Module 1

A journey from high level languages, through assembly, to the running process

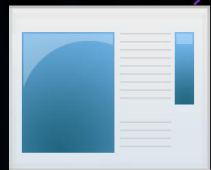# Basics of PE (Portable Executable)

Basics of a PE file

- PE (Portable Executable) is a native executable format on Windows
- PE files:
  - user mode: EXE, DLL
  - kernel mode: driver (.sys), kernel image (ntoskrnl.exe)
  - UEFI (run in SMM – System Managemant Mode)
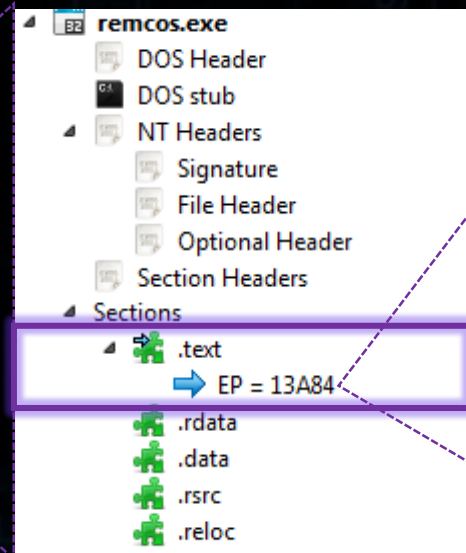 - Also OBJ files have structures similar to PE

Basics of a PE file

- PE (Portable Executable) contains information:
  - What to execute: the compiled code
  - How to execute: headers with data necessary for loading it

Basics of a PE file

- PE format is based on a Unix format COFF – that was used in VAX/VMS

- It was introduced as a part of specification Win32

- Throughout many years, the core of the format didn't change, only some new fields of some structures have been added

- Since introduction of 64 bit environment, PE needed to be adjusted to it: 64 bit PE was introduced

- Also, new variants have been introduced, like .NET PE – containing additional structures with intermediate code and metadata

Basics of a PE file

- PE file structure: the DOS part (legacy) and the Windows Part
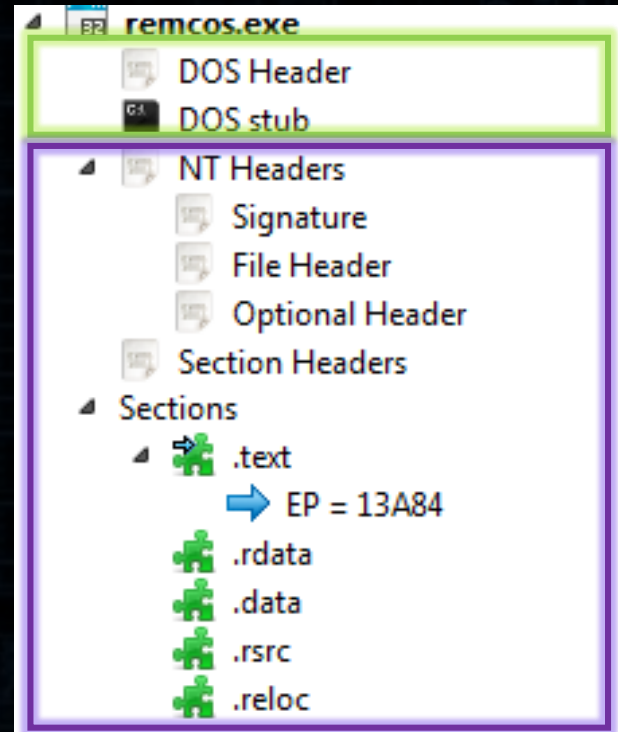
Basics of a PE file

- DOS Header: only e_magic, and e_lfnew must be filled:

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

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/pe_hdr.h

Basics of a PE file

- DOS Header: fields to remember

```
typedef struct _IMAGE_DOS_HEADER {
    WORD    e_magic; // Magic number → „MZ"
...
    LONG    e_lfanew; // points to NT header
} IMAGE_DOS_HEADER; *PIMAGE_DOS_HEADER;
```

```
typedef struct _IMAGE_NT_HEADERS32/64 {
    DWORD Signature;        → Magic number „PE\0\0"
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32/64 OptionalHeader;
} IMAGE_NT_HEADERS64;
```

Let's have a look in PE-bear...

Basics of a PE file

- FileHeader: fields to remember

```
typedef struct _IMAGE_NT_HEADERS32/64 {
    DWORD Signature;                              "PE\0\0"
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32/64 OptionalHeader;
} IMAGE_NT_HEADERS64;
```

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine; // Specifies the architecture
    WORD    NumberOfSections; // How many sections?
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Let's have a look in PE-bear...

