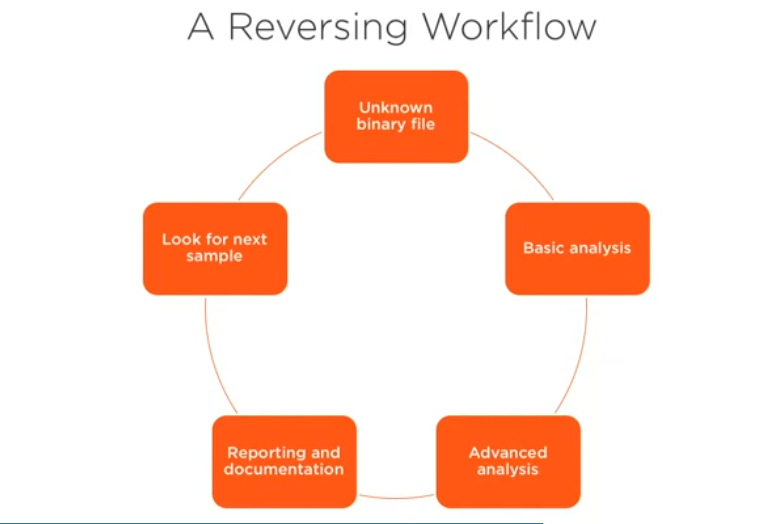
Reverse Engineering process

Native code: code that is compiled from languages such as C and C++

Instruction set architectures (ISA): abstract model of a computer defines what is needed to program it

Examples of ISA include Intel x86 and x86-64, ARM and AMD

Reversing workflow



* It’s started with unknown binary file. This could be malicious sample that was just captured on your network or an endpoint
* Basic analysis: to understand what is the file type is, what file format it used
* Advanced analysis: this includes analyzing those program instructions
* Reporting and documentation: if you found a bug the client needs to know what that bug is.

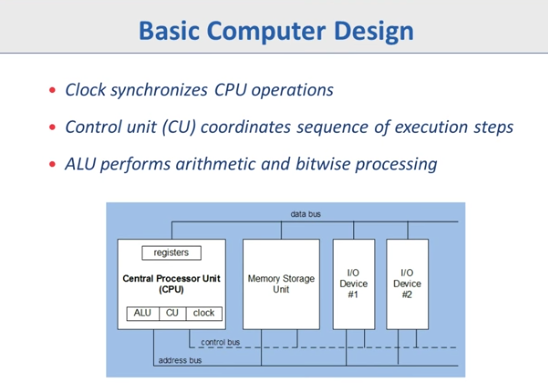
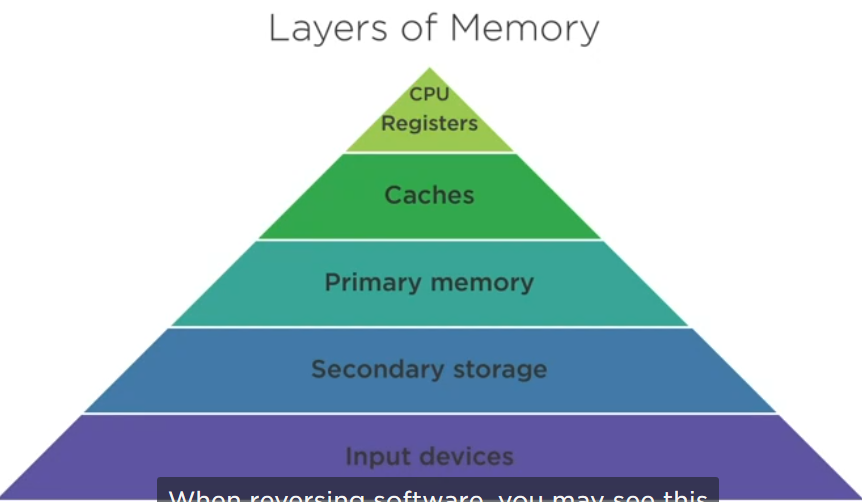
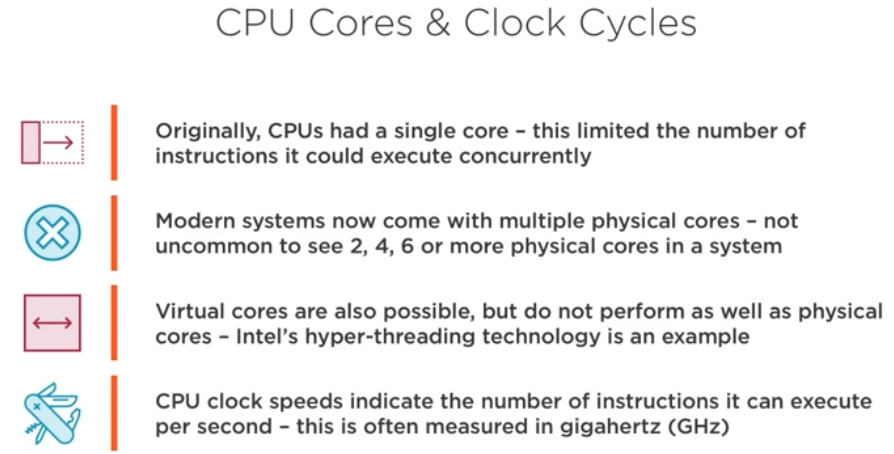
Code obfuscation: make the code more difficult to analyze

How does CPU work?

1. when you double click on the EXE file this is causes the operating system loader to take the executable from disk that store the exe file (secondary storage) and move it into primary memory or RAM.
2. Once it loaded the CPU can take over and fetch the first instruction out of memory
3. One instruction is fetched, it is then decoded and then executed
4. This process is repeat as long as the program is running

It is actually the CPU core that is responsible for that

CPU clock speed indicates the number of the instructions it can execute per second this often measured in gigahertz (GHz)



Registers: is a very very small and high speed storage unit

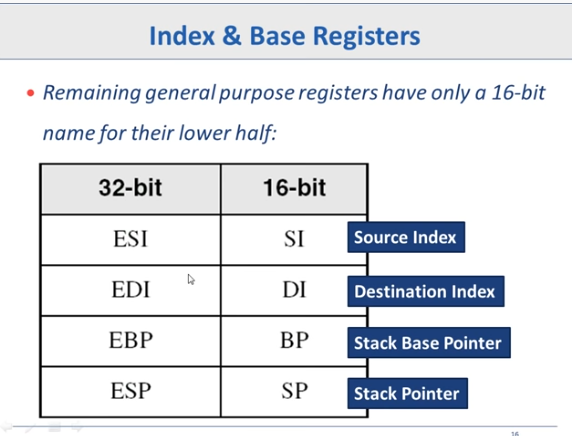
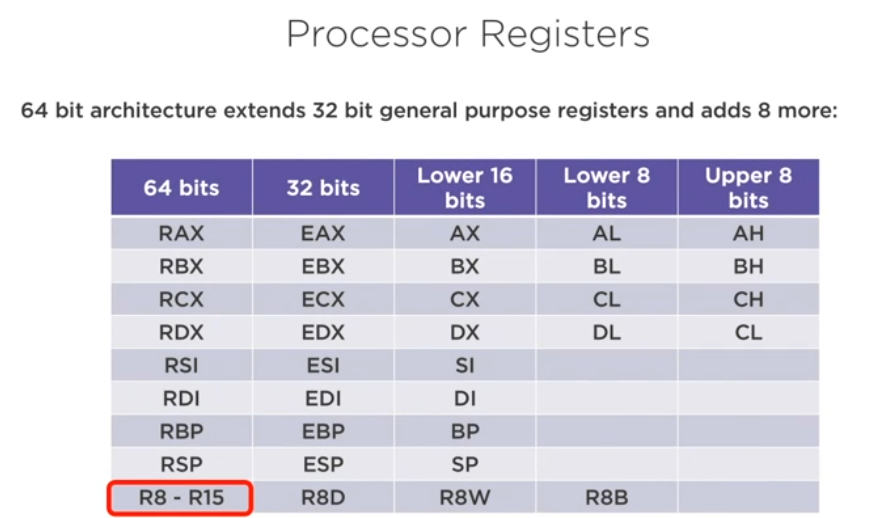
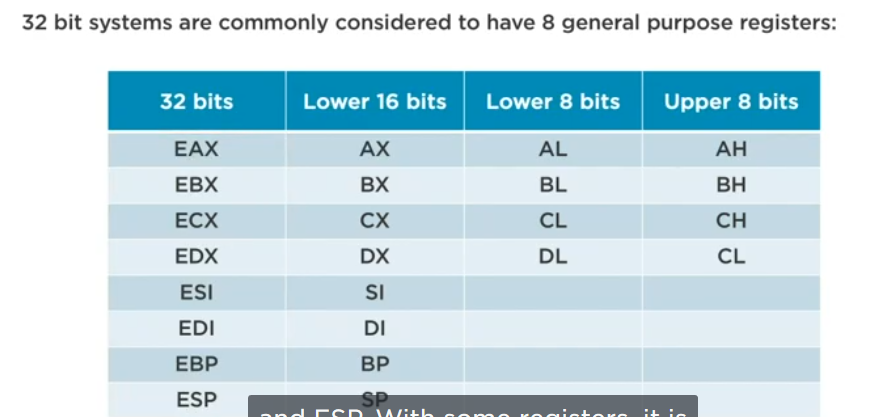
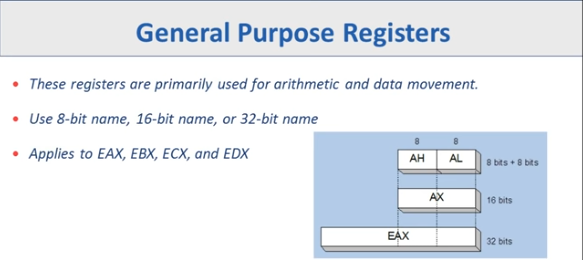
بعض البروسيسوز الريجستر بيبقي 8 بت وبعضها 16 بت وبعضها 32 بت

