

Module 1

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

https://github.com/hasherezade/malware_training_voll

Basics of PE (Portable Executable)



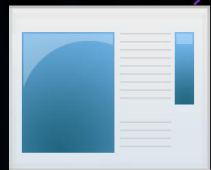
Basics of a PE file

- 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 Management Mode)
 - Also OBJ files have structures similar to PE

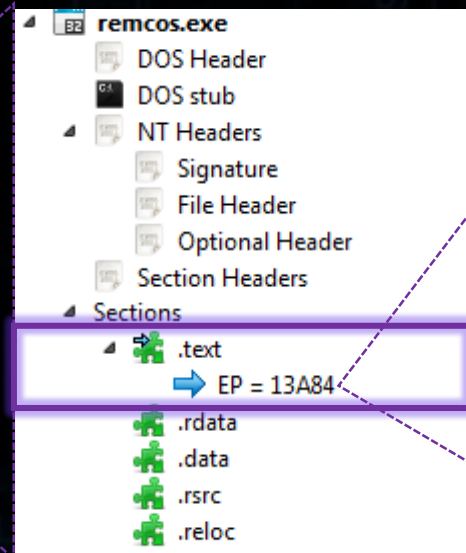


Basics of a PE file

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



remcos.exe



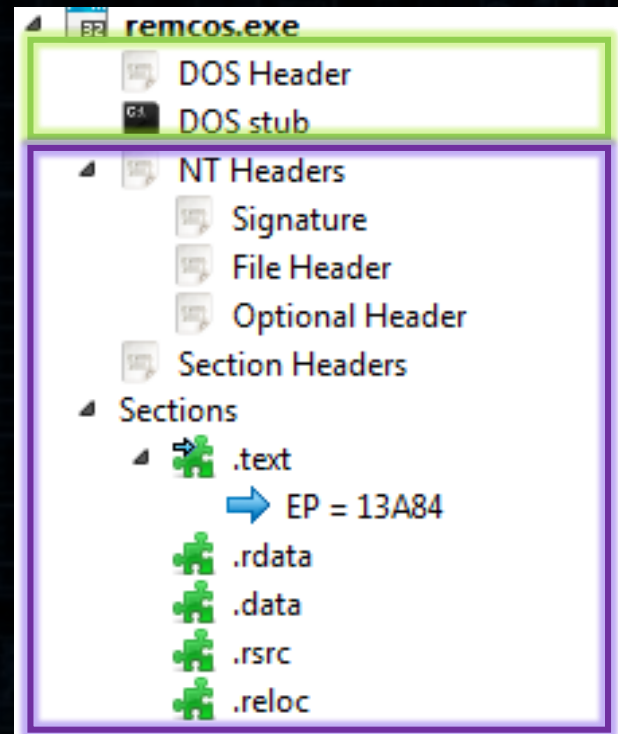
	Hex	Disasm
413A84	55	PUSH EBP
413A85	8BEC	MOV EBP, ESP
413A87	6AFF	PUSH -1
413A89	68805F4100	PUSH 0X415F08
413A8E	68103C4100	PUSH 0X413C10
413A93	64A100000000	MOV EAX, DWORD PTR FS:[0]
413A99	50	PUSH EAX
413A9A	64892500000000	MOV DWORD PTR FS:[0], ESP
413AA1	83EC68	SUB ESP, 0X68
413AA4	53	PUSH EBX

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_lfanew 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;
```

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_HEADERS32/64;
```

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;  
    IMAGE_FILE_HEADER FileHeader;  
    IMAGE_OPTIONAL_HEADER32/64 OptionalHeader;  
} IMAGE_NT_HEADERS32/64;
```

```
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

- OptionalHeader: 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_HEADERS32/64;
```

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;       //size of all the headers to map  
    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[EDIRECTORY_ENTRIES_NUM];  
} IMAGE_OPTIONAL_HEADER64;
```


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

```
#define IMAGE_FIRST_SECTION( ntheadr ) ((PIMAGE_SECTION_HEADER) \
((ULONG_PTR)ntheadr) + \
FIELD_OFFSET( IMAGE_NT_HEADERS, OptionalHeader ) + \
((ntheadr)->FileHeader.SizeOfOptionalHeader) \
))
```

A macro in
winnt.h
pointing the first
section



Disasm: [.rsrc] to [.reloc] General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports Resources								
+ [icon]								
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--			