Basics of a PE file

- OptionaHeader: fields to remember

```
typedef struct _IMAGE_NT_HEADERS32/64 {
    DWORD Signature;                   → „PE\0\0"
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32/64 OptionalHeader;
} IMAGE_NT_HEADERS64;
```
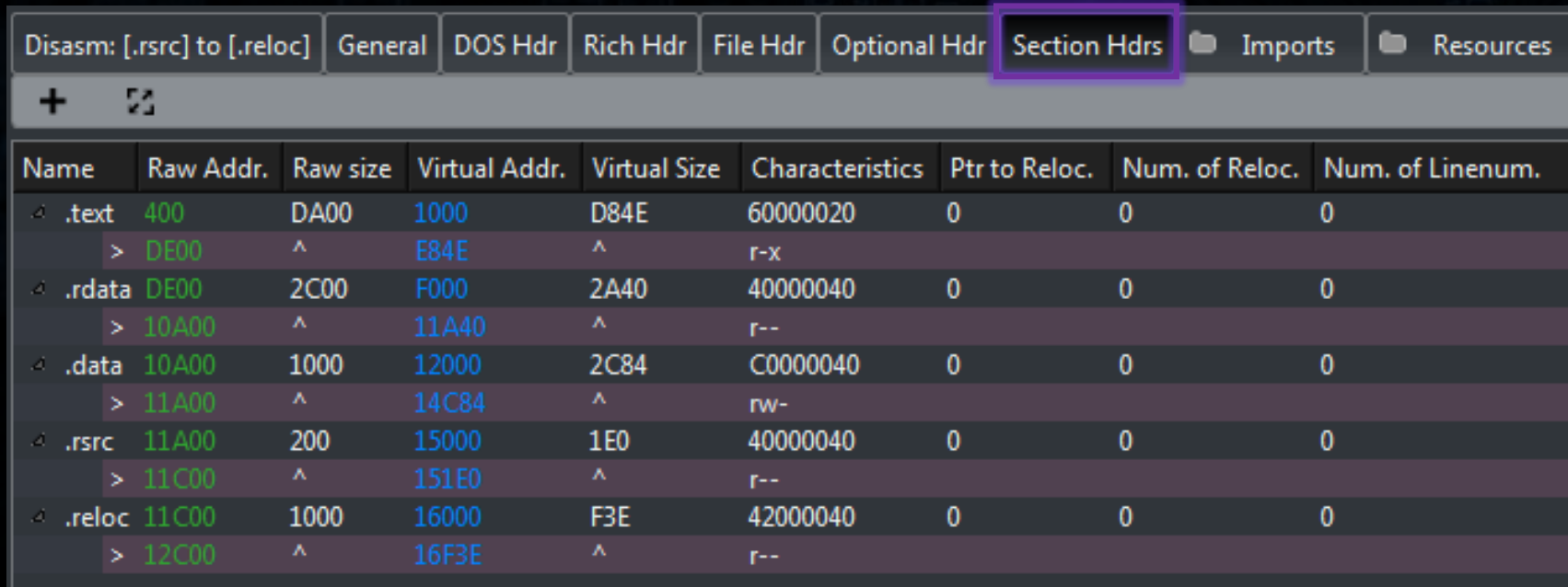
Let's have a look in PE-bear...

