer to symbol table
- 0 number of symbols
- E0 size of optional header
- 102 characteristics
  - Executable
  - 32 bit word machine

Lister - [c:\Programare\Exemple\guess.exe]

File Edit Options Encoding Help

```
00000000: 4D 5A 90 00 03 00 00 00|04 00 00 00 FF FF 00 00 | MZ.  └  ┘   ÿÿ
00000010: B8 00 00 00 00 00 00 00|40 00 00 00 00 00 00 00 | ¸       @
00000020: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00 |
00000030: 00 00 00 00 00 00 00 00|00 00 00 00 F8 00 00 00 |             ø
00000040: 0E 1F BA 0E 00 B4 09 CD|21 B8 01 4C CD 21 54 68 | ♬.º♬ ´.Í!¸.LÍ!Th
00000050: 69 73 20 70 72 6F 67 72|61 6D 20 63 61 6E 6E 6F | is program canno
00000060: 74 20 62 65 20 72 75 6E|20 69 6E 20 44 4F 53 20 | t be run in DOS
00000070: 6D 6F 64 65 2E 0D 0D 0A|24 00 00 00 00 00 00 00 | mode....$
00000080: 9B C5 AD 89 DF A4 C3 DA|DF A4 C3 DA DF A4 C3 DA | ■Å-■ß¤ÃÚß¤ÃÚß¤ÃÚ
00000090: 6B 38 2C DA DD A4 C3 DA|BC F9 C2 DB DC A4 C3 DA | k8,ÚÝ¤ÃÚ¼ùÂÛÜ¤ÃÚ
000000A0: BC F9 C6 DB C8 A4 C3 DA|BC F9 C7 DB D2 A4 C3 DA | ¼ùÆÛÈ¤ÃÚ¼ùÇÛÒ¤ÃÚ
000000B0: 02 5B 08 DA DD A4 C3 DA|DF A4 C2 DA 99 A4 C3 DA | ┐[◘ÚÝ¤ÃÚß¤ÂÚ■¤ÃÚ
000000C0: B1 F9 C6 DB DE A4 C3 DA|B1 F9 3C DA DE A4 C3 DA | ±ùÆÛÞ¤ÃÚ±ù<ÚÞ¤ÃÚ
000000D0: B1 F9 C1 DB DE A4 C3 DA|52 69 63 68 DF A4 C3 DA | ±ùÁÛÞ¤ÃÚRichß¤ÃÚ
000000E0: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00 |
000000F0: 00 00 00 00 00 00 00 00|50 45 00 00 4C 01 09 00 |         PE  L..
00000100: 71 72 D1 57 00 00 00 00|00 00 00 00 E0 00 02 01 | qrÑW        à ┐.
00000110: 0B 01 0E 00 00 52 00 00|00 44 00 00 00 00 00 00 | ♂.♬  R   D
```

Numărul de secțiuni: 0x0009

```
typedef struct _IMAGE_NT_HEADERS {
   DWORD Signature; /* "PE"\0\0 */    /* 0x00 */
   IMAGE_FILE_HEADER FileHeader;      /* 0x04 */
   IMAGE_OPTIONAL_HEADER OptionalHeader;    /* 0x18 */
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Markerul PE: la adresa 0x000000F8 este 0x4550 adică PE

Procesorul pe care poate rula aplicația: 0x014C Intel 386