Basics of a PE file: sections

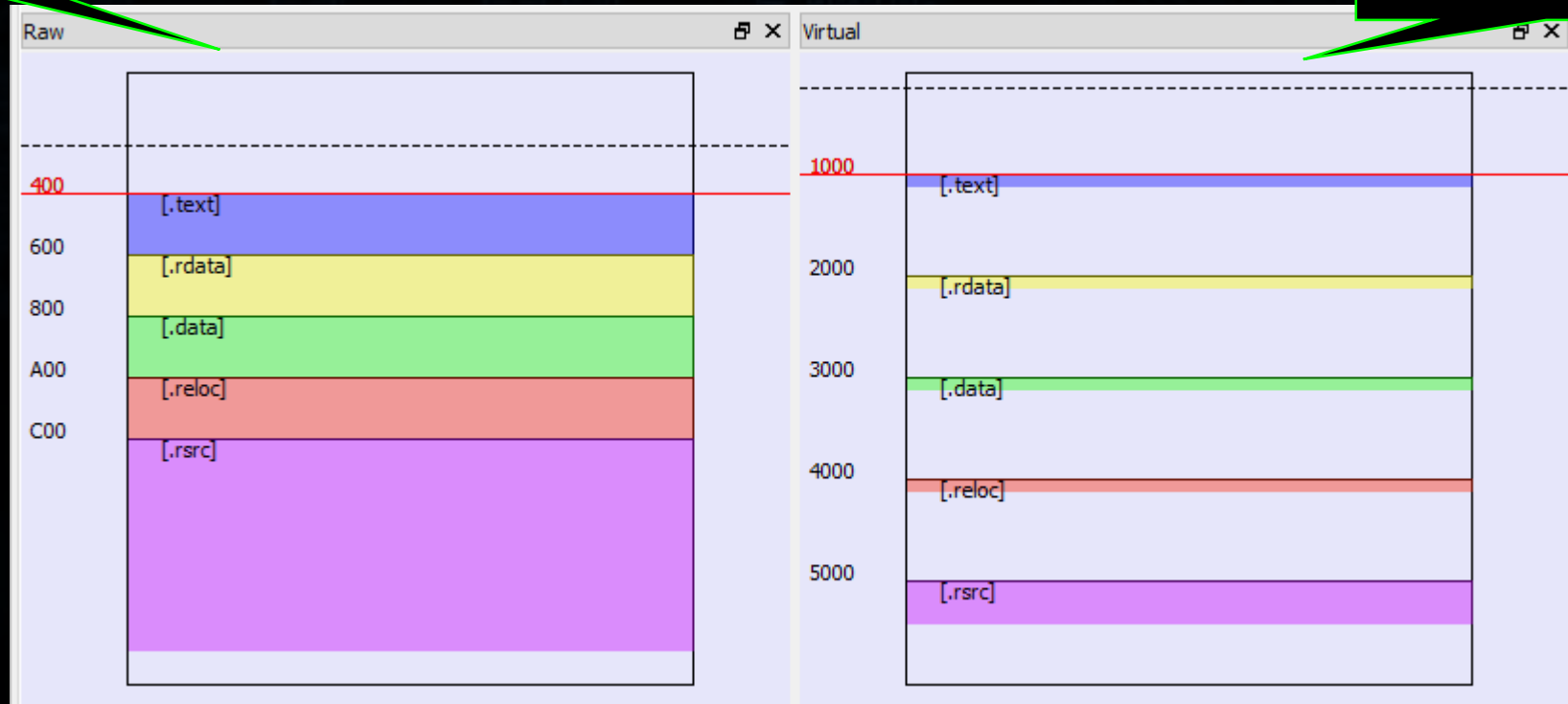
- on the disk PE is stored in a raw format (the unit is defined by File Alignment)
- In memory PE is mapped to its virtual format (the unit is defined by Section Alignment) – usually of the granularity of one page (0x1000)

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

Raw
(file on the disk)

Virtual
(mapped in the process
memory)



Basics of a PE file: caves

- The space reserved for a section is always **rounded up to some unit** (**FileAlignment** in a **raw** format, **SectionAlignment** in **virtual**)
- The size of the **actual section content** may be smaller
- The additional space is unused, and filled with padding. It is called a **section cave**.
- Caves may be virtual or raw
- Sometimes they may be used for installing code implants