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;               // type: NT32 ? NT64?
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint; // where the execution starts?
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;           //default load base
    DWORD       SectionAlignment;    //unit in memory
    DWORD       FileAlignment;       //unit on disk
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;         //size of the loaded PE
    DWORD       SizeOfHeaders;       //offset where the sections hdrs start
    DWORD       CheckSum;
    WORD        Subsystem; // is it a console app? a driver? etc.
    WORD        DllCharacteristics; // features enabled
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[DIRECTORY_ENTRIES_NUM];
} IMAGE_OPTIONAL_HEADER64;
```

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/pe_hdr.h

Basics of a PE file: sections

- PE is divided into sections with different permissions

- Sections introduce a logical layout of the binary, that compilers/linkers can follow

- Dividing PE on section improves security: the code is isolated from the data

- HOWEVER:
  - if DEP is disabled, page without execution permission can still be executed
  - The section containing the Entry Point will always be treated as executable

Basics of a PE file: sections

- PE sections are defined by sections header

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	DA00	1000	D84E	60000020	0	0	0
>	DE00	^	E84E	^	r-x			
.rdata	DE00	2C00	F000	2A40	40000040	0	0	0
>	10A00	^	11A40	^	r--			
.data	10A00	1000	12000	2C84	C0000040	0	0	0
>	11A00	^	14C84	^	rw-			
.rsrc	11A00	200	15000	1E0	40000040	0	0	0
>	11C00	^	151E0	^	r--			
.reloc	11C00	1000	16000	F3E	42000040	0	0	0
>	12C00	^	16F3E	^	r--			

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/remap.h

Basics of a PE file: sections

- on the disk PE is stored in a raw format (the unit is defined by File Alignment)

- In memory PE is mapped to its virtual format (the unit is defined by Section Alignment) – usually of the granularity of one page (0x1000)

Disasm: [.rsrc] to [.reloc]	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	
Offset	Name				Value	Value
118	Base of Data				F000	
11C	Image Base				400000	
120	Section Alignment				1000	
124	File Alignment				200	

Basics of a PE file: sections
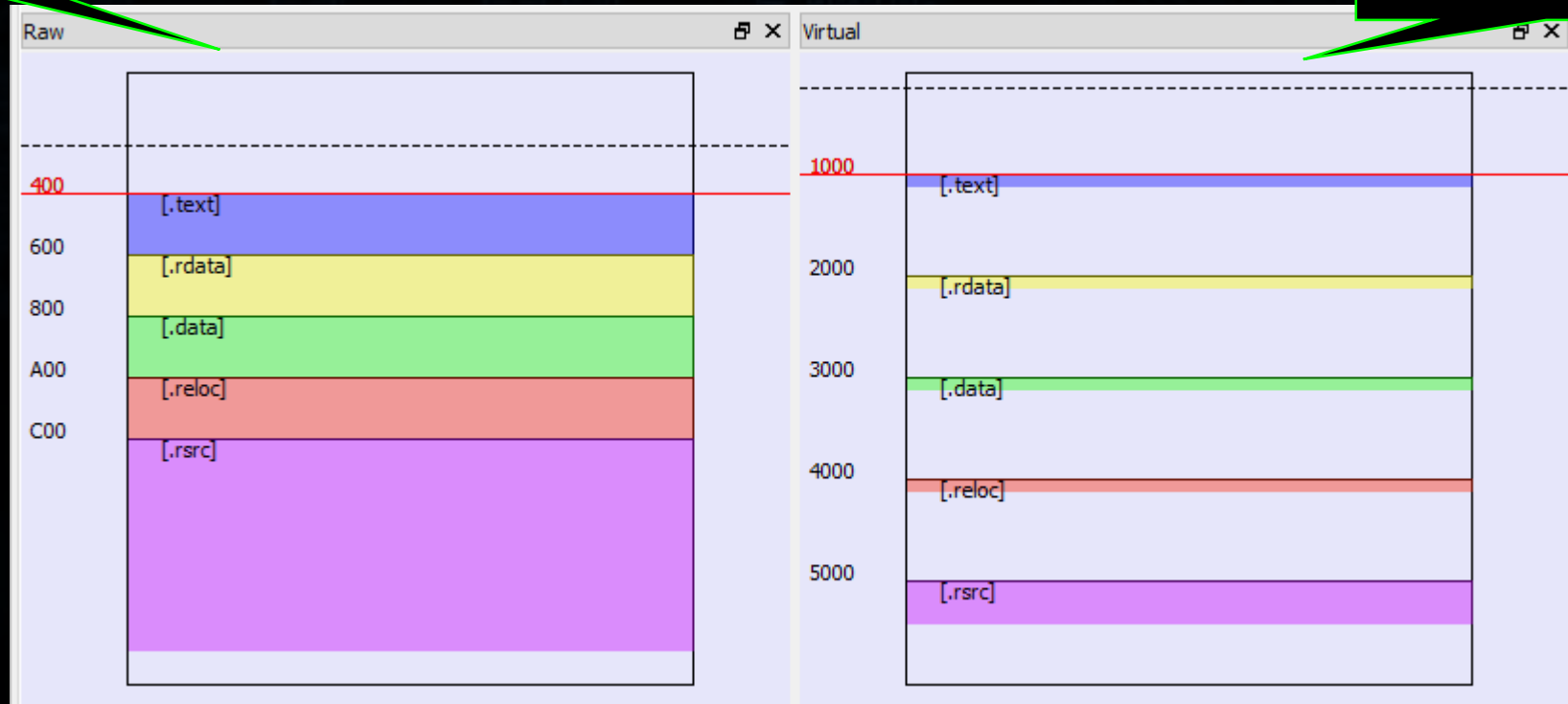
Basics of a PE file: sections

- **Raw** addresses (in file) usually correspond to **virtual** addresses (in memory) and vice versa

Raw 0x400 = RVA 0x1000

- **RVA** : Relative Virtual Address (without Image Base)