```
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;
```

Dimensiunea header-ului opțional: 0x00E0

Structura fișierelor PE

- **PE file optional header**
  - Structura nu este opțională pentru fișiere PE !
  - Furnizează informații importante în completarea celor din IMAGE_FILE_HEADER

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  /* Standard fields */
  WORD  Magic; /* 0x10b or 0x107 */ /* 0x00 */
  BYTE  MajorLinkerVersion;
  BYTE  MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;        /* 0x10 */
  DWORD BaseOfCode;
  DWORD BaseOfData;
  /* NT additional fields */
  DWORD ImageBase;
  DWORD SectionAlignment;       /* 0x20 */
  DWORD FileAlignment;
  WORD  MajorOperatingSystemVersion;
  WORD  MinorOperatingSystemVersion;
  WORD  MajorImageVersion;
  WORD  MinorImageVersion;
  WORD  MajorSubsystemVersion;      /* 0x30 */
  WORD  MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;           /* 0x40 */
  WORD  Subsystem;
  WORD  DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;      /* 0x50 */
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
  /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Câmpuri standard (comune cu COFF)

Câmpuri adiționale

Tipul fișierului – setată la 0x10b pentru executabil normal

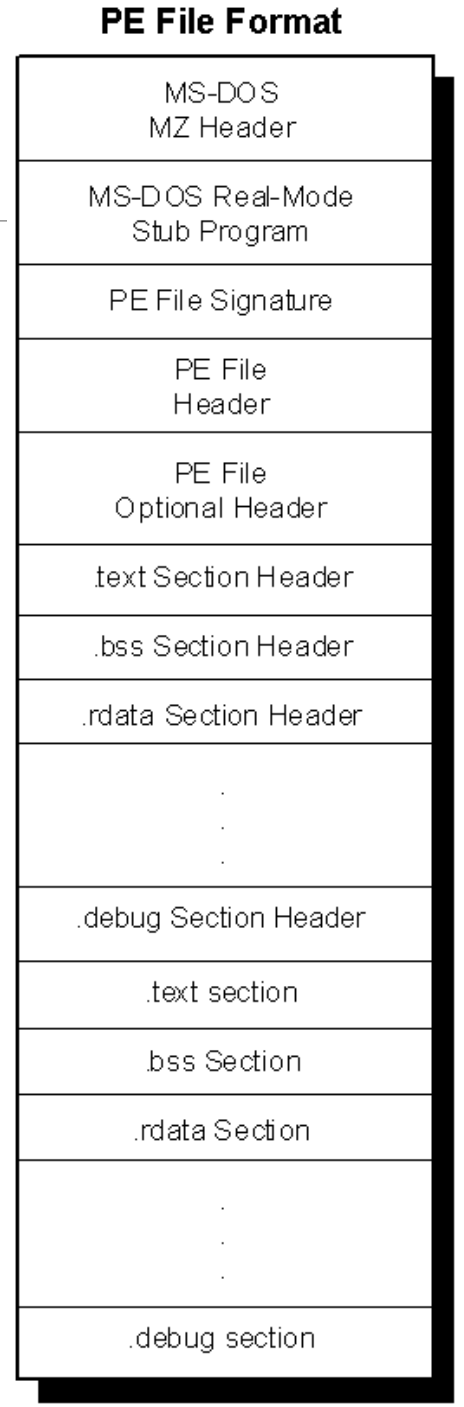
Versiunea de linkeditor

Dimensiunea codului executabil – suma dacă sunt mai multe secțiuni de cod

Dimensiunea datelor inițializate repectiv neinițializate

Adresa relativă (la ImageBase) de unde se va începe execuția codului
(și imlicit sfârșitul IAT – Import Address Table)

Adresele de început relative (la ImageBase) pentru secțiunea de cod (.text) respectiv secțiunea de date neinițializate (.bbs)



Structura fișierelor PE

- **PE file optional header**
  - Structura nu este opțională pentru fișiere PE !
  - Furnizează informații importnate în completarea celor din IMAGE_FILE_HEADER

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  /* Standard fields */
  WORD  Magic; /* 0x10b or 0x107 */ /* 0x00 */
  BYTE  MajorLinkerVersion;
  BYTE  MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;        /* 0x10 */
  DWORD BaseOfCode;
  DWORD BaseOfData;
  /* NT additional fields */
  DWORD ImageBase;
  DWORD SectionAlignment;           /* 0x20 */
  DWORD FileAlignment;
  WORD  MajorOperatingSystemVersion;
  WORD  MinorOperatingSystemVersion;
  WORD  MajorImageVersion;
  WORD  MinorImageVersion;
  WORD  MajorSubsystemVersion;      /* 0x30 */
  WORD  MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;           /* 0x40 */
  WORD  Subsystem;
  WORD  DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;      /* 0x50 */
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
  /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Câmpuri standard (comune cu COFF)

Câmpuri adiționale

Adresa de bază – adresa preferată unde să fie încărcată imaginea în memorie

Aliniamentul secțiunilor (în octeți) și unitatea de bază ocupată de o secțiune – cel puțin o pagină. Secțiunile se aliniază la multiplu de această dimensiune

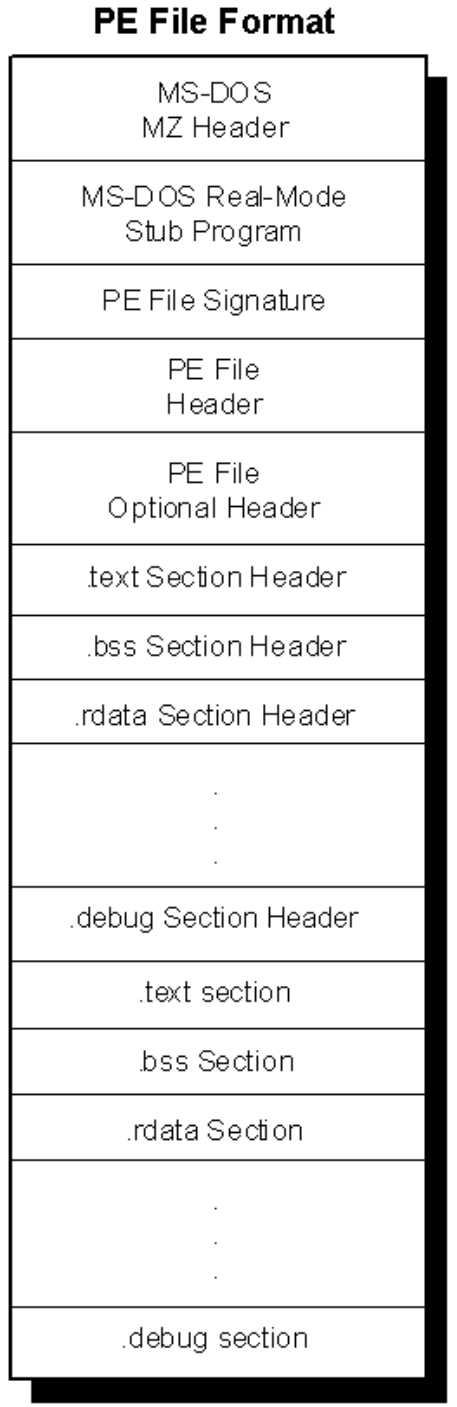
Granularitatea minimă a datelor brute din secțiuni. Alinierea se face începând de la un multiplu al acestei valori. Implicit este 0x200 octeți (512 octeți).

Versiunea minimă a SO Windows necesară pentru rularea aplicației

Versiunea aplicației

Dimensiunea regiunii din spațiul de adrese virtual care trebuie rezervată pentru maparea executabilului (cu toate headerele și secțiunile)

- Exemplu: Avem un sistem cu dimensiunea de pagină 4KB, și un executabil cu 11 secțiuni (inclusiv headerele) – fiecare mai mică decât o pagină
  - Dacă alinierea se face la limita de 65535 octeți, atunci SizeOfImage este 11 * 65535 = 720896 (176 de pagini)