Registers can access by its name not address

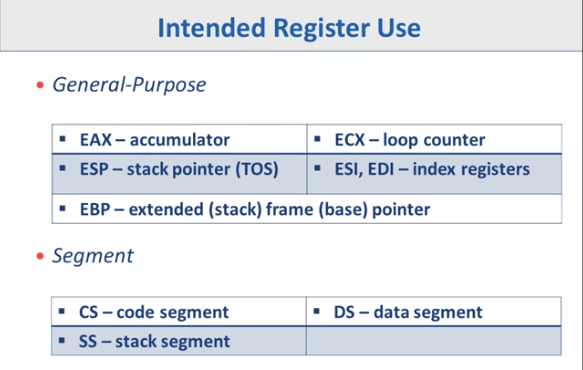
Registers with 32-bit named general purpose segment

Registers with 16-bit named segment registers

EAX for example E for extended that mean that register was 16 bit then become 32 bit



And these commonly used for copy operations



32-bit pointer registers:

EIP >> instruction pointer tells the CPU where to go in memory to fetch the next instruction

ESP >> stack pointer this is responsible for keeping track of the top of the stack in memory

EBP >> base pointer this is used in order to create the stack frame for a function that is the space in the stack that is used for arguments and local variables

64-bit pointer registers:

RIP: Instruction pointer

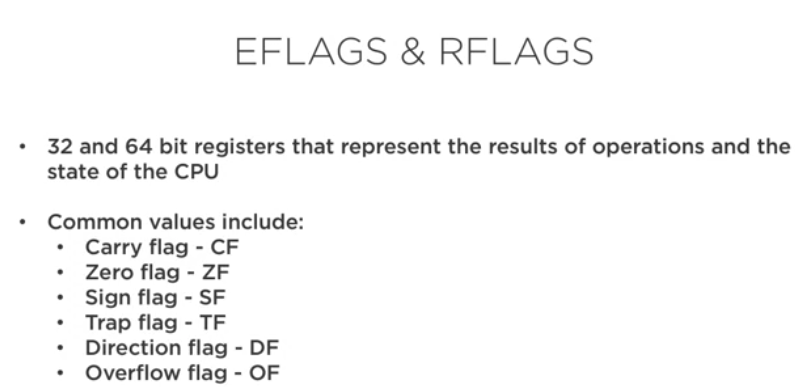
RSP: Stack pointer

RBP: Base pointer

The same mission but for 64-bit systems

EFLAGS & REFLAGS

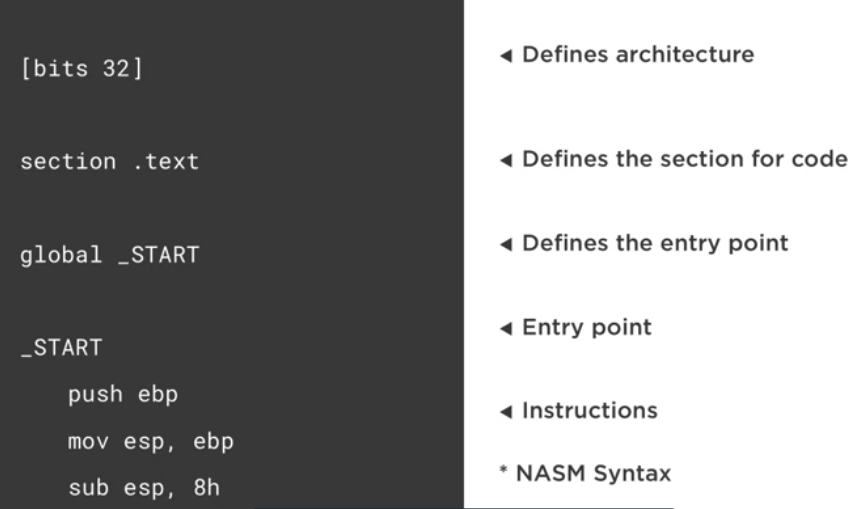
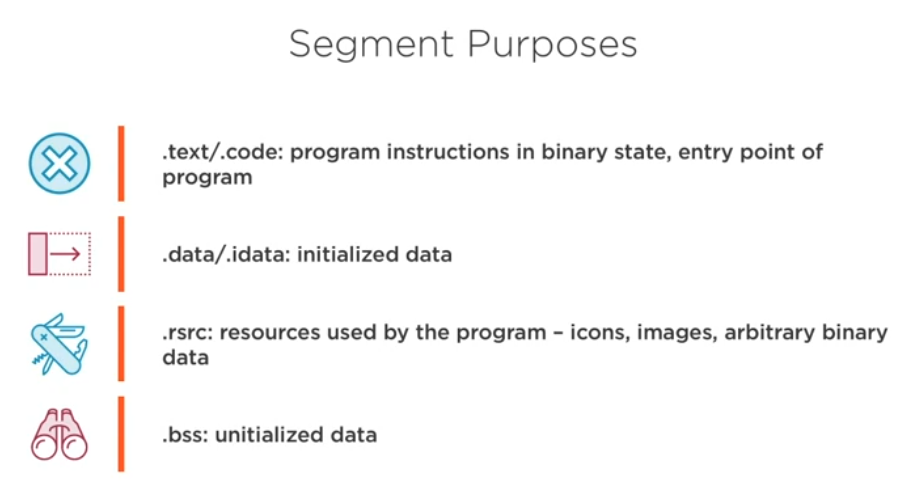
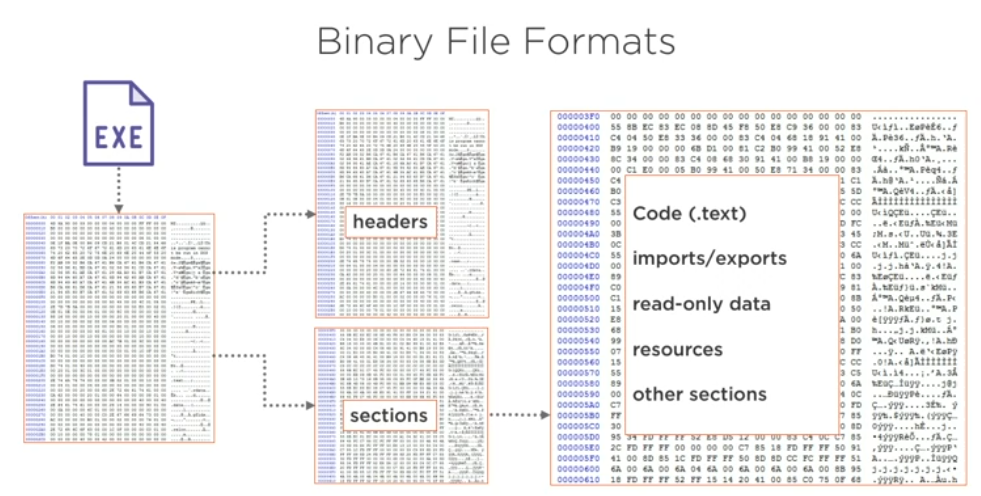
Represent the status of CPU or result for an arithmetic operation