- **VA**: absolute Virtual Address (with Image Base)

400	[.text]
600	[.rdata]
800	[.data]
A00	[.reloc]
C00	[.rsrc]

Virtual	
1000	[.text]
2000	[.rdata]
3000	[.data]
4000	[.reloc]
5000	[.rsrc]

RVA 0x2000 = raw 0x600

RVA 0x5015 = raw 0xC15
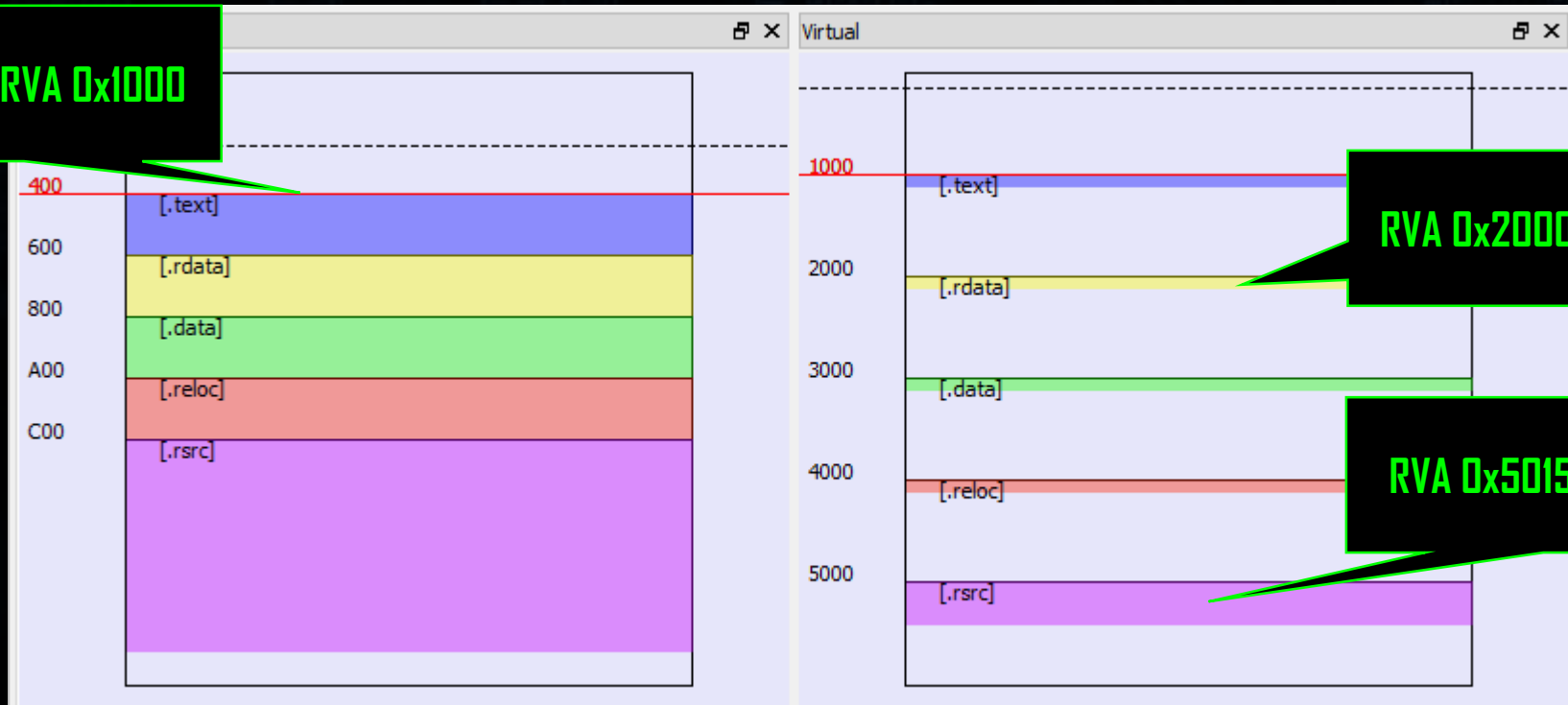
Basics of a PE file: sections

- Raw addresses (in file) *usually* correspond to virtual addresses (in memory) and vice versa

Raw 0x700 = RVA 0x2100

RVA 0x2400 -> invalid raw

- **RVA** : Relative Virtual Address (without Image Base)

- **VA**: absolute Virtual Address (with Image Base)

Virtual

400	1000 [.text]
600	
[.rdata]	2000 [.rdata]
800	
[.data]	
A00	3000 [.data]
[.reloc]	
C00	4000 [.reloc]
[.rsrc]	
	5000 [.rsrc]

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/remap.h
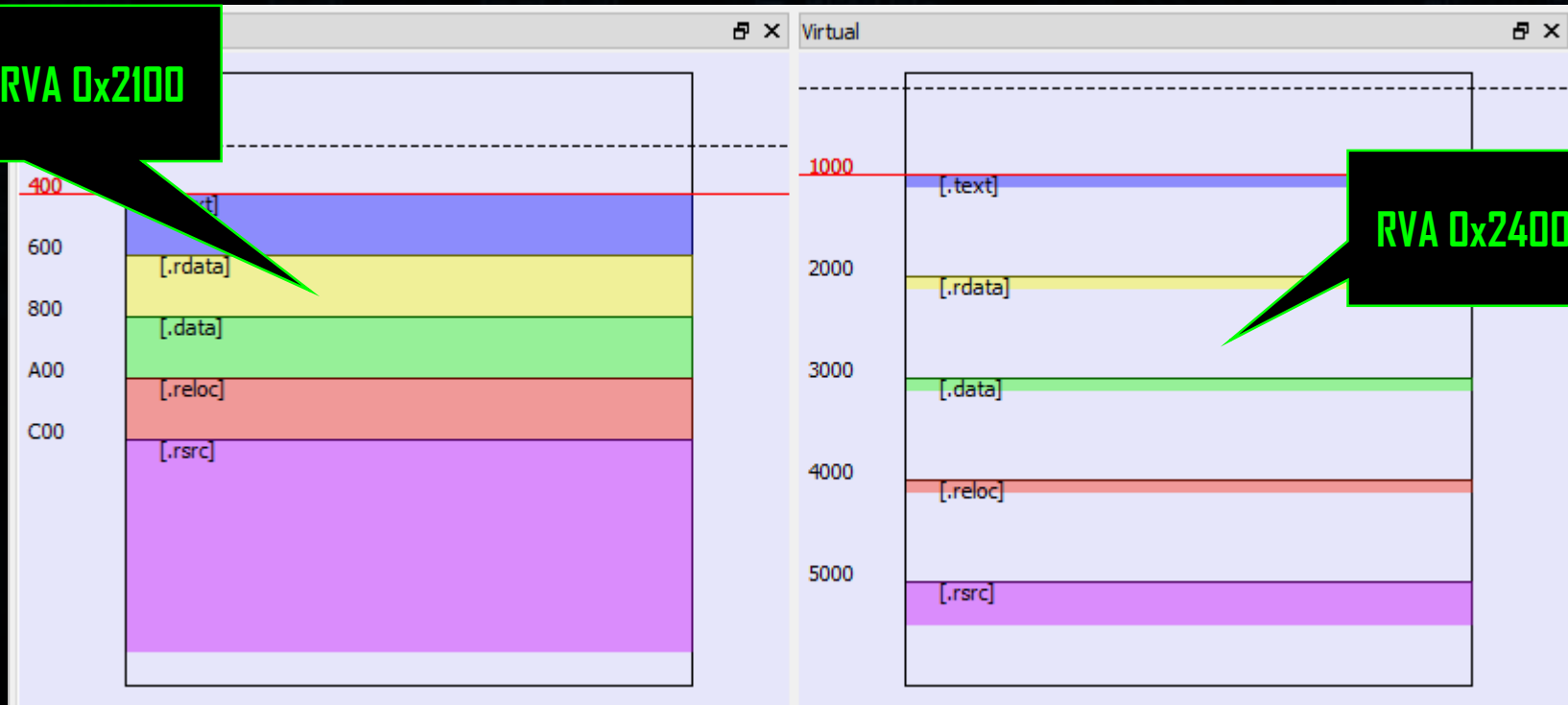
Basics of a PE file: sections

- Raw addresses (in file) *usually* correspond to virtual addresses (in memory) and vice versa
  - However:
    - Some sections can be unpacked in memory and not filled in the file
    - Some addresses may not be mapped (present in the file, but not in the memory image)

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
UPX0	400	0	1000	17000	E0000080	0	0	0
UPX1	400	A000	18000	A000	E0000040	0	0	0
.rsrc	A400	7000	22000	7000	C0000040	0	0	0

Raw