  - Dacă alinierea se face la limita de 4KB, atunci SizeOfImage este 11 * 4096 = 45056 (11 pagini)



Structura fișierelor PE

- **PE file optional header**
  - Structura nu este opțională pentru fișiere PE !
  - Furnizează informații importnate în completarea celor din IMAGE_FILE_HEADER

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  /* Standard fields */
  WORD  Magic; /* 0x10b or 0x107 */ /* 0x00 */
  BYTE  MajorLinkerVersion;
  BYTE  MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;        /* 0x10 */
  DWORD BaseOfCode;
  DWORD BaseOfData;
  /* NT additional fields */
  DWORD ImageBase;
  DWORD SectionAlignment;       /* 0x20 */
  DWORD FileAlignment;
  WORD  MajorOperatingSystemVersion;
  WORD  MinorOperatingSystemVersion;
  WORD  MajorImageVersion;
  WORD  MinorImageVersion;
  WORD  MajorSubsystemVersion;      /* 0x30 */
  WORD  MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;               /* 0x40 */
  WORD  Subsystem;
  WORD  DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;      /* 0x50 */
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
  /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Câmpuri standard (comune cu COFF)

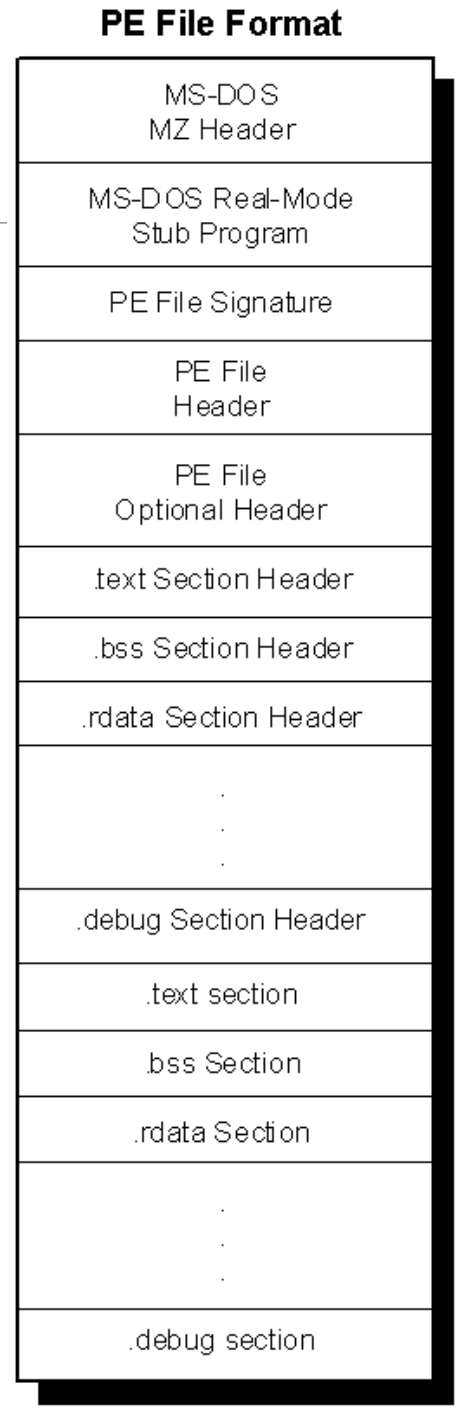
Câmpuri adiționale

Dimensiunea tuturor header-elor din PE (MS-DOS header, PE file header, PE optional header, și PE section header) rotunjit în sus la un multiplu de FileAlignment

Checksum pentru validarea executabilului la încărcare

Subsistemul destinație pentru aplicație

Indică dacă funcția de inițializare a DLL-ului este apelat – este setat 0 și se apelează de fiecare dată când
- dll-ul este încărcat prima dată / se termină
- un thread se termină / este creat



Structura fișierelor PE

- **PE file optional header**
  - Structura nu este opțională pentru fișiere PE !
  - Furnizează informații importnate în completarea celor din IMAGE_FILE_HEADER

Câmpuri standard (comune cu COFF)

Câmpuri adiționale

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  /* Standard fields */
  WORD  Magic; /* 0x10b or 0x107 */ /* 0x00 */
  BYTE  MajorLinkerVersion;
  BYTE  MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;        /* 0x10 */
  DWORD BaseOfCode;
  DWORD BaseOfData;
  /* NT additional fields */
  DWORD ImageBase;
  DWORD SectionAlignment;       /* 0x20 */
  DWORD FileAlignment;
  WORD  MajorOperatingSystemVersion;
  WORD  MinorOperatingSystemVersion;
  WORD  MajorImageVersion;
  WORD  MinorImageVersion;
  WORD  MajorSubsystemVersion;      /* 0x30 */
  WORD  MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;           /* 0x40 */
  WORD  Subsystem;
  WORD  DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;          /* 0x50 */
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
  /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Dimensiunea inițială a memoriei **rezervate** pentru stiva threadului. Implicit 1MB

Dimnesiunea inițială a memoriei **alocate** pentru stiva threadului. Implicit 1 sau 2 pagini

Dimensiunea inițială a memoriei **rezervate** pentru heap-ul implicit al procesului

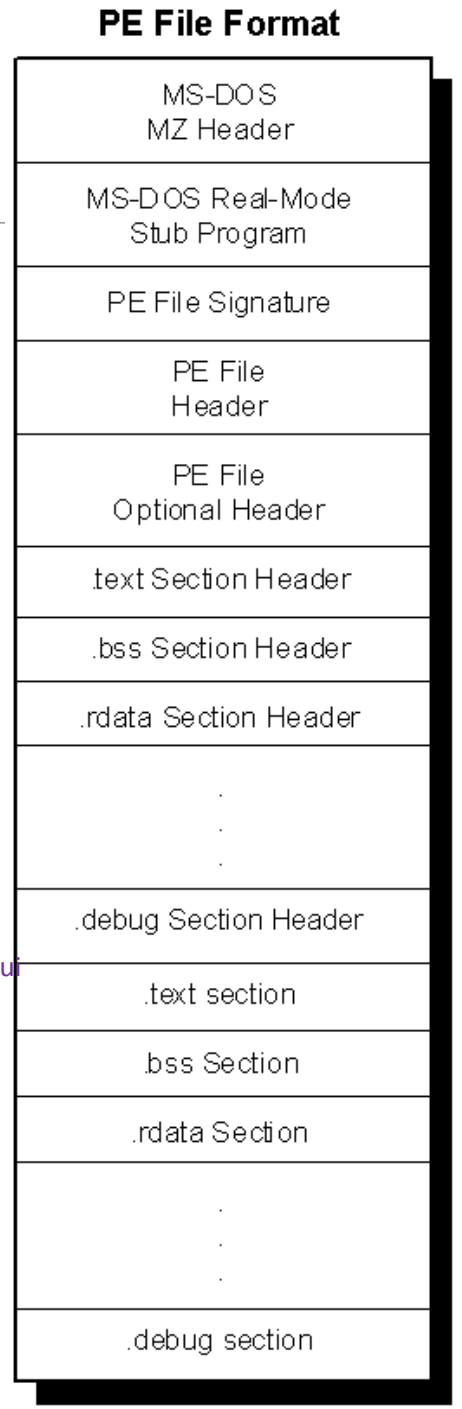
Dimensiunea inițială a memoriei **alocate** pentru heap-ul implicit al procesului. Implicit 1 pagină

Flag-uri legate de debugging (învechite)

Numărul de intrări valide în tabelul următor (maxim 16 intrări)

Tablou de structuri IMAGE_DATA_DIRECTORY care indică părți importante ale fișierului PE – adresa de începul și dimensiunea.

- Prima intrare: adresa și dimnesiunea tabelei de funcții exportante
- A doua intrare: adresa și dimnesiunea tabelei de funcții importate

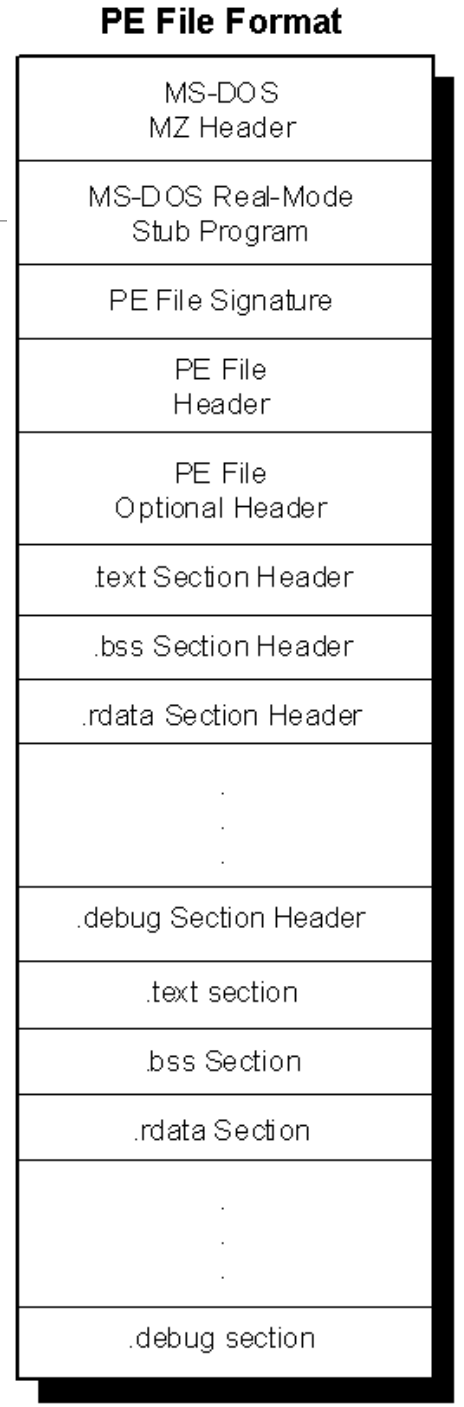


Structura fișierelor PE

- **PE file optional header**
  - Structura nu este opțională pentru fișiere PE !
  - Furnizează informații importnate în completarea celor din IMAGE_FILE_HEADER