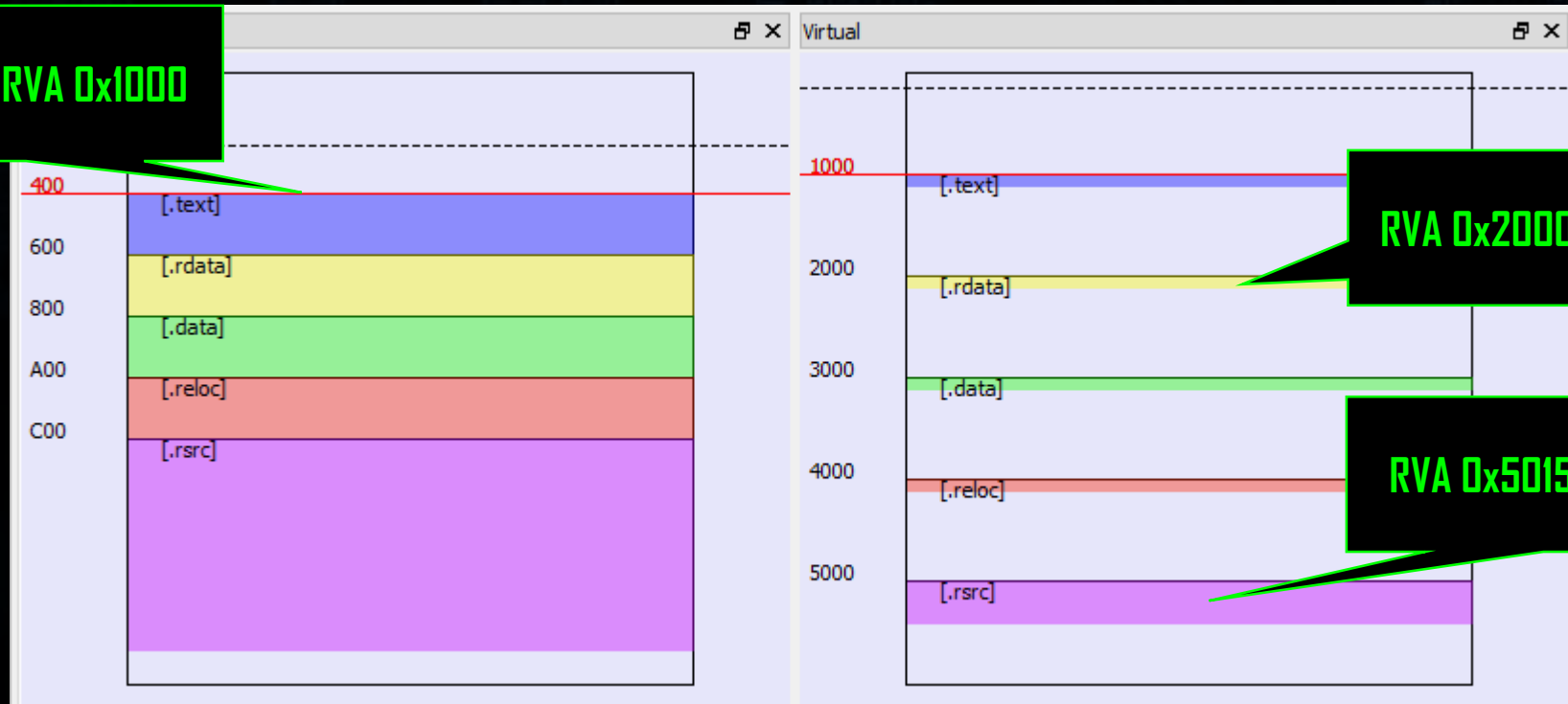


Basics of a PE file: addresses

- **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)