```
400
        [UPX1]
A400
        [.rsrc]
```
Firefox Setup Stub 38.0.5.exe

Virtual

```
1000
        [UPX0]


18000
        [UPX1]
22000
        [.rsrc]
```

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/remap.h

Basics of a PE file

- The most information lies in data directories

| Disasm: [.rsrc] to [.reloc] | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs |

Offset	Name	Value	Value
150	Size of Heap Reserve	100000	
154	Size of Heap Commit	1000	
158	Loader Flags	0	
15C	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
160	Export Directory	0	0
168	Import Directory	1133C	28
170	Resource Directory	15000	1E0
178	Exception Directory	0	0
180	Security Directory	0	0
188	Base Relocation Table	16000	9AC
190	Debug Directory	0	0
198	Architecture Specific Data	0	0
1A0	RVA of GlobalPtr	0	0
1A8	TLS Directory	0	0
1B0	Load Configuration Directory	10E78	40
1B8	Bound Import Directory in headers	0	0
1C0	Import Address Table	F000	138
1C8	Delay Load Import Descriptors	0	0
1D0	.NET header	0	0

Basics of a PE file: Relocation

- Relocation Table

Basics of a PE file: Relocation

1. PE comes with some default base address in the header
2. All the absolute addresses inside the PE assume that it was loaded at this base

Base Address = 400000

46E040 = 400000 + 6E040

Offset	Name	Value
114	Base of Code	1000
118	Base of Data	F000
11C	Image Base	400000

Hex		Disasm	
E977	E8E84AFFFF	▲	CALL 0X453464
45E97C	8BF0		MOV ESI, EAX
45E97E	FF1540E04600	▼	CALL DWORD PTR [0X46E040] [KERNEL32.dll].GetLastError
45E984	50		PUSH EAX
45E985	E8984AFFFF	▲	CALL 0X453422

Basics of a PE file: Relocation

- In the past EXEs were usually loaded at their default base (only DLLs didn't have to)

- Nowadays most PEs load at a dynamic base (due to ASLR)

- A flag in the header determines if a dynamic base will be used

https://github.com/hasherezade/malware_training_vol1/blob/main/exercises/module1/lesson2_pe/pe_snippets/relocate.h

Basics of a PE file: Relocation

- If the PE was loaded at a different base than the one defined in the header, all its fields using absolute addresses must be recalculated (rebased)

46E040 = 400000 + 6E040

	Hex		Disasm
45E977	E8E84AFFFF	▲	CALL 0X453464
	8BF0		MOV ESI, EAX
	FF15 40E04600	▼	CALL DWORD PTR [0X46E040] [KERNEL32.dll].GetLastError
45E984	50		PUSH EAX
45E985	E8984AFFFF	▲	CALL 0X453422

002B0000	00001000	
002C0000	00001000	pe-sieve32.exe
002C1000	0006D000	".text"
0032E000	00010000	".rdata"
0033E000	00006000	".data"
00344000	00022000	".rsrc"
00366000	00006000	".reloc"

Load base = 2C0000

32E040 = 2C0000 + 6E040

```
0031E977    E8 E84AFFFF    call pe-sieve32.313464
            8BF0           mov esi,eax
            FF15 40E03200  call dword ptr ds:[<&GetLastError>]
            50             push eax
0031E985    E8 984AFFFF    call pe-sieve32.313422
```

Basics of a PE file: Relocation

- How does PE know **where** are the fields that needs to be **rebased**?

Basics of a PE file: Relocation

- How does PE know **where** are the fields that needs to be **rebased**?

- They are listed in the Relocation Table!

Basics of a PE file: Relocation

- Let's open one of our sample PEs in PE-bear and see the relocation table

- Check the code snippet to see how the relocation table is processed

```
Exercise time...
```

Basics of a PE file:
Imports & Exports

Most executables use some functions exported by other modules (external libraries)

1. If we use a static library, the linker will automatically add the external code into our PE

2. If we use a dynamic library (DLL), the used functions will be listed in the Import Table of our PE, and dynamic linking will be done when the PE is loaded

3. Alternatively, we can load a DLL by ourselves using `LoadLibrary` and fetch the exported function via `GetProcessAddress`

Basics of a PE file: Exports

- Export Table

Basics of a PE file: Exports

1. DLLs are libraries of functions for other PEs to use

2. An Export Table is a catalogue allowing to find and use a particular function

Basics of a PE file: Exports