```
typedef struct _IMAGE_OPTIONAL_HEADER {
   /* Standard fields */
   ... ............
   DWORD NumberOfRvaAndSizes;
   IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
   /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific
                                                  // Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration
                                                  // Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory
                                                  // in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import
                                                  // Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```



Exemplu

```
typedef struct _IMAGE_OPTIONAL_HEADER {
   /* Standard fields */
   WORD  Magic;        /* 0x010b */ /* 0x00 */
   BYTE  MajorLinkerVersion;
   BYTE  MinorLinkerVersion;

   DWORD SizeOfCode;

   DWORD SizeOfInitializedData;
   DWORD SizeOfUninitializedData;

   DWORD AddressOfEntryPoint;        /* 0x10 */

   DWORD BaseOfCode;
   DWORD BaseOfData;
   ...........
```

0x010B executabil normal

0x00005200

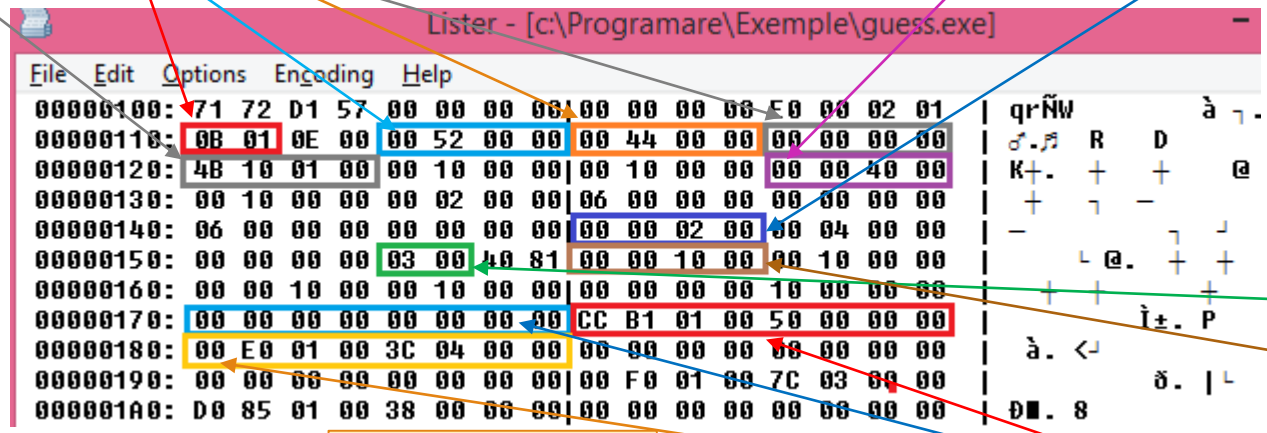
0x00004400

0x00000000

Adresa de bază: 0x00400000

Dimensiunea stub + header-e + toate secțiunile: 0x00020000

```
   ........
   /* NT additional fields */
   DWORD ImageBase;
   DWORD SectionAlignment;       /* 0x20 */
   DWORD FileAlignment;
   WORD  MajorOperatingSystemVersion;
   WORD  MinorOperatingSystemVersion;
   WORD  MajorImageVersion;
   WORD  MinorImageVersion;
   WORD  MajorSubsystemVersion;      /* 0x30 */
   WORD  MinorSubsystemVersion;
   DWORD Win32VersionValue;
   DWORD SizeOfImage;
   DWORD SizeOfHeaders;
   DWORD CheckSum;           /* 0x40 */
```



Resource Table – adresa relativă și dimensiune

Export Table – adresa relativă și dimensiune

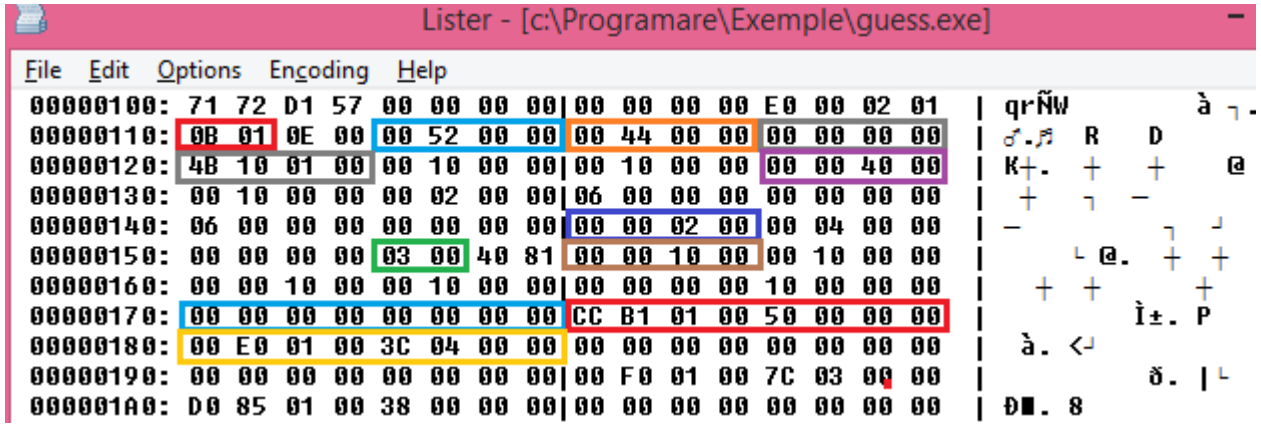
Import Table – adresa relativă și dimensiune

Subsistemul: 3 - Consola

```
   WORD  Subsystem;
   WORD  DllCharacteristics;
   DWORD SizeOfStackReserve;
   DWORD SizeOfStackCommit;
   DWORD SizeOfHeapReserve;       /* 0x50 */
   DWORD SizeOfHeapCommit;
   DWORD LoaderFlags;
   DWORD NumberOfRvaAndSizes;
   IMAGE_DATA_DIRECTORY DataDirectory
   [IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
   /* 0xE0 */
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Exemplu

OPTIONAL HEADER VALUES
- 10B magic # (PE32)
- 14.00 linker version
- 5200 size of code
- 4400 size of initialized data
- 0 size of uninitialized data
- 1104B entry point (0041104B) @ILT+70(_mainCRTStartup)
- 1000 base of code
- 1000 base of data



400000 image base (00400000 to 0041FFFF)
- 1000 section alignment
- 200 file alignment
- 6.00 operating system version
- 0.00 image version
- 6.00 subsystem version
- 0 Win32 version
- 20000 size of image
- 400 size of headers
- 0 checksum
- 3 subsystem (Windows CUI)
- 8140 DLL characteristics
  - Dynamic base
  - NX compatible
  - Terminal Server Aware

100000 size of stack reserve
- 1000 size of stack commit
- 100000 size of heap reserve
- 1000 size of heap commit
- 0 loader flags

16 number of directories
- 0 [0] RVA [size] of Export Directory
- 1B1CC [50] RVA [size] of Import Directory
- 1E000 [43C] RVA [size] of Resource Directory
- 0 [0] RVA [size] of Exception Directory
- 0 [0] RVA [size] of Certificates Directory
- .....

Structura fișierelor PE

- **Tabela de secțiuni**
  - Fiecare intrare este un header de secțiune – mai întâi apar toate header-e de secțiune și apoi secțiunile efective
  - O intrare are 40 octeți și furnizează informații incluzând adresa de început și dimensiunea fiecărei secțiuni din PE

```
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;
```

Numele secțiunii. Max. 8 caractere – adesea primul este . (punct)
Poate ocupa toți cei 8 octeți → atunci lipsește '\0' !!!

Pentru PE: Dimnesiunea secțiunii în memorie – înainte de rotunjiri date de aliniere. (Pentru OBJ este adresa fizică a secțiunii)

Adresa relativă (RVA) a secțiunii

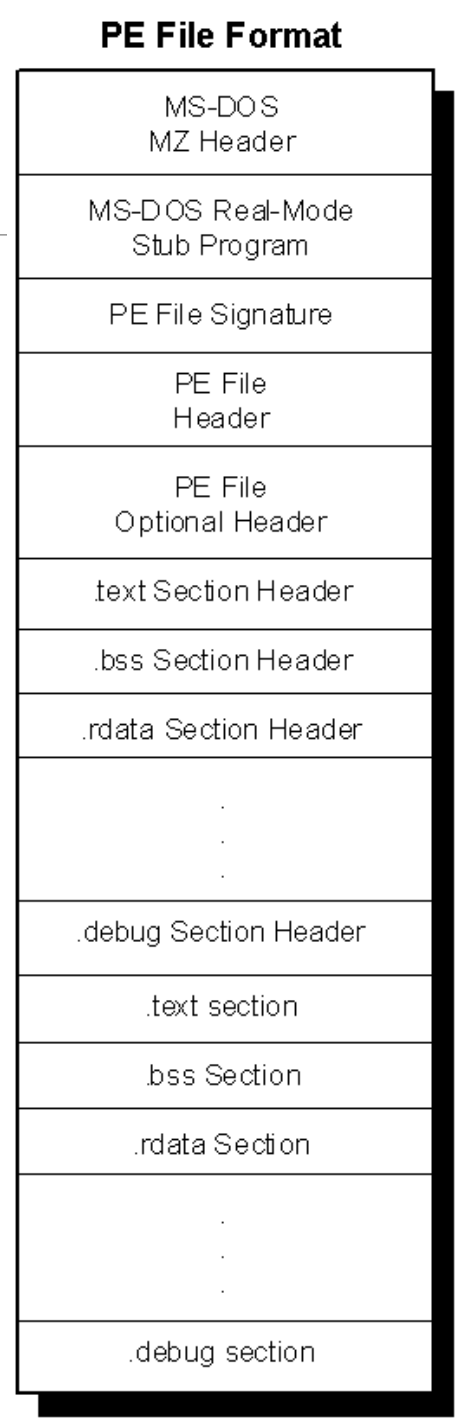
Dimensiunea efectivă a secțiunii pe disc – înainte de a fi rotunjită pt alinieri

Pointer (FA – File Address) la prima pagină de date din secțiune

Relevante doar pentru fișiere OBJ

Caracteristici (flaguri): indică atributele secțiunii – drepturi de acces etc.

- 0x00000020 – secțiune de cod (ex. .text)
- 0x00000040 – secțiune de date inițializate (ex. .data)
- 0x00000080 – secțiune de date neinițializate (ex. .bss)
- 0x10000000 – secțiune partajată (ex. dll-uri)
- 0x20000000 – secțiune executabilă (de regulă în combinație cu 0x00000020)
- 0x40000000 – secțiune cu drept de citire (cele mai multe secțiuni)
- 0x80000000 – secțiune cu drept de scriere – dacă nu este setat secțiunea devine read-only sau execute-only (ex. .data și .bss)



Exemplu

