0111011101100101
0110110001101100
0100000001100100
0110111101101110
0110010101000001

Malware Analysis
(part A)

Davide Balzarotti
davide.balzarotti@eurecom.fr

Malware Analysis

$\approx$

Adversarial Binary Analysis

So, what is a Binary?

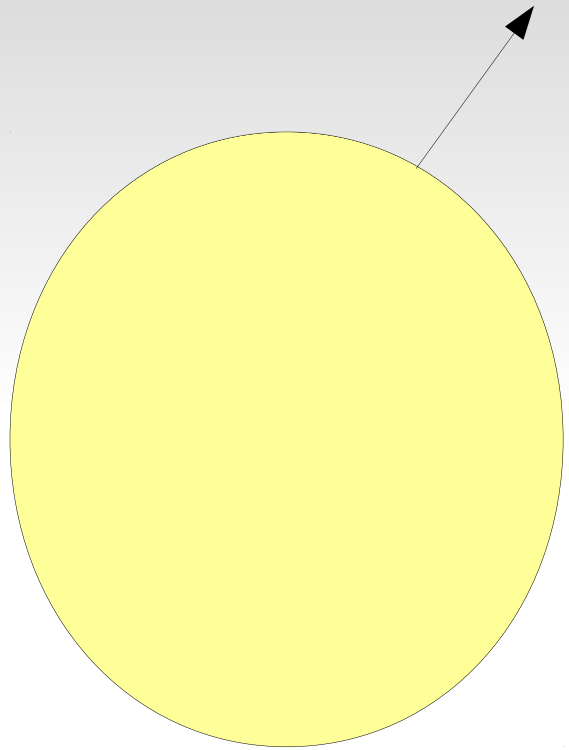
- The term binary can refer to
  - Program binary – executable format that contains instructions that can be loaded and executed by a computer
  - Data binary – non-text file containing generic data (word documents, pictures, videos, ...)

- Program binaries are typically produced by a compiler (e.g., GCC or Microsoft Visual Studio)

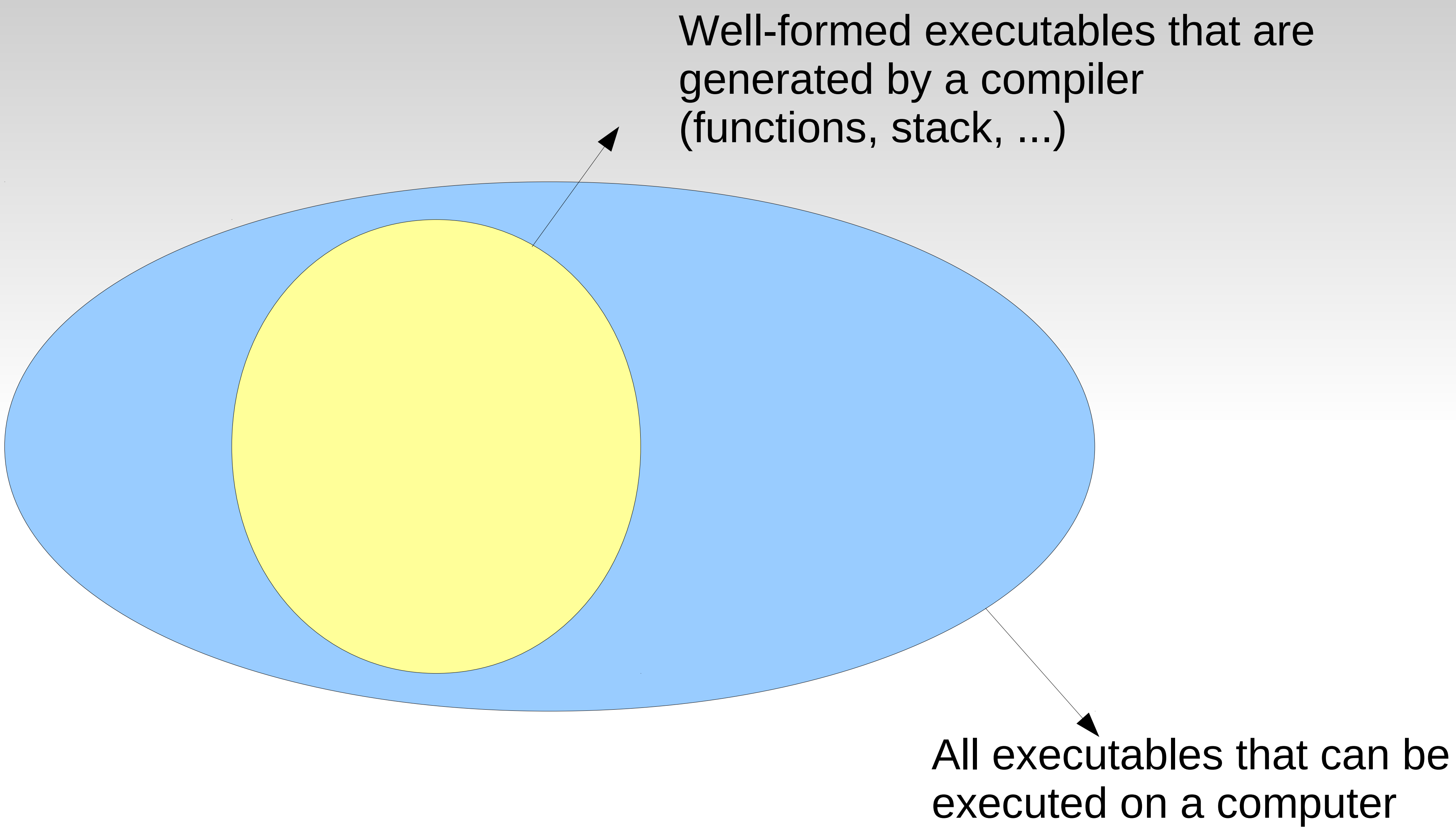
  … but this is not always the case

The World of Binaries

Well-formed executables that are generated by a compiler (functions, stack, ...)



The World of Binaries



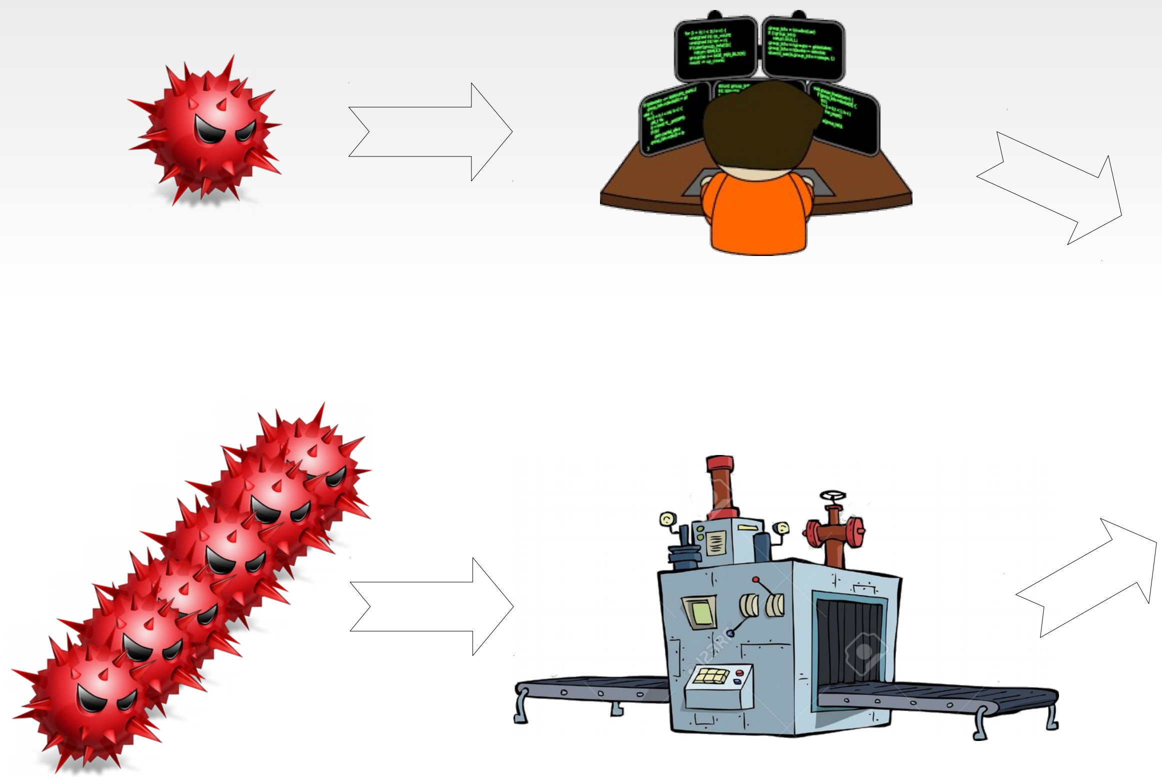
Binary Code

- It comes in different forms:
  - executable files
  - libraries
  - firmware images
  - process core dumps
  - system-wide memory dumps
  - ...

Malware Analysis

- *Adversarial:* malicious binaries are
  - Typically stripped of all the symbols
  - Often obfuscated and packed
  - Full of anti-debugging and anti-analysis tricks, suicide bombs, and checks for analysis environments
- The goal is to understand...
  - ...what the malware does
  - ...how it does it
  - ...which ones are its conditions (triggers)
  - ...if it is a modified version of another malware

Malware Analysis



Behavior

Triggers

Malware or Goodware?

Type and Family

Binary Analysis Techniques

Static Analysis

- Examine the binary without running it
- The only option when the program cannot be run
  (partial memory dump, missing pieces, unavailable architecture,...)
- It can tell you everything the program can do

Binary Analysis Techniques

Static Analysis

- Examine the binary without running it
- The only option when the program cannot be run
  (partial memory dump, missing pieces, unavailable architecture,...)
- It can tell you everything the program can do

Dynamic Analysis

- Run the program and observe its behavior
- It tells you exactly what the program does when it is executed in a *given environment* and with a *particular input*
- Can be language agnostic

Precision vs Coverage

- Dynamic analysis techniques...
  - Are more precise: they can observe the instructions executed and the values of registers and memory
  - Achieve a smaller coverage: they observe one execution path at the time
- Static analysis techniques...
  - Are less precise: need to reason about the program behavior without actually executing the code
  - Achieve a larger coverage: can reason about all possible executions at the same time

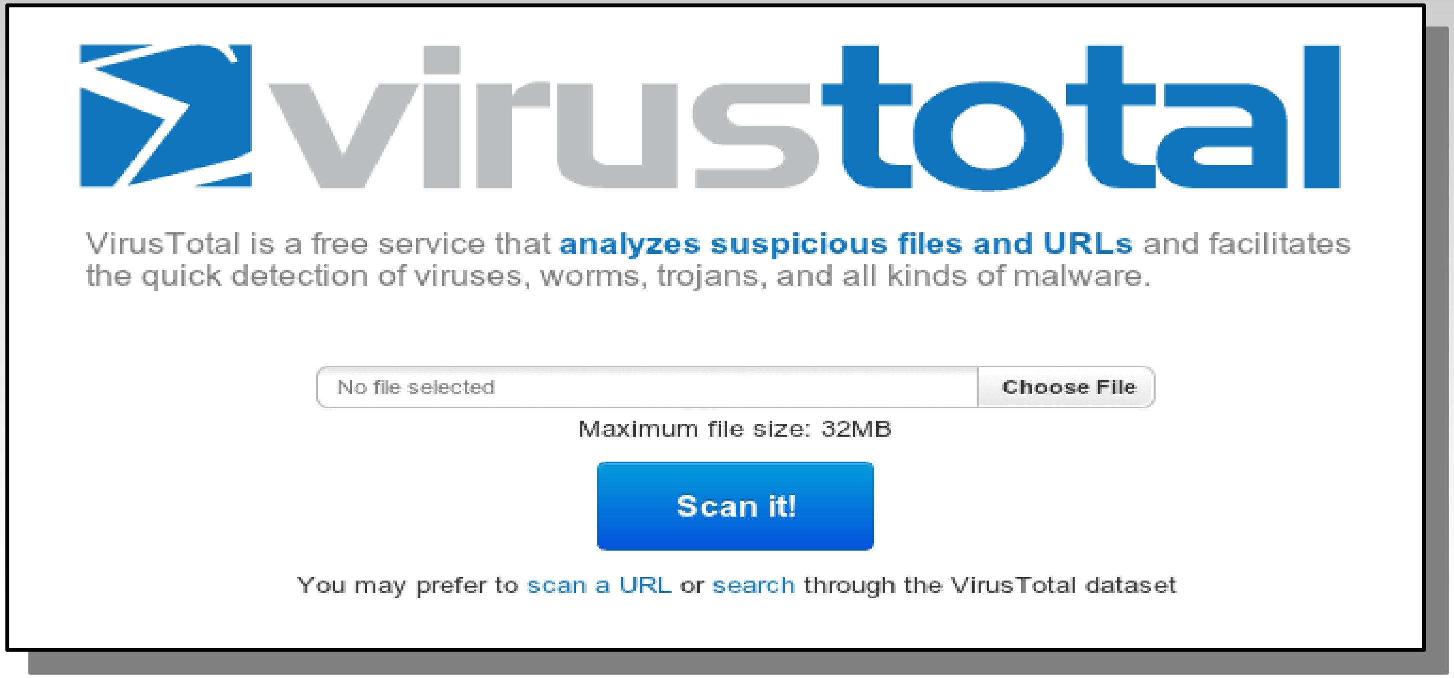
Summary

- Black Box File Analysis
- Static Analysis
- Dynamic Analysis
- Automation & Scalability

Black Box Analysis

- Is it a known binary?
  - → Check file hash
- Is it similar to something we already know?
  - → Signatures
- Hints on what the malware does
  - → Embedded strings
  - → Imported libraries
  - → File headers and symbols

Checking the File Hash



- Submit files -or- check for an hash (MD5 or SHA256)
- Report the result of ~54 antivirus systems
- It knows more than 1B files

Antivirus	Result	Update
AhnLab-V3	Win-Trojan/Pakes.191398.B	20120410
AntiVir	BDS/Bifrose.Gen	20120410
Antiy-AVL	Trojan/Win32.Win32.gen	20120410
Avast	Win32:Agent-ABW [Trj]	20120410
AVG	BackDoor.Generic2.HOC	20120411
BitDefender	Backdoor.Blackhole.2005.K	20120411
ByteHero	-	20120407
CAT-QuickHeal	Backdoor.BlackHole.orj	20120410
ClamAV	PUA.Packed.ASPack	20120411
Commtouch	W32/D_Downloader!GSA	20120411
Comodo	TrojWare.Win32.Trojan.Agent.Gen	20120410
DrWeb	Trojan.PWS.Kpo	20120411
Emsisoft	Trojan.Win32.Pakes!IK	20120411
eSafe	Win32.BDSBifrose	20120408
eTrust-Vet	-	20120410
F-Prot	W32/D_Downloader!GSA	20120410

Antivirus	Result	Update
AhnLab-V3	Win-Trojan/Pakes.191398.B	20120410
AntiVir	BDS/Bifrose.Gen	20120410
Antiy-AVL	Trojan/Win32.Win32.gen	20120410
Avast	Win32:Agent-ABW [Trj]	20120410
AVG	BackDoor.Generic2.HOC	20120411
BitDefender	Backdoor.Blackhole.2005.K	20120411
ByteHero	-	20120407
CAT-QuickHeal	Backdoor.BlackHole.orj	20120410
ClamAV		20120411
Commtouch		20120411
Comodo		20120410
DrWeb		20120411
Emsisoft	Trojan.Win32.Pakes!IK	20120411
eSafe	Win32.BDSBifrose	20120408
eTrust-Vet	-	20120410
F-Prot	W32/D_Downloader!GSA	20120410

You can use AVClass to parse a VT report and output the most likely family name of a sample

Write Your Own Signatures

- Yara is a language to describe byte-level patterns, and a tool to match the patterns against arbitrary files
  - Kind of a custom `grep`
- Each rule is a text-based description composed by
  - some meta information
  - a list of strings that define patterns
  - a condition that defines how the string must appear in the target file
- Runs on Linux, Windows, and Mac OS X, either from the command line or through python bindings
  - Multi-threaded support for parallel scans

/* YARA */

Yara

- Matching one or more rules against a file:

```
$ yara rule-file <target-file>

$ yara rule-file <PID>
```

- From Python:

```
import yara

rules   = yara.compile("/home/foo/yrules")
matches = rules.match("/home/foo/malware")
matches = rules.match(some_string)
```

Yara Rules

```
rule foobar : banker-tag
{
  meta:
    description = “Banker Foobar”

  strings:
    $a = "__SYSTEM__" nocase wide ascii
    $b = {78 [40-48] 45 ?? 5F A? }
    $c = /md5: [0-9a-zA-Z]{32}/
    uint32(uint32(0x3C)) == 0x00004550

  condition:
   filesize > 200k
   (#a>2 and $b) or ($c in (100.. filesize))
}
```

* for more info check the Yara online user manual

- Is it a known binary?
  - Check file hash → **MD5 on VirusTotal**
- Is it similar to something we already know?
  - Signatures → **Submit to VirusTotal**
    Use Yara rules
- Hints on what the malware does
  - Embedded strings
  - Imported libraries
  - File headers and symbols

strings

```
> strings -a -t d malware
    ...
   936 connect
   944 fork
    ...
   2709 [^_]
   2800 193.253.230.214
   2818 /bin/ls
   2992 http://weird-domain.com
   3168 -cra-n_qqub
   1669 GCC: (GNU) 4.2.4 (Ubuntu 4.2.4-1ubuntu1)
   ...
```

**Remember to check for Unicode strings (especially in Windows binary) using the* `-e` *option*

Libraries

- Static Linking
  - Static libraries (`.a` in Linux, `.lib` in Windows) are copied into the executable at compile time
  - If symbols are missing, it is hard to distinguish the library code from the program code → analysis much more complex
  - Rare in Windows, quite common in Linux

Libraries

- [Implicit] Dynamic Linking
  - Dynamic libraries (`.so` in Linux, `.dll` in Windows) are loaded when the program starts
  - Check with `dependency-walker` (for Windows), `ldd` or `lddtree` (for Linux)

Libraries

- [Explicit] Dynamic Linking (aka Runtime Linking)
  - To load an additional library (not listed in the binary header) at runtime
  - Quite common in Windows (`LoadLibrary` / `GetProcAddress`)
  - Not so common in Linux, but sometimes used to load plugins (`dlopen` / `dlsym`)

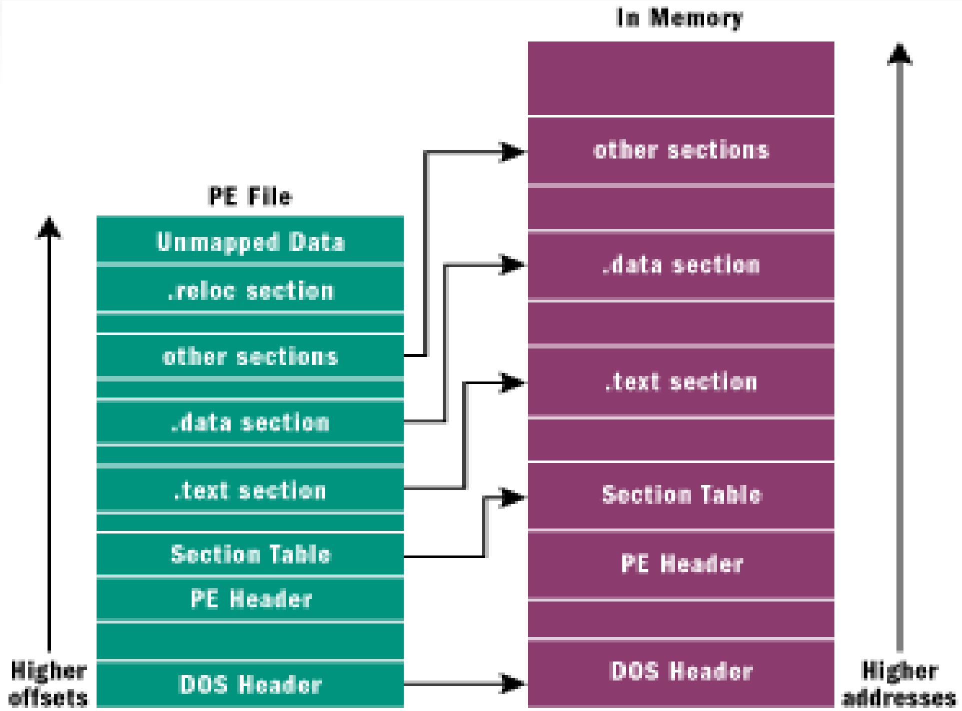
Binary File Formats

- Define what the file looks like on disk and how it should be loaded in memory
- Portable Executable (PE)
  - Used to represent executables, object code, and DLLs in 32-bit and 64-bit versions of Windows operating systems
- Executable and Linkable Format (ELF)
  - Used to represent executables, object code, shared libraries, and core dumps
  - Adopted by many unix-like operating systems (Linux, Solaris, BSDs, …), game consoles (Sony Playstation, Wii, Dreamcast, …) mobile phones (Android, Symbian, Sony, ...)

Portable Executable (PE)

- File format introduced by Microsoft as part of the original Win32 specifications
  - Derived from the earlier Unix Common Object File Format (COFF) found on VAX/VMS
  - Contains executable and library code (the only difference is one bit)

- Data structure on disk are the same that are mapped in memory (even though the offset may differ)



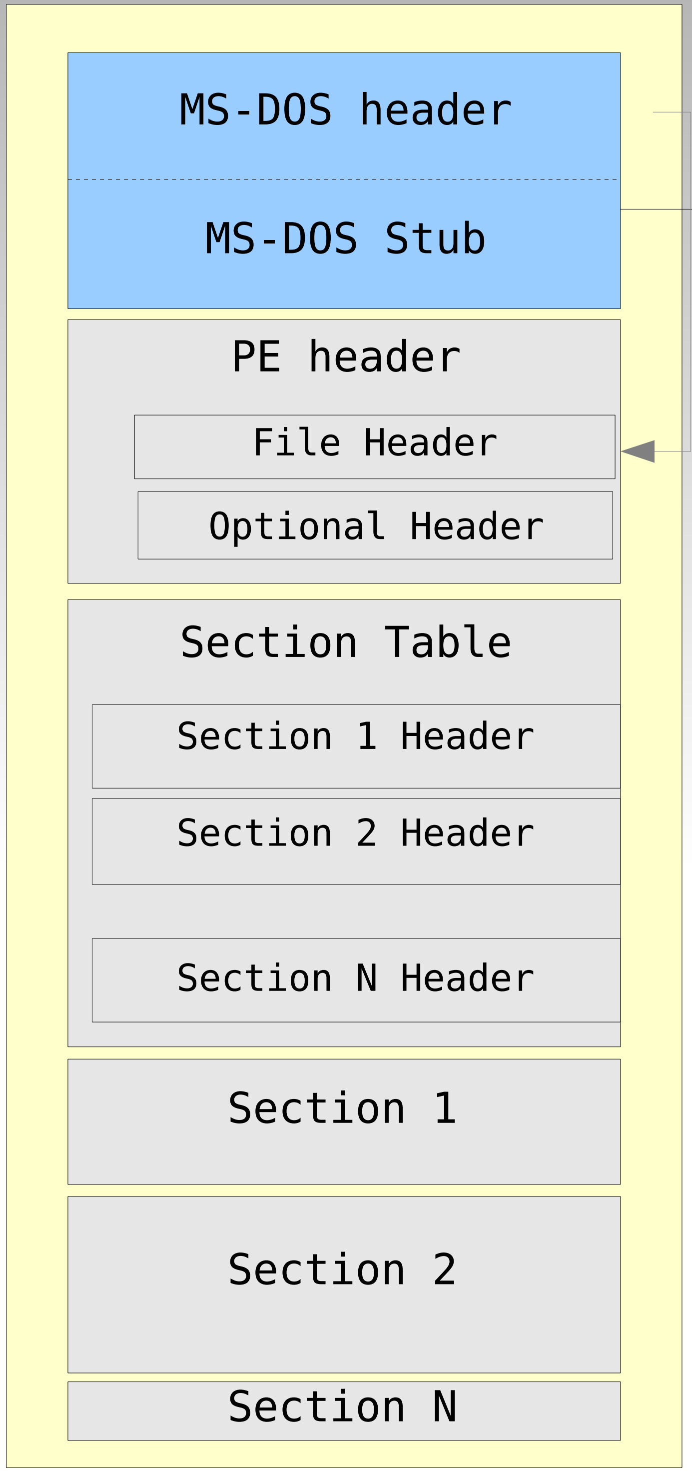
Sections

- A file contains multiple sections, each representing either code or data
  - Global variables
  - Import/export tables
  - Resources
  - Relocation info
- Each section has its own set of attributes
  - **R**ead / **W**rite / **E**xecute
  - Shared between all the processes running the executable

Compilers have a standard set of sections that they generate, but programmers are free to create and name their own sections

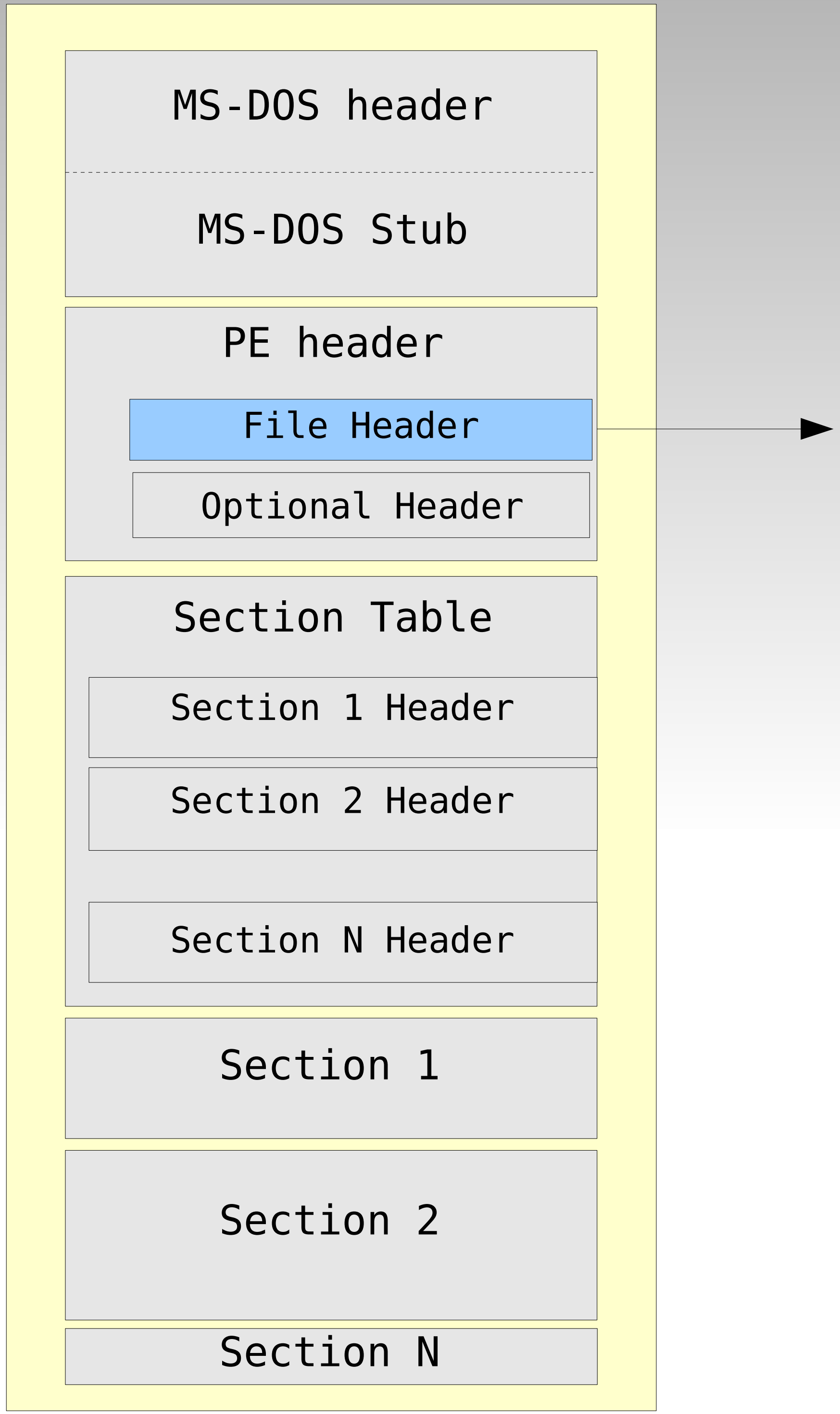
Locating Info inside a PE

- Memory addresses are expressed using Relative Virtual Addresses (RVA)
  - VAs (virtual addresses) are obtained by adding RVAs the base address of the executable in memory
- A Data Directory array is used to locate other artifacts and data structures inside the PE
  - Import and export tables, security certificates, resources, exception handler table, debug information, ...
- Each imported DLL has a structure in the PE, containing the name of the DLL and an array of function pointers known as the Import Address Table (IAT)
  - All external functions calls in the program go through the IAT



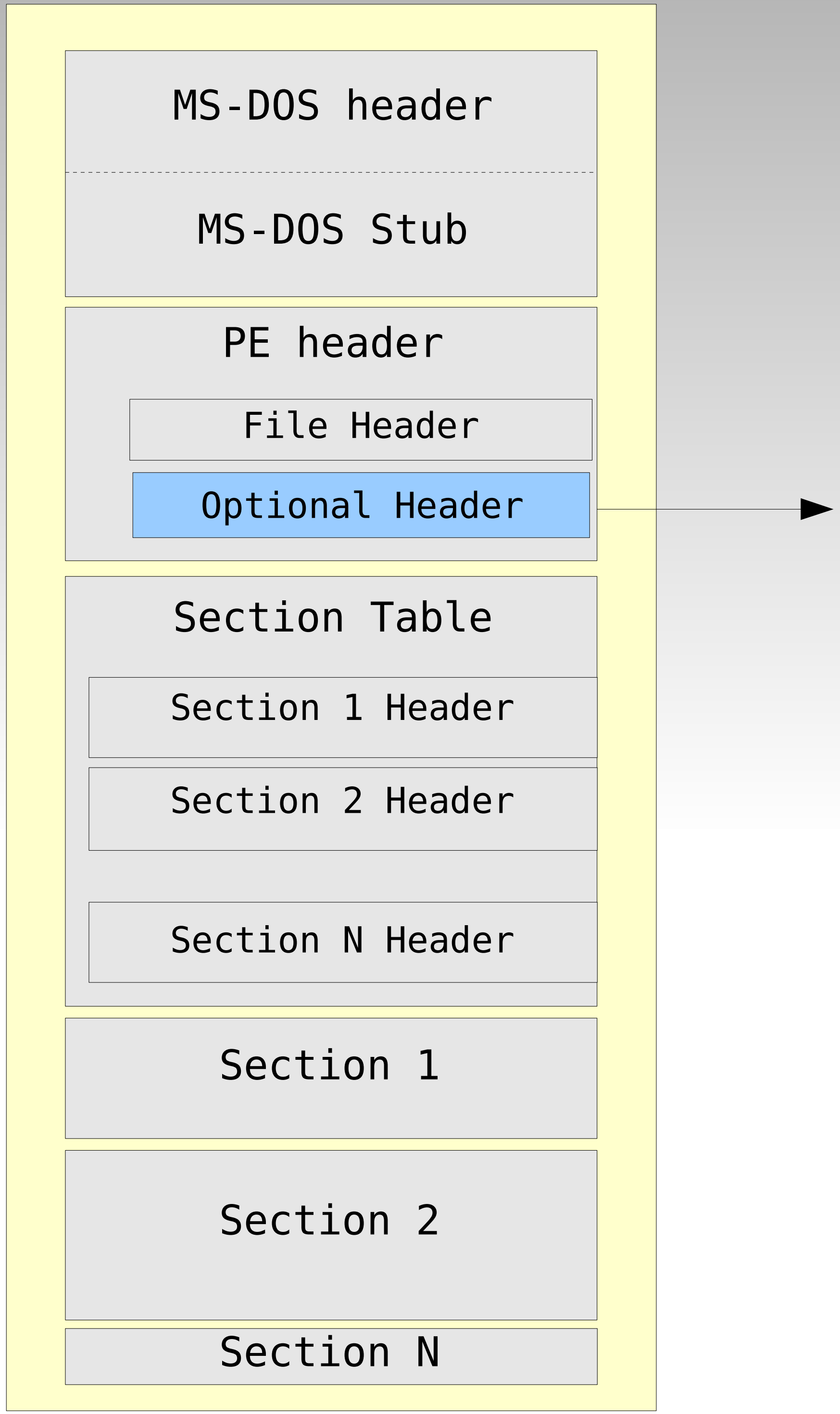
PE file begins with a small MS-DOS Executable

- When executed on a machine without Windows, the program prints a message saying that Windows is required to run the program

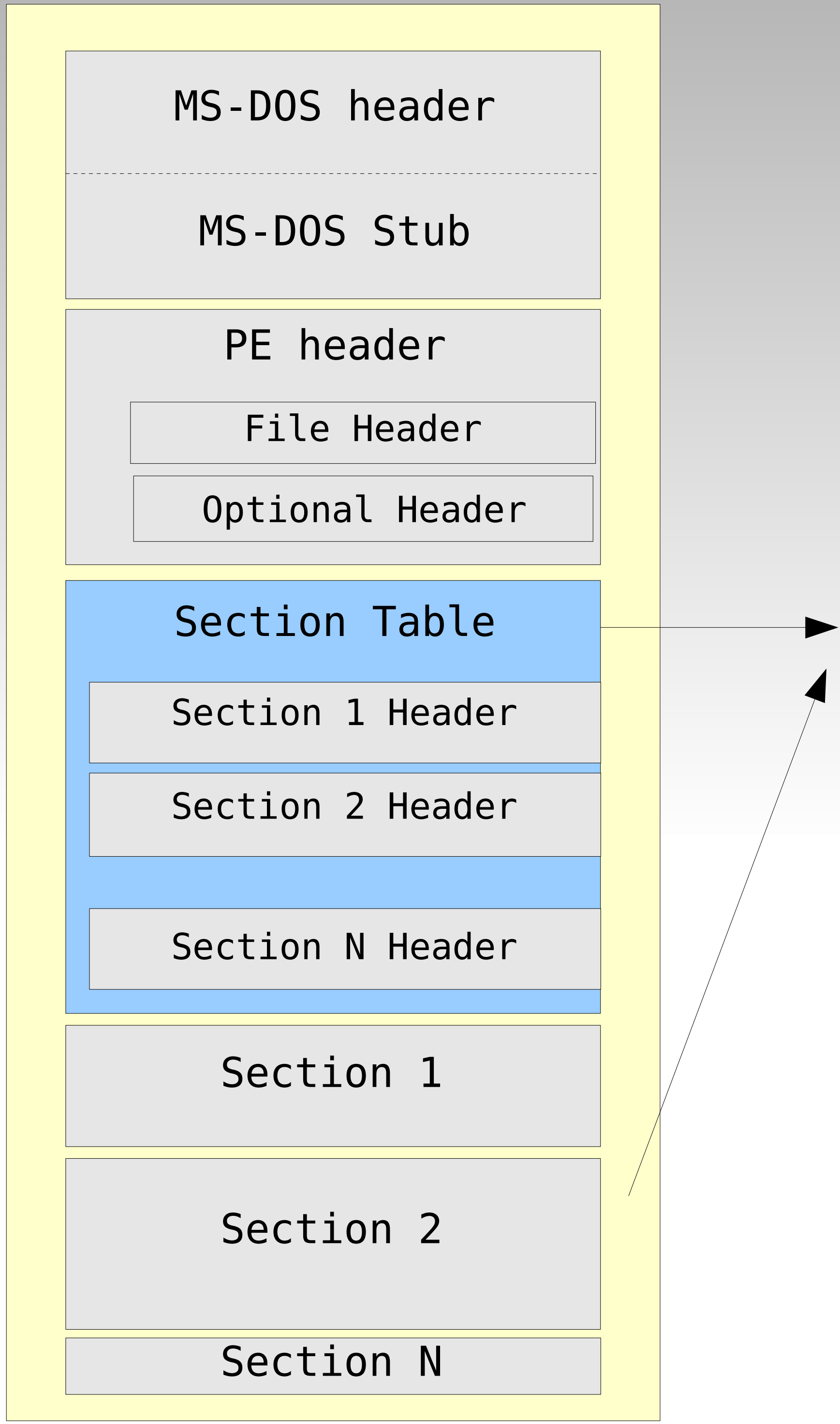


Basic information about the file
(same as the old COFF format)

- 32/64 bits
- Number of Sections
- Creation time
- EXE or Library Flag
- ...

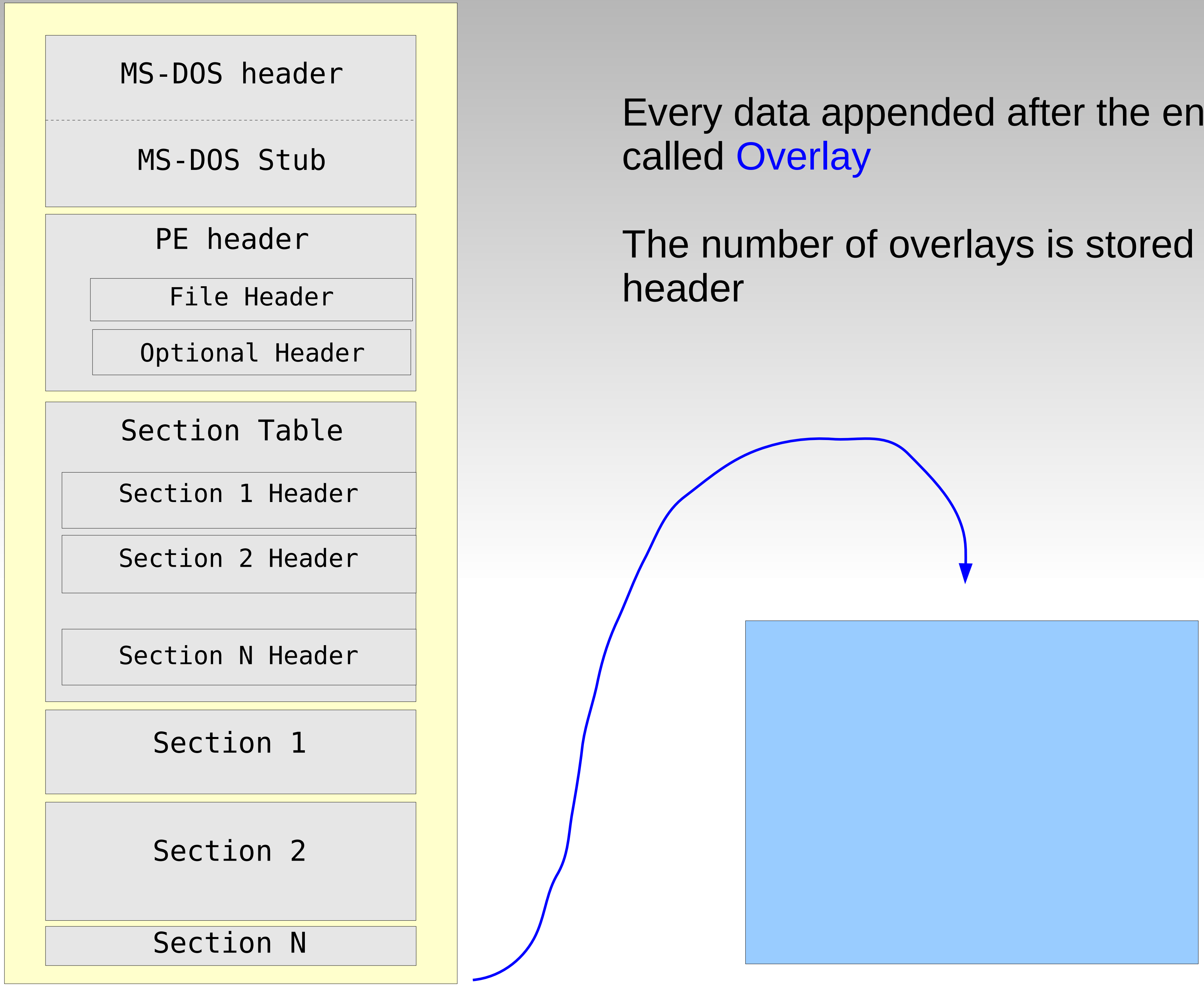


- Linker version
- Size of Code
- Entry point address
- Base of code (RVA)
- Size of headers
- Section Alignment
- Subsystem
  (console, GUI, device driver, OS2, Posix)
- Data directory



Common Sections

- `.text` The default code section
- `.data` The default read/write data section. Global variables typically go here
- `.rdata` The default read-only data section. String literals and C++ vtables are examples of items put here
- `.idata` The imports table
- `.rsrc` The resources (icons, dialogs, ...)
- `.crt` Data used by the C++ runtime. Function pointers that are used to call the constructors and destructors of static C++ objects
- `.pdata` The exception table
- `.reloc` Relocation data for the loader



Every data appended after the end of the PE file is called Overlay

The number of overlays is stored in the DOS header

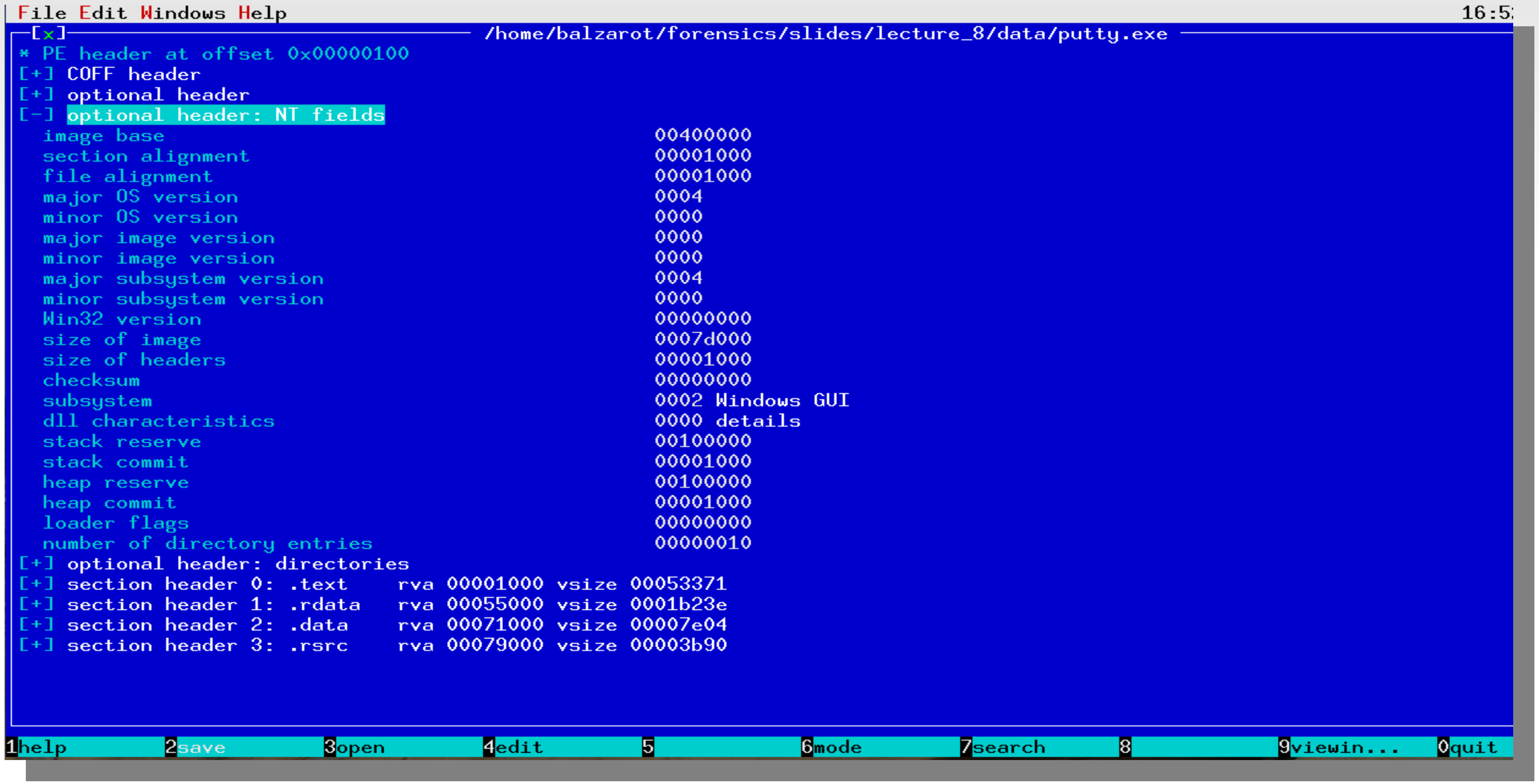
Loading Libraries

- The `.idata` section contains the import directory table, which has one entry for each imported library
- Imported functions can be listed by name or by ordinal
  - The ordinal represents the position of the function in the DLL Export Address table
  - If listed by name, the linker does a binary search of the Export Name Pointers table of the corresponding DLL to lookup the symbol
- After the linker (part of the Windows kernel) locates the function address, it stores it inside the IAT
  - Functions call in the program code use the IAT as intermediate table containing the addresses of the functions

More info: http://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-two

HT

- Open source editor/viewer/analyzer for executables
  - Works in Linux (`apt-get install ht`) and Windows
  - Support for PE32, PE64, ELF, DOS, Java



Playing with PE Files

```python
import pefile
sample = pefile.PE('program.exe')
sample.show_warnings()
```

- Very complete and flexible python library to analyze PE files
- Also supports modification of each field
  (e.g., you can change the entrypoint and save the file)

```python
sample.OPTIONAL_HEADER.AddressOfEntryPoint
sample.OPTIONAL_HEADER.ImageBase
print sample.FILE_HEADER
sample.print_info()# print all the PE headers
                     in a human-readable form
```

Playing with PE Files

List of imported symbols

```
for entry in sample.DIRECTORY_ENTRY_IMPORT:
  print entry.dll
  for imp in entry.imports:
    print '\t', hex(imp.address), imp.name
```

List of sections

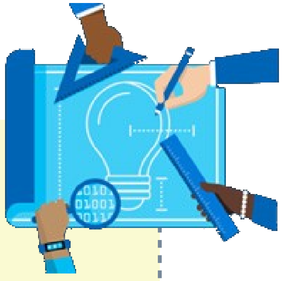
```
for section in sample.sections:
  print (section.Name,
         hex(section.VirtualAddress),
         section.Misc_VirtualSize,
         section.SizeOfRawData,
         section.get_entropy(),
         section.get_hash_md5())
```

Anomalous PEs

- Suspicious PE attributes or characteristics can be used to flag potentially malicious binaries
  - Suspicious API functions in the IAT
  - Weird entry points (not in the `.code` or `.text` segments)
  - Sections with extremely high entropy (sign of packers)
  - Invalid timestamps
  - Suspicious overlays
  - …
- `pescanner.py` is a little tool based on pefile that performs this kind of check and prints the suspicious entries

PE Reconnaissance

- PEStudio is a tool (for Windows) that is gaining popularity to survey PE files.
- It contains a number of byte signatures and rules to detect anomalous PE values
  - All data is stored in plain JSON files



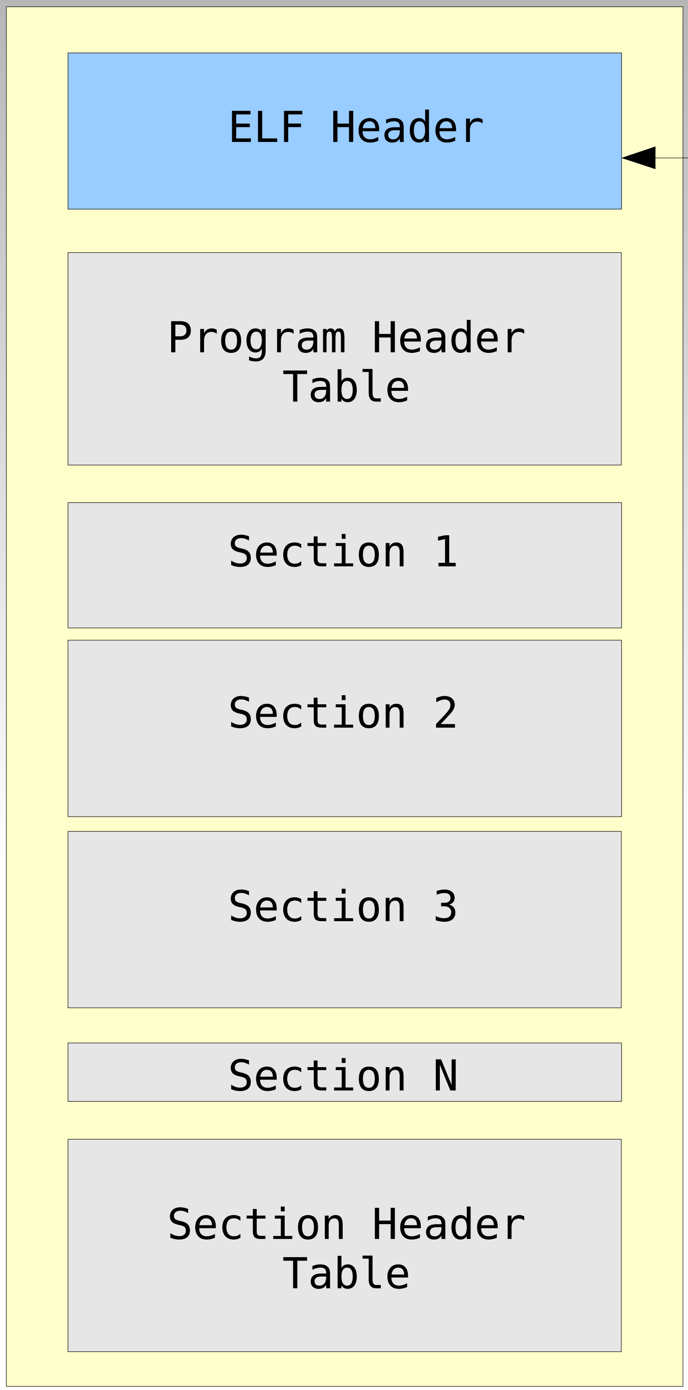
Implement a python-based command-line tool to match PEStudio rules

Import Hash

- Introduced by Mandiant to track related malware samples
  - An `impHash` is an hash of the library/API names and their specific order within a PE executable
  - Apparently robust against small changes / variations in the code
  - Ask Google for the Python code to compute the hash
- Already integrated in several tools and services (including VirusTotal)

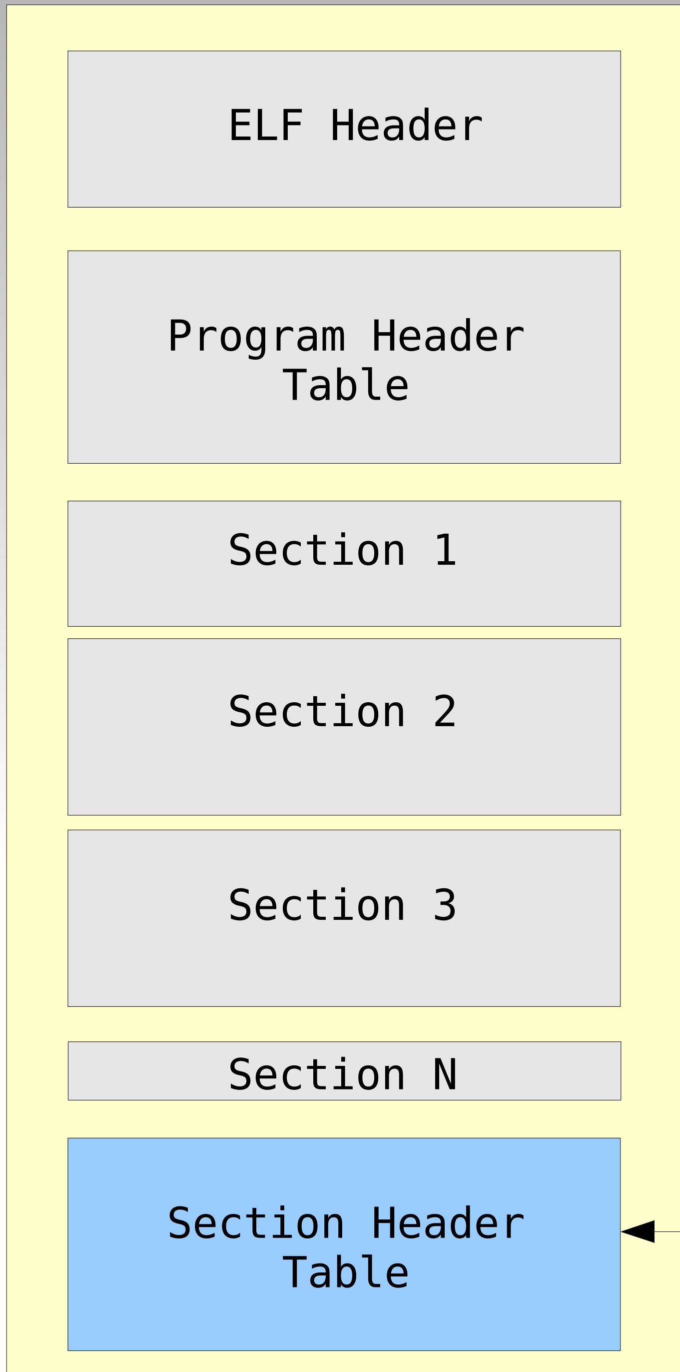
Executable and Linking Format (ELF)

- Introduced in UNIX SVR4 in 1989
  - Now adopted by Linux, Solaris, FreeBSD, HP-UX, …
  - Designed to be flexible and extensible, and not bound to any particular processor or architecture
  - Can store executables, relocatable objects, shared libraries, and core dump files
- Contains three headers (`ELF`, `section`, and `program`)
- Dual nature
  - Compilers and linkers treat the file as a number of logical sections described by the section header
  - System loaders treat the file as a number of segments described by the program header (each segment normally contains several sections)



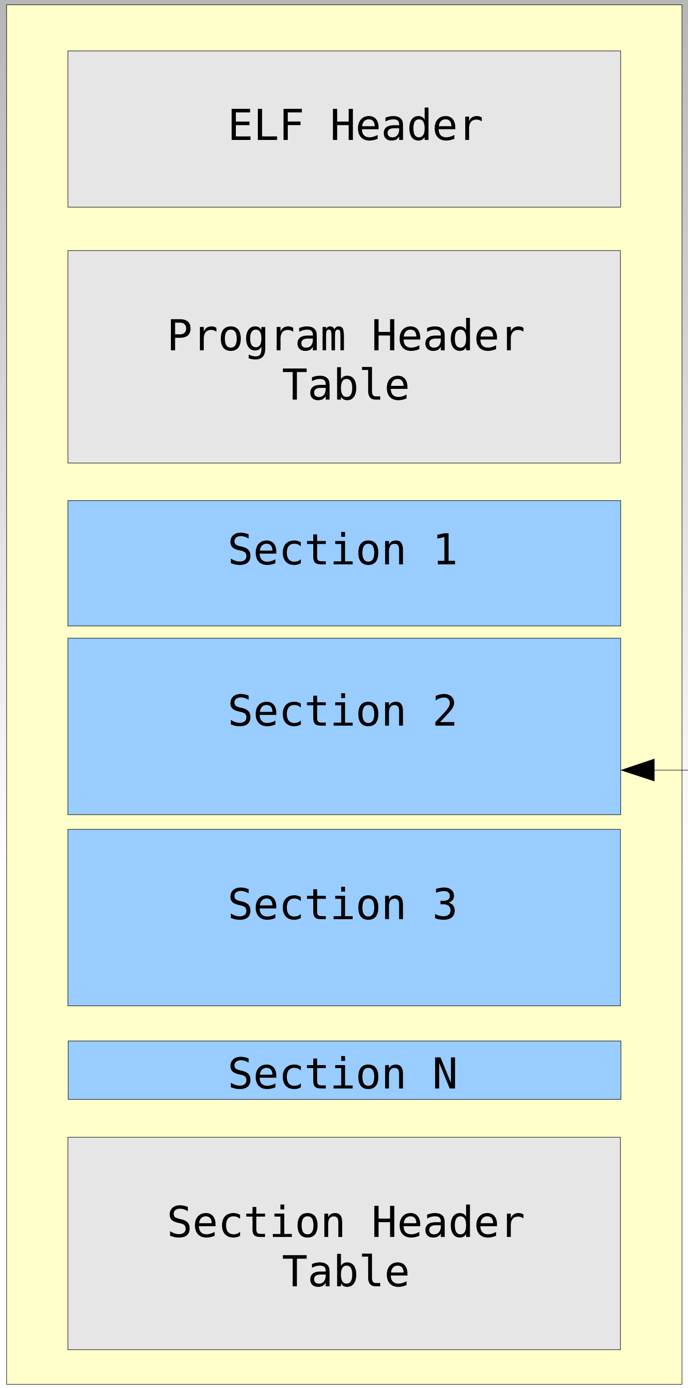
Always located at the beginning of the file

- Magic Number (75 45 4C 46)
- File type (executable, library, ...)
- Machine architecture
- Code entry point
- Program header offset
- Number of program headers
- Section header offset
- Number of section headers



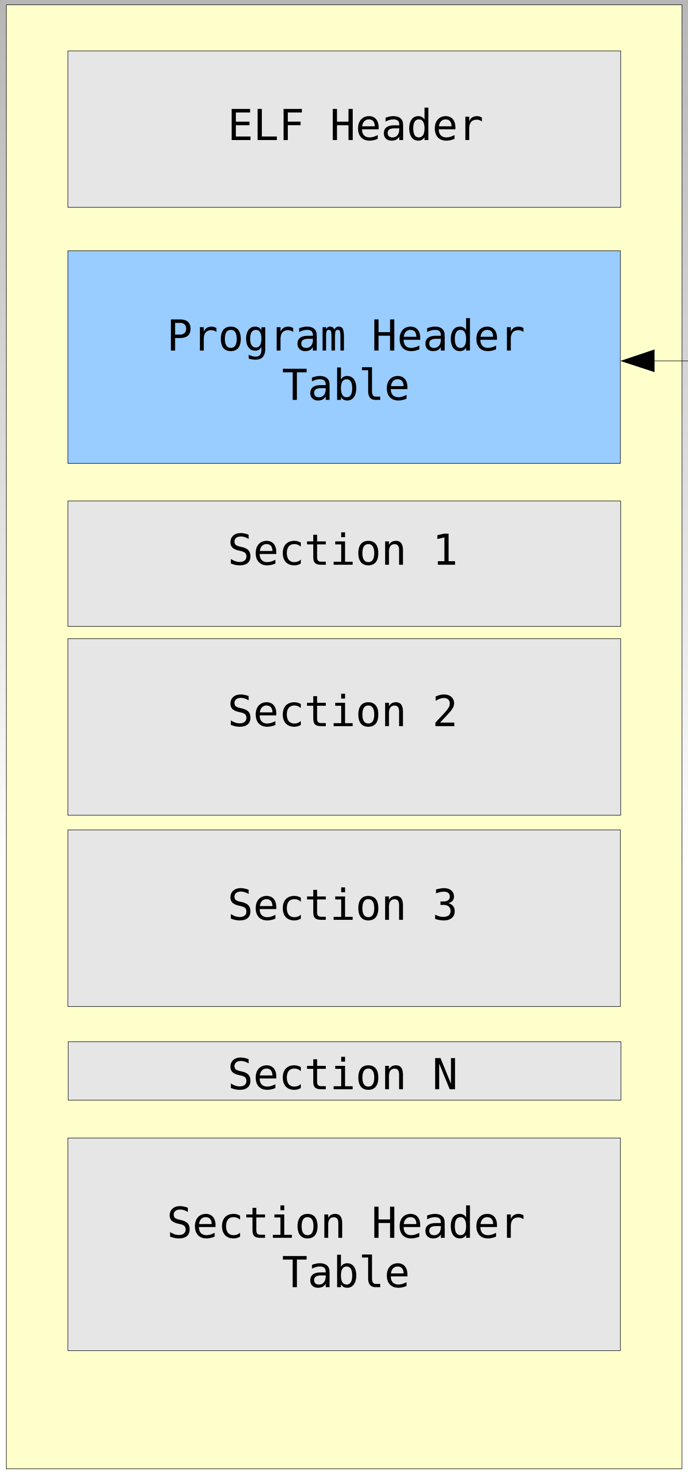
Array of structures
One entry for each section in the file:

```
sh_name      # index in the section header
               string table
sh_type      # what kind of info are stored
               in the section
sh_flags     # write, alloc, exec
sh_addr      # base address in memory
sh_offset    # location in the ELF file
sh_size
sh_link
sh_info
sh_addralign
sh_entsize
```



ELF sections:

```
.bss          (uninitialized data)
.data         (initialized data)
.rodata       (read-only data)
.text         (code)
.symtab       (symbol table)
.got .plt     (used by dyn linker)
.ctors        (constructors)
.dtors        (destructors)
...
```



Array of structures:

```
p_type    # Type of the segment
p_offset  # Position in the ELF file
p_vaddr   # Address in memory
p_paddr
p_filesz  # Size on disk
p_memsz   # Size in memory
p_flags   # Read / Write / Execute
p_align   # Alignment in memory
```

Common `p_type` values:

```
PHDR    # used to load the program table
          itself in memory
LOAD    # Loadable segment
DYNAMIC # Dynamic linking information
INTERP  # Path name to invoke as an
          Interpreter
```
(normally points to the dynamic linker)

More on Section Headers

- Since the loader and dynamic linker only reason in terms of segments, section information is not required at runtime
  - All the info required by the linker at runtime are in the PT_DYNAMIC segment
  - However, using the section header is simpler.. and so most of the tools (gdb, objdump, readelf, ht, …) rely on it
- You can get rid of the section header by truncating the file:

```
truncate -s $(readelf -h file.elf | grep -F 'Start of section
headers' | awk '{print $5}') file.elf
```

Symbol Table

- Holds information needed to locate the program symbols
- Each symbol has
  - `Section` (to which it relates to)
  - `Name`
  - `Value`
  - `Size`
  - `Type` (object, function, file, no type)
  - `Binding`
    - `Local` (visible only to the object file that defines it)
    - `Global` (visible to all the object files combined)
    - `Weak` (like global, but it can be overridden by other definitions)

Stripped Binaries

- The symbol table can be removed from the binary

```
$ strip -s <program>
```

- In a stripped binary
  - The dynamic symbol names are preserved
    (for functions that have to be imported from shared libraries)
  - All the names of the program functions and global variables are lost
- Particularly bad when the program is statically linked
  - Hundreds of nameless library functions are mixed with the program code

Let's Have a Look

```
$ readelf <options>  filename
```

- `-h -S -l` → print the ELF, section, or program headers
- `-e` → print all the headers
- `-s` → print the symbols
- `-n` → print the notes
- `-d` → print the dynamic section
- `-r` → print relocation section

```
$ ldd program
$ lddtree program
```

- List the shared libraries required by the program

Process Creation

- The kernel loads the segments defined by the program headers into the process memory
  - If there is an interpreter defined, the kernel loads this binary as well
- The kernel sets up the stack and jump to the interpreter's entry point
  - If there's no interpreter, the process entry point is used

Functions and Global Symbols

- The address of global symbols imported from external libraries are computed when the binary is loaded in memory
  - But the `.text` segment is read-only.. so it cannot be modified
  - So, every time the code has to reference a global symbol, it does that through a Global Offset Table (GOT) in the data section
  - At run-time, the GOT entries are modified by the dynamic linker to point to the intended data

Functions and Global Symbols

- The address of global symbols imported from external libraries are computed when the binary is loaded in memory
  - But the `.text` segment is read-only.. so it cannot be modified
  - So, every time the code has to reference a global symbol, it does that through a Global Offset Table (GOT) in the data section
  - At run-time, the GOT entries are modified by the dynamic linker to point to the intended data
- If the code needs to call a function in a different module, the dynamic linker creates an array of read-only jump stubs, called Procedure Linking Table (PLT)
  - The stubs use entries in the GOT to invoke the right function
- Shared library code is Position Independent (PIC) and does not need relocation

Lazy Binding

- To improve the performance, entries in the GOT related to external functions are resolved at the first invocation

```
80483d5 <main>:
…
80483e5 call 80482f0 <printf@plt>
```

More info:
http://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one

Lazy Binding

- To improve the performance, entries in the GOT related to external functions are resolved at the first invocation

```
80483d5 <main>:
…
80483e5 call 80482f0 <printf@plt>
```

```
80482e0 <plt[0]>
80482e0 push        dword ptr [8049ff8h]
80482e6 jmp         dword ptr [8049ffch]
...
80482f0 <plt[2]>
80482f0 jmp  dword ptr [data_804a000]
80482f6 push        0
80482fb jmp         80482e0h
```

Lazy Binding

- To improve the performance, entries in the GOT related to external functions are resolved at the first invocation

```
8049ff4 <GOT>:
…
804a000 dd 80482f6
```

```
80483d5 <main>:
…
80483e5 call 80482f0 <printf@plt>
```

```
80482e0 <plt[0]>
80482e0 push        dword ptr [8049ff8h]
80482e6 jmp         dword ptr [8049ffch]
...
80482f0 <plt[2]>
80482f0 jmp  dword ptr [data_804a000]
80482f6 push        0
80482fb jmp         80482e0h
```

Push `0`, the offset in the `.rel.plt` section for the symbol to load
Check with `readelf -r`

Lazy Binding

- To improve the performance, entries in the GOT related to external functions are resolved at the first invocation

```
80483d5 <main>:
…
80483e5 call 80482f0 <printf@
```

```
8049ff4 <GOT>:
…
<GOT+8>: resolver
…
804a000 dd 80482f6
```

```
80482e0 <plt[0]>
80482e0 push        dword ptr [8049ff8h]
80482e6 jmp         dword ptr [8049ffch]
...
80482f0 <plt[2]>
80482f0 jmp  dword ptr [data_804a000]
80482f6 push        0
80482fb jmp         80482e0h
```

Lazy Binding

- To improve the performance, entries in the GOT related to external functions are resolved at the first invocation

```
80483d5 <main>:
…
80483e5 call 80482f0 <printf@
```

```
8049ff4 <GOT>:
…
<GOT+8>: resolver
…
804a000 dd <addr of printf>
```

```
80482e6 <plt[0]>
80482e6 push        dword ptr [8049ff8h]
80482e6 jmp         dword ptr [8049ffch]
...
80482e6 <plt[2]>
80482f0 jmp  dword ptr [data_804a000]
80482f6 push        0
80482fb jmp         80482e0h
```

Resolver function in the dynamic linker

Playing with ELF Files

lief

- Framework developed by Quarkslab to parse, modify, and abstract PE, ELF, MachO files.

```
import lief
b = lief.parse(FILENAME)
for s in b.sections:
  print s.name, s.entropy

for symbol in b.symbols:
    print(symbol)
```

Stripped & Statically Linked

- The worse case scenario for reverse engineering
- Need to separate the library code from the program code
  - Look for known functions in the binary
  - Unfortunately, libraries are re-compiled quite often and the byte representation of the same function may change
  - Static libraries contain many object files, and not all are included in the final program
- Solution:
  - Build a database of signatures of library functions
  - Match all signatures against the binary
  - Use the positive match to re-construct a symbol table
- Example: IDA Pro FLIRT signatures or Radare zignatures

- Is it a known binary?
  - Check file hash → MD5 on VirusTotal, AVClass
- Is it similar to something we already know?
  - Signatures → Submit to VirusTotal, Yara
- What does the malware do?
  - Embedded strings → strings
  - Imported libraries → ldd, dependency-walker
  - File headers and symbols → readelf, ht
    pefile, lief