```
00311F49   .  JNZ pe_unmap.003120E2
00311F4F   .  PUSH 0                            ┌pOverlapped = NULL
00311F51   .  LEA EDX,DWORD PTR SS:[EBP-1014]   │
00311F57   .  PUSH EDX                          │pBytesRead
00311F58   .  PUSH 1000                         │BytesToRead = 1000 (4096.)
00311F5D   .  LEA EDX,DWORD PTR SS:[EBP-1004]   │
00311F63   .  PUSH EDX                          │Buffer
00311F64   .  PUSH DWORD PTR DS:[ECX+EAX]       │hFile
00311F67   .  CALL DWORD PTR DS:[<&KERNEL32.ReadFile>  └ReadFile
00311F6D   .  TEST EAX,EAX
```

DS:[0031F008]=768E96FB (kernel32.ReadFile)

We call a function from a DLL...

```
Address   Hex dump                            ASCII
0031F008  FB 96 8E 76 FF 98 8F 76  û‹Žv ˜v
0031F010  A8 3E 8E 76 95 A2 03 77  ¨>Žv•¢ w
```

**...and the execution is redirected to
the exported function**

C CPU - main thread, module kernel32

```
768E96FB  PUSH 0C                                          ReadFile
768E96FD  PUSH kernel32.768E9758
768E9702  CALL kernel32.768EBD70
768E9707  MOV EDI,DWORD PTR SS:[EBP+14]
768E970A  TEST EDI,EDI
768E970C  JE SHORT kernel32.768E9711
768E970E  AND DWORD PTR DS:[EDI],0
768E9711  MOV ESI,DWORD PTR SS:[EBP+8]
```

Basics of a PE file: Exports

1. Functions can be exported by a name or by ordinal (a number)
2. Some exports can be forwarded (pointing to other functions, in other DLLs)

Basics of a PE file: Exports

- Forwarded functions

Basics of a PE file: Imports

- Import Table

Basics of a PE file: Imports

- Dynamic linking is done when a PE is loaded

- The loader walks through the Import Table of the PE
  - loads needed DLLs
  - searches the imported functions in the export table of the DLL
  - fills the thunks via which the PE is going to make calls to the exported functions with appropriate addresses

Basics of a PE file: Imports

```
00311F49   .  JNZ pe_unmap.003120E2
00311F4F   .  PUSH 0                                  ┌pOverlapped = NULL
00311F51   .  LEA EDX,DWORD PTR SS:[EBP-1014]         │
00311F57   .  PUSH EDX                                │pBytesRead
00311F58   .  PUSH 1000                               │BytesToRead = 1000 (4096.)
00311F5D   .  LEA EDX,DWORD PTR SS:[EBP-1004]         │
00311F63   .  PUSH EDX                                │Buffer
00311F64   .  PUSH DWORD PTR DS:[ECX+EAX]             │hFile
00311F67   .  CALL DWORD PTR DS:[<&KERNEL32.ReadFile> └ReadFile
00311F6D   .  TEST EAX,EAX
```

We call a function from a DLL...

```
DS:[0031F008]=768E96FB (kernel32.ReadFile)
```

Address	x dump	ASCII
0031F008	96 8E 76 FF 9░ 8F 76	ūⁱ░v šČv
0031F010	░ 8E 76 95 A2 03 ░	Ė>░v└ó♥w

...via thunk that was filled with the address of the exported function

General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	📁 Imports

✣ ✚ 🗋

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDate
1013C	KERNEL32.dll	77	FALSE	11364	0

KERNEL32.dll [77 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
F000	VirtualAlloc	-	1149C	768F2FB6	-	5AB
F004	VirtualFree	-	114AC	768F1DA4	-	5AE
F008	ReadFile	-	114C8	768E96FB	-	458
F00C	GetCommandLi...	-	114D4	768F98FF	-	1E2

Basics of a PE file: Imports

```
00311F49   .  JNZ pe_unmap.003120E2
00311F4F   .  PUSH 0                               ┌pOverlapped = NULL
00311F51   .  LEA EDX,DWORD PTR SS:[EBP-1014]      │
00311F57   .  PUSH EDX                             │pBytesRead
00311F58   .  PUSH 1000                            │BytesToRead = 1000 (4096.)
00311F5D   .  LEA EDX,DWORD PTR SS:[EBP-1004]      │
00311F63   .  PUSH EDX                             │Buffer
00311F64   .  PUSH DWORD PTR DS:[ECX+EAX]          │hFile
00311F67   .  CALL DWORD PTR DS:[<&KERNEL32.ReadFile>  └ReadFile
00311F6D   .  TEST EAX,EAX
```

DS:[0031F008]=768E96FB (kernel32.ReadFile)

We call a function from a DLL...

Address	Hex dump	ASCII
0031F008	FB 96 8E 76 FF 98 8F 76	û|Av ÿ˜v
0031F010	A8 3E 8E 76 95 A2 03 77	¨>Av•¢w

C CPU - main thread, module kernel32