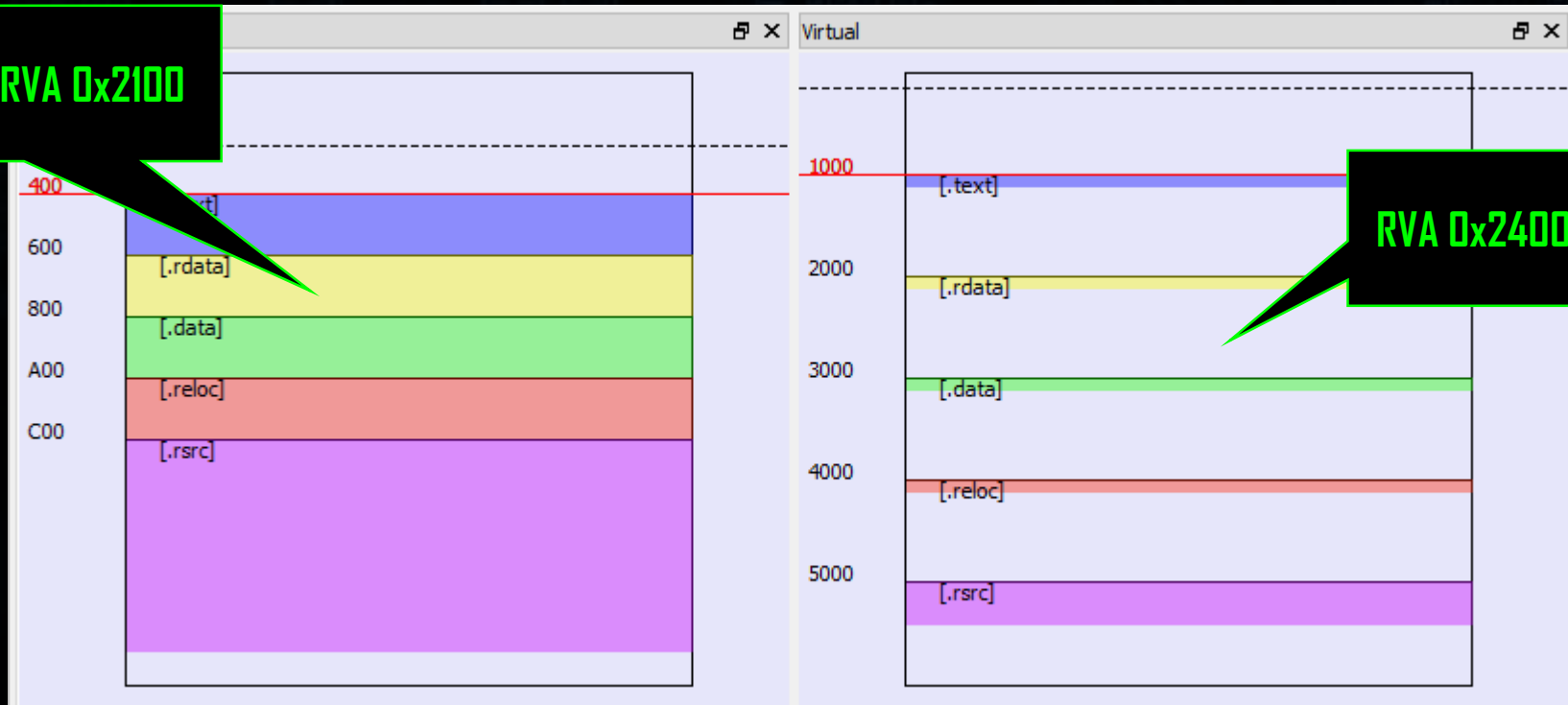


Basics of a PE file: addresses

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

Raw 0x700 = RVA 0x2100

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



RVA 0x2400 -> invalid raw

Basics of a PE file: addresses

- **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)

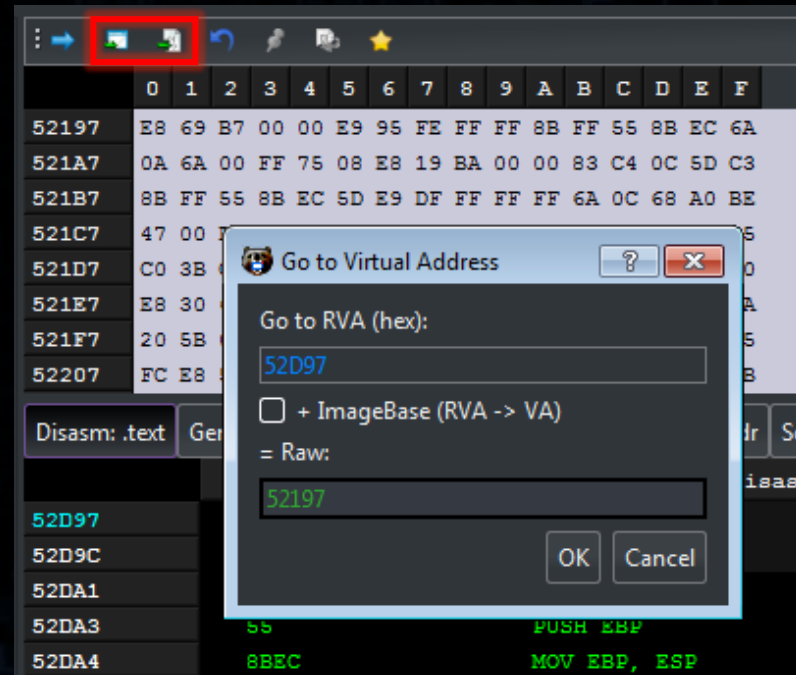
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
UPX0	400	0	1000	17000	E0000080	0	0	0
UPX1	400	A000	18000	A000	E0000040	0	0	0
.rsrc	A400	7000	22000	7000	C0000040	0	0	0

Raw	Virtual
400	1000
A400	18000
	22000

Firefox Setup Stub 38.0.5.exe

Basics of a PE file: addresses

- Let's open one of our sample PEs in PE-bear and see the section table
- Try converting various addresses from Raw format to Virtual, follow and observe



Exercise time...

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

	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

Hdr

File Hdr

Optional Hdr

Section Hdrs

Imports

Resources

BaseReloc.

Offset	Page RVA	Block Size	Entries Count
11C00	1000	74	36
11C74	2000	AC	52
11D20	3000	A0	4C
11DC0	4000	A0	4C

Relocation Block [54 entries]

Offset	Value	Type	Offset from Page	Reloc RVA
11C08	3072	32 bit field	72	1072
11C0A	3088	32 bit field	88	1088
11C0C	309D	32 bit field	9D	109D
11C0E	30CD	32 bit field	CD	10CD
11C10	314C	32 bit field	14C	114C
11C12	31B3	32 bit field	1B3	11B3
11C14	31CE	32 bit field	1CE	11CE
11C16	31E4	32 bit field	1E4	11E4
11C18	321F	32 bit field	21F	121F
11C1A	3229	32 bit field	229	1229
11C1C	323D	32 bit field	23D	123D
11C1E	3266	32 bit field	266	1266

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

Header	Rich Hdr	File Hdr	Optional Hdr
Offset	Name	Value	
114	Base of Code	1000	
118	Base of Data	F000	
11C	Image Base	400000	

Base Address = 400000

$46E040 = 400000 + 6E040$

	Hex	Disasm
45E977	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

Disasm: .text	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
Offset	Name	Value	Value				
144	Subsystem	3	Windows c				
146	DLL Characteristics	8140					
		40	DLL can move				
		100	Image is NX compatible				
		8000	TerminalServer aware				
148	Size of Stack Reserve	100000					

DLL Characteristics: DLL can move

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**)

Disassembly Window:

