





Lecture 19: PE Loaders

Agenda

- PE File Format
- DOS Header
- NT Headers
- Imports
- Relocations (if time)

Next Time

- Writing a Packer
- Writing a Crypter
- What our PE Loader will not handle well
- Matryoshka Dolls and PE Loaders

PE File Format

Tools

In this lecture, we will use x64dbg, and PE-Bear to explore the PE file format.

As a sample, lets use Calc.exe (64 bit)

Run `$path = Get-Command calc.exe` to find the path to calc.exe on your machine

Run `PE-Bear.exe $path.Source` (in powershell)

Calc.exe

File Settings View Compare Info

calc.exe

- DOS Header
- DOS stub
- NT Headers
 - Signature
 - File Header
 - Optional Headers
- Section Headers
 - .text → EP = C70
 - .rdata
 - .data
 - .pdata
 - .rsrc
 - .reloc

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
C70	48	83	EC	28	E8	2B	FA	FF	FF	48	83	C4	28	E9	7E	FD		H	.	i	(+	ú	y	Y	H	.	A	(~	ý				
C80	FF	FF	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC		y	y	i	i	i	i	i	i	i	i	i	i	i	i	i			
C90	48	83	EC	28	48	8B	01	81	38	63	73	6D	E0	75	23	83		H	.	i	(H	.	.	.	8	c	s	m	à	u	#	.		
CA0	78	18	04	75	1D	8B	48	20	8D	81	E0	FA	6C	E6	83	F8		x	.	u	.	.	.	H	.	.	.	à	ú	1	æ	.	ø		
CB0	02	76	08	81	F9	00	40	99	01	75	07	FF	15	5F	09	00		.	v	.	ù	.	.	.	0	.	u	.	y	.	.	.			
CC0	00	CC	33	C0	48	83	C4	28	C3	CC	CC	CC	CC	CC	CC	CC		.	¿	3	A	H	.	.	.	A	(A	i	i	i	i	i		
CD0	48	83	EC	28	48	8D	0D	B5	FF	FF	FF	FF	15	9F	08	00		H	.	i	(H	.	.	.	µ	y	y	y	.	.	.			
CE0	00	33	C0	48	83	C4	28	C3	CC	CC	CC	CC	CC	CC	CC	FF	25		.	3	A	H	.	.	.	A	(A	i	i	i	i	i		
CF0	4C	09	00	00	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	48	83	EC	18		L	i	i	i	i	i	i	i			
D00	33	D2	48	8D	41	FF	48	83	F8	FD	77	3C	B8	4D	5A	00		3	0	H	.	A	y	Y	H	.	.	ø	y	w	<	.	M	Z	.

Disasm: .text	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports	Resources	Exception	BaseReloc	Debug	LoadConfig	Hint
1870	4883EC28												
1874	E82BFAFFFF												
1879	4883C428												
187D	E97EFDFFFF												
1882	CC												
1883	CC												
1884	CC												
1885	CC												
1886	CC												
1887	CC												
1888	CC												
1889	CC												
188A	CC												
188B	CC												
188C	CC												
188D	CC												
188E	CC												
188F	CC												
1890	4883EC28												
1894	488B01												
1897	813863736DE0												
189D	7523												
189F	83781804												
18A3	751D												
18A5	8B4820												
18A8	8D81E0FA6CE6												
18AE	83F802												
18B1	7608												
18B3	81F909409901												
18B9	7507												
18BB	FF155F090000												
18C1	CC												

DOS Header

- The DOS header contains the magic bytes MZ that identify it as a PE
- The final entry (referenced as `->e_lfanew`) is the address of NT Headers
- The next block of bytes contain the DOS stub

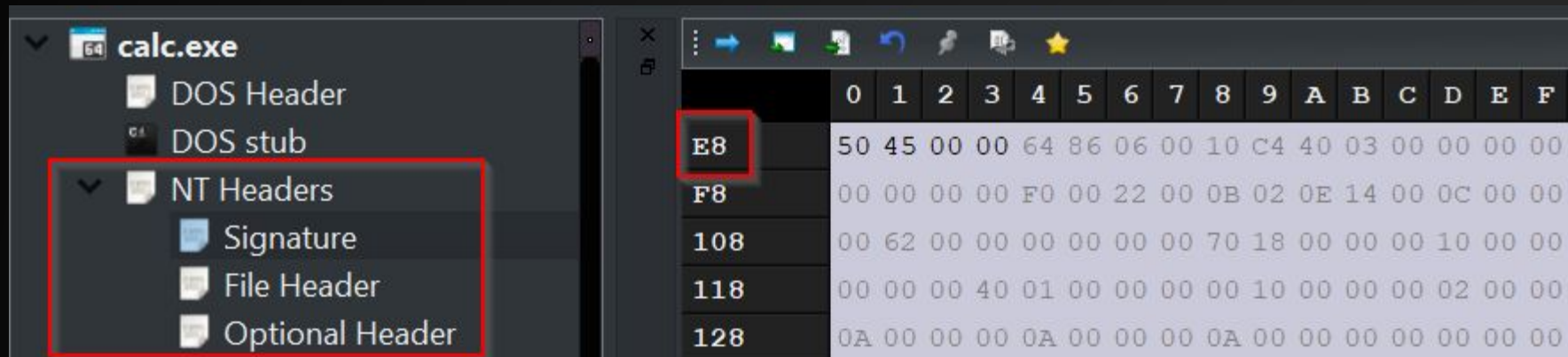
Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	E8