Assembler: translate the instructions into machine language

Headers in EXE files give the operating system and reverse tool information about the binary file

Sections contain code, resources, imports, exports and other important data for that binary file



- A reverser: who take the machine code and translate it back to assembly language that can understand

- The output of Disassembler is assembly

- The output of Decompiler is original code (high level language)

- OPCODES are simply the binary representation of the instruction and any operands – often displayed as hexadecimal values



- As a reverser you want to take the machine code and by disassembler or decompiler you want to translate it to original instructions

**Essential instructions**

There are approximately 1503 instructions defined in intel’s ISA

We can organize it in few primary categorize:

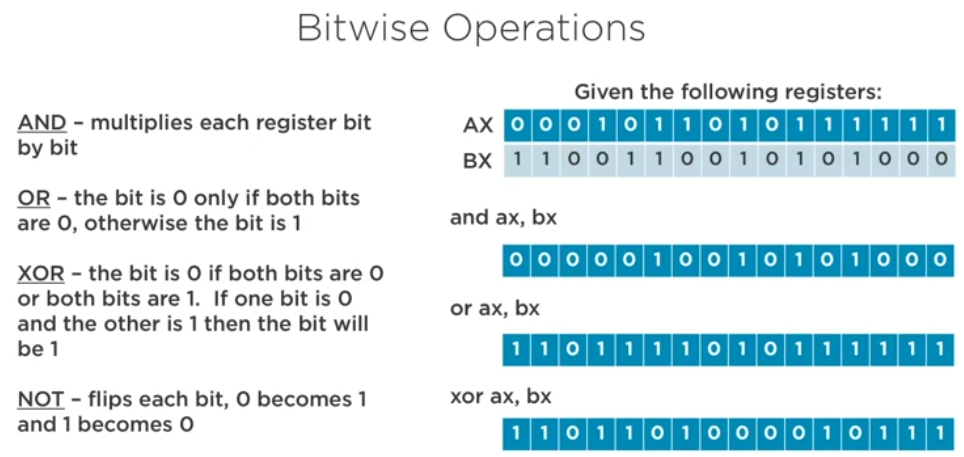
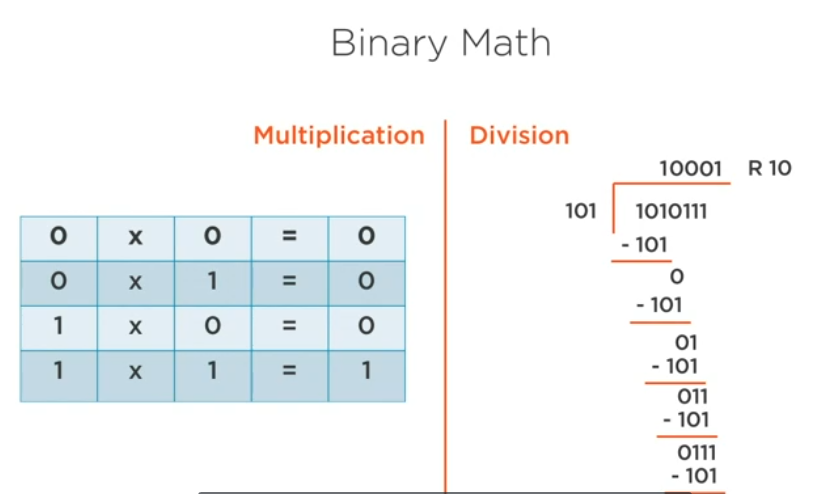
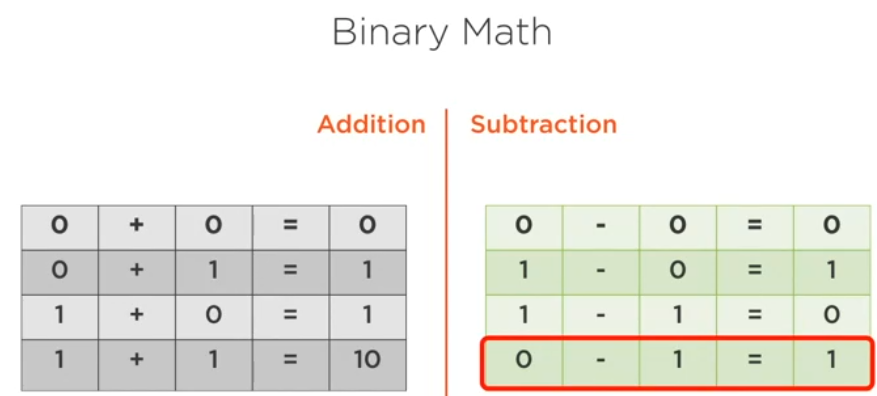
- arithmetic

- memory

- comparison

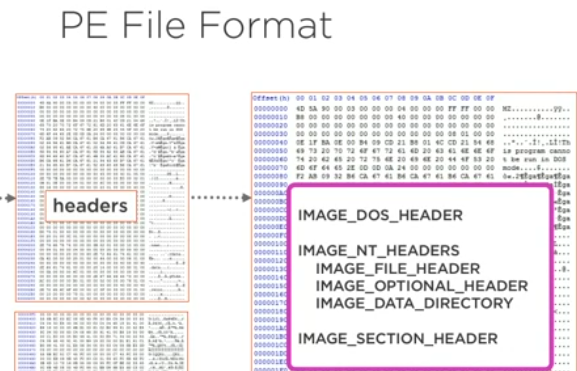
- control flow

- bit-manipulation



PE (Portable Executable File):

There is a flag or a bit that differentiate between PE and DLL because the two are the same binary format

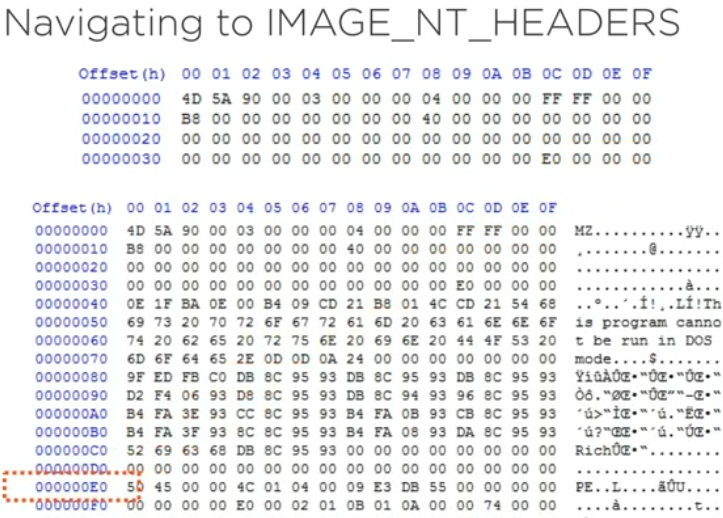


These structures define and provide the organization for the binary content within the file format

**IMAGE\_DOS\_HEADER**: is the first **64 bytes of every PE files**. This actually consists of 19 members, Two the most important are **magic and e\_lfanew**

**Magic**: contains values 4Dh,5Ah (mz)

**e\_lfanew:** is a DWORD and this referee to go to the next location (NT\_HEADERS)



The E0 taken us to the next structure NT\_HEADERS

This structure begins with the **signature** which is **4 bytes**

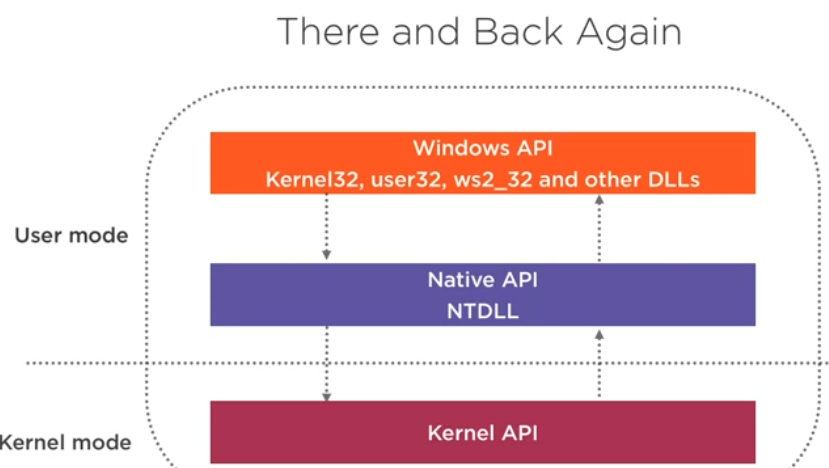
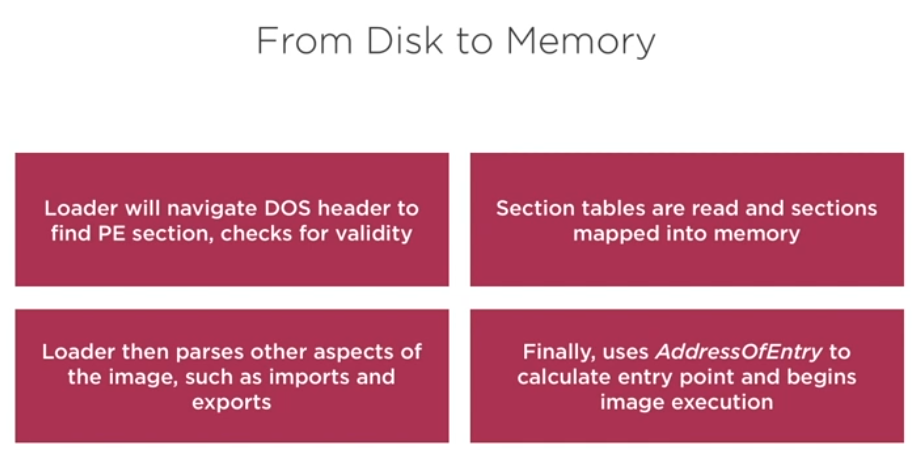
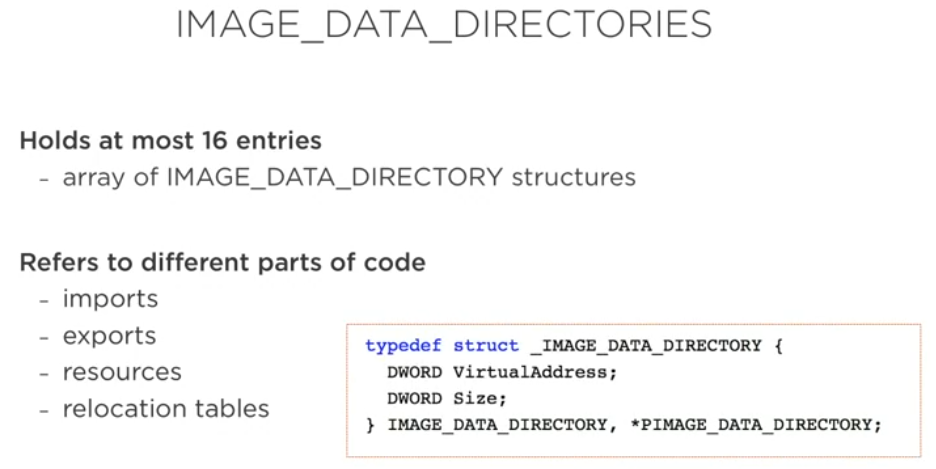
IMAGE\_NT\_HEADERS: contain information such as where to find the entry point of the program, the image base where we will find this code within virtual memory and whether it uses such protection like ASLR or DEP (Data execution prevention)  
contains three members

**Signature** which is DWORD

**FileHeader**

**OptionalHeader**

**IMAGE\_OPTIONAL\_HEADERS:** contains a lot of information like **AddressOfEntryPoint**, **ImageBase** that identifies where in the memory the program will loaded,**ImageCharacteristics** that defines whether it is EXE or DLL



Looking at PE file data:

- characteristics and properties

- code and data

- Imports and exports