Address	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

Memory Dump Window:

Address	Value	Comment
002B0000	00001000	
002C0000	00001000	pe-sieve32.exe
002C1000	0000D000	".text"
0032E000	00010000	".rdata"
0033E000	00006000	".data"
00344000	00022000	".rsrc"
00366000	00006000	".reloc"

Annotations:

- $46E040 = 400000 + 6E040$ (points to the absolute address in the disassembly window)
- Load base = 2C0000** (points to the PE header in the memory dump)
- $32E040 = 2C0000 + 6E040$ (points to the relative address in the disassembly window)

Disassembly Window (Rebased):

Address	Hex	Disasm
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 GetProcAddress

Basics of a PE file: Exports

- Export Table

	Data Directory	Address	Size
168	Export Directory	B4FC4	A7FA
170	Import Directory	BF7C0	1F4
178	Resource Directory	C7000	528
180	Exception Directory	0	0

Disasm: .text	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Exports	Im
✦								
Offset	Name	Value	Meaning					
B47CC	MajorVersion	0						
B47CE	MinorVersion	0						
B47D0	Name	B8502	KERNEL32.dll					
B47D4	Base	1						
B47D8	NumberOfFunctions	54F						
B47DC	NumberOfNames	54F						
B47E0	AddressOfFunctions	B4FEC						
B47E4	AddressOfNames	B6528						
B47E8	AddressOfNameOrdinals	B7A64						
Exported Functions [1359 entries]								
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder			
B47EC	1	53C33	B88C2	BaseThreadInit...				
B47F0	2	BF311	BC10A	InterlockedPus...	NTDLL.RtlInterlockedPushListSList			
B47F4	3	BEDC7	B850F	AcquireSRWLoc...	NTDLL.RtlAcquireSRWLockExclusive			
B47F8	4	BEDE8	B8527	AcquireSRWLoc...	NTDLL.RtlAcquireSRWLockShared			
B47FC	5	45911	B853C	ActivateActCtx				
B4800	6	370DF	B854B	AddAtomA				
B4804	7	425F5	B8554	AddAtomW				

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

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

The screenshot displays the 'Exports' tab in a debugger, showing the export table of a PE file. The 'Exported Functions' table lists functions with their offsets, ordinals, names, and RVA values. A red arrow points from the 'ReadFile' entry (offset B56E8, ordinal 3C0, name BCF72) to the '496FB' address in the 'Disasm' window. The 'Disasm' window shows the assembly code at the selected address, which is 'PUSH OXC' (hex 6A0C).

Offset	Name	Value	Meaning
B47CC	MajorVersion	0	
B47CE	MinorVersion	0	
B47D0	Name	B8502	KERNEL32.dll
B47D4	Base	1	
B47D8	NumberOfFunc...	54F	

Offset	Ordinal	Function	Name RVA	Name	Forwarder
B56E0	3BE	60E73	BCF4F	ReadConsoleW	
B56E4	3BF	42C62	BC50C	ReadDirectoryChangesW	
B56E8	3C0	496FB	BCF72	ReadFile	
B56EC	3C1	63D99	BCF7B	ReadFileEx	

Disasm: .text	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Exports	Impo
496FB	6A0C							
496FD	685897E277							
49702	E869260000							
49707	8B7D14							
4970A	85FF							
4970C	7403							
4970E	832700							
49711	8B7508							
49714	83FEF4							
49717	0F84A50B0200							

Basics of a PE file: Exports

```
00311F49 . JNZ pe_unmap.003120E2
00311F4F . PUSH 0
00311F51 . LEA EDX,DWORD PTR SS:[EBP-1014]
00311F57 . PUSH EDX
00311F58 . PUSH 1000
00311F5D . LEA EDX,DWORD PTR SS:[EBP-1004]
00311F63 . PUSH EDX
00311F64 . PUSH DWORD PTR DS:[ECX+EAX]
00311F67 . CALL DWORD PTR DS:[&KERNEL32.ReadFile]
00311F6D . TEST EAX,EAX

pOverlapped = NULL
pBytesRead
BytesToRead = 1000 (4096.)
Buffer
hFile
```

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	Av sCv
0031F010	A8 3E 8E 76 95 A2 03 77	E>AvL0w

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

CPU - main thread, module kernel32

Address	Disassembly	Comment
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

kernel32.dll

Exported Functions [1359 entries]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
B4B28	D0	2028C	B9588	DelayLoadFailureHook	
B4B2C	D1	4266A	B959D	DeleteAtom	
B4B30	D2	BEF89	B95A8	DeleteBoundaryDescriptor	NTDLL.RtlDeleteBoundaryDescriptor
B4B34	D3	BEFAB	B95C1	DeleteCriticalSection	NTDLL.RtlDeleteCriticalSection
B4B38	D4	34F66	B95D7	DeleteFiber	
B4B3C	D5	447CB	B95E3	DeleteFileA	
B4B40	D6	89993	B95EF	DeleteFileTransactedA	

ntdll.dll

Exported Functions [1990 entries]				
Offset	Ordinal	Function RVA	Name RVA	Name
3614C	2E6	556D1	3ECD1	RtlDeleteBoundaryDescriptor
36150	2E7	59AC5	3ECED	RtlDeleteCriticalSection
36154	2E8	5DD50	3ED06	RtlDeleteElementGenericTable
36158	2E9	DC18	3ED23	RtlDeleteElementGenericTableAvl
3615C	2EA	76E87	3ED43	RtlDeleteHashTable
36160	2EB	B6994	3ED56	RtlDeleteNoSplay

	Disasm	Hint
59AC5	★ MOV EDI, EDI	RtlDeleteCriticalSection
59AC7	PUSH EBP	
59AC8	MOV EBP, ESP	
59ACA	PUSH -2	
59ACC	PUSH 0X77F10DE8	
59AD1	PUSH 0X77EDE0ED	

Basics of a PE file: Imports

- Import Table

	Data Directory	Address	Size
160	Export Directory	0	0
168	Import Directory	1133C	28
170	Resource Directory	15000	1E0

General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports Resources BaseReloc. Load

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1013C	KERNEL32.dll	77	FALSE	11364	0	0	114BA	F000

KERNEL32.dll [77 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
F000	VirtualAlloc	-	1149C	1149C	-	5AB
F004	VirtualFree	-	114AC	114AC	-	5AE
F008	ReadFile	-	114C8	114C8	-	458
F00C	GetCommandLineA	-	114D4	114D4	-	1E2
F010	IsDebuggerPresent	-	114E6	114E6	-	383
F014	EncodePointer	-	114FA	114FA	-	13C
F018	DecodePointer	-	1150A	1150A	-	117
F01C	IsProcessorFeaturePresent	-	1151A	1151A	-	388

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

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
00311F51 . LEA EDX,DWORD PTR SS:[EBP-1014]
00311F57 . PUSH EDX
00311F58 . PUSH 1000
00311F5D . LEA EDX,DWORD PTR SS:[EBP-1004]
00311F63 . PUSH EDX
00311F64 . PUSH DWORD PTR DS:[ECX+EAX]
00311F67 . CALL DWORD PTR DS:[&KERNEL32.ReadFile]
00311F6D . TEST EAX,EAX
```

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

Address	hex dump	ASCII
0031F008	96 8E 76 FF 96 8E 76 00 Av s dv	
0031F010	8E 76 95 A2 03 17 E0 AvL d w	

pOverlapped = NULL

pBytesRead

BytesToRead = 1000 (4096.)

Buffer

hFile

ReadFile

We call a function from a DLL...

...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?	OriginalFirstThunk	TimeDateStamp
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
00311F51 . LEA EDX,DWORD PTR SS:[EBP-1014]
00311F57 . PUSH EDX
00311F58 . PUSH 1000
00311F5D . LEA EDX,DWORD PTR SS:[EBP-1004]
00311F63 . PUSH EDX
00311F64 . PUSH DWORD PTR DS:[ECX+EAX]
00311F67 . CALL DWORD PTR DS:[&KERNEL32.ReadFile]
00311F6D . TEST EAX,EAX

pOverlapped = NULL
pBytesRead
BytesToRead = 1000 (4096.)
Buffer
hFile
```

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	Av sCv
0031F010	A8 3E 8E 76 95 A2 03 77	E>AvL0w

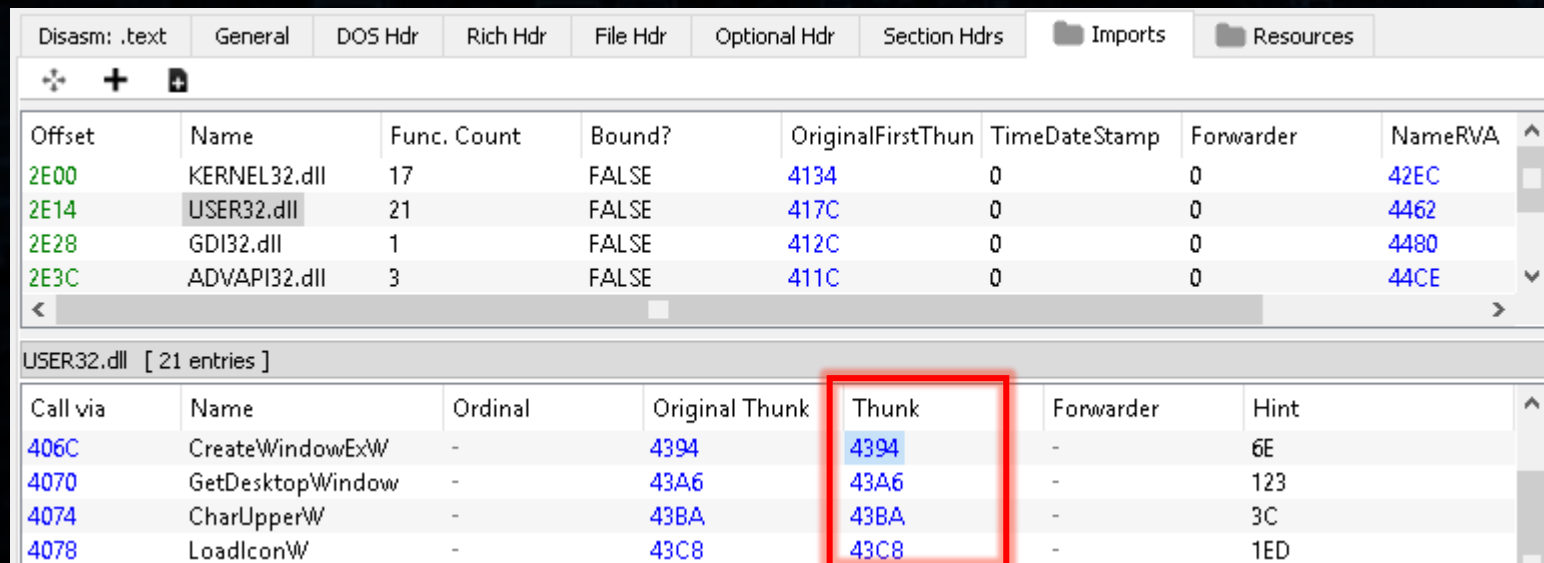
...and the execution is redirected to the exported function