MZ magic Bytes

Address of NT Headers

NT Headers

- Signatures
- File Header
- Optional Header



calc.exe

- DOS Header
- DOS stub
- NT Headers
 - Signature
 - File Header
 - Optional Header

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E8	50	45	00	00	64	86	06	00	10	C4	40	03	00	00	00	00
F8	00	00	00	00	F0	00	22	00	0B	02	0E	14	00	0C	00	00
108	00	62	00	00	00	00	00	00	70	18	00	00	00	10	00	00
118	00	00	00	40	01	00	00	00	00	10	00	00	00	02	00	00
128	0A	00	00	00	0A	00	00	00	0A	00	00	00	00	00	00	00

Signature

- Usually 4 bytes containing
- “PE\0\0”
- For our purposes, it is only used to verify the file format.

A screenshot of a hex editor window. The top toolbar contains icons for navigation and editing. The main area displays a hex dump with columns for offsets (0-15) and corresponding byte values. The first row, starting at offset E8, shows the bytes 50, 45, 00, 00, which are highlighted in yellow. These bytes represent the ASCII string "PE\0\0". The second row shows bytes 64, 86, 06, 00, 10, C4, 40, 00, and the third row shows 00, 00, 00, 00, F0, 00, 22, 00, 0B, 02, 0E, 10. To the right of the hex dump, the ASCII representation of the bytes is shown, with "P E . ." corresponding to the first row.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E8	50	45	00	00	64	86	06	00	10	C4	40	00	00	00	00	00
F8	00	00	00	00	F0	00	22	00	0B	02	0E	10	00	00	00	00
108	00	62	00	00	00	00	00	00	70	18	00	00	00	00	00	00

File Headers

Following the Signature, we have the File Headers. This gives us

- The number of sections (NumberOfSections)
- Whether or not we have a DLL/EXE (Characteristics)
- The Compilation timestamp

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
Offset	Name	Value		Meaning			
EC	Machine	8664		AMD64 (K8)			
EE	Sections Count	6		6			
F0	Time Date Stamp	340c410		Friday, 24.09.1971 16:02:24 UTC			
F4	Ptr to Symbol Table	0		0			
F8	Num. of Symbols	0		0			
FC	Size of OptionalHeader	f0		240			
▼ FE	Characteristics	22					

Optional Headers

I don't know why it is listed as optional. I don't think a PE can run without this section (but I could be wrong?)

The optional headers will contain the data required to load the PE

Specifically, we will use values found here to build the IAT, and perform Base Relocations

Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	140000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. (Minor)	0	
134	Win32 Version Value	0	
138	Size of Image	8000	
13C	Size of Headers	400	
140	Checksum	14163	
144	Subsystem	2	Windows GUI
146	DLL Characteristics	C160	
		40	DLL can move
		100	Image is NX compatible
		4000	Guard
		8000	TerminalServer aware
148	Size of Stack Reserve	80000	
150	Size of Stack Commit	2000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	

Optional Headers (pt 1)

- Magic: Architecture of image
- Entry Point: Relative virtual address (RVA) from the Base Address
- Image Base: (preferred) Base address: Where in memory the PE “prefers” to be loaded. If the location is unavailable, the Image needs to be relocated

Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	140000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. Minor	0	
134	Win32 Version Value	0	
138	Size of Image	B000	
13C	Size of Headers	400	
140	Checksum	14163	
144	Subsystem	2	Windows GUI
146	DLL Characteristics	C160	
		40	DLL can move
		100	Image is NX compatible
		4000	Guard
		8000	TerminalServer aware
148	Size of Stack Reserve	80000	
150	Size of Stack Commit	2000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	

Optional Headers (pt 2)

- **SizeOfImage:** the virtual size of the image
- **SizeOfHeaders:** the size of the headers
- **DLLCharacteristics:** flags including knowledge of hardening features such as ASLR/ CFG...etc. Not super important for us other than assuming knowledge of ASLR.

Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	140000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. Minor	0	
134	Win32 Version Value	0	
138	Size of Image	8000	
13C	Size of Headers	400	
140	Checksum	14163	
144	Subsystem	2	Windows GUI
146	DLL Characteristics	C160	
		40	DLL can move
		100	Image is NX compatible
		4000	Guard
		8000	TerminalServer aware
148	Size of Stack Reserve	80000	
150	Size of Stack Commit	2000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	

Size of the Image in bytes, as well as the headers

Subsystem (are we a console application or GUI?)

Working with C/C++

Let's assume we have a pointer to a PE's bytes. I.e. imagine we read the raw bytes of a PE from disk, and have a pointer to the start of the buffer

There are several structures we can use to parse the bytes

We will assume that the Executables are all 64bit, but with a little bit of care, we can make our code work for 32bit and 64 bit PEs. (Note that we need a 32 bit exes to launch 32 bit applications. Same for 64 bit)

DOS Header:

```
typedef struct IMAGE_DOS_HEADER {    // DOS .EXE header
    WORD   e_magic;                  // Magic number
    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
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

DOS Header

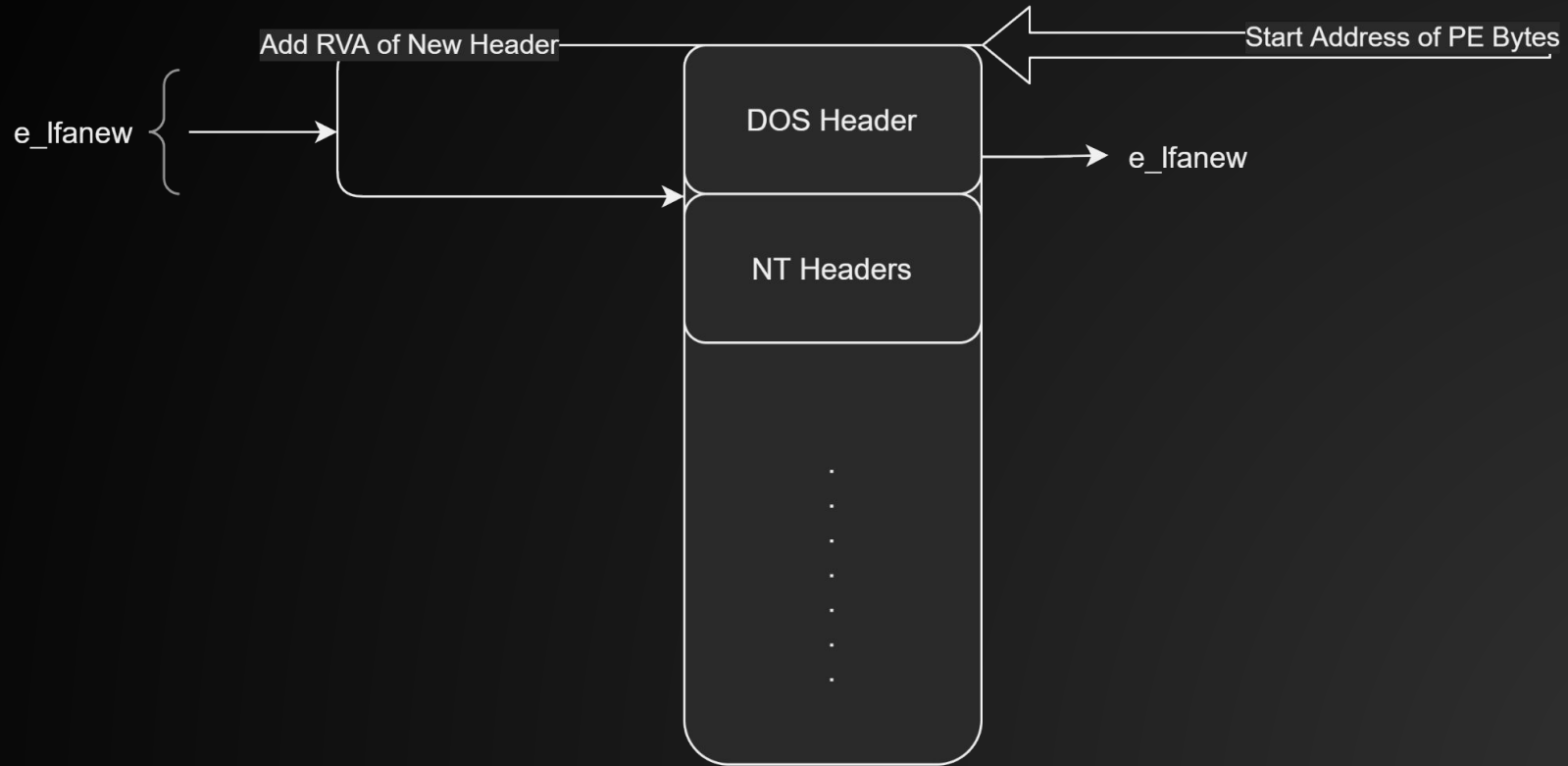
- If we have a pointer to the start of our PE Bytes, we can simply set the start of a DOS_HEADER struct to our PE bytes
-
- Defined in <winnt.h>
- ```
LONG e_lfanew; // File address of new exe header
```
- The offset above gives us the start address of the NT Headers

# NT Headers

To calculate the address of the NT Headers, we add the offset `e_lfanew` to the start of the address of the DOS Header, which is itself the start address of the PE Bytes.

- I.e. if the value of `e_lfanew` is  $X$ , and the start memory address of our PE buffer is  $Y$ , then the address of the beginning of the NT Headers struct is  $Y + X$

# NT Headers



# NT Headers

IMAGE\_NT\_HEADERS is actually a macro that expands to the relevant Struct for 32bit PEs and 64bit PEs respectively

32 bit: IMAGE\_NT\_HEADERS32

64 bit: IMAGE\_NT\_HEADERS64

# NT Headers 64 bit

```
typedef struct _IMAGE_NT_HEADERS64 {
 DWORD Signature;
 IMAGE_FILE_HEADER FileHeader;
 IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

# NT Header → FileHeader

```
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;
```

# NT Header → FileHeader

We need to Parse the

- **Machine** to ensure we have the correct Arch
- **NumberOfSections**: this allows us to iterate through all sections of the PE



# NT Headers → OptionalHeaders

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
 WORD Magic;
 BYTE MajorLinkerVersion;
 BYTE MinorLinkerVersion;
 DWORD SizeOfCode;
 DWORD SizeOfInitializedData;
 DWORD SizeOfUninitializedData;
 DWORD AddressOfEntryPoint;
 DWORD BaseOfCode;
 ULONGLONG ImageBase;
 DWORD SectionAlignment;
 DWORD FileAlignment;
 WORD MajorOperatingSystemVersion;
 WORD MinorOperatingSystemVersion;
 WORD MajorImageVersion;
 WORD MinorImageVersion;
 WORD MajorSubsystemVersion;
 WORD MinorSubsystemVersion;
 DWORD Win32VersionValue;
 DWORD SizeOfImage;
 DWORD SizeOfHeaders;
 DWORD CheckSum;
 WORD Subsystem;
 WORD DllCharacteristics;
 ULONGLONG SizeOfStackReserve;
 ULONGLONG SizeOfStackCommit;
 ULONGLONG SizeOfHeapReserve;
 ULONGLONG SizeOfHeapCommit;
 DWORD LoaderFlags;
 DWORD NumberOfRvaAndSizes;
 IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

## Optional Headers → \_\_\_\_\_

There is a lot going on in this structure but lets gleam a few important values from it that will come in handy:

- 1) ImageBase: a memory address (preferred) to load the image at. If we can't load the PE here, we should load it somewhere else.
- 2) AddressOfEntryPoint: the RVA of the PE's entry function. (I.e., ImageBase + AddressOfEntryPoint → Virtual Address of the entry point)
- 3) SizeOfHeaders: the size of the PE's headers

# Steps to Load a PE

- 1) Get a pointer to the start of your PE Bytes.
- 2) Parse the DOS Header, and the NT Header
- 3) Map the PE into Memory
- 4) Resolve Imports
- 5) Perform Base Relocations
- 6) Fix Section Memory Protections (technically optional but you shouldn't mark everything as R/W/X)
- 7) Pass execution to the entrypoint

Note steps 2 and 3 can be interchanged.

# Mapping the PE into Memory

From the optional Headers, we know the size of the PE is  
`ntHeader--> OptionalHeader-->SizeOfHeaders`

We can use this to Allocate enough memory to hold all of the sections of our PE

```
BYTE* ImageBase = (BYTE*)::VirtualAlloc(NULL,
 sizeofImage,
 MEM_RESERVE | MEM_COMMIT,
 PAGE_READWRITE
);
```

This gives us a pointer to a buffer in memory that we can read from and write to.

# Breaking down the VirtualAlloc Args

NULL: I want the OS to figure out where in virtual memory to give me space. Alternatively, we could pass in a specific address.

sizeofImage: how many bytes I want

MEM\_RESERVE | MEM\_COMMIT: reserve the contiguous memory, and ensure it is filled with 0s. Distinction between reserving and committing is a bit technical, so we omit details. But these are usually the parameters you want when calling VirtualAlloc

PAGE\_READWRITE: I want to be able to read from and write to this block of memory

# Mapping Sections

Recall that our PE has a preferred base address.

Since ASLR is enabled on most systems, there is a chance the address is unavailable.

We could make a call to `VirtualAlloc` with the preferred base address, check if it fails than set the parameter to `NULL` but in my experiment, we rarely get our desired address.

For simplicity, I skip this extra step, but just know we are deviating from what the actual PE loader does.

# Mapping the PE into Memory

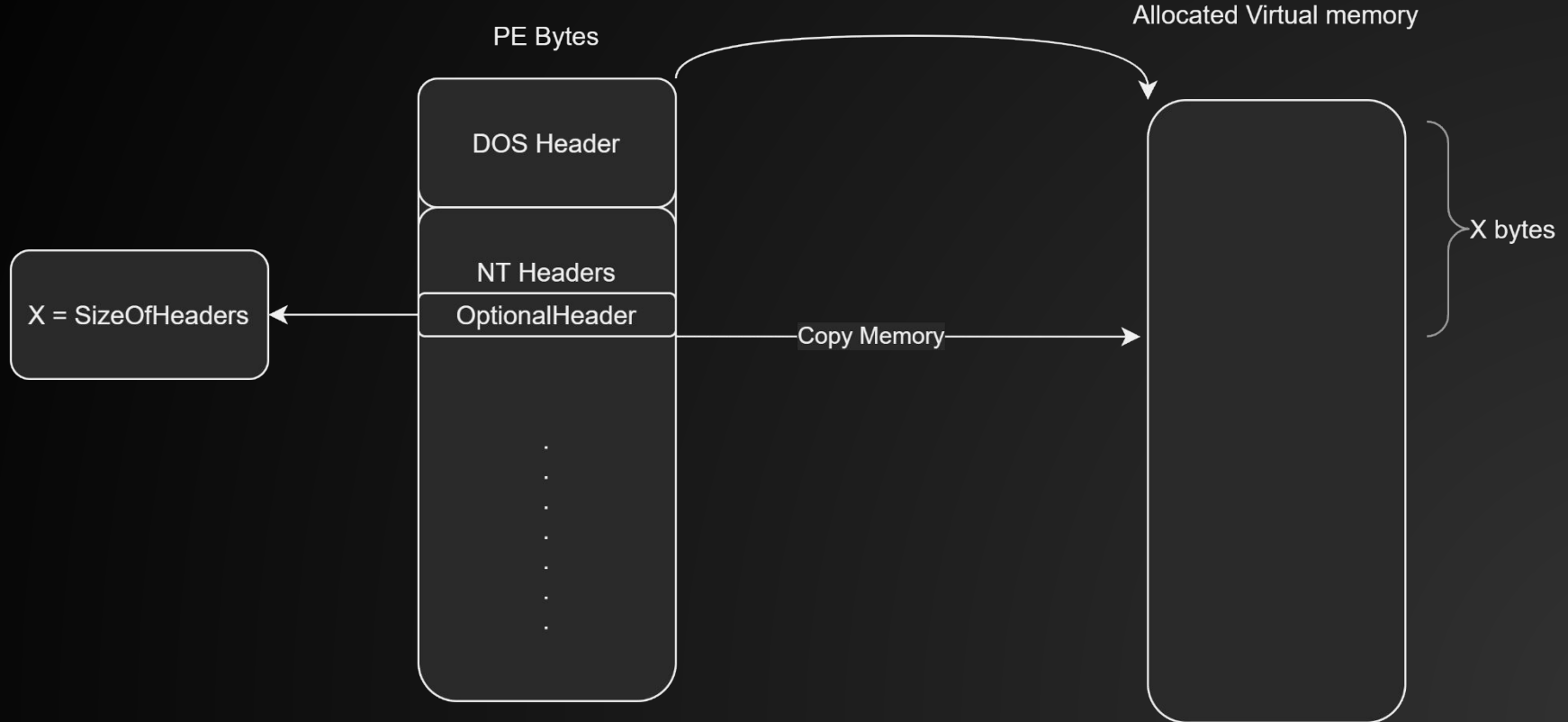
First, we need to copy the headers of the PE.

The first X bytes of the PE are the PE headers

X is located in the optional Header under `SizeOfHeaders`

To map the headers into memory, we simply copy X bytes starting at the beginning of the PE bytes, into the first X bytes of the newly allocated Buffer

# Copying The Headers into Memory





# PE Sections

# PE Sections Struct

```
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;
```

# Copying The Sections into Memory

We must now iterate over each section (i.e. `.text` `.data` `.rdata` ...etc) and map it into memory at the correct offset in our newly allocated buffer

The sections should come after the header

Sections are also structs whos definitions can be found in `winnt.h`

# Iterating over Sections

- There is a handy macro that will give us the address of the first section: `IMAGE_FIRST_SECTION()`
- This macro returns a pointer to an array of Image sections
- We retrieved the number of sections in the previous step, and we can now iterate

# Copying Sections

For each section struct, we need to get

- The VirtualAddress: RVA of the section
- PointerToRawData: a pointer to the section bytes
- SizeOfRawData: the number of bytes in the section

Add the RVA to the base address of the PE to get a pointer to the start of this section.

Copy SizeOfRawData bytes from PointertoRawData to BaseAddress + RVA

The Characteristic value also tells us what memory protection is required for the section

# Handling Imports

Now that we have all of our sections mapped into memory, it is time to build the IAT

The Imports are described in the PE Header inside of the Optional Header

The Data Directories is struct corresponding to a table

| ... | Number of RVAs and Sizes          | RV      |      |
|-----|-----------------------------------|---------|------|
| ▼   | Data Directory                    | Address | Size |
| ... | Export Directory                  | 0       | 0    |
| ... | Import Directory                  | 2794    | A0   |
| ... | Resource Directory                | 5000    | 4710 |
| ... | Exception Directory               | 4000    | F0   |
| ... | Security Directory                | 0       | 0    |
| ... | Base Relocation Table             | A000    | 2C   |
| ... | Debug Directory                   | 2320    | 54   |
| ... | Architecture Specific Data        | 0       | 0    |
| ... | RVA of GlobalPtr                  | 0       | 0    |
| ... | TLS Directory                     | 0       | 0    |
| ... | Load Configuration Directory      | 2010    | 118  |
| ... | Bound Import Directory in headers | 0       | 0    |
| ... | Import Address Table              | 2128    | 140  |
| ... | Delay Load Import Descriptors     | 0       | 0    |
| ... | .NET header                       | 0       | 0    |

# IAT

Recall that the Import Address Table (IAT) is a table of function pointers

These function pointers are references to functions from loaded Libraries

The notation we use to reference a function from a DLL is with a \$  
I.e. user32.dll\$MessageBoxA

# Building up the IAT

We locate the Import Address Table by parsing the optional headers, which contains the data directory

The DataDirectory is an array of `IMAGE_DATA_DIRECTORY` structure

```
typedef struct _IMAGE_DATA_DIRECTORY {
 DWORD VirtualAddress;
 DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

| ... | Number of RVAs and Sizes | ... | Address | Size |
|-----|--------------------------|-----|---------|------|
| ... |                          |     |         |      |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 2794    | A0   |
| ... |                          |     | 5000    | 4710 |
| ... |                          |     | 4000    | F0   |
| ... |                          |     | 0       | 0    |
| ... |                          |     | A000    | 2C   |
| ... |                          |     | 2320    | 54   |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 2010    | 118  |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 2128    | 140  |
| ... |                          |     | 0       | 0    |
| ... |                          |     | 0       | 0    |



# Getting the Import Address table

- Given the start address of the DataDirectory (cast as the correct type), we can reference the start of the Import Address Table
  - `ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT];`
- The IAT contains the functions address, but we don't know which entry the function pointer belongs to
- This information is contained elsewhere

# Import Directory Table

- The IDT is a null terminated array of `IMAGE_IMPORT_DESCRIPTOR` structs
- I sometimes absent mindedly call this the Import Descriptor Table (which is NOT a thing), because it contains the descriptions of required modules
- The Import Directory table, contains entries for modules loaded by the image
- The Loader uses this table to build the IAT with function pointers to absolute Virtual Addresses.
- Recall these entries should point to the correct offset in a loaded DLL

TLDR: this is where we find our null terminated array of DLLs required by our PE

# Import Lookup Table

The ILT is also a null terminated array of `IMAGE_THUNK_DATA` structs

`IMAGE_THUNK_DATA` expands to `IMAGE_THUNK_DATA32/64` depending on the architecture.

These structs contain the functions within the DLL that we need to load.

```
typedef struct _IMAGE_THUNK_DATA64 {
 union {
 ULONGLONGT ForwarderString; // PBYTE
 ULONGLONGT Function; // PDWORD
 ULONGLONGT Ordinal;
 ULONGLONGT AddressOfData; // PIMAGE_IMPORT_BY_NAME
 } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

# ILT Parsing

- The ILT can specify imported functions by either Name or Ordinal value
- For ordinal fields, the union in IMAGE\_THUNK\_DATA will have the most significant bit set to 1
  - In the above case, we can extract the the ordinal name from the least significant bytes
- If the bit is not set, the name of the function is contained in an IMAGE\_IMPORT\_BY\_NAME struct

# Data Directories

The directories appear in a fixed order, (Export Directory <\*> followed by Import Directory <\*> followed by Resource Directory <\*>...etc)

The values in the Data directory that we need to read/modify are

- 1) The import directory
- 2) Import Address Directory Table (IDT)
- 3) Import Address Table (IAT)

# Data Directories

The IDT gives us information about what imports are required for the PE to load

The IAT provides the programmer an interface to reference an RVA that will contain a trampoline to the actual address of the imported function at run time.

The Import Directory RVA points to a NULL terminated array of `IMAGE_IMPORT_DESCRIPTOR` structs

# IMAGE\_IMPORT\_DESCRIPTOR

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
 DWORD Characteristics; // 0 for terminating null import descriptor
 DWORD OriginalFirstThunk; // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
 } DUMMYUNIONNAME;
 DWORD TimeDateStamp; // 0 if not bound,
 // -1 if bound, and real date\time stamp
 // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
 // O.W. date/time stamp of DLL bound to (Old BIND)

 DWORD ForwarderChain; // -1 if no forwarders
 DWORD Name;
 DWORD FirstThunk; // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

# Iterating over Libraries

- For each Image import descriptor, attempt to get a handle to the library. If this fails, panic.
- Add the data directories `IMAGE_DIRECTORY_ENTRY_IMPORT` RVA to the base address to get the start of the NULL terminated array of `IMAGE_IMPORT_DESCRIPTOR` structs
- We must now walk this array.
- Each Image Descriptor will have a
  - → Name (the DLL to load)
  - → `IMAGE_THUNK_DATA*` which points to “THUNK” (functions resolved at a later date)
- Call `LoadLibraryA` on → Name
- For each image descriptor, we must now iterate through the list of functions required for that library



# Iterating over Functions in a Module

- Parse the OriginalFirstThunk and FirstThunk entry points for this library
- These entries are both null terminated `IMAGE_THUNK_DATA` structs.
- The OriginalFirstThunk entry points to an array of references to the functions to import from the external library. This is exactly the Import Lookup Table.
- FirstThunk points to a list of addresses that gets filled with pointers to the imported symbols. This is exactly the import Address table.

# IMAGE\_THUNK\_DATA

Also a macro that expands to the 32bit/64bit version

```
typedef struct _IMAGE_THUNK_DATA64 {
 union {
 ULONGLONG ForwarderString; // PBYTE
 ULONGLONG Function; // PDWORD
 ULONGLONG Ordinal;
 ULONGLONG AddressOfData; // PIMAGE_IMPORT_BY_NAME
 } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

# Image thunk Data

We recover the function name either via Ordinal or via Name.

We read the function name from our lookup table, as well as the RVA

The Function field needs to be patched with the functions address

We call GetProcAddress (either on the ordinal or the Name) and set the value of the lookup table to the address of the loaded function

Repeat this for each function, and each library!

# Debugging

## Tips for Debugging:

- Look at the entry point of the PE you are loading.
- Notice the symbols you see for functions are actually references to the IAT
- Break at the entrypoint of the manually loaded PE and compare it to the legitimately loaded PE
- If these symbols differ, the IAT was not created properly
- Example: detecting (an actual bug) that ate my Wednesday night.

# Tips

- Breakpoint at VirtualAlloc
- MessageBoxA ← just put these wherever you want to break at.  
Or you can compile with symbols/Configure VSCode.
- Breakpoint at CreateThread/However you pass execution.
  - Set a breakpoint at the entry point (in this case, register R8 should have the address of the entry point)
- Ctrl-f9 (execute until function return) is your friend!
- F8 step over a function call (run until return then call ret)
- F7 step into (follow RIP into the function)

# Tips Continued

- Debugging ACCESS\_VIOLATION:
  - This error means you
    - Tried to read memory you are not allowed to read
    - Tried to write to memory your are not allowed to write to
    - Tied to execute memory you are not allowed to execute
- For the PE loader, you either
  - Messed up the memory protections
  - Have invalud function pointers in your IAT
  - Did not perform relocations properly
  - \*Other PE stuff (TLS callbacks)

# Relocations

- Recall that the ImageBase is the preferred base address of the PE once it is mapped into memory
- If we were able to give the PE its preferred address, then we are done! Pass execution off to the entry point.
- If not, we have more work to do.
- If the PE has acknowledged the existence of ASLR, we can map it into a different location. If not, panic.

# Dealing with Absolute Addresses

Our PE, while aware of ASLR, might still references addresses that are absolute addresses.

In this case, we need to find all of them, calculate their offset from the preferred address, and patch them

The PE file format makes this easy for us with a structure called the relocation table

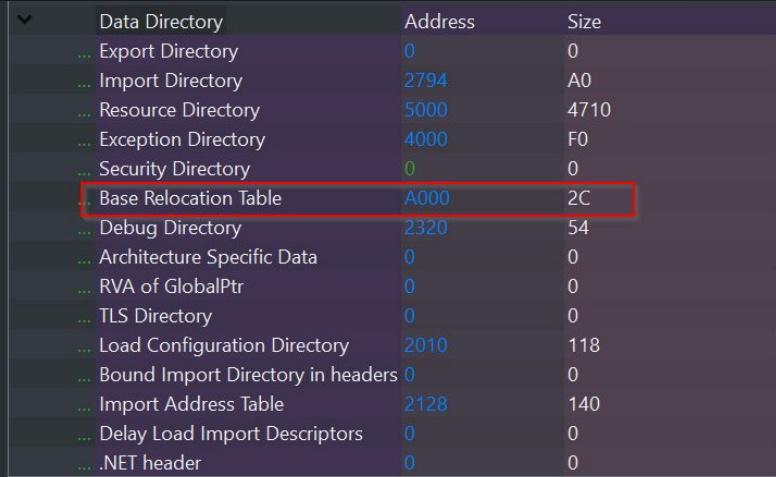


# IMAGE\_BASE\_RELOCATION

The VirtualAddress is a relative Virtual address, where we can relocate a block

The SizeOfBlock is...well the size of the block

```
typedef struct _IMAGE_BASE_RELOCATION
{
 DWORD VirtualAddress;
 DWORD SizeOfBlock;
} IMAGE_BASE_RELOCATION, *PIMAGE_BASE_RELOCATION;
```



|     | Data Directory                    | Address | Size |
|-----|-----------------------------------|---------|------|
| ... | Export Directory                  | 0       | 0    |
| ... | Import Directory                  | 2794    | A0   |
| ... | Resource Directory                | 5000    | 4710 |
| ... | Exception Directory               | 4000    | F0   |
| ... | Security Directory                | 0       | 0    |
| ... | Base Relocation Table             | A000    | 2C   |
| ... | Debug Directory                   | 2320    | 54   |
| ... | Architecture Specific Data        | 0       | 0    |
| ... | RVA of GlobalPtr                  | 0       | 0    |
| ... | TLS Directory                     | 0       | 0    |
| ... | Load Configuration Directory      | 2010    | 118  |
| ... | Bound Import Directory in headers | 0       | 0    |
| ... | Import Address Table              | 2128    | 140  |
| ... | Delay Load Import Descriptors     | 0       | 0    |
| ... | .NET header                       | 0       | 0    |

# Relocations

If the Base Address of the allocated memory matches the preferred base address of the PE, there is nothing to be done!

Else, we might need to patch some values.

We can parse the relocation table from the option headers inside of the Data Directory

We can also compute the “delta” in base addresses by taking the preferred base address and subtracting it from the mapped address.

# Base Relocation

All memory addresses in the code and data sections are stored relative to the address defined by ImageBase in the OptionalHeader.

If the PE can't be loaded at its preferred memory address, the references must reflect this!

The PE stores informations about all these references in the base relocation table inside of the Data Directory

# Relocation

Each entry has  $(\text{SizeOfBlock} - \text{IMAGE\_SIZEOF\_BASE\_RELOCATION}) / 2$  entries where each entry is a word.

The first 4 bits define the type of relocation

the lower 12 bits define the offset relative to the VirtualAddress.

```
typedef struct _IMAGE_BASE_RELOCATION {
 DWORD VirtualAddress;
 DWORD SizeOfBlock;
} IMAGE_BASE_RELOCATION;
```

# It's just a switch case

IMAGE\_REL\_BASED\_ABSOLUTE : nothing to be done

IMAGE\_REL\_BASED\_HIGHLOW: Add the delta between the preferred base address and the allocated memory block to the 32 bits found at the offset.

IMAGE\_REL\_BASED\_DIR64: Same as above, but addresses are 64 bit.

# TLS Callbacks

I am just going to give you this one.

If you want to use multiple threads, you need to run TLS callbacks that prepare Thread Local Storage. Run this right before passing off execution

```
void HandleTLSCallbacks (void* lpImageBase, IMAGE_NT_HEADERS* ntHeaders){
 if(ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].Size)
 {
 PIMAGE_TLS_DIRECTORY tls = (PIMAGE_TLS_DIRECTORY)((UINT_PTR)lpImageBase +
ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress);
 PIMAGE_TLS_CALLBACK *callback = (PIMAGE_TLS_CALLBACK *) tls->AddressOfCallBacks;

 while(*callback)
 {
 wprintf(L"[+] TLS callback at %p\n", (void*) callback);
 (*callback)((LPVOID) lpImageBase, DLL_PROCESS_ATTACH, NULL);
 callback++;
 }
 } else{
 wprintf(L"No Tls Callbacks! \n");
 }
}
```

# Packers/Crypters

- Yara rules will help us identify static content in PEs.
- We can no longer fully rely on them to detect the malware
- Since we can run a PE from memory, we can embed a compressed, and encrypted PE inside of another PE
- We can, however, use it to detect the stub that unpacks the malware
- Usually though, we want to unpack the malware to identify not just the packer but also the final payload
- This also has implications in how First/second stage payloads work

# UnPacking Malware

Let's assume for now that the unpacked payload is a .exe PE.

Most stubs that unpack malware need to do the following

- 1) Recover the raw PE Bytes (decompress, decrypt...etc)
- 2) Allocate memory, some of which needs to be executable
- 3) Copy the PE into memory
- 4) Resolve Import Address Table
- 5) Handle Relocations
- 6) TLS callbacks
- 7) Pass execution to the entry point



# Unpacking: Useful Breakpoints

- As always, look at the imports. There might, however, might not be any!
- If we can catch the loader/packer before it passes off execution, we can simply dump the PE from memory
  - Note that the PE we dump might be memory mapped, in which case we will need to perform the opposite of what our loader does!
- Set a Breakpoint at VirtualAlloc(Ex), or other functions used to allocate memory (HeapAlloc, NtAllocateVirtualMemory...etc)
  - Run until Return. The value in RAX will be the base address of the allocate memory
  - Follow this address in dump and keep an eye on this until you see the sections copied into memory

# Matryoshka Dolls

- Malware authors will routinely Compose Packers
- This can get very annoying, very fast.
- You can actually run the same packer multiple times.
- For example, we can UPX pack our loader to load a UPX packed exe.
- First, we need to unpack the real loader, then we unpack the packed payload, then we can dump the actual payload



# Building your own Packer

- Compile a stub to unpack your code that decrypts, and decompresses data embedded in the PE. An example of this could be a resource file
- Python pefile makes interacting with PE resources easy
- This way, you can use python to compress and encrypt your payload, and you can embed the result as a resource in a fixed place.
- The primary purpose of a packer is two fold:
- 1) Hash Busting: old signatures no longer work on the packed malware. You can (and should!) write yara rules for the stub-- but it is very easy to also hash bust the stub!
- 2) Frustrating the reverse engineer.

# Automation

- Scripting a debugger : set breakpoints at common functions found in Packers and scan memory for the magic MZ
- Unpack.me: unpacking sandbox. Sadly for you all, it is 32bit only >:)
- PE-Sieve: incredible tool by Hasherezade that checks for injected PEs/Shellcode in a process
- Writing a static unpacker for a specific packer you see a lot can take time, but depending on the circumstances could also be worth it!