```
typedef struct _IMAGE_SECTION_HEADER {
BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
} Misc;

DWORD VirtualAddress;

DWORD SizeOfRawData;

DWORD PointerToRawData;
.............

DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Numele secțiunii
- Indică rolul secțiunii

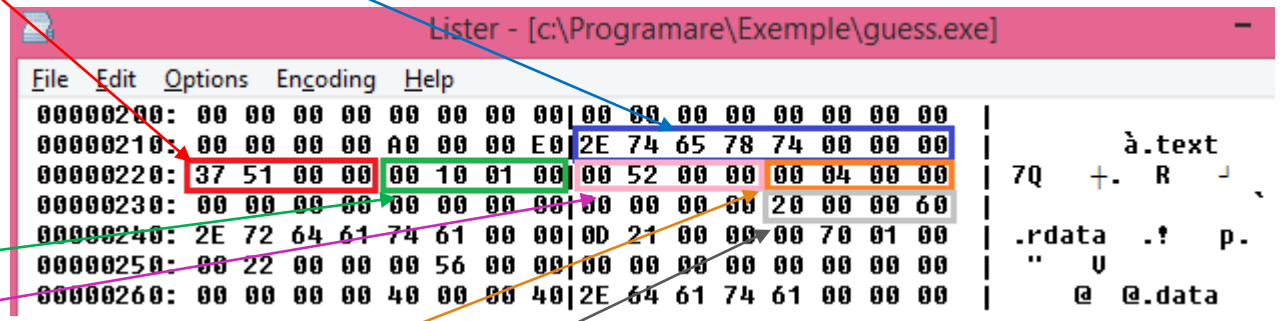
Dimensiunea în memorie

Adresa relativă a secțiunii

Dimensiunea pe disc

Pointer la prima pagină de date din secțiune

Atributele secțiunii



Secțiuni tipice pentru PE

Secțiune de cod: .text

- Cod executabil
- Dacă sunt mai multe fișiere sursă sunt compilate și linkeditate împreună, toate secțiunile de cod sunt concatenate și formează o singură secțiune de cod în fișierul executabil – analogic și pentru date inițializate, date neinițializate
- Aici se află și IAT – Import Address Table – imediat înainte de entry-point-ul modulului

Secțiuni de date:

- .data – data globale sau statice inițializate
- .bss – data globale sau statice neinițializate
- .rdata – date read-only: constante și directorul de debug

Secțiune de resurse: .rsrc

Secțiune de relocare: .reloc

- Conține tabela de relocări de bază
  - Utilizat în cazul în care încărcătorul nu a reușit să încarce modulul acolo unde linkeditorul ar fi așteptat să fie

Secțiune de exporturi: .edata

- .edata – informații despre funcții și date exportate altor module (de regulă apare doar la dll-uri)

Secțiune de importuri

- .idata – informații despre funcții și date importante din alte dll-uri

Secțiune de date locale threadului (thread local storage): .tls

Secțiunea de exporturi (.edata)

- **Tabela de exporturi**
  - Secțiunea de exporturi (.edata) conține informații despre funcții și date exportate altor module (de regulă apare doar la dll-uri)
  - Începe cu tabela de exporturi IMAGE_EXPORT_DIRECTORY (tabelă cu o singură intrare – indică locațiile și dimensiunile altor tabele de exporturi)

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;
    DWORD   AddressOfNames;
    DWORD   AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Adresa relativă (RVA) a șirului care conține numele dll-ului

Numărul de ordine al primei funcții exportate (numerele de ordine trebuie să existe în `AddressOfNameOrdinals`)

Numărul de funcții exportate de modul – numărul de elemente din tabela `AddressOfFunctions`

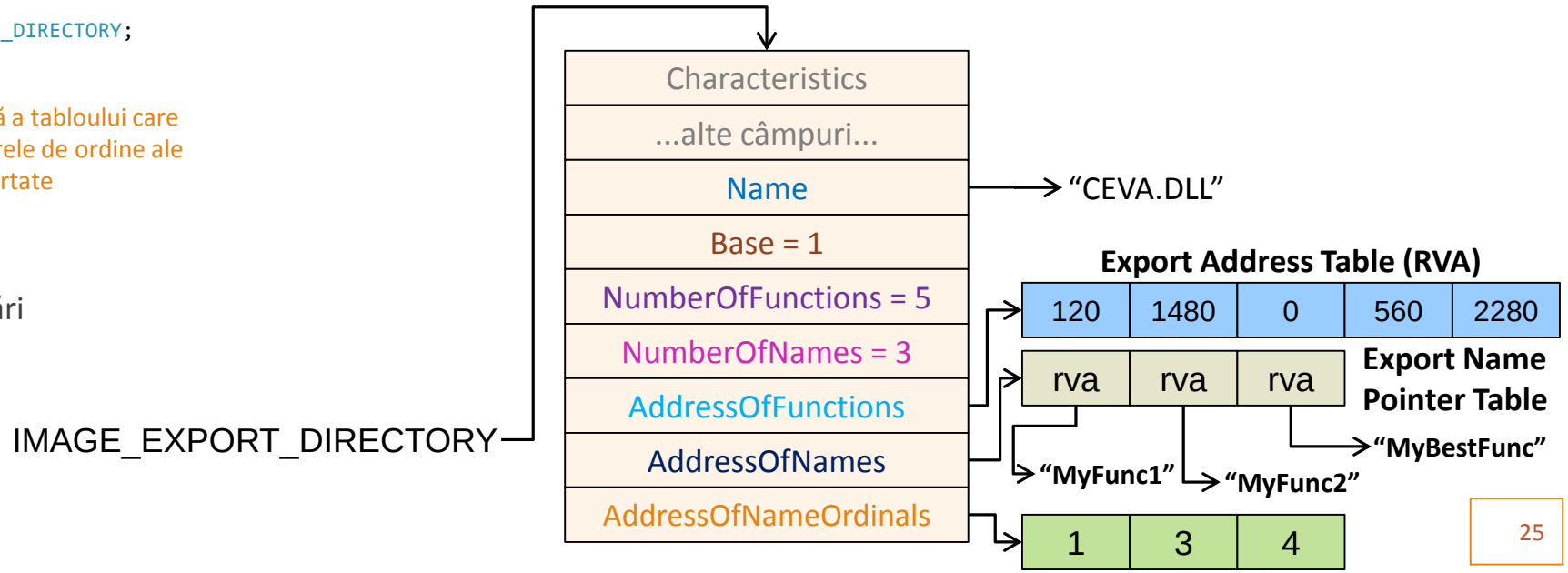
Numărul de elemente din tabloul `AddressOfNames` – poate fi mai mic decât `NumberOfFunctions`

Adresa relativă (RVA) a tabloului cu adresele (entry point-urile - RVA) funcțiilor exportate

Adresa relativă (RVA) a tabloului cu numele funcțiilor exportate

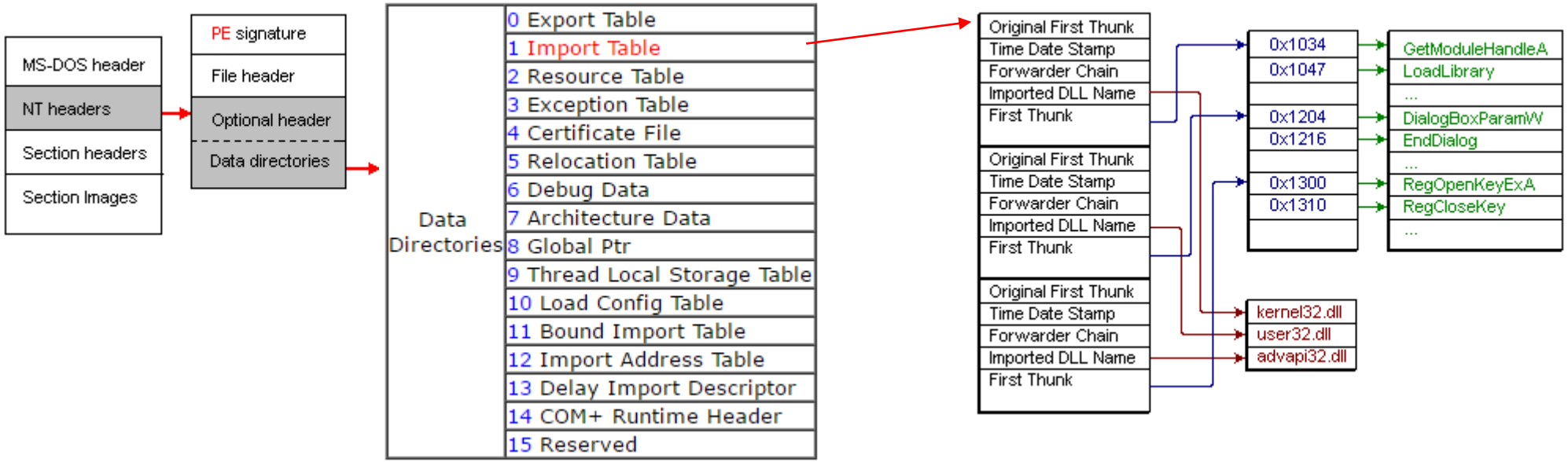
Adresa relativă a tabloului care conține numerele de ordine ale funcțiilor exportate

- Trei tabele importante pentru exportări
  - Export Address Table
  - Export Name Pointer Table
  - Export Ordinal Table



Secțiunea de importuri (.idata)

- **Tabela de importuri**
  - Secțiunea de importuri (.idata) conține informații despre funcții și date importate din alte module (de regulă din dll-uri)



Secțiunea de importuri (.idata)

- **Tabela de importuri**
  - Secțiunea de importuri (.idata) conține informații despre funcții și date importate din alte module (de regulă din dll-uri)
  - Începe cu descriptorul tabelei de importuri IMAGE_IMPORT_DESCRIPTOR (vezi și structurile IMAGE_IMPORT_BY_NAME și IMAGE_THUNK_DATA)

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    DWORD   OriginalFirstThunk; // RVA to original unbound IAT
    DWORD   TimeDateStamp;      // 0 if not bound,
    DWORD   ForwarderChain;     // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;         // RVA to IAT (if bound this
                                // IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

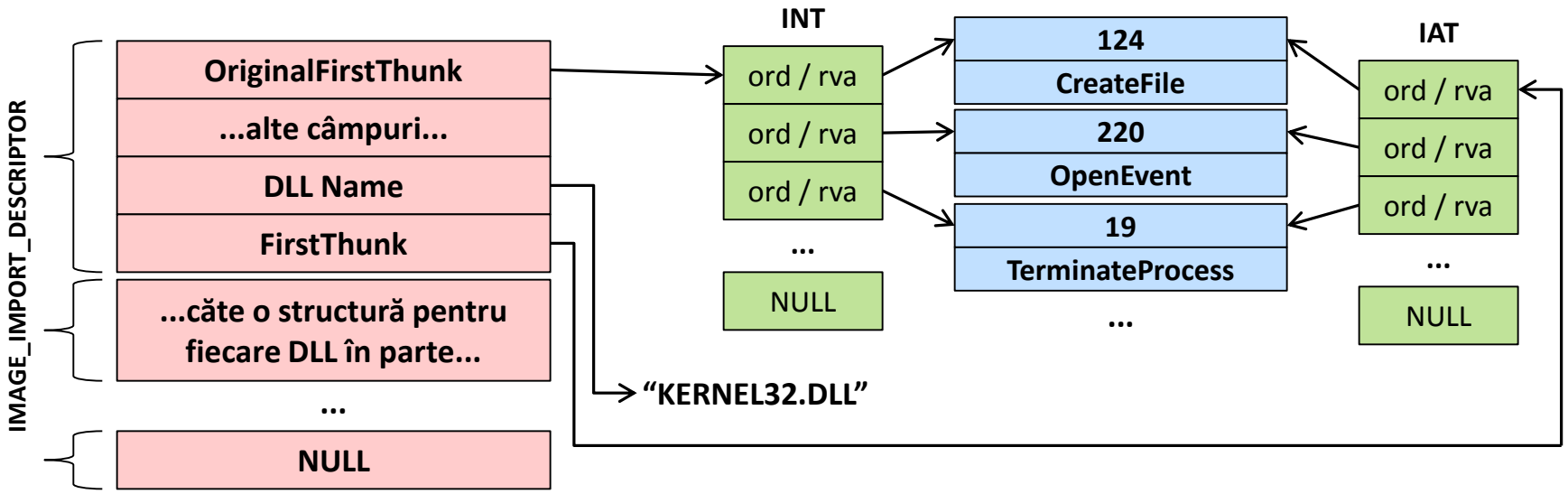
Adresa relativă a tabelei Import Name Table (conține structuri IMAGE_THUNK_DATA)
- Bitul cel mai semnificat indică dacă se face importul în funcție de
  - numărul de ordine (import by ordinal – bit 0) sau nume (import by name – bit 1)

Timpul la care s-a generat executabilul

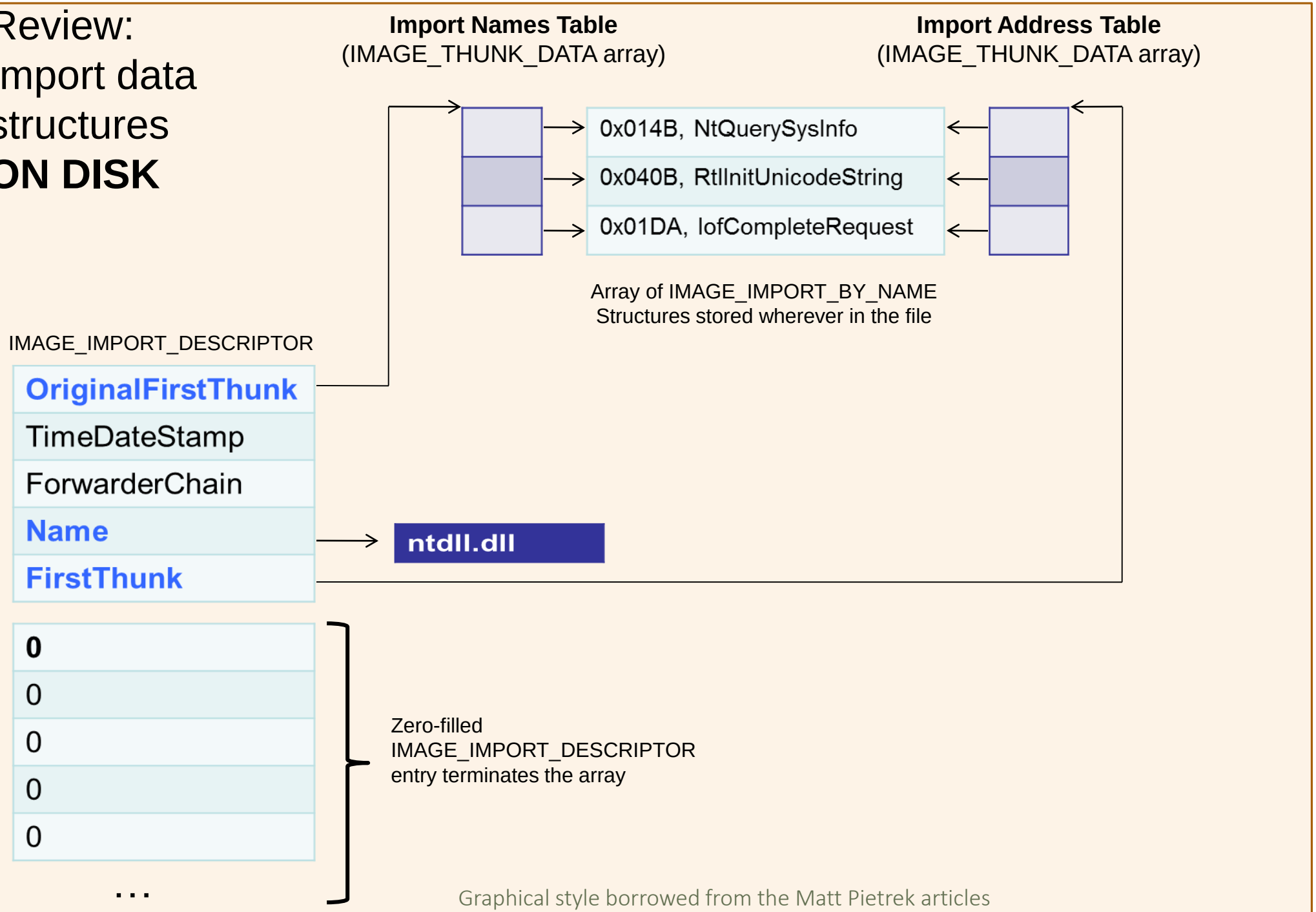
Legat de forwarding

Adresa relativă a șirului care conține numele dll-ului importat

Adresa virtuală (VA) a tabelei Import Address Table
- Adresele efective după relocare

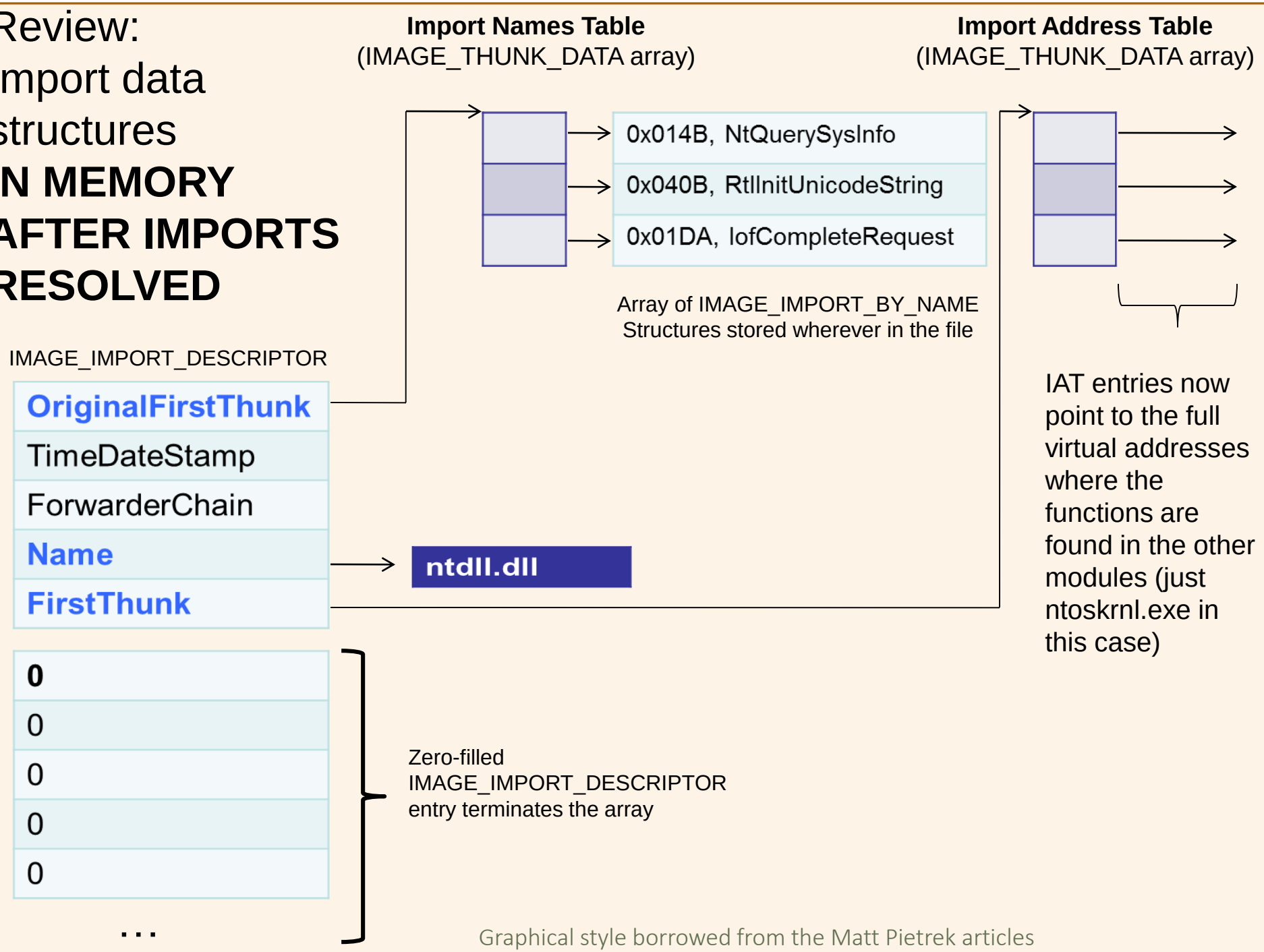


Review: Import data structures **ON DISK**



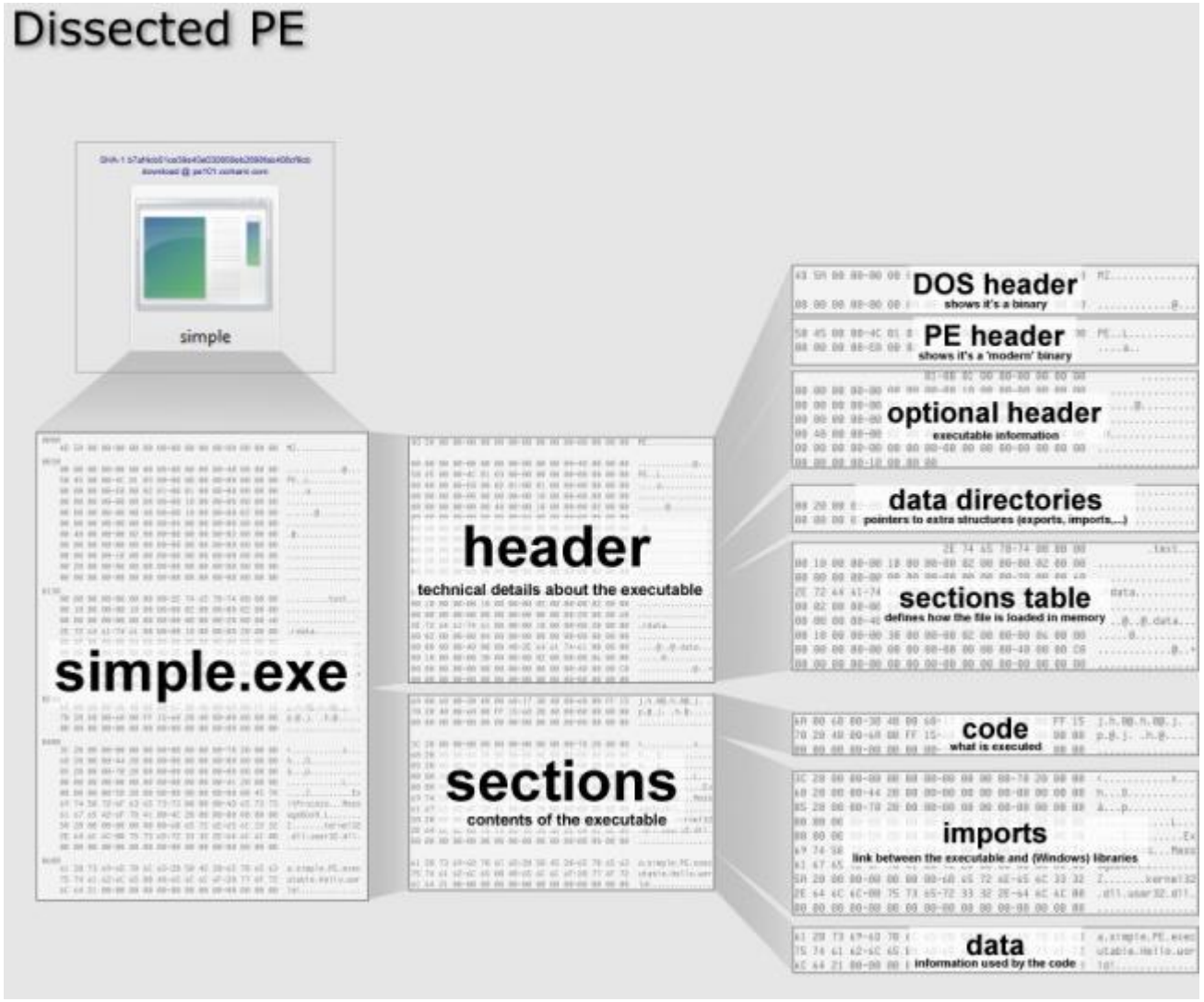
Graphical style borrowed from the Matt Pietrek articles

Review: Import data structures **IN MEMORY AFTER IMPORTS RESOLVED**



Graphical style borrowed from the Matt Pietrek articles

Fișiere PE


Dissected PE
simple
simple.exe
header
technical details about the executable
sections
contents of the executable
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports,...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code


Hexadecimal dump
ASCII dump
Fields
Values
Explanation
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code
e_magic
e_lfanew
'MZ'
0x40
constant signature
offset of the PE Header
Signature
Machine
NumberOfSections
SizeOfOptionalHeader
Characteristics
'PE', 0, 0
0x14c [intel 386]
3
0xe0
0x102 [32b EXE]
constant signature
processor: ARM/MIPS/Intel/...
number of sections
relative offset of the section table
EXE/DLL/...
Magic
AddressOfEntryPoint
ImageBase
SectionAlignment
FileAlignment
MajorSubsystemVersion
SizeOfImage
SizeOfHeaders
Subsystem
NumberOfRvaAndSizes
0x10b [32b]
0x1000
0x400000
0x1000
0x200
4 [NT 4 or later]
0x4000
0x200
2 [GUI]
16
32 bits/64 bits
where execution starts
address where the file should be mapped in memory
where sections should start in memory
where sections should start on file
required version of Windows
total memory space required
total size of the headers
driver/graphical/command line/...
number of data directories
ImportsVA
0x2000
RVA of the imports
Sections table
Name
VirtualSize
VirtualAddress
SizeOfRawData
PointerToRawData
Characteristics
.text
0x1000
0x1000
0x200
0x200
CODE EXECUTE READ
.rdata
0x1000
0x2000
0x200
0x400
INITIALIZED READ
.data
0x1000
0x3000
0x200
0x600
DATA READ WRITE
For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.
x86 assembly
Equivalent C code
push 0
push 0x403000
push 0x403017
push 0
call [0x402070]
push 0
call [0x402068]
MessageBox(0, "Hello World!","a simple PE executable", 0);
ExitProcess(0);
Imports structures
Consequences
descriptors
0x203c
0x204c, 0
0x2078
kernel32.dll
0,ExitProcess
0x2068
0x204c, 0
0x2044
0x205a, 0
0x2085
user32.dll
0,MessageBoxA
0x2070
0x205a, 0
after loading,
0x402068 will point to kernel32 dll's ExitProcess
0x402070 will point to user32 dll's MessageBoxA
Strings
a simple PE executable\0
Hello world!\0

Fișiere PE

Loading process

❶ Headers

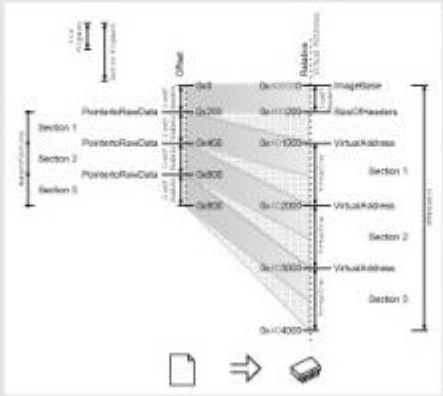
the *DOS Header* is parsed
the *PE Header* is parsed
(its offset is *DOS Header's e_lfanew*)
the *Optional Header* is parsed
(it follows the *PE Header*)

❷ Sections table

Sections table is parsed
(it is located at: offset (*OptionalHeader*) + *SizeOfOptionalHeader*)
it contains *NumberOfSections* elements
it is checked for validity with alignments:
FileAlignments and *SectionAlignments*

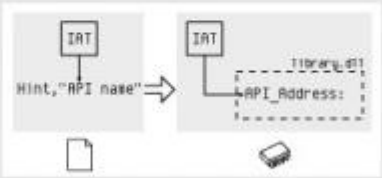
❸ Mapping

the file is mapped in memory according to:
the *ImageBase*
the *SizeOfHeaders*
the Sections table



❹ Imports

DataDirectories are parsed
they follow the *OptionalHeader*
their number is *NumOfRVAAndSizes*
imports are always #2
imports are parsed
each descriptor specifies a *DLLname*
this DLL is loaded in memory
IAT and *INT* are parsed simultaneously
for each API in *INT*
its address is written in the *IAT* entry



❺ Execution

Code is called at the *EntryPoint*
the calls of the code go via the IAT to the APIs

Notes

MZ HEADER *aka* DOS_HEADER
Starts with 'MZ' (initials of *Mark Zbikowski* MS-DOS developer)
PE HEADER *aka* IMAGE_FILE_HEADERS / COFF file header
Starts with 'PE' (Portable Executable)
OPTIONAL HEADER *aka* IMAGE_OPTIONAL_HEADER
Optional only for non-standard PEs but required for executables
RVA Relative Virtual Address
Address relative to ImageBase (at ImageBase, RVA = 0)
Almost all addresses of the headers are RVAs
In code, addresses are *not* relative.

INT Import Name Table
Null-terminated list of pointers to Hint, Name structures
IAT Import Address Table
Null-terminated list of pointers
On file it is a copy of the INT
After loading it points to the imported APIs
HINT
Index in the exports table of a DLL to be imported
Not required but provides a speed-up by reducing look-up

Materiale de studiu

Visual Studio, Microsoft Portable Executable and Common Object File Format Specification

- Vezi resursele de pe moodle

The life of binaries, http://opensecuritytraining.info/LifeOfBinaries.html