CPU - main thread, module kernel32

Address	Disassembly	Comment
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: Imports

- Raw: before filling imports



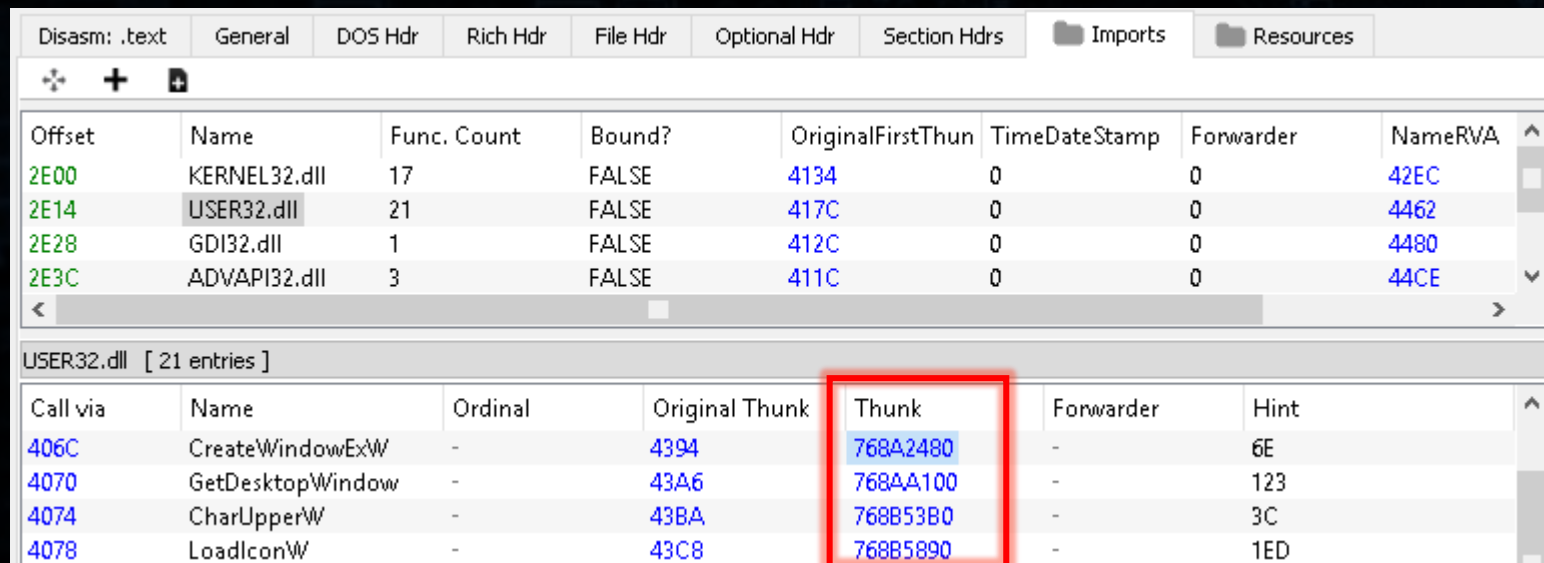
The screenshot shows the PE Explorer interface with the 'Imports' tab selected. The top table lists imported DLLs: KERNEL32.dll, USER32.dll, GDI32.dll, and ADVAPI32.dll. The 'USER32.dll' entry is selected, and the bottom table shows its 21 entries. The 'Thunk' column in the bottom table is highlighted with a red box, showing the original thunks for 'CreateWindowExW', 'GetDesktopWindow', 'CharUpperW', and 'LoadIconW'.