```
768E96FB   PUSH 0C                                          ReadFile
768E96FD   PUSH kernel32.768E9758
768E9702   CALL kernel32.768EBD70
768E9707   MOV EDI,DWORD PTR SS:[EBP+14]
768E970A   TEST EDI,EDI
768E970C   JE SHORT kernel32.768E9711
768E970E   AND DWORD PTR DS:[EDI],0
768E9711   MOV ESI,DWORD PTR SS:[EBP+8]
```

...and the execution is redirected to the exported function

Basics of a PE file: Imports

- Raw: before filling imports

Basics of a PE file: Imports

- Loaded: after filling imports – thunks are filled with addresses of exported functions

Basics of a PE file: Imports

Basics of a PE file: Imports

- Import Table: structure
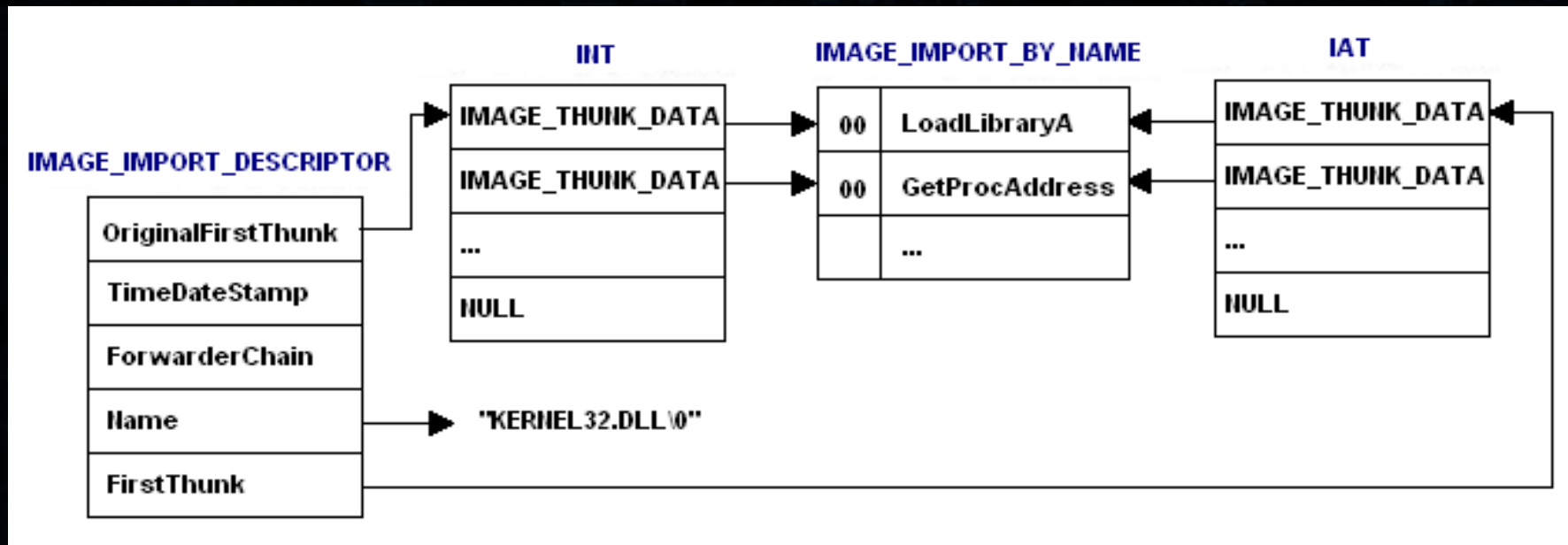
Basics of a PE file: Imports

- Let's open one of our sample PEs in PE-bear and see the import table. Find the corresponding DLLs and their exports.

- Check the code snippets to see how the import and export tables are processed

Exercise time...

Exercise

- Compile the given code of a custom PE loader and get familiar with it
  - https://github.com/hasherezade/malware_training_vol1/tree/main/exercises/module1/lesson2_pe

Further readings...

- MSDN documentation:
  - https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
- Classic articles about PE by Matt Pietrek:
  - https://bytepointer.com/resources/pietrek_in_depth_look_into_pe_format_pt1.htm -
  - https://bytepointer.com/resources/pietrek_in_depth_look_into_pe_format_pt2.htm
  - https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN