

Life Of Binaries

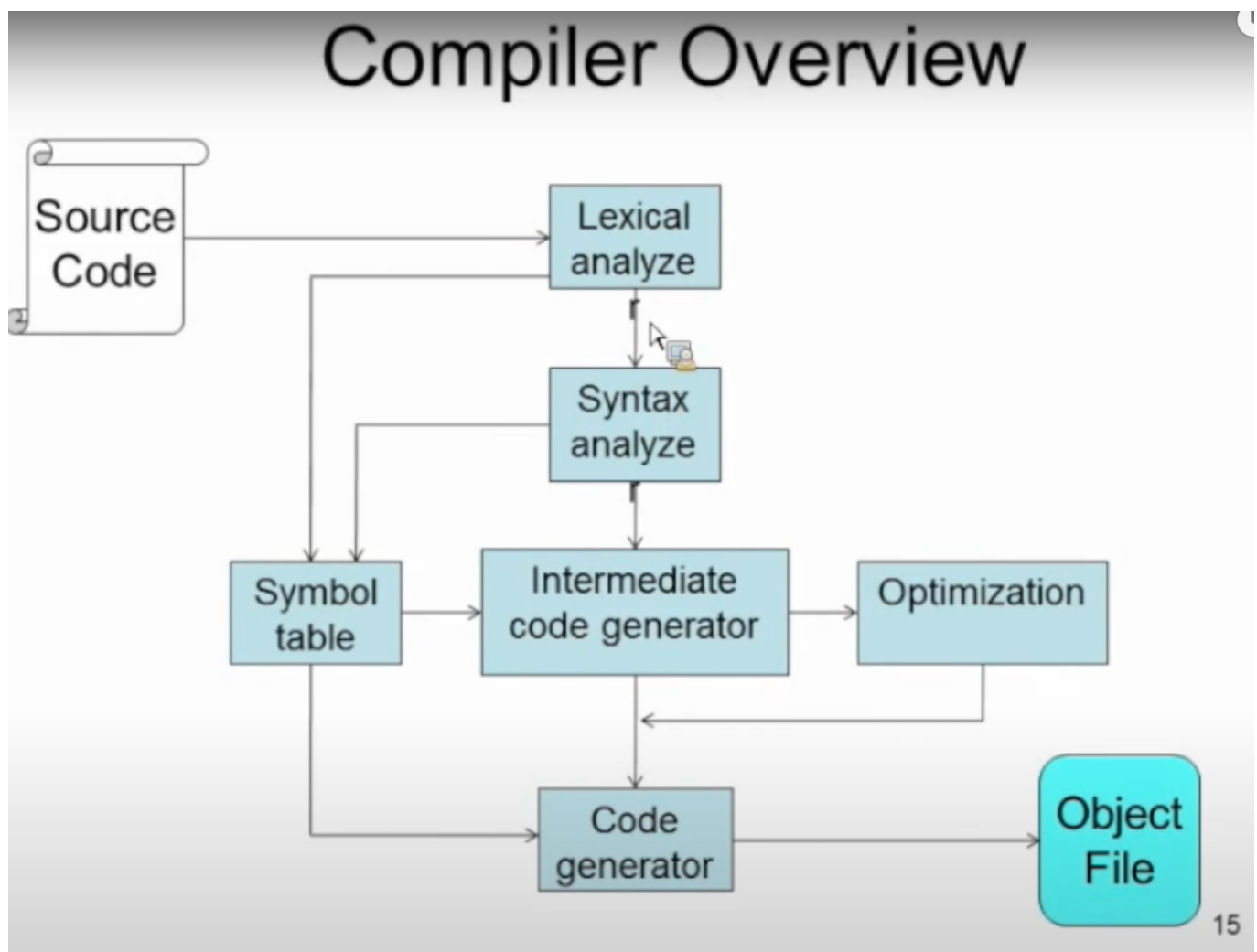
When we write a program in C or C++, and compile it, first, we flat everything on the surface. like all header files, all external/internal libraries are taken from their sources and compiler compiles this code. At the end of the process, we are ended up with a object file, **a.o**

so during the linking process, linker links all the object and other components and links them together to create the binary.

Upon running, the binary is processed by the OS loader and dynamic linker(runtime linker) for linking the libraries.

so lets say we used **printf**, in runtime linker period, it is obvious that we did not write the function, it is defined elsewhere so it marks down in the binary that **this person needs printf from this library.**

And so loader takes them to memory and you get your program running afterwards.



So linker is basically :

There are 2 or more objects spit out from the previous process, they need to be linked together in the desired way. So this sorts and orders these objects and

their internals to create an executable or a library file.

Compilers:

compilers go with the expression trees. In the code, there is an expression tree where code and data are in harmony. For example you declare a variable, and only after that you use it. So, it is like create stack, open space for var, assign value to a var etc. this is the tree.

Loader loads the binary and checks the FP. Frame pointer register. It can be ARM, X86, PPC etc. In case of X86, starts with the regular stack building.

push ebp.

then, for sake of argument, we have `int a = 4;` what happens, push 4 here. So compiler follows this kind of a tree logic.

Executables:

Windows ==> PE (Portable Executable)

Linux ==> ELF (Executable and linkable Format)

Mac ==> Mach-o (Mach Object)

There are different target binary formats :

Executable:

.exe on windows, no suffix on Linux. A program which will either stand completely on its own containing all necessary code, or which will request external libraries that it will depend on.

Dynamic Library:

.dll on windows and .so in linux. (.so is shared object) needs to be loaded by some other program in order for any of the code to be executed. the library may have some code which is **automatically** executed at load time. (the `DllMain()` on windows or `init()` on linux) This is as opposed to a library which executes none of its own code and only provides codes to other programs

So these have `main()` functions basically, when they are called, they get executed

It starts running initialization code, and step by step calls other functions or libraries.

this is where attackers employ **DLL injection attacks**

Static Library:

.lib on windows and .a on Linux. Static libraries are a bunch of object files with some specific header info to describe the organization of files.

these are used when you want to compile all files together to later be linked against statically. so you say to your linker basically that do not use the `printf` or `scanf` function in the standard library but use the one I gave you through the static library.

**Common Windows PE file Extensions: **

- exe ==> executable file
- dll ==> dynamic link library
- sys/drv ==> System File (kernel driver)
- ocx => ActiveX control
- .cpl==>Control panel
- .scr ==> screensaver

So screensavers are full executables, which can deliver malware!

PE DOS HEADER:

When a PO Dos file is opened and seen, there are bunch of header files are imported and executed. Two of these are

WORD e_magic ==> magic number and **LONG e_lfanew** ==> file address of new exe header, its an offset to the next instruction

These are contained, alondside many other, under a typedef struct IMAGE_DOS_HEADER typedef function.

so this is a exe header.

e_magic ==> is always going to be set to ASCII 'MZ' which is from Mark Zbikowski who developed MS-DOS

For mot windoes programs the DOS header contains a stub DOS program which does nothing but print out **This program cannot be run in DOS mode**

e_lfanew ==> this is what we care about mostly. this specifies a file offset where PE header can be found(a file pointer gibi dusun.)

Ben **Peview** ile rastgele bir program actim mesela 😊 karsima cikan ilk sey :

```
00000040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ....!..L.!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  F3 86 9A 8E B7 E7 F4 DD  B7 E7 F4 DD B7 E7 F4 DD  ....
```

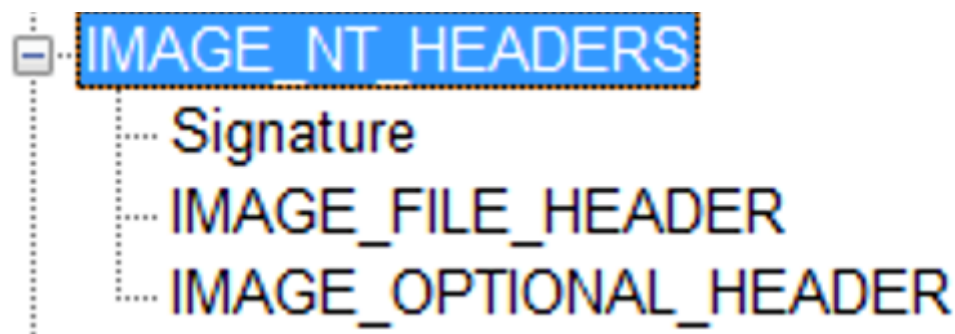
also at the very first line is MZ 😊

and also we have file offset.

so there is .exe, there is 'MZ' and rthere is DOS MODE sign. three of these are indicates that we are dealign with a windows file 😊

PE NT HEADER ED HEADER:

this is Image NT Headers. which contains 3 headers actually. signature, file header, optional header.



Signature == 0x00004550 also known as ASCII string "PE" in little endian order in DWORD. Otherwise, just a holder for two other **embedded** (not pointed to) structs.

IMAGE_FILE_HEADER and **IMAGE_OPTIONAL_HEADER** are embedded in there, they are not pointed at.

IMAGE_FILE_HEADER

We care about couple of stuff in this header.

the important ones are :

WORD Machine

Machine specifies what architecture this is supposed to run on. This is our first indicator about 32 or 64 bit binary.

check the data section in the header.

Value of **014C** = x86 binary, PE32 binary.

Value of **8664** = X86-64 binary yani AMD64 yani 64bit yani PE32+ binary.

8664 makes sense, 86_64 😊

This does not have to be 100 percent accurate, but gives a clue.

WORD NumberOfSections

Tells how many section headers will be there later.

DWORD TimeDateStamp

this is pretty interesting field. It is a Unix timestamp (seconds since epoch where epoch is 00:00:00 UTC on Jan 1, 1970) and is set at link time.

so basically how many seconds have elapsed since the epoch.

Can be used as a unique version for the given file. (this file is compiled on tatata timestamp.) can be used to know when a file was linked (**useful for determining whether an attacker tool is fresh or correlating with other forensic evidence keeping in mind that attackers CAN manipulate it.**)

WORD Characteristics

Characteristics field is used to specify many things like line numbers that are stripped, 32 bit word machine, that it is a system file, it can handle >x GB of ram etc.

IMAGE_OPTIONAL_HEADER :

this is a field we do not really care about. SizeOfOptionalHeader can theoretically be shrunk to exclude data directory fields which the linker does not need to include. PointerToSymbolTable , NumberOfSymbols not used anymore now that the debug info is stored in a separate file.

OPTIONAL HEADER:

Optional header is not at all optional! It has to be there!

In this header there are 6 entries we care about:

Magic

Optional Header 0.

Magic is the true determinant of whether this is a PE32 or PE32+ binary. Depending on the value, the optional header will be interpreted as having a couple of 32 or 64 bit fields.

0x10B ==> 32 bits, PE32 **0x20B** ==> 64 bit , PE32+

0x14C is giving hint maybe, in file headers. but the magic number in optional headers tells OS to how to load, so it is determinant of architecture.

so one field(0x14c) can say it is 32 but this may say it is 64, we take this one.

AddressOfEntryPoint

specifies the RVA(relative address) of where the loader starts executing code once it is completed loading the binary. Do not assume it just points to the beginning of .text section, or even the start of main().

This is basically saying once you finish all loading and stuff, go to this address and start executing starting from this address.

this is very important!

so you have a malware and you dont know where it begins executing or it is stripped off, what you can do is to check this value and put a breakpoint for the debugger AND YOU CATCH IT BEFORE IT RUNS ANY CODE!

Except for one caveat, which is TLS, which will be covered later on.

ImageBase

Image Base is the information pertains to where this program wants to be located in the memory. So in 64 bit system, it wants a 64 bit base, which corresponds to ULONGLONG type

In 64 bits, the type is ULONGLONG.

Thus, it specifies the preferred virtual memory location where the beginning of the binary should be replaced. If the binary cannot be loaded at ImageBase(for example something else is running that memory) then the loader picks an unused memory range. Then , every location in the binary which was compiled

assuming that the binary was loaded at ImageBase must be fixed by adding the difference between the actual ImageBase minus desired Image base.

The list of places must be fixed and kept in a special **relocations** (.reloc) section.

Hence, Microsoft does not support position-independent code.

On linux systems, shared libraries will be executed as position independent code.

SectionAlignment

specifies that sections must be aligned on boundaries which are multiples of this value. for example, if it is 0x1000, then you might expect to see sections starting agt 0x1000, 0x2000, 0x5000 etc.

0x1000 is size of a page in intel 32 systems.

FileAlignment

data was written to binary in chunks no smaller than this value. some common values are 0x200, (512, the size of a HDD sector) and 0x80,(size of sector in floppy.) SSDs use 4KB size.

this is from AskUbuntu.com:

In the old days, 512 byte sectors was the norm for disks. The system used to read/write sectors only one sector at a time, and that was the best that the old hard drives could do.

Now, with modern drives being so dense, and so fast, and so smart, reading/writing sectors only one sector at a time really slows down total throughput.

The trick was... how do you speed up total throughput, but still maintain compatibility with old/standard disk subsystems? You create a 4096 block size that are made up of eight 512 byte physical sectors. 4096 is now the minimum read/write transfer to/from the disk, but it's handed off in compatible 512 byte chunks to the OS.

This means that even if the system only needs one 512 byte sector of information, the drive reads eight 512 byte sectors to get it. If however, the system needs the next seven sectors, it's already read them, so no disk I/O needs to occur... hence a speed increase in total throughput.

Modern operating systems can fully take advantage of native 4K block sizes of modern drives.

Also according to Intel.com, **Intel® Solid State Drives support a 512 byte and 4096 byte (4K) physical sector size.**

EZ. Take this much of a bunch and write it this much. 512 bloklar halinde yani.

If you don't have multiples of hex 0x200, then you need to put some padding file before putting the next section. Let's say your file is 10 bytes,

you write it and pad it until the next file offset, end of 512. and keep writing the next section

SO FILE ALIGNMENT IS HOW YOU ALIGN ITEMS ON DISK WHEREAS SECTION ALIGNMENT IS HOW YOU ALIGN THEM ON MEMORY

SizeOfImage

this is the amount of contiguous memory that must be reserved to load the binary into memory.

As loader goes through code and accumulates all the required amount of memory, this plus this plus this etc, so it knows exactly how much memory it needs. `SizeOfImage` basically gives this total amount.

This is the total size of the binary once it is mapped to the memory

DllCharacteristics

Specifies some more important **security options like ASLR and non-executable memory regions for the loader and the effects are not limited to DLLs**

- **DLLCharacteristics** specifies some important security options like ASLR and non-executable memory regions for the loader, and the effects are not limited to DLLs.


```

#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040 // DLL can move.
#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY 0x0080 // Code integrity image
#define IMAGE_DLLCHARACTERISTICS_NX_COMPAT 0x0100 // Image is NX compatible
#define IMAGE_DLLCHARACTERISTICS_NO_SEH 0x0400 // Image does not use SEH. No SE handler may reside in this image
      
```
- `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` is set when linked with the `/DYNAMICBASE` option. This is the flag which tells the OS loader that this binary supports ASLR. Must be used with the `/FIXED:NO` option for .exe files otherwise they won't get relocation information.
- `IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY` says to check at load time whether the digitally signed hash of the binary matches.
- `IMAGE_DLLCHARACTERISTICS_NX_COMPAT` is set with the `/NXCOMPAT` linker option, and tells the loader that this image is compatible with Data Execution Prevention (DEP) and that non-executable sections should have the NX flag set in memory (we learn about NX in the Intermediate x86 class)
- `IMAGE_DLLCHARACTERISTICS_NO_SEH` says that this binary never uses structured exception handling, and therefore no default handler should be created (because in the absence of other options that SEH handler is potentially vulnerable to attack.)

If the **Fixed Base Address** is set to Generate a relocation section, it means that `/FIXED:NO` flag is set and it generates relocations.

pFile	Data	Description	Value
000000F8	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000FA	0A	Major Linker Version	
000000FB	00	Minor Linker Version	
000000FC	00000A00	Size of Code	
00000100	00000C00	Size of Initialized Data	
00000104	00000000	Size of Uninitialized Data	
00000108	000012B1	Address of Entry Point	
0000010C	00001000	Base of Code	
00000110	00002000	Base of Data	
00000114	00400000	Image Base	
00000118	00001000	Section Alignment	
0000011C	00000200	File Alignment	
00000120	0005	Major O/S Version	
00000122	0001	Minor O/S Version	
00000124	0000	Major Image Version	
00000126	0000	Minor Image Version	
00000128	0005	Major Subsystem Version	
0000012A	0001	Minor Subsystem Version	
0000012C	00000000	Win32 Version Value	
00000130	00000000	Size of Image	
00000134	00000400	Size of Headers	
00000138	0000A6D4	Checksum	
0000013C	0003	Subsystem	IMAGE_SUBSYSTEM_WINDOWS_CUI
0000013E	8140	DLL Characteristics	
	0040		IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
	0100		IMAGE_DLLCHARACTERISTICS_NX_COMPAT
	8000		IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE
00000140	00100000	Size of Stack Reserve	
00000144	00001000	Size of Stack Commit	
00000148	00100000	Size of Heap Reserve	
0000014C	00001000	Size of Heap Commit	
00000150	00000000	Loader Flags	

in this picture, the part under the DLLsections on the right panel shows some info

DynamicBase == ASLR is ON NX COMPAT == DEP/NX feaure

and on the left panel, down there are sections , particularly **SECTION .reloc** this holds the data for relocations

So Does this support ASLR? means is the Dynamic Base Flag is set?.

DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

Array of officially 16 things and this is the storage of pointers from all of this other data structure that we are go over.

This holds pointers of what exports it holds,

it holds pointers to what functions and import it holds

it holds the pointers where the debug infromation can be found, where the digital signature is, where the relocation informatiion is, where the resources are etc.

super informative header!!

It is like a big map that points at tll the other data structures.

An example can be seen below:

00000154	00000010	Number of Data Directories	
00000158	00000000	RVA	EXPORT Table
0000015C	00000000	Size	
00000160	00002224	RVA	IMPORT Table
00000164	0000003C	Size	
00000168	00004000	RVA	RESOURCE Table
0000016C	000001B4	Size	
00000170	00000000	RVA	EXCEPTION Table
00000174	00000000	Size	
00000178	00000000	Offset	CERTIFICATE Table
0000017C	00000000	Size	
00000180	00005000	RVA	BASE RELOCATION Table
00000184	00000148	Size	
00000188	000020D0	RVA	DEBUG Directory
0000018C	0000001C	Size	
00000190	00000000	RVA	Architecture Specific Data
00000194	00000000	Size	
00000198	00000000	RVA	GLOBAL POINTER Register
0000019C	00000000	Size	
000001A0	00000000	RVA	TLS Table
000001A4	00000000	Size	
000001A8	00002108	RVA	LOAD CONFIGURATION Table
000001AC	00000040	Size	
000001B0	00000000	RVA	BOUND IMPORT Table
000001B4	00000000	Size	
000001B8	00002000	RVA	IMPORT Address Table
000001BC	000000A8	Size	
000001C0	00000000	RVA	DELAY IMPORT Descriptors
000001C4	00000000	Size	
000001C8	00000000	RVA	CLI Header
000001CC	00000000	Size	
000001D0	00000000	RVA	
000001D4	00000000	Size	

Playing bitHunt Scavenger level 2 here is my result

```
Correct! Score = 2000

Congratulations, you passed round 2!
It took you 22 minutes, 2 seconds for this round.
And so far it's taken you a total time of 22 minutes, 2 seconds.

Oh no! You've been teleported back to previous rounds!
Show that you haven't forgotten all the old material already!

What is the IMAGE_DOS_HEADER.e_magic in ASCII?
Answer: 0
```

SECTION HEADERS:

Section group portions of code or data which have similar purpose, or should have similar memory permissions.

Number of section headers can be found where? at **File headers** section.

This groups similar memory permissions, bunches them up and then let the OS loader loads same permissions once they are mapped in memory

So think of the role of linker? links all these different objects and data etc? It works like this:

Take this bunch, this is executable but not writable; this data is only readable, that one is executable and writable. these bunches and permissions.

and then linker links all of them into the final form of the binary

so what are the common section names?

- **.text**

Code which should never be paged out of memory to disk. This is where the actual code goes. Especially in windows kernel, some code can be load onto memory or etc, but .text one is always non-pageable, it never is paged to disk.

it always stays on memory

- **.data**

read/write data(globals). So for example some code resides in stack, like local variable etc. (in linux this is .rodata) these are strings, global variables that can be used in different places.

- **.rdata**

read-only data(strings)

- **.bss**

Block Started by Symbol or Block Storage Start, depending on who you ask 😊 In practice, the .bss seems to merged into the .data section by the linker for the binaries

Takes up space in memory but not in disk. actually linker links and merges the bss section with the text section or it is merged with the data section.

- **.idata**

Import address table. In practice, seems to get merged with .text or .rdata

- **.edata**

Export information

All these can be seen across different file formats, different operating systems. so it is a shared value.

idata and edata will get merged with .data section so we won't see them in PE analysis section in section.

- **PAGE***

code/data which it is fine to page out to disk if you are running low in memory.

Essentially means that these section names in kernel modules will be like **Page verify** or **Page Log**(lock?). so these sections are prefixed with **PAGE**.

- **.reloc**

Relocation information for where to modify hardcoded addresses which assume that the code was loaded at its preferred base address in memory.

if data needs to be moved around, it is needed to be known that this is the data structure, here is the important piece of information etc.

- **.rsrc**

Resources. Lots of possible stuff from icons to other embedded binaries. the section has structures organizing it sort of like a filesystem.

some rootkits actually hide behind this and show themselves as resources, then when they are dumped they are loaded into kernel modules and running in the kernel etc.

Stuxnet for example, used explorer.exe and .rsrc to load itself into the system.

- **.pdata**

ARM, MIPS, and SH Windows CE compiles use PDATA structures to aid in stack walkint at run-time. This structure aids in debugging and exception processing.

NOTE: IMMEDIATELY AFTER THE OPTIONAL HEADER , THE OS LOADER EXPECTS DATA SECTION. IF THERE IS EVEN A BYTE LONG PADDING, IT CONFUSES THE OS LOADER.

this is how Section Headers are :

Round1Q0.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000288	00000290	00000294	00000298	0000029C	000002A0	000002A4	000002A8	000002AA	000002AC
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000008AE	00001000	00000A00	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000057A	00002000	00000600	00000E00	00000000	00000000	0000	0000	40000040
.data	00000358	00003000	00000200	00001400	00000000	00000000	0000	0000	C0000040
.rsrc	000001B4	00004000	00000200	00001600	00000000	00000000	0000	0000	40000040
.reloc	00000156	00005000	00000200	00001800	00000000	00000000	0000	0000	42000040

lets explain those column names:

SECTION HEADERS EXPLAINED:

- **NAME[8]**

Name is a byte array of ASCII characters. It is NOT guaranteed to be **null terminated**. So if you are trying to parse a PE file yourself oyu need to be aware of that.

You can put whatever you want in there.it is generally for the human or linker, loader does not care about it at all.

- **Virtual Address**

is the RVA of the section relative to OptionalHeader.ImageBase

- **Pointer to Raw Data**

is a relative offset from the beginning of the file which says where the actual section data is stored.

NOTE

RVA == the virtual address.

where is it mapped on the memory?

Actual Value = ImageBase + RVA.

Because RVA is an offset, we can just add RVA and ImageBase and get the Actual base value, which is the mapped address on the memory

PE IMPORTS:

Static Linking vs Dynamic Linking:

Static linking is on compile time VERSUS dynamic is runtime!

with static linking, you literally just include a copy of every helper function you use inside the executable you are generating.

Dynamic linking is when you **resolve pointers** to functions inside libraries **at runtime**

A statically linked executable is bloated compared to a dynamically linked one. But, on the other it is standalone, without outside dependencies.