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA
2E00	KERNEL32.dll	17	FALSE	4134	0	0	42EC
2E14	USER32.dll	21	FALSE	417C	0	0	4462
2E28	GDI32.dll	1	FALSE	412C	0	0	4480
2E3C	ADVAPI32.dll	3	FALSE	411C	0	0	44CE

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
406C	CreateWindowExW	-	4394	4394	-	6E
4070	GetDesktopWindow	-	43A6	43A6	-	123
4074	CharUpperW	-	43BA	43BA	-	3C
4078	LoadIconW	-	43C8	43C8	-	1ED

Basics of a PE file: Imports

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



The screenshot shows the PE Explorer interface with the 'Imports' tab selected. The top table lists imported DLLs, with 'USER32.dll' selected. Below it, the 'USER32.dll [21 entries]' table is displayed. A red rectangle highlights the 'Thunk' column in this table, which contains addresses of exported functions from USER32.dll.

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA
2E00	KERNEL32.dll	17	FALSE	4134	0	0	42EC
2E14	USER32.dll	21	FALSE	417C	0	0	4462
2E28	GDI32.dll	1	FALSE	412C	0	0	4480
2E3C	ADVAPI32.dll	3	FALSE	411C	0	0	44CE

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
406C	CreateWindowExW	-	4394	768A2480	-	6E
4070	GetDesktopWindow	-	43A6	768AA100	-	123
4074	CharUpperW	-	43BA	768B53B0	-	3C
4078	LoadIconW	-	43C8	768B5890	-	1ED

Basics of a PE file: Imports

General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
+	+	+	+	+	+	+
Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	
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

DLL Base + Function RVA

Example:
 $768A0000 + 496FB = 768E96FB$

Disasm: .text

General

DOS Hdr

Rich Hdr

File Hdr

Optional Hdr

Section Hdrs

Exports

✱

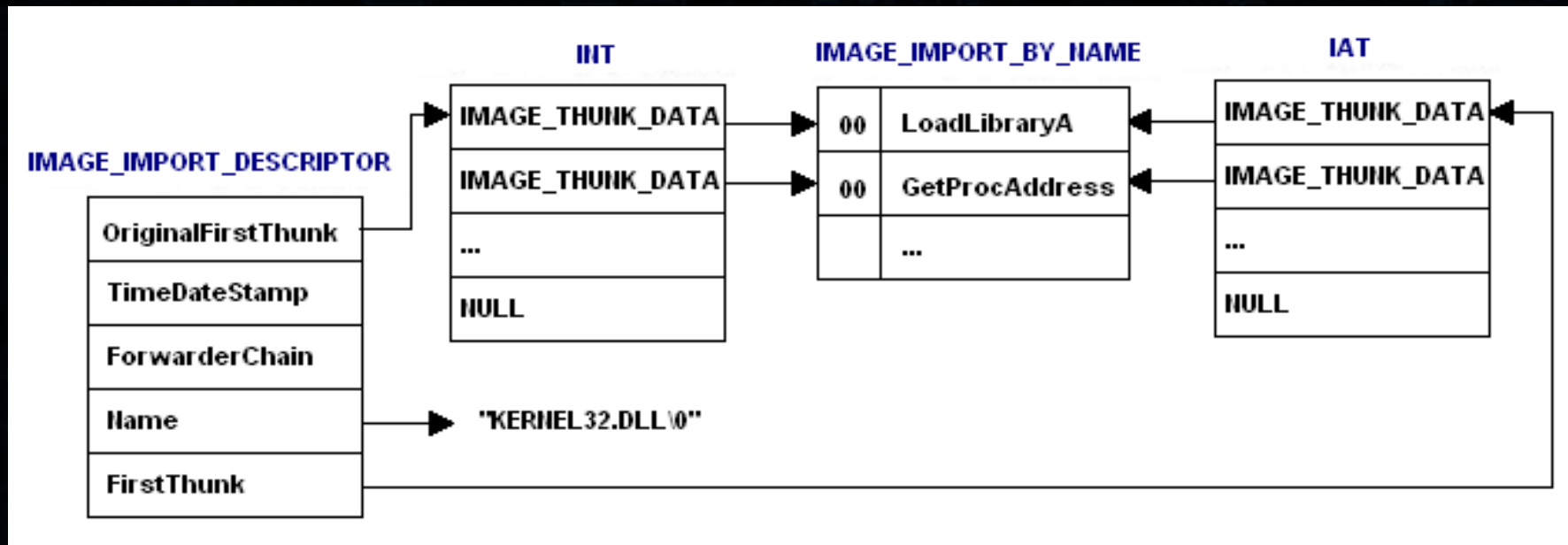
Offset	Name	Value	Meaning
B47CC	MajorVersion	0	
B47CE	MinorVersion	0	
B47D0	Name	B8502	KERNEL32.dll
B47D4	Base	1	
B47D8	NumberOfFunc...	54F	

Exported Functions [1359 entries]

Offset	Ordinal	Function	Name RVA	Name	Forwarder
B56E0	305	60E73	BCF4F	ReadConsoleW	
B56E4	3BF	42C62	BCF5C	ReadDirectoryChangesW	
B56E8	3C0	496FB	BCF72	ReadFile	
B56EC	3C1	63D99	BCF7B	ReadFileEx	

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_voll/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](https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN)