

Binary Analysis - 1

REVERSE ENGINEERING

João Paulo Barraca

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Binary Objects



Binary files

- The result of a compilation process
 - Translating high level code (C/C++, etc...) into native code or bytecode
- Code is encapsulated in a binary format
 - It's not a raw file with unstructured bytes
- Target system (CPU or VM) will process the resulting code
 - Which may be only part of the file content

PE and ELF Formats

Linux: Executable and Linkable Format (ELF).

- **Windows:** Portable Executable (PE) format (.exe, .dll).
- Key sections of a binary:
 - `.text`: Contains the executable instructions (code).
 - `.data`: Initialized global and static variables.
 - `.rdata`/`.rodata`: Read-only data (e.g., hardcoded strings).

hello.c

Source code

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    printf("Hello World\n");

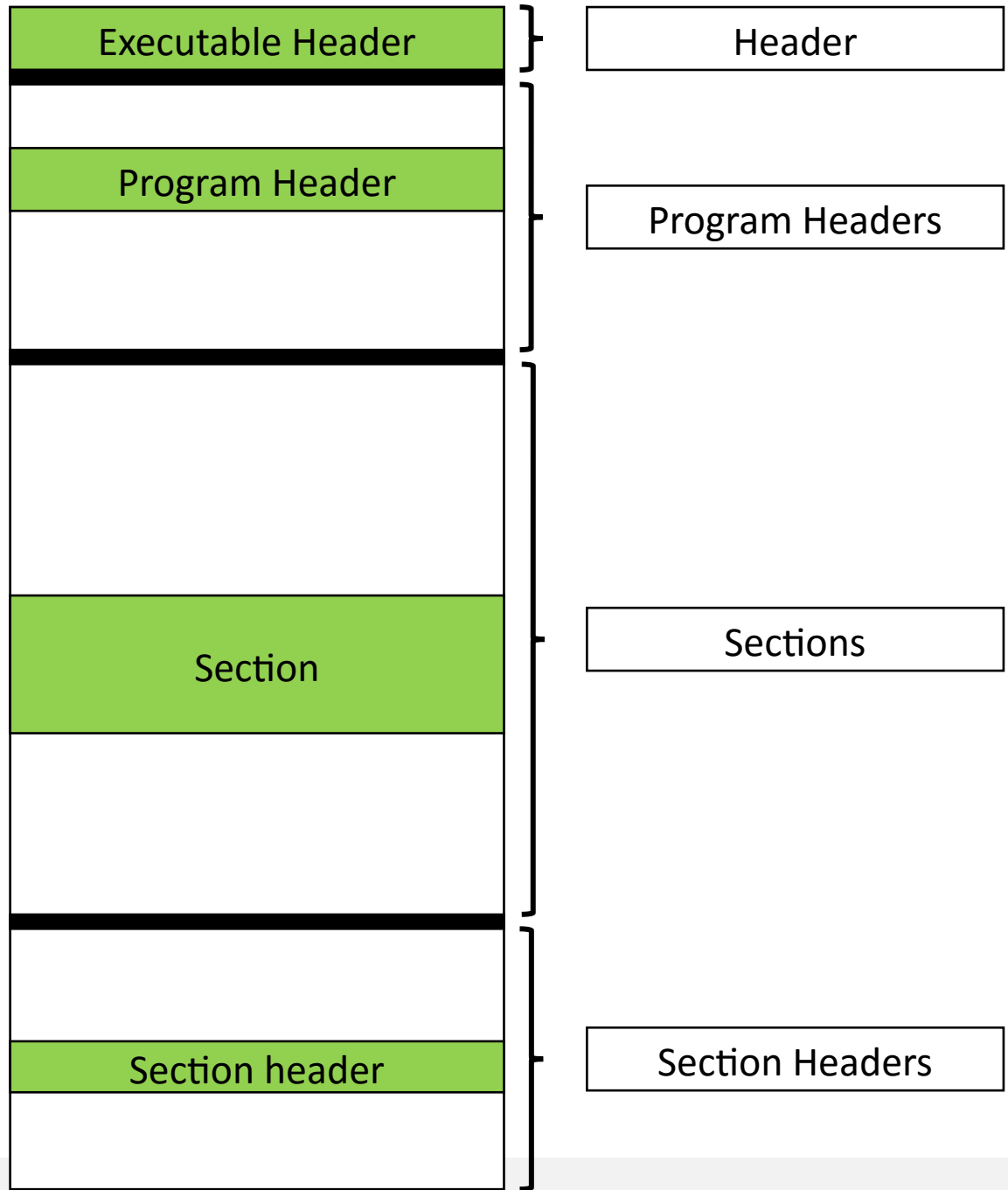
    return 0;
}
```

ELF Files



ELF – Executable and Linkable Format

- Container for executable files, object files, shared libraries, and core dumps
 - And other things out of this context like in Android
- Composed by several headers and sections:
 - Executable Header
 - Several Program Headers (optional)
 - Several Sections, with a header and content



ELF Headers

Executable Header

- Mandatory header, with basic information about the file
 - Architecture
 - Entry Point
 - Header locations and number
 - Type
 - Type of data
- Follow the structure **Elf64_Ehdr**
 - defined in `/usr/include/elf.h`

```
1 $ readelf -h hello
2
3 ELF Header:
4   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
5   Class:                               ELF64
6   Data:                                   2's complement, little endian
7   Version:                               1 (current)
8   OS/ABI:                                UNIX - System V
9   ABI Version:                           0
10  Type:                                   DYN (Shared object file)
11  Machine:                                Advanced Micro Devices X86-64
12  Version:                                0x1
13  Entry point address:                    0x1050
14  Start of program headers:                64 (bytes into file)
15  Start of section headers:               14688 (bytes into file)
16  Flags:                                   0x0
17  Size of this header:                     64 (bytes)
18  Size of program headers:                 56 (bytes)
19  Number of program headers:               11
20  Size of section headers:                 64 (bytes)
21  Number of section headers:                30
22  Section header string table index:       29
```

ELF Headers

Section Headers

- Sections are unstructured placeholders of data (frequently code) **targeting the Linker**
 - Some sections are well known and follow a defined structure
 - Some sections can be arbitrary binary blob
 - Some sections may contain content not useful for execution
 - Section order is irrelevant
 - Symbols, relocation information is stored in sections
- Headers describe the properties of each section
 - Name, type, flags, address when loaded, file offset, size, information...
- Files without linking, may omit section headers

```
1 $ readelf -S hello |grep "\["
2
3 [Nr] Name Type Address Offset
4 [ 0] NULL 0000000000000000 00000000
5 [ 1] .interp PROGBITS 00000000000002a8 000002a8
6 [ 2] .note.ABI-tag NOTE 00000000000002c4 000002c4
7 [ 3] .note.gnu.build-i NOTE 00000000000002e4 000002e4
8 [ 4] .gnu.hash GNU_HASH 0000000000000308 00000308
9 [ 5] .dynsym DYNSYM 0000000000000330 00000330
10 [ 6] .dynstr STRTAB 00000000000003d8 000003d8
11 [ 7] .gnu.version VERSYM 000000000000045a 0000045a
12 [ 8] .gnu.version_r VERNEED 0000000000000468 00000468
13 [ 9] .rela.dyn RELA 0000000000000488 00000488
14 [10] .rela.plt RELA 0000000000000548 00000548
15 [11] .init PROGBITS 0000000000001000 00001000
16 [12] .plt PROGBITS 0000000000001020 00001020
17 [13] .plt.got PROGBITS 0000000000001040 00001040
18 [14] .text PROGBITS 0000000000001050 00001050
19 [15] .fini PROGBITS 00000000000011c4 000011c4
20 [16] .rodata PROGBITS 0000000000002000 00002000
21 [17] .eh_frame_hdr PROGBITS 0000000000002010 00002010
22 [18] .eh_frame PROGBITS 0000000000002050 00002050
23 [19] .init_array INIT_ARRAY 0000000000003de8 00002de8
24 [20] .fini_array FINI_ARRAY 0000000000003df0 00002df0
25 [21] .dynamic DYNAMIC 0000000000003df8 00002df8
26 [22] .got PROGBITS 0000000000003fd8 00002fd8
27 [23] .got.plt PROGBITS 0000000000004000 00003000
28 [24] .data PROGBITS 0000000000004020 00003020
29 [25] .bss NOBITS 0000000000004030 00003030
30 [26] .comment PROGBITS 0000000000000000 00003030
31 [27] .symtab SYMTAB 0000000000000000 00003050
32 [28] .strtab STRTAB 0000000000000000 00003650
33 [29] .shstrtab STRTAB 0000000000000000 00003853
```

Executable Symbols

Tables

- Symbols are names identifying addresses of a binary
 - Have a type, such as Function, and including Undefined
 - E.g. functions create symbols, especially external functions (puts)
- ELF files have two symbol tables
 - .dynsym: symbols which will be allocated to memory when the program loads.
 - In the example, puts is provided by libc, required for operation, and exists as a dynamic symbol
 - .symtab: contains all symbols, including many used for linking and debugging, but not related to code required for execution.
 - These areas will not be allocated (mapped) to RAM
 - Extremely useful to identify the name of functions/sections when reversing!

Executable Symbols

Stripping

- Only symbols in the `.dyntab` are required
 - Identify allocated sections
 - Identify symbols that must be resolved in external libraries
 - Used for Dynamic Linking when the program is loaded
- **Stripping** is the process of removing unused symbols and code from a binary
 - Stripped binaries take less space, and are not reversed so easily
 - There is no hints about the purpose of a function from its name

```
1 $ readelf --syms hello
```

```
2
```

```
3 Symbol table '.dynsym' contains 7 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_deregisterTMCloneTab
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
3:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
4:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
5:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_registerTMCloneTable
6:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@GLIBC_2.2.5 (2)

```
12
```

```
13 Symbol table '.symtab' contains 64 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000000002a8	0	SECTION	LOCAL	DEFAULT	1	
2:	00000000000002c4	0	SECTION	LOCAL	DEFAULT	2	
...							
48:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@@GLIBC_2.2.5
49:	0000000000004030	0	NOTYPE	GLOBAL	DEFAULT	24	__edata
50:	00000000000011c4	0	FUNC	GLOBAL	HIDDEN	15	__fini
51:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_
52:	0000000000004020	0	NOTYPE	GLOBAL	DEFAULT	24	__data_start
53:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
54:	0000000000004028	0	OBJECT	GLOBAL	HIDDEN	24	__dso_handle
55:	0000000000002000	4	OBJECT	GLOBAL	DEFAULT	16	__IO_stdin_used
56:	0000000000001160	93	FUNC	GLOBAL	DEFAULT	14	__libc_csu_init
57:	0000000000004038	0	NOTYPE	GLOBAL	DEFAULT	25	__end
58:	0000000000001050	43	FUNC	GLOBAL	DEFAULT	14	__start
59:	0000000000004030	0	NOTYPE	GLOBAL	DEFAULT	25	__bss_start
60:	0000000000001135	34	FUNC	GLOBAL	DEFAULT	14	main
61:	0000000000004030	0	OBJECT	GLOBAL	HIDDEN	24	__TMC_END__
62:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_registerTMCloneTable
63:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@@GLIBC_2.2

```
34
```

Binary is stripped of extra symbols

Only the `.dysym` table is kept
Required for identifying allocatable areas
Notice as all symbols here are undefined (must be dynamically linked)

```
1 $ strip hello
2
3 $ readelf --syms hello
4
5 Symbol table '.dysym' contains 7 entries:
6   Num:      Value              Size Type   Bind   Vis      Ndx Name
7     0: 0000000000000000         0 NOTYPE LOCAL  DEFAULT UND
8     1: 0000000000000000         0 NOTYPE WEAK   DEFAULT UND _ITM_deregisterTMCloneTab
9     2: 0000000000000000         0 FUNC  GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
10    3: 0000000000000000         0 FUNC  GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
11    4: 0000000000000000         0 NOTYPE WEAK   DEFAULT UND __gmon_start__
12    5: 0000000000000000         0 NOTYPE WEAK   DEFAULT UND _ITM_registerTMCloneTable
13    6: 0000000000000000         0 FUNC  WEAK   DEFAULT UND __cxa_finalize@GLIBC_2.2.5 (2)
```

What is inside an Object File?

- An Object File contains information required to execute a program (not only code)
 - May not include all implementation, as this can be dynamically loaded
- Information is kept in sections, which are processed differently. Some are:
 - **.rodata**: readonly data, containing strings
 - **.got**: Global Offset Table - maps symbols to memory locations (offsets).
 - **.plt**: Procedure Linkage Table – uses the PLT to transfer execution to the correct location of a symbol, dealing with external symbols and fixing the GOT
 - **.bss**: **Block Starting Symbol** – contains uninitialized variables
 - **.dynsym**: List of symbols in allocatable memory
 - ... many others:
 - To read sections: `readelf -S hello`
 - To dump all code: `objdump -M intel -d hello`

```
1 $ objdump -sj .rodata hello
2
3 hello:      file format elf64-x86-64
4
5 Contents of section .rodata:
6 2000 01000200 48656c6c 6f20576f 726c6400  ....Hello World.
```

ELF Sections

.init and .fini

- Contains executable code required before/after the binary entry point is executed
 - Initialization tasks to prepare/clean the memory space
- Some uses:
 - prepare profiling tasks (`__gmon_start__`)
 - Invoke global constructors/destructors (C++)
 - Save program

```
1 $ objdump -M intel -d -j .init hello
2
3 hello:      file format elf64-x86-64
4
5
6 Disassembly of section .init:
7
8 00000000000001000 <_init>:
9     1000:      48 83 ec 08      sub    rsp,0x8
10    1004:      48 8b 05 dd 2f 00 00  mov    rax,QWORD PTR [rip+0x2fdd]      # 3fe8 <__gmon_start__>
11    100b:      48 85 c0          test   rax,rax
12    100e:      74 02           je     1012 <_init+0x12>
13    1010:      ff d0          call   rax
14    1012:      48 83 c4 08      add    rsp,0x8
15    1016:      c3            ret
```

ELF Sections

.text section

- Contains the main program code
 - The main target of a Reverse Engineering activity
 - Allocated as executable and read-only
 - Contains the user code, and additional code created by the compiler
 - Cleanup/initialization functions, stack guards, etc..
- In this section resides the program entry point
 - When the binary is loaded, execution flow is transferred that address
 - **Related** to the `main` function in a C program (but not the main)

```
13 | Entry point address: | 0x1050
```

ELF Sections

.text section: Entry Point

The **hello** program entry point address

```
1 $ objdump -M intel -d -j .text hello
2
3 hello:      file format elf64-x86-64
4
5
6 Disassembly of section .text:
7
8 0000000000001050 <_start>:
9   1050:    31 ed      xor     ebp,ebp
10  1052:    49 89 d1   mov     r9,rdx
11  1055:    5e        pop     rsi
12  1056:    48 89 e2   mov     rdx,rsp
13  1059:    48 83 e4 f0 and     rsp,0xfffffffffffffff0
14  105d:    50        push    rax
15  105e:    54        push    rsp
16  105f:    4c 8d 05 5a 01 00 00 lea     r8,[rip+0x15a]          # 11c0 <__libc_csu_fini>
17  1066:    48 8d 0d f3 00 00 00 lea     rcx,[rip+0xf3]         # 1160 <__libc_csu_init>
18  106d:    48 8d 3d c1 00 00 00 lea     rdi,[rip+0xc1]         # 1135 <main>
19  1074:    ff 15 66 2f 00 00   call   QWORD PTR [rip+0x2f66] # 3fe0 <__libc_start_main@GLIBC_2.2.5>
20  107a:    f4        hlt
21  107b:    0f 1f 44 00 00   nop   DWORD PTR [rax+rax*1+0x0]
22
23 ...
```

Loads the address of the main function into **RDI** (first argument) of a function

Calls [__libc_start_main@GLIBC 2.2.5](#) which transfers control to the program **main** function

ELF Sections

.bss, .data, .rodata

- **.rodata**: Read only data
 - Stores constant values
 - Mapped to a page marked as read only
- **.data**: Area with information to initialize variables
 - As the data can be modified, the section is writable
- **.bss**: Uninitialized variables
 - Memory is allocated for a variable that may be required, but nothing else is done
 - As there is no data associated, the .bss doesn't take space on the binary. Only instructs the system to reserve memory.

ELF Sections

.plt, .got, .got.plt

- **Procedure Linkage Table and Global Offset Table**
 - **.PLT**: Code to relocate symbols
 - **.GOT**: Array with addresses of each symbol requiring relocation
 - **.got** is similar to **.got.plt** but it's writable, while **.got** may be marked as Read Only as a security measure (-z relro)
 - Using a table (GOT) allows patching this table, while keeping libraries in same address, shared to multiple processes
- **Sections required for lazy binding (real time relocation)**
 - Linker needs to resolve the effective address of a code identified by a symbol (e.g., **puts**)
 - The code may be on the program, or on an external library, mapped to the virtual memory
 - **.plt** and **.got** ensure the symbol location is found and the code jumps around correctly
 - This is executed as the symbols are required! (LAZY)
 - On Linux, the Env Variable **LD_BIND_NOW** forces linking by the linker (on program load)
 - Will increase performance during execution, but will slow down startup

ELF Sections

Lazy Binding

(1) The **puts** function is called. The function is on an external library, and it must be relocated. So, it jumps to the **puts@plt**

```
1 0000000000001020 <.plt>:
2   1020:      push  QWORD PTR [rip+0x2fe2]      # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
3   1026:      jmp   QWORD PTR [rip+0x2fe4]      # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
4   102c:      nop   DWORD PTR [rax+0x0]
5
6 0000000000001030 <puts@plt>:
7   1030:      jmp   QWORD PTR [rip+0x2fe2]      # 4018 <puts@GLIBC_2.2.5>
8   1036:      push  0x0
9   103b:      jmp   1020 <.plt>
10
11 ....
12
13 0000000000001135 <main>:
14 ...
15   114b:      call  1030 <puts@plt>
16 ...
17
18 0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
19   4000:      f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .=.
20   ...
21   4018:      36 10 00 00 00 00 00 00 6.....
```


ELF Sections

Lazy Binding

(4) At the dynamic linker, it searches for the symbols in the mapped libraries and writes a value to the GOT at 0x4018.

Then he calls that address.

```
1 0000000000001020 <.plt>:
2   1020:      push   QWORD PTR [rip+0x2fe2]      # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
3   1026:      jmp    QWORD PTR [rip+0x2fe4]      # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
4   102c:      nop    DWORD PTR [rax+0x0]
5
6 → 0000000000001030 <puts@plt>:
7   1030:      jmp    QWORD PTR [rip+0x2fe2]      # 4018 <puts@GLIBC_2.2.5>
8   1036:      push   0x0
9   103b:      jmp    1020 <.plt>
10
11 ....
12
13 0000000000001135 <main>:
14 ...
15   114b:      call   1030 <puts@plt>
16 ...
17
18 0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
19   4000:      f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .=.
20   ...
21   4018:      36 10 00 00 00 00 00 00 6.....
```

ELF Sections

Lazy Binding

(2.1) At the PLT, the code doesn't jump to the final location, as it is not known (yet)

Instead, it jumps to an entry at the GOT (0x4018).

If the program is executing, and it is the second time `puts` is called, the entry has `0x7fffffff651910`, which points to the real `puts`.

This was obtained by loading the binary in GDB and using GEF

```
1 0000000000001020 <.plt>:
2   1020:      push   QWORD PTR [rip+0x2fe2]      # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
3   1026:      jmp    QWORD PTR [rip+0x2fe4]      # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
4   102c:      nop    DWORD PTR [rax+0x0]
5
6 → 0000000000001030 <puts@plt>:
7   1030:      jmp    QWORD PTR [rip+0x2fe2]      # 4018 <puts@GLIBC_2.2.5>
8   1036:      push   0x0
9   103b:      jmp    1020 <.plt>
10
11 ....
12
13 0000000000001135 <main>:
14 ...
15   114b:      call   1030 <puts@plt>
16 ...
17 gef> got
18
19 GOT protection: Partial RelRO | GOT functions: 1
20
21 [0x8004018] puts@GLIBC_2.2.5 → 0x7fffffff651910
```

ELF Sections

.rel.*, .rela.*

- Tables containing information to the dynamic linker about the required relocations
 - R_X86_64_GLOB_DAT: GOT offset should be filled with the symbol address (Lines 8-12)
 - R_X86_64_JUMP_SLO: Jump Slots to be represented in the `.got.plt` and `.plt` sections as shown previously (Line 16)

```
1 $ readelf --relocs hello
2
3 Relocation section '.rela.dyn' at offset 0x488 contains 8 entries:
4   Offset          Info          Type          Sym. Value     Sym. Name + Addend
5   000000003de8     000000000008  R_X86_64_RELATIVE          1130
6   000000003df0     000000000008  R_X86_64_RELATIVE          10f0
7   000000004028     000000000008  R_X86_64_RELATIVE          4028
8   000000003fd8     000100000006  R_X86_64_GLOB_DAT 0000000000000000 __ITM_deregisterTMClone + 0
9   000000003fe0     000300000006  R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
10  000000003fe8     000400000006  R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
11  000000003ff0     000500000006  R_X86_64_GLOB_DAT 0000000000000000 __ITM_registerTMCloneTa + 0
12  000000003ff8     000600000006  R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0
13
14 Relocation section '.rela.plt' at offset 0x548 contains 1 entry:
15   Offset          Info          Type          Sym. Value     Sym. Name + Addend
16  000000004018     000200000007  R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

ELF Sections

.dynamic section

- Contains information instructing the operating system/dynamic linker to load the binary
 - Address of important tables
 - Flags
 - Required libraries
 - Debug flags
 - INIT/FINI addresses

```
1 $ readelf --dynamic hello
2
3 Dynamic section at offset 0x2df8 contains 26 entries:
4   Tag                Type                Name/Value
5   0x0000000000000001 (NEEDED)           Shared library: [libc.so.6]
6   0x000000000000000c (INIT)             0x1000
7   0x000000000000000d (FINI)             0x11c4
8   0x0000000000000019 (INIT_ARRAY)      0x3de8
9   0x000000000000001b (INIT_ARRAYSZ)      8 (bytes)
10  0x000000000000001a (FINI_ARRAY)      0x3df0
11  0x000000000000001c (FINI_ARRAYSZ)      8 (bytes)
12  0x000000006ffffef5 (GNU_HASH)           0x308
13  0x0000000000000005 (STRTAB)           0x3d8
14  0x0000000000000006 (SYMTAB)           0x330
15  0x000000000000000a (STRSZ)            130 (bytes)
16  0x000000000000000b (SYMENT)           24 (bytes)
17  0x0000000000000015 (DEBUG)            0x0
18  0x0000000000000003 (PLTGOT)           0x4000
19  0x0000000000000002 (PLTRELSZ)         24 (bytes)
20  0x0000000000000014 (PLTREL)           RELA
21  0x0000000000000017 (JMPREL)           0x548
22  0x0000000000000007 (RELA)             0x488
23  0x0000000000000008 (RELASZ)           192 (bytes)
24  0x0000000000000009 (RELAENT)          24 (bytes)
25  0x000000006fffffff (FLAGS_1)          Flags: PIE
26  0x000000006ffffffe (VERNEED)          0x468
27  0x000000006fffffff (VERNEEDNUM)       1
28  0x000000006ffffff0 (VERSYM)           0x45a
29  0x000000006ffffff9 (RELACOUNT)        3
30  0x0000000000000000 (NULL)             0x0
```

ELF Program Headers

Overview

- Provide a **segment view** of the binary, complementing the **section view**
 - Type of segment, offset in the binary file, alignments, virtual addresses to be considered
 - **Target the operating system that will load the program and not the linker** as the sections do

```
1 $ readelf --wide --segments hello
2
3 Elf file type is DYN (Shared object file)
4 Entry point 0x1050
5 There are 11 program headers, starting at offset 64
6
7 Program Headers:
8   Type           Offset      VirtAddr           PhysAddr           FileSiz  MemSiz   Flg  Align
9   PHDR           0x000040   0x0000000000000040 0x0000000000000040 0x000268 0x000268 R    0x8
10  INTERP         0x0002a8   0x00000000000002a8 0x00000000000002a8 0x00001c 0x00001c R    0x1
11      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
12  LOAD           0x000000   0x0000000000000000 0x0000000000000000 0x000560 0x000560 R    0x1000
13  LOAD           0x001000   0x0000000000001000 0x0000000000001000 0x0001cd 0x0001cd R E  0x1000
14  LOAD           0x002000   0x0000000000002000 0x0000000000002000 0x000158 0x000158 R    0x1000
15  LOAD           0x002de8   0x0000000000003de8 0x0000000000003de8 0x000248 0x000250 RW  0x1000
16  DYNAMIC        0x002df8   0x0000000000003df8 0x0000000000003df8 0x0001e0 0x0001e0 RW  0x8
17  NOTE          0x002c4    0x0000000000002c4 0x0000000000002c4 0x000044 0x000044 R    0x4
18  GNU_EH_FRAME   0x002010   0x0000000000002010 0x0000000000002010 0x00003c 0x00003c R    0x4
19  GNU_STACK     0x000000   0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
20  GNU_RELRO     0x002de8   0x0000000000003de8 0x0000000000003de8 0x000218 0x000218 R    0x1
```

ELF Program Headers

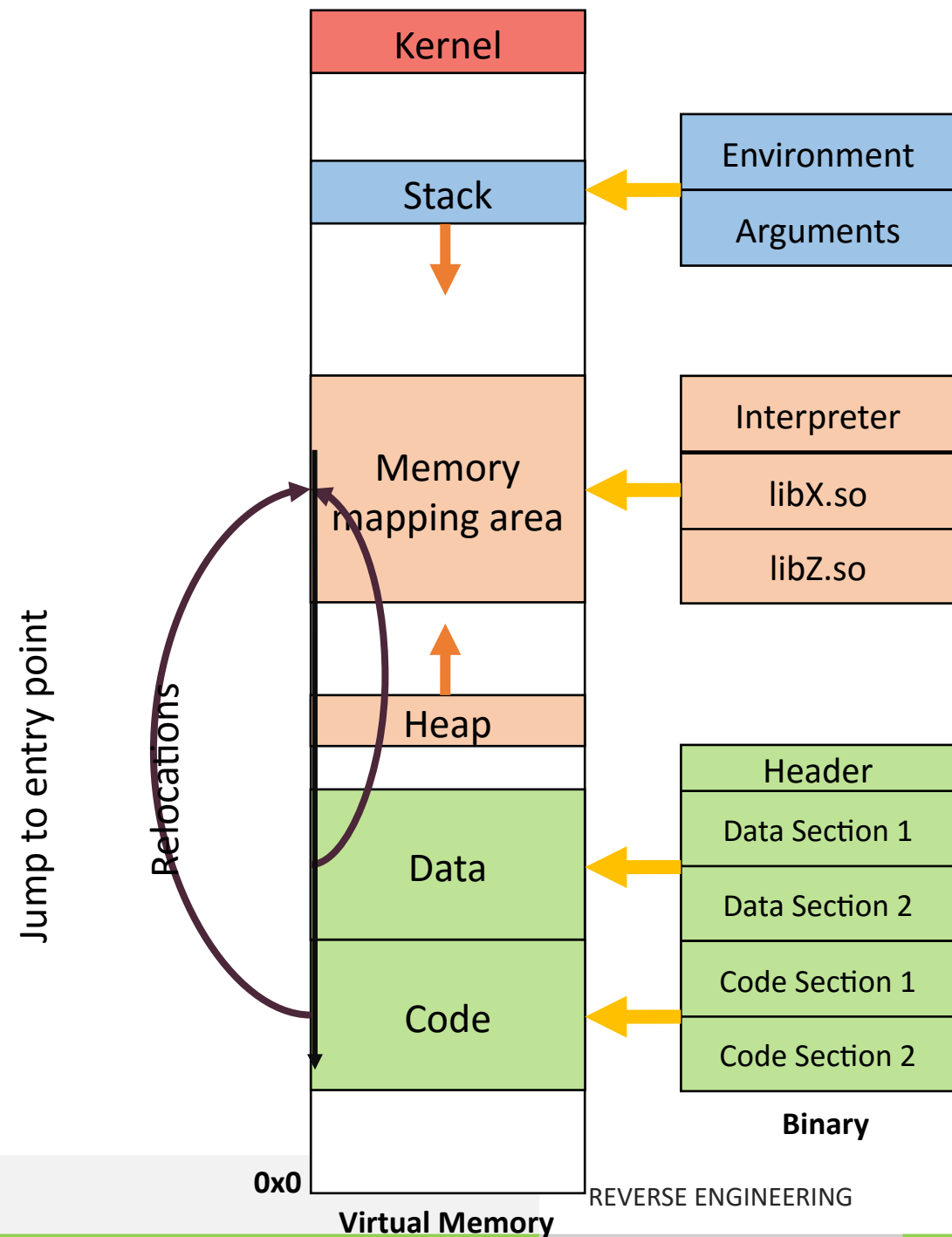
Types

- **LOAD:** Segment should be loaded in memory
- **INTERP:** Segment containing the name of the interpreter to be used
- **DYNAMIC:** Segment containing the `.dynamic` section, to be used by the interpreter

```
1 $ readelf --wide --segments hello
2
3 Elf file type is DYN (Shared object file)
4 Entry point 0x1050
5 There are 11 program headers, starting at offset 64
6
7 Program Headers:
8   Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg  Align
9   PHDR            0x000040 0x0000000000000040 0x0000000000000040 0x000268 0x000268 R    0x8
10  INTERP          0x0002a8 0x00000000000002a8 0x00000000000002a8 0x00001c 0x00001c R    0x1
11    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
12  LOAD            0x000000 0x0000000000000000 0x0000000000000000 0x000560 0x000560 R    0x1000
13  LOAD            0x001000 0x0000000000001000 0x0000000000001000 0x0001cd 0x0001cd R E  0x1000
14  LOAD            0x002000 0x0000000000002000 0x0000000000002000 0x000158 0x000158 R    0x1000
15  LOAD            0x002de8 0x0000000000003de8 0x0000000000003de8 0x000248 0x000250 RW  0x1000
16  DYNAMIC          0x002df8 0x0000000000003df8 0x0000000000003df8 0x0001e0 0x0001e0 RW  0x8
17  NOTE            0x0002c4 0x00000000000002c4 0x00000000000002c4 0x000044 0x000044 R    0x4
18  GNU_EH_FRAME    0x002010 0x0000000000002010 0x0000000000002010 0x00003c 0x00003c R    0x4
19  GNU_STACK       0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
20  GNU_RELRO       0x002de8 0x0000000000003de8 0x0000000000003de8 0x000218 0x000218 R    0x1
```

How are objects loaded?

- File is split according to existing sections
 - Each loaded at a different location (with different access attributes)
- Libraries are also mapped in the program address space
 - All code from libraries is present
- Stack grows downwards, heap grows upwards
 - On modern OS, growth may be limited, not on microcontrollers
- Interpreter is required to setup the binary in memory
 - `ld-Linux.so` or `ntdll.dll`
 - `readelf -p .interp filename`
 - Will handle relocations, resolving required symbols
 - If lazy-loading is used, relocation is done when the symbol is first used



PE Files

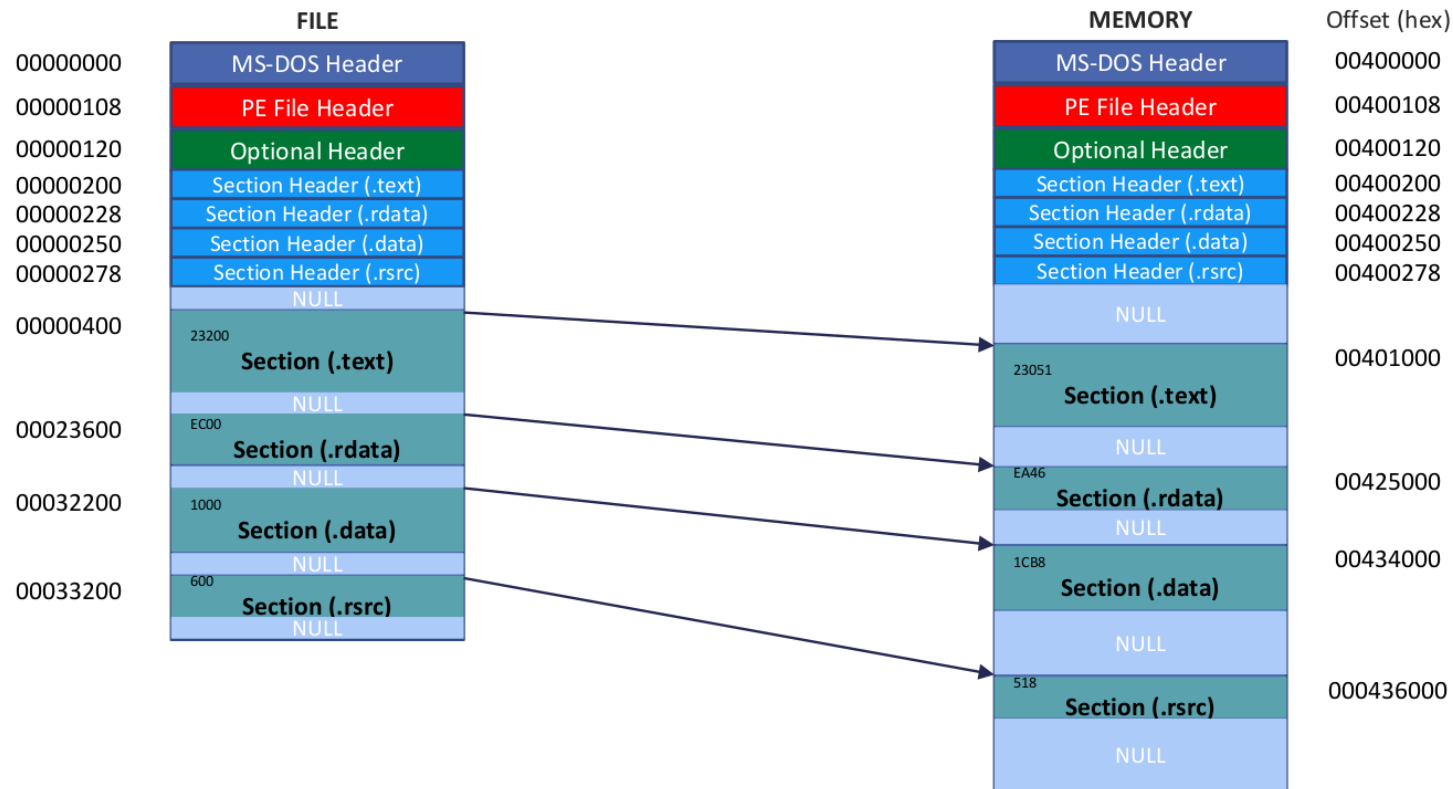


Portable Executable

- Format used for executables, object code and libraries in Windows
 - Includes .exe, .sys, .dll, .mui, .ocx, .scr, .tsp, .cpl, .drv, .ax, .acm and others
- Includes information about all requirements to load the code
 - External libraries required
 - Symbols exported
 - Resources
 - Icons, certificates, signatures
- From the *Common Object File Format* (COFF) specification

Portable Executable

- Magic: MZ (4D 5A)
 - Mark Zbikowski
- Structure is similar to ELF's
 - Additional stubs for DOS
 - and Windows NT
- Section header lists sections
- Each section provides a specific content to be loaded
 - .text, .rdata, .data, .rsrc



Portable Executable

> DosHeader		0h	40h	struct IMAGE_D...	
> DosStub		40h	D8h	struct IMAGE_D...	
> NtHeader		128h	108h	struct IMAGE_N...	
> SectionHeaders[7]		230h	118h	struct IMAGE_S...	
> Section[0]	.text	400h	11A400h	struct IMAGE_S...	
> Section[1]	.rdata	11A800h	49600h	struct IMAGE_S...	
> Section[2]	.data	163E00h	D400h	struct IMAGE_S...	
> Section[3]	.pdata	171200h	A200h	struct IMAGE_S...	
> Section[4]	._RDATA	17B400h	200h	struct IMAGE_S...	
> Section[5]	.rsrc	17B600h	C6000h	struct IMAGE_S...	
> Section[6]	.reloc	241600h	1800h	struct IMAGE_S...	
> ImportDescriptor[0]	SHLWAPI.dll	15F3F8h	14h	struct IMAGE_I...	
> ImportDescriptor[1]	IPHLAPI.DLL	15F40Ch	14h	struct IMAGE_I...	
> ImportDescriptor[2]	WS2_32.dll	15F420h	14h	struct IMAGE_I...	
> ImportDescriptor[3]	MPR.dll	15F434h	14h	struct IMAGE_I...	
> ImportDescriptor[4]	COMCTL32.dll	15F448h	14h	struct IMAGE_I...	
> ImportDescriptor[5]	VERSION.dll	15F45Ch	14h	struct IMAGE_I...	

```

00:0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ýý..
00:0010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00:0030 00 00 00 00 00 00 00 00 00 00 00 00 28 01 00 00 .....(....
00:0040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..°..'Í!..LÍ!Th
00:0050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00:0060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00:0070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...$......
00:0080 17 DF E8 F3 53 BE 86 A0 53 BE 86 A0 53 BE 86 A0 .BèóS%t S%t S%t
00:0090 18 C6 85 A1 56 BE 86 A0 18 C6 83 A1 96 BE 86 A0 .E...jV%t .Æfj-~%t
00:00A0 53 BE 86 A0 52 BE 86 A0 91 3F 7B A0 52 BE 86 A0 S%t R%t '?{ R%t
00:00B0 91 3F 82 A1 41 BE 86 A0 91 3F 85 A1 41 BE 86 A0 '?;A%t '?...A%t
00:00C0 91 3F 83 A1 37 BE 86 A0 18 C6 82 A1 4A BE 86 A0 '?fj7%t .Æ;J%t
00:00D0 18 C6 80 A1 51 BE 86 A0 18 C6 87 A1 66 BE 86 A0 .Æ€;Q%t .Æ‡;f%t
00:00E0 53 BE 87 A0 8E BC 86 A0 A1 3C 82 A1 50 BE 86 A0 S%t Ž%t ;<;P%t
00:00F0 A1 3C 83 A1 02 BE 86 A0 A1 3C 79 A0 52 BE 86 A0 ;<fj.%t ;<y R%t
00:0100 53 BE 11 A0 52 BE 86 A0 A1 3C 84 A1 52 BE 86 A0 S%. R%t ;<..R%t
00:0110 52 69 63 68 53 BE 86 A0 00 00 00 00 00 00 00 00 RichS%t .....
00:0120 00 00 00 00 00 00 00 00 50 45 00 00 64 86 07 00 .....PE...df..
00:0130 8E FC 55 66 00 00 00 00 00 00 00 00 F0 00 22 00 Žuuf.....ð."
00:0140 0B 02 0E 27 00 A4 11 00 00 C2 15 00 00 00 00 00 00 ...'µ...Â.....
00:0150 58 2B 0E 00 00 10 00 00 00 00 00 40 01 00 00 00 X+.....@.....
00:0160 00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00 .....
00:0170 06 00 00 00 00 00 00 00 00 B0 27 00 00 00 04 00 00 .....°'.....
00:0180 D3 62 24 00 02 00 60 81 00 00 10 00 00 00 00 00 00 Ób$...'.....
00:0190 00 10 00 00 00 00 00 00 00 00 10 00 00 00 00 00 .....
00:01A0 00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00 .....
00:01B0 00 00 00 00 00 00 00 00 F8 0B 16 00 08 02 00 00 .....ð.....
00:01C0 00 30 1B 00 90 5F 0C 00 00 70 1A 00 44 A0 00 00 00 .0..._...p...D ..
00:01D0 00 2E 24 00 B0 27 00 00 00 90 27 00 90 17 00 00 ..$.°'...'.....
00:01E0 B0 6D 14 00 38 00 00 00 00 00 00 00 00 00 00 00 °m..8.....
00:01F0 00 00 00 00 00 00 00 00 80 6F 14 00 28 00 00 00 .....€o..(....
00:0200 70 6C 14 00 40 01 00 00 00 00 00 00 00 00 00 00 pl..@.....
00:0210 00 C0 11 00 18 16 00 00 00 00 00 00 00 00 00 00 .Ä.....
00:0220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00:0230 2E 74 65 78 74 00 00 00 CE A3 11 00 00 10 00 00 .text...í£.....
00:0240 00 A4 11 00 00 04 00 00 00 00 00 00 00 00 00 00 00 .µ.....
00:0250 00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00 .... ..rdata..
00:0260 3E 95 04 00 00 C0 11 00 00 96 04 00 00 A8 11 00 >...Ä...-.....
00:0270 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 .....@.....@...@
00:0280 2E 64 61 74 61 00 00 00 5C 0E 04 00 00 60 16 00 .data...\.....
00:0290 00 D4 00 00 00 3E 16 00 00 00 00 00 00 00 00 00 .ô...>.....
00:02A0 00 00 00 00 40 00 00 C0 2E 70 64 61 74 61 00 00 .....@..Ä.pdata..
00:02B0 44 A0 00 00 00 70 1A 00 00 A2 00 00 00 12 17 00 D ...p...C.....
00:02C0 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 .....@...@...@
00:02D0 5F 52 44 41 54 41 00 00 F4 01 00 00 00 20 1B 00 .RDATA..ô....
00:02E0 00 02 00 00 00 B4 17 00 00 00 00 00 00 00 00 00 .....
00:02F0 00 00 00 00 40 00 00 40 2E 72 73 72 63 00 00 00 .....@..@.rsrc...
00:0300 90 5F 0C 00 00 30 1B 00 00 60 0C 00 00 B6 17 00 _...0...°'.....
00:0310 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 .....@...@...@
00:0320 2E 72 65 6C 6F 63 00 00 90 17 00 00 00 90 27 00 .reloc.....'

```

Dynamic Linker



Dynamic Linker

- Vital for the loading process, and can help reversing a program
 - Provide information about the loaded libraries
 - Help debugging the linking process
 - Force linking with custom libraries
 - And many other
- Communication is achieved through environmental variables
 - In the format LD_*
 - Setting a variable, or setting a variable with a specific value, activates Linker features

Dynamic Linker

LD_LIBRARY_PATH

- A list of directories in which to search for ELF libraries at execution time.
 - The items in the list are separated by either colons or semicolons
 - A zero-length directory name indicates the current working directory.

- Activating: `LD_LIBRARY_PATH=libs ./progrname`
 - Linker will look into `./libs` while loading libraries for the program
 - Allows having a different set of libraries for the program (E.g., debug versions)

Dynamic Linker

LD_BIND_NOW

- Causes the dynamic linker to **resolve all symbols at program startup** instead of deferring function call resolution to the point when they are first referenced.
 - Especially useful for debug as all symbols point to their correct location
- Activated by setting the variable: `LD_BIND_NOW=1 progname`

```
gef> got

GOT protection: Partial RelRO | GOT functions: 4

[0x8004018] pthread_create@GLIBC_2.2.5 → 0x8001036
[0x8004020] printf@GLIBC_2.2.5 → 0x7ffffff618560
[0x8004028] pthread_exit@GLIBC_2.2.5 → 0x8001056
[0x8004030] exit@GLIBC_2.2.5 → 0x8001066
```

LD_BIND_NOW not set

```
gef> got

GOT protection: Partial RelRO | GOT functions: 4

[0x8004018] pthread_create@GLIBC_2.2.5 → 0x7ffffff797280
[0x8004020] printf@GLIBC_2.2.5 → 0x7ffffff618560
[0x8004028] pthread_exit@GLIBC_2.2.5 → 0x7ffffff7981d0
[0x8004030] exit@GLIBC_2.2.5 → 0x7ffffff5f9ea0
```

LD_BIND_NOW is set

Dynamic Linker

LD_DEBUG

- Output verbose debugging information about the the dynamic linking
 - Allows tracing the operation of the linker
 - Debug where libraries are loading from
 - Determine if libraries are being loaded and which symbols trigger the event
 - Determine the search path used looking for libraries
- The content of this variable is one of more of the following categories, separated by colons/commas, spaces:
 - help, all, bindings, files, reloc, scopes, statistics, symbols, unused, version
- Use: `LD_DEBUG=option programname`

Dynamic Linker

LD_DEBUG

```
1 $ LD_DEBUG=all ./hello_thread
2 ▼ ...
3
4 7441: relocation processing: /lib/x86_64-linux-gnu/libc.so.6 (lazy)
5 7441: symbol=_res; lookup in file=./hello_thread [0]
6 7441: symbol=_res; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
7 7441: symbol=_res; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
8 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `_res' [GLIBC_2.2.5]
9 7441: symbol=stderr; lookup in file=./hello_thread [0]
10 7441: symbol=stderr; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
11 7441: symbol=stderr; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
12 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `stderr' [GLIBC_2.2.5]
13 7441: symbol=error_one_per_line; lookup in file=./hello_thread [0]
14 7441: symbol=error_one_per_line; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
15 7441: symbol=error_one_per_line; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
16 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `error_one_per_line' [GLIBC_2.2.5]
17 7441: symbol=__morecore; lookup in file=./hello_thread [0]
18 7441: symbol=__morecore; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
19 7441: symbol=__morecore; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
20 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__morecore' [GLIBC_2.2.5]
21 7441: symbol=__key_encryptsession_pk_LOCAL; lookup in file=./hello_thread [0]
22 7441: symbol=__key_encryptsession_pk_LOCAL; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
23 7441: symbol=__key_encryptsession_pk_LOCAL; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
24 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__key_encryptsession_pk_LOCAL' [GLIBC_2.2.5]
25 7441: symbol=__libpthread_freeres; lookup in file=./hello_thread [0]
26 7441: symbol=__libpthread_freeres; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
27 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libpthread.so.0 [0]: normal symbol `__libpthread_freeres'
28 7441: symbol=__progname_full; lookup in file=./hello_thread [0]
29 7441: symbol=__progname_full; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
30 7441: symbol=__progname_full; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
31 7441: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__progname_full' [GLIBC_2.2.5]
32 7441: symbol=__ctype32_tolower; lookup in file=./hello_thread [0]
33 7441: symbol=__ctype32_tolower; lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
34 7441: symbol=__ctype32_tolower; lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
```

Dynamic Linker

LD_PRELOAD

- A list of additional, user-specified, ELF shared objects to be loaded before all others.
 - This feature can be used to **selectively override functions** in other shared objects.
 - Symbols present in the provided ELF Shared objects are used instead of the original
 - Only the functions available in the shared object will be over written

- Use: `LD_PRELOAD=./liboverride.so progname`
 - Useful to provide custom implementations of any function in the program
 - Custom implementation can call the original implementation through manual symbol loading

hello_thread.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define NUM_THREADS    5
5
6  void *PrintHello(void *threadid)
7  {
8      long tid;
9      tid = (long)threadid;
10     printf("Hello World! It's me, thread #%ld!\n", tid);
11     pthread_exit(NULL);
12 }
13
14 int main(int argc, char *argv[])
15 {
16     pthread_t threads[NUM_THREADS];
17     int rc;
18     long t;
19     for(t=0;t<NUM_THREADS;t++){
20         printf("In main: creating thread %ld\n", t);
21         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
22         if (rc){
23             printf("ERROR; return code from pthread_create() is %d\n", rc);
24             exit(-1);
25         }
26     }
27
28     pthread_exit(NULL);
29 }
```

hello_thread.c

Dynamic symbols

```
1 $ readelf --dyn-syms hello_thread
2
3 Symbol table '.dynsym' contains 10 entries:
4   Num:      Value              Size Type   Bind   Vis      Ndx Name
5   0: 0000000000000000          0 NOTYPE LOCAL DEFAULT UND
6   1: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND pthread_create@GLIBC_2.2.5 (2)
7   2: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND _ITM_deregisterTMCloneTab
8   3: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (3)
9   4: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (3)
10  5: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND __gmon_start__
11  6: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND pthread_exit@GLIBC_2.2.5 (2)
12  7: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND exit@GLIBC_2.2.5 (3)
13  8: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND _ITM_registerTMCloneTable
14  9: 0000000000000000          0 FUNC   WEAK   DEFAULT UND __cxa_finalize@GLIBC_2.2.5 (3)
```

hello_thread.c

liboverride.c – compile with `gcc -shared -fPIC -o liboverride.so liboverride.c -ldl`

```
1  #define _GNU_SOURCE
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <dlfcn.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8
9  void pthread_exit(){
10     void (*orig_pthread_exit)(void) = dlsym(RTLD_NEXT, "pthread_exit");
11
12     printf("pthread_exit entry\n");
13     orig_pthread_exit();
14     printf("pthread_exit exit\n");
15 }
16
17 int pthread_create(void* a, void* b, void * c, void* d){
18     int (*orig_pthread_create)(void*, void*, void*, void*) = dlsym(RTLD_NEXT, "pthread_create");
19     int r = orig_pthread_create(a, b, c, d);
20     printf("pthread_create exit: ret=%d", r);
21     return r;
22 }
23 }
```

Manually load original function

Call original function

hello_thread.c

Left: standard execution, right: LD_PRELOAD overriding some functions

```
1 $ ./hello_thread
2 In main: creating thread 0
3 In main: creating thread 1
4 Hello World! It's me, thread #0!
5 In main: creating thread 2
6 Hello World! It's me, thread #1!
7 In main: creating thread 3
8 Hello World! It's me, thread #2!
9 In main: creating thread 4
10 Hello World! It's me, thread #3!
11 Hello World! It's me, thread #4!
```

```
1 LD_PRELOAD=./liboverride.so ./hello_thread
2 In main: creating thread 0
3 pthread_create entry: 0x7ffff9f3b5b0 (nil) 0x7f861ef96165 (nil)
4 pthread_create exit: ret=0
5 In main: creating thread 1
6 pthread_create entry: 0x7ffff9f3b5b8 (nil) 0x7f861ef96165 0x1
7 Hello World! It's me, thread #0!
8 pthread_create exit: ret=0
9 In main: creating thread 2
10 pthread_create entry: 0x7ffff9f3b5c0 (nil) 0x7f861ef96165 0x2
11 Hello World! It's me, thread #1!
12 pthread_exit entry
13 pthread_create exit: ret=0
14 In main: creating thread 3
15 pthread_exit entry
16 Hello World! It's me, thread #2!
17 pthread_exit entry
18 pthread_create entry: 0x7ffff9f3b5c8 (nil) 0x7f861ef96165 0x3
19 pthread_create exit: ret=0
20 In main: creating thread 4
21 pthread_create entry: 0x7ffff9f3b5d0 (nil) 0x7f861ef96165 0x4
22 Hello World! It's me, thread #3!
23 pthread_exit entry
24 Hello World! It's me, thread #4!
25 pthread_exit entry
26 pthread_create exit: ret=0
27 pthread_exit entry
```



PE Files



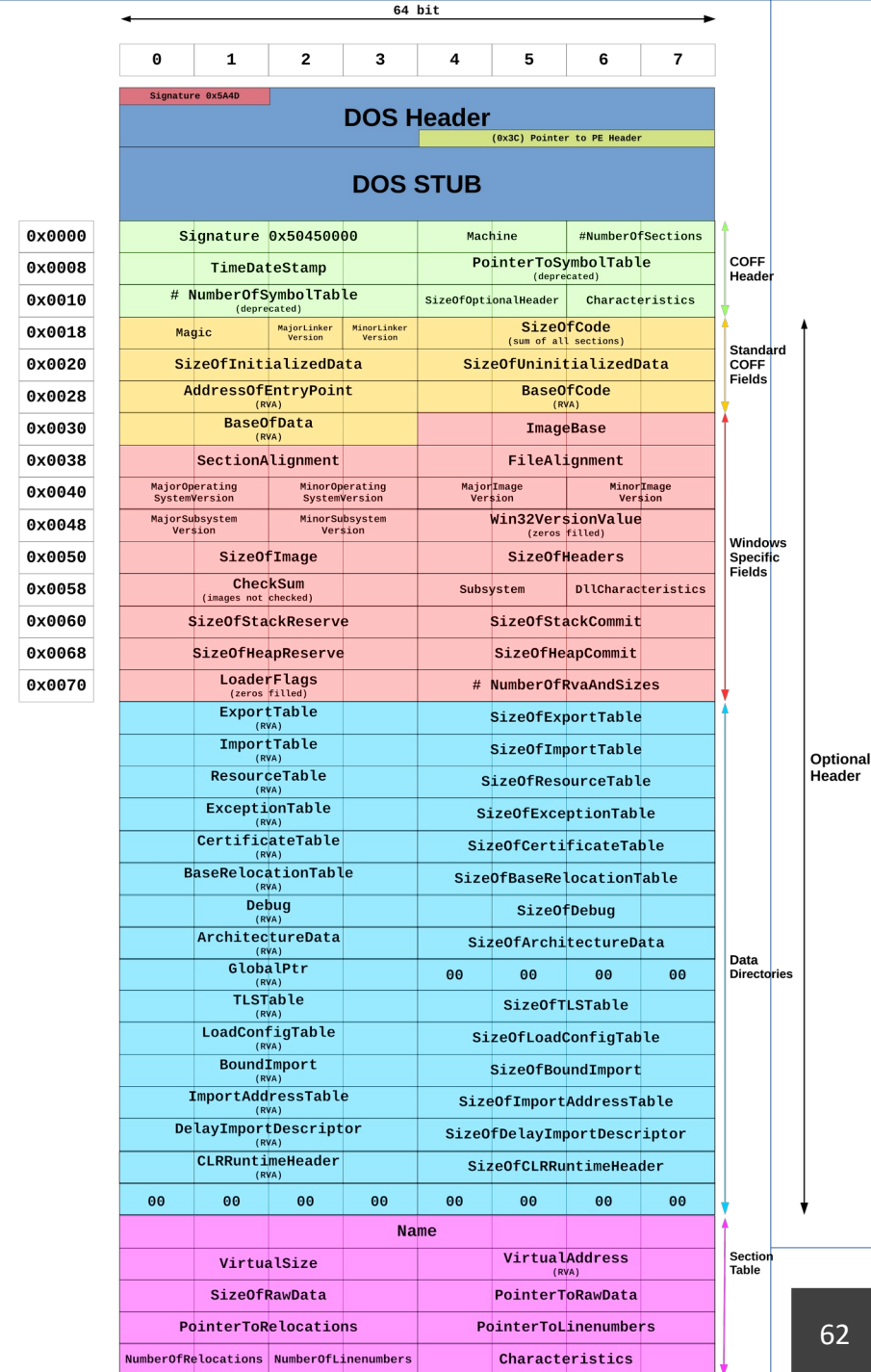
Portable Executable Files (PE)

The base of Windows Software

- The Portable Executable (PE) format is the standard data structure used by the Windows operating system for executables, object code, and DLLs.
- Derived from the earlier Common Object File Format (COFF) used in Unix.
- "Portable" refers to its versatility across different Windows architectures (x86, x64, ARM).
- Common Extensions: `.exe`, `.dll`, `.sys` (drivers), `.scr` (screensavers).

Macro-Structure of a PE File

- A PE file is a highly organized database containing:
 - Headers: instructions on how to load the file.
 - Sections: the actual code and data.
- **DOS Header:** Legacy compatibility.
- **PE Headers (NT Headers):** The core metadata.
- **Section Headers:** The map of the data payload.
- **Sections:** The actual code, variables, and resources.

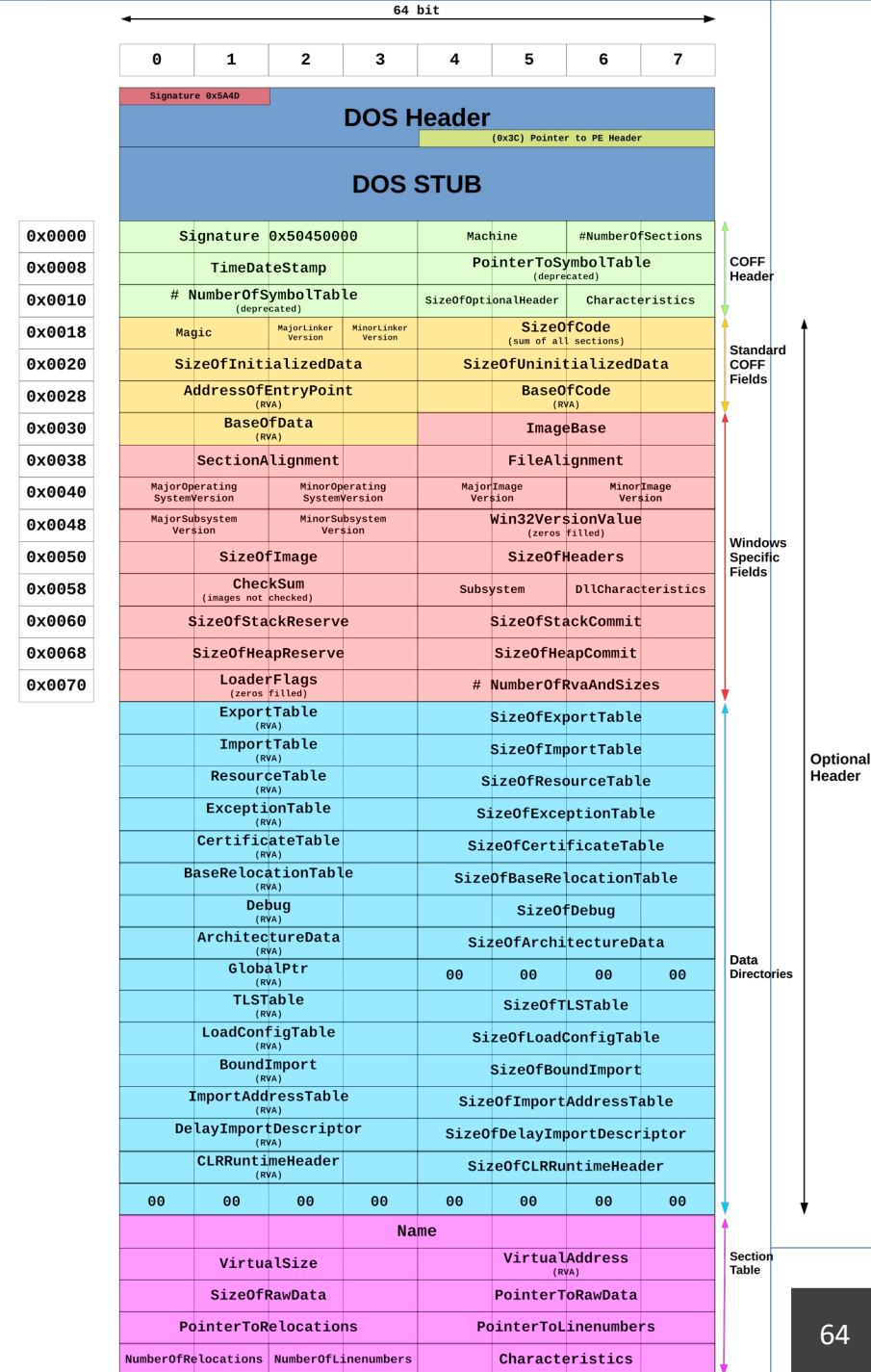


MS-DOS Header & Stub

- **Magic Bytes:** PE file starts with the characters MZ (hex: 4D 5A)
 - Named after Mark Zbikowski, an early MS-DOS architect.
- **DOS Stub:** A tiny, valid MS-DOS program embedded right after the header.
 - In DOS, it prints: "This program cannot be run in DOS mode."
- **The Golden Pointer:** The most critical field here is `e_lfanew` (offset 0x3C).
 - It acts as a pointer telling the OS exactly where the real PE header begins.

The PE Header (NT Headers)

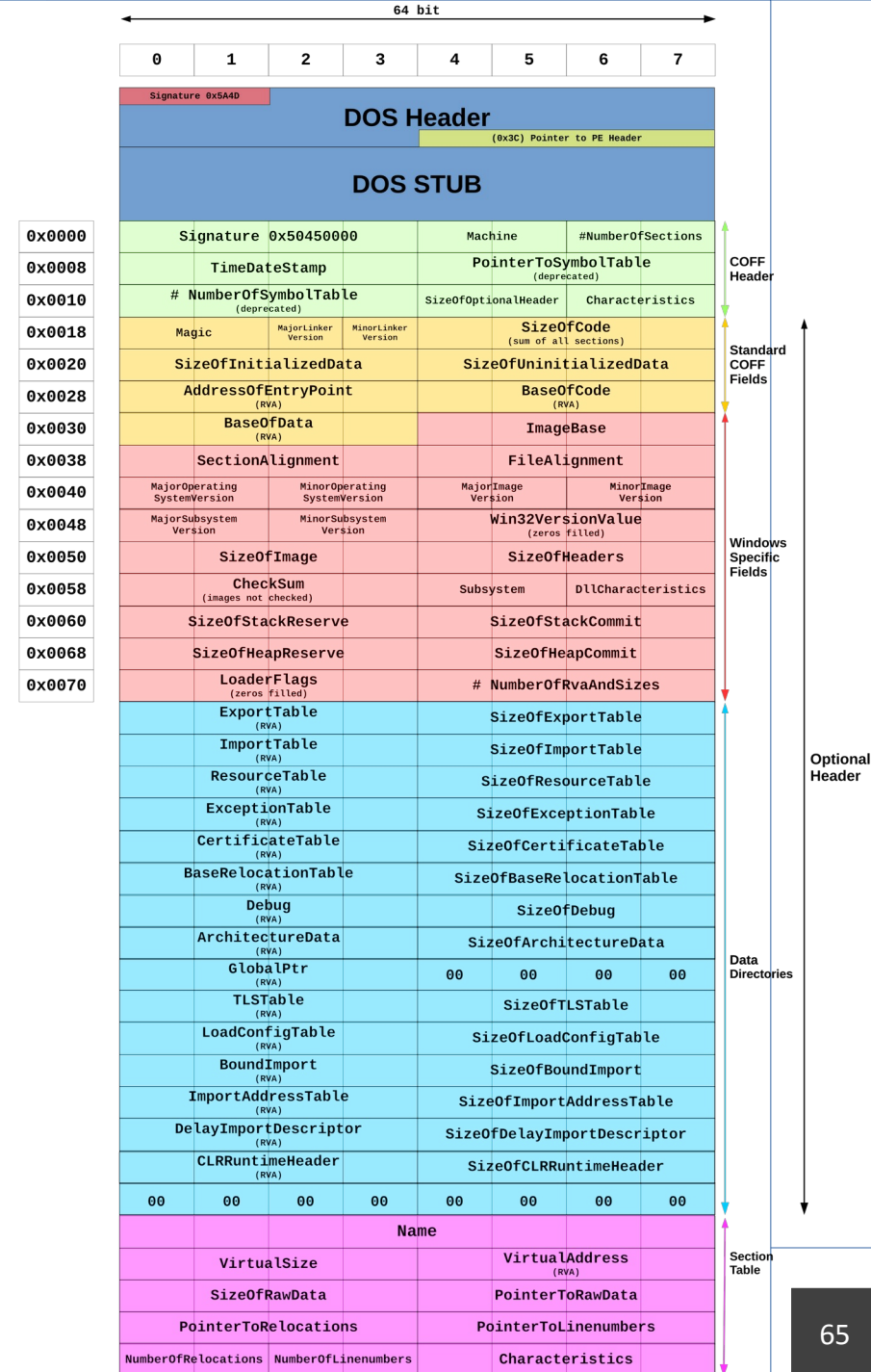
- At the e_lfanew offset: begins `PE\0\0` (hex: `50 45 00 00`).
- **File Header:** Contains basic physical layout data:
 - Target Machine Architecture (e.g., Intel 386, AMD64).
 - Number of Sections.
 - TimeDateStamp (when it was compiled).
- **Optional Header:** (Mandatory for executables). Contains crucial execution data:
 - AddressOfEntryPoint: Where the code starts executing.
 - ImageBase: The preferred memory address where the file wants to be loaded
 - often `0x00400000` for old 32-bit exes.



Data Directories

- Pointers to important data tables stored within the file's sections.
- Export Directory:** Lists the functions this file offers to other programs (crucial for .dll files).
- Import Directory:** Lists the external functions (APIs) this file needs from the OS
 - e.g., CreateFileW from kernel32.dll
- Others: CertificateTable, ResourceTable....

Security Note: Malware analysts heavily scrutinize the Import Address Table (IAT) to guess what a suspicious program might do based on the APIs it requests.



Concepts of Alignment and RVA

- When a PE file is launched, the Windows Loader doesn't just copy it bit-for-bit into RAM. It maps it based on defined boundaries.
- **Raw Offset** (File Alignment): How data is packed on the hard drive
 - typically aligned to 512 bytes
- **Virtual Address** (Section Alignment): How data is stretched out in RAM
 - typically aligned to 4096 bytes / 4KB pages
- **Relative Virtual Address (RVA)**: An address in memory relative to the program's starting point (ImageBase).

$$\textit{Absolute Address} = \textit{ImageBase} + \textit{RVA}$$

Where the Data Lives

Section Name	Purpose	Permissions
.text	Contains the executable CPU instructions (the code).	Read, Execute
.data	Initialized global and static variables.	Read, Write
.rdata	Read-only data (e.g., hardcoded strings, const variables).	Read
.bss	Uninitialized global variables (takes up no space on disk, only in memory).	Read, Write
.rsrc	Resources like icons, dialog boxes, and embedded manifests.	Read
.reloc	Relocation tables (used if the file cannot load at its preferred ImageBase).	Read

Metadata

- PE have additional information that can help analysis.

- Language, compiler, debug information, strings,

- ▼ PE64

- Operation system: Windows(Vista)[AMD64, 64-bit, Console]

- Linker: Microsoft Linker(14.36.35723)

- Compiler: Microsoft Visual C/C++(19.36.35723)[LTCG/C++]

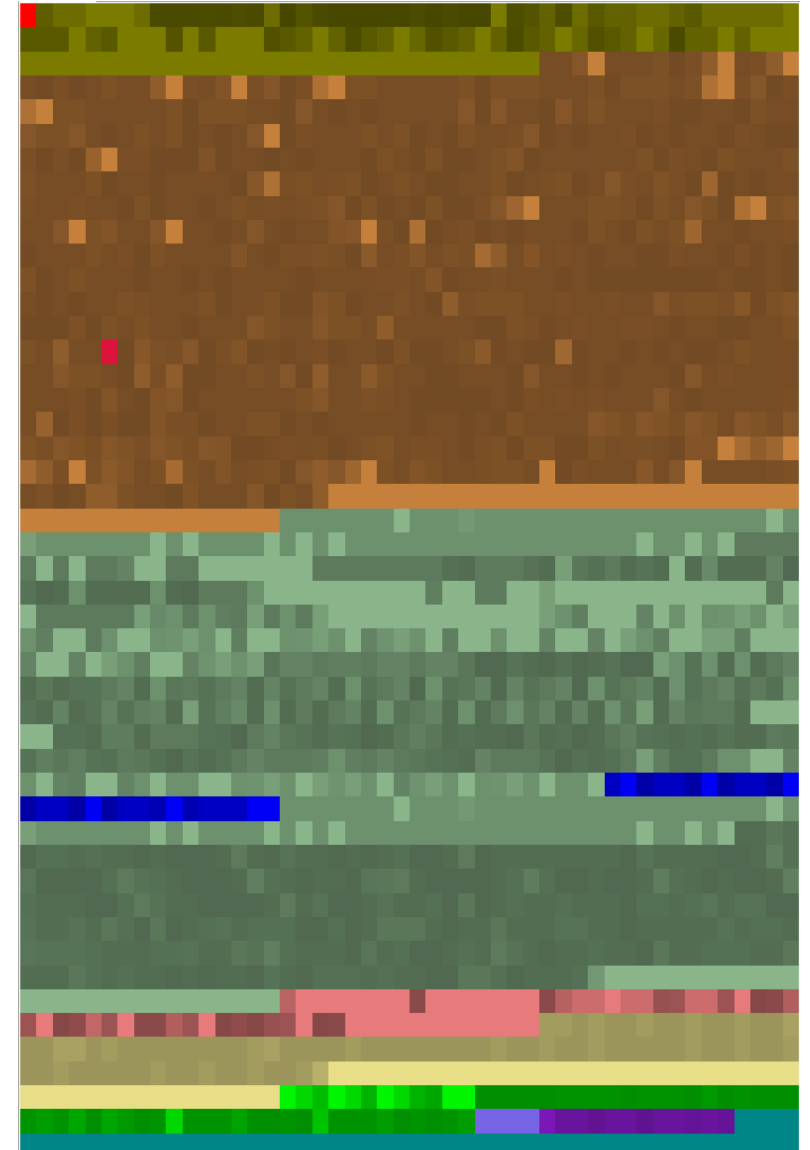
- Language: C++

- Library: Microsoft C/C++ Runtime[dynamic]

- Tool: Visual Studio(2022, v17.6)

- ▼ Debug data: Binary[Offset=0x295c,Size=0x5c]

- Debug data: PDB file link(7.0)



Metadata

- PE have additional information that can help analysis.

- Language, compiler, debug information, strings,

- ▼ PE64

- Operation system: Windows(Vista)[AMD64, 64-bit, Console]

- Linker: Microsoft Linker(14.36.35723)

- Compiler: Microsoft Visual C/C++(19.36.35723)[LTCG/C++]

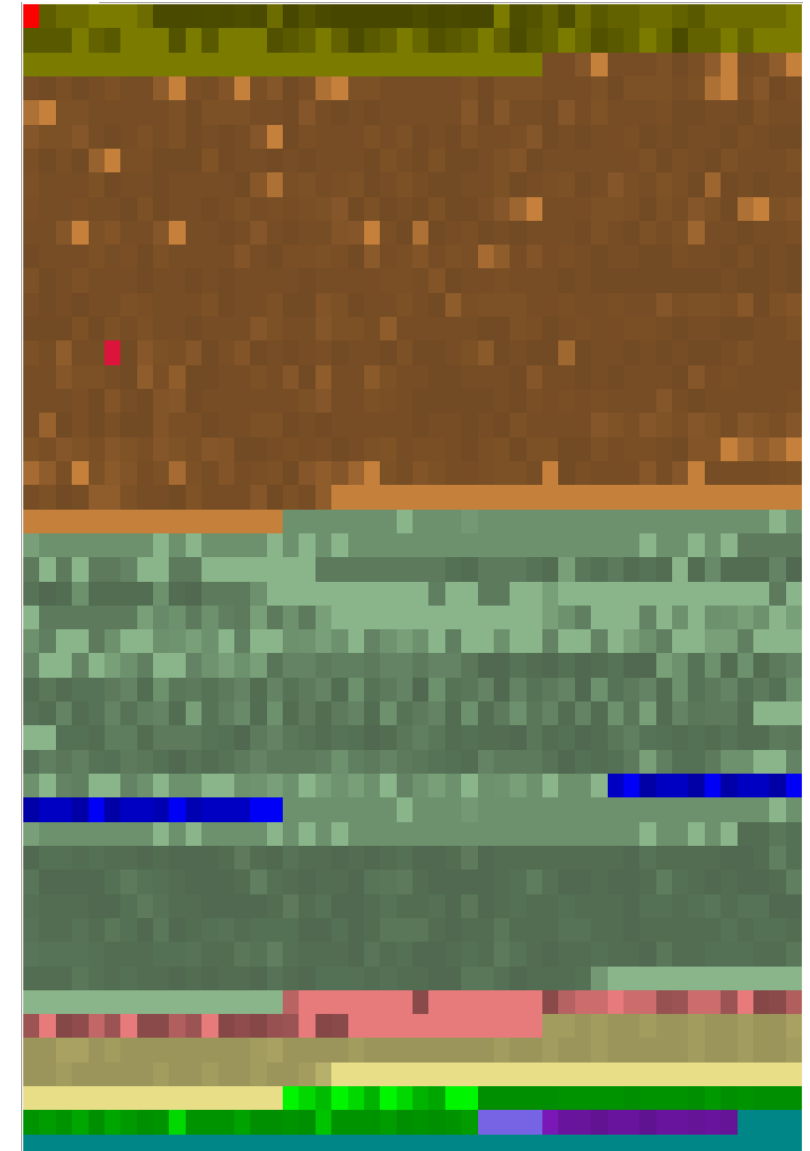
- Language: C++

- Library: Microsoft C/C++ Runtime[dynamic]

- Tool: Visual Studio(2022, v17.6)

- ▼ Debug data: Binary[Offset=0x295c,Size=0x5c]

- Debug data: PDB file link(7.0)



Language and Compiler Considerations

The Impact of the Compiler

- Not all PE or ELF files are created equal!
 - The language and compiler used to build the executable fundamentally change its internal structure.
- A file written in C/C++ looks entirely different from one written in C# or Go.

- **The Golden Rule:** Identifying the compiler or framework (using tools like Detect It Easy) is the first step in analysis because it dictates which tools and methodologies you must use.

Delphi Binaries

A popular rapid application development (RAD) language based on Object Pascal. Frequent in enterprise tools and specific malware families (banking trojans).

• Structure & Quirks

- **Section Names:** Often uses CODE, DATA, and BSS instead of the standard C-style .text and .data.
- **Strings:** Uses Pascal-style strings. Instead of ending with a null byte (\0), the first byte(s) indicate the length of the string.
- **Calling Convention:** Heavily relies on the fastcall convention
 - Arguments through EAX, EDX, and ECX registers rather than the stack.

• Analysis Process

- Standard disassemblers (Ghidra, IDA) work, but finding the WinMain entry point can be messy due to the Visual Component Library (VCL) initialization code.
- Specialized tools like Interactive Delphi Reconstructor (IDR) to recover form layouts and event handlers.

00458264	45	??	45h	E
00458265	00	??	00h	
00458266	0e	??	0Eh	
00458267	43	??	43h	C
00458268	52	??	52h	R
00458269	4b	??	4Bh	K
0045826a	42	??	42h	B
0045826b	54	??	54h	T
0045826c	4d	??	4Dh	M
0045826d	6f	??	6Fh	o
0045826e	75	??	75h	u
0045826f	73	??	73h	s
00458270	65	??	65h	e
00458271	4d	??	4Dh	M
00458272	6f	??	6Fh	o
00458273	76	??	76h	v
00458274	65	??	65h	e
00458275	11	??	11h	
00458276	00	??	00h	
00458277	08	??	08h	
00458278	87	??	87h	
00458279	45	??	45h	E
0045827a	00	??	00h	
0045827b	0a	??	0Ah	
0045827c	43	??	43h	C
0045827d	52	??	52h	R
0045827e	4b	??	4Bh	K
0045827f	42	??	42h	B
00458280	54	??	54h	T
00458281	43	??	43h	C
00458282	6c	??	6Ch	l
00458283	69	??	69h	i
00458284	63	??	63h	c
00458285	6b	??	6Bh	k
00458286	11	??	11h	
00458287	00	??	00h	

00412b43	8d 95 f8	LEA	EDX=>local_10c, [EBP + 0xfffffef8]
	fe ff ff		
00412b49	8b 45 08	MOV	EAX,dword ptr [EBP + Stack[0x4]]
00412b4c	8b 40 fc	MOV	EAX,dword ptr [EAX + -0x4]
00412b4f	e8 14 05	CALL	FUN_00403068
	ff ff		
00412b54	8d 85 f8	LEA	EAX=>local_10c, [EBP + 0xfffffef8]
	fe ff ff		
00412b5a	89 45 f8	MOV	dword ptr [EBP + local_c],EAX
00412b5d	c6 45 fc 04	MOV	byte ptr [EBP + local_8],0x4
00412b61	8d 45 f8	LEA	EAX=>local_c, [EBP + -0x8]
00412b64	50	PUSH	EAX
00412b65	6a 00	PUSH	0x0
00412b67	8d 95 f4	LEA	EDX=>local_110, [EBP + 0xfffffef4]
	fe ff ff		
00412b6d	a1 b4 a1	MOV	EAX=>PTR_DAT_004105f0, [PTR_PTR_0045a1b4]
	45 00		
00412b72	e8 21 2e	CALL	FUN_00405998
	ff ff		

Strings encoding and Calling Convention

Go (Golang) Binaries

Increasingly popular for cross-platform development and modern malware

•Structure & Quirks

- **Statically Linked:** Go binaries compile the entire runtime and all dependencies into the PE file, resulting in massive file sizes.
- **String Blobs:** Strings are not null-terminated. They are stored in massive, continuous blobs and referenced by a pointer and a length value.
- **The `pcIntab`:** The Program Counter Line Number Table. This is Go's internal mapping system that often contains unstripped function names and file paths.

•Analysis Process

- Opening a raw Go binary in a disassembler yields thousands of unlabeled, chaotic functions.
- **Restore symbols before analyzing.** Tools like GoReSym or other scripts parse the `pcIntab` and automatically rename the thousands of functions to their original Go names.

.NET Binaries (C# / VB.NET)

Programs built on Microsoft's .NET framework.

• Structure & Quirks

- The PE file is essentially a hollow shell. The standard PE Entry Point simply points to a tiny stub (usually an imported function called `_CorExeMain` from `mscorlib.dll`).
- This stub's only job is to load the Common Language Runtime (CLR).
- The actual logic is not x86/x64 assembly; it is stored as Microsoft Intermediate Language (MSIL) alongside rich metadata in a specific `.text` directory.

• Analysis Process

- Do not use traditional disassemblers (like IDA or Ghidra) for basic .NET analysis. You will only see the CLR loader stub.
- Use specialized .NET decompilers like dnSpy, ILSpy, or dotPeek. Because of the rich metadata, these tools can often recover the exact, highly readable C# source code (unless it has been aggressively obfuscated).

```

1
2 void entry(void)
3
4 {
5     /* WARNING: Could not recover jumtable at 0x004055be. Too many branches */
6     /* WARNING: Treating indirect jump as call */
7     _CorExeMain();
8     return;
9 }
10

```

```

*****
*                               FUNCTION                               *
*****
undefined4 __fastcall InitializeComponent(byte param_1, ...
    EAX:4      <RETURN>
    CL:1       param_1
    EDX:4      param_2
.NET CLR Managed Code
InitializeComponent
db[4711]
00402264 00 02 73
    1a 00 00
    0a 7d 01 ...
- 00402264 [0]      0h, 2h, 73h, 1Ah,
- 00402268 [4]      0h, 0h, Ah, 7Dh,
- 0040226c [8]      1h, 0h, 0h, 4h,
- 00402270 [12]     2h, 73h, 1Bh, 0h,
- 00402274 [16]     0h, Ah, 7Dh, 2h,
- 00402278 [20]     0h, 0h, 4h, 2h,
- 0040227c [24]     73h, 1Ch, 0h, 0h,
- 00402280 [28]     Ah, 7Dh, 3h, 0h,
- 00402284 [32]     0h, 4h, 2h, 73h,
- 00402288 [36]     1Ch, 0h, 0h, Ah,
- 0040228c [40]     7Dh, 9h, 0h, 0h,
- 00402290 [44]     4h, 2h, 73h, 1Ch,
- 00402294 [48]     0h, 0h, Ah, 7Dh,
- 00402298 [52]     Ah, 0h, 0h, 4h,
- 0040229c [56]     2h, 73h, 1Ch, 0h,
- 004022a0 [60]     0h, Ah, 7Dh, Bh,

```

Rust Binaries

A modern, memory-safe systems programming language. It is becoming increasingly popular for both legitimate high-performance applications and advanced malware

• Structure & Quirks:

- **LLVM Optimization:** Rust relies on the LLVM compiler infrastructure. Its aggressive optimizations make the resulting assembly look highly convoluted and alien compared to standard C/C++.
- **Name Mangling:** Rust encodes function names with complex mangling schemes (either legacy hash-based or the newer v0 scheme), resulting in extremely long, unreadable names (e.g., `_ZN3std2io5stdio6_print17he4e057f5c3547846E`).
- **Fat Pointers:** Strings and slices are not simple memory addresses or null-terminated. They are "fat pointers" that store both the memory address and the length.
- **Panic Handlers:** Rust's built-in error handling (panic!) relies on specific runtime functions. Even in stripped binaries, these panic handlers often survive and can leak original source file paths and line numbers.

• Analysis Process:

- Traditional tools (IDA, Ghidra) work, but separating the massive Rust standard library from the actual user code is difficult.
- Demangling is step one. Use tools like `rustfilt` or ensure your disassembler has an active Rust demangling script. To find the true main function, it is often easier to trace backwards from the main wrapper inside the Rust standard library initialization code.

```

162 uStack_70 = 0;
163 local_88 = 0;
164 uStack_80 = 0;
165 local_c8._48_8_ = 0;
166 uStack_90 = 0;
167 local_c8._32_8_ = 0;
168 local_c8._40_8_ = 0;
169 local_c8._16_8_ = 0;
170 local_c8._24_8_ = 0;
171 local_c8.__align_0_4_ = 0;
172 local_c8.__align_4_2_ = 0;
173 local_c8.__align_6_2_ = 0;
174 local_c8._8_8_ = 0;
175 local_38 = (_func_5327 *)0x0;
176 sigaction(0xb, (sigaction *)0x0, (sigaction *)&local_c8);
177 if ((pollfd)local_c8.__align == (pollfd)0x0) {
178     if (DAT_00150948 == '\0') {
179         DAT_00150948 = '\x01';
180         /* try { // try from 00111a37 to 00111ac5 has its CatchHandler @ 00111bf1 */
181         DAT_00150940 = FUN_00147150();
182         if (iVar5 == 0) {
183             FUN_001485d0(pcVar17, pcVar8);
184         }
185         bVar18 = false;
186     }
187     uStack_40._0_4_ = 0x8000004;
188     local_c8.__align = (long)FUN_0014ad20;
189     sigaction(0xb, (sigaction *)&local_c8, (sigaction *)0x0);
190 }
191 sigaction(7, (sigaction *)0x0, (sigaction *)&local_c8);
192 if ((pollfd)local_c8.__align == (pollfd)0x0) {
193     if (DAT_00150948 == '\0') {
194         DAT_00150948 = '\x01';
195         DAT_00150940 = FUN_00147150();
196         if (bVar18) {
197             FUN_001485d0(pcVar17, pcVar8);
198         }
199     }
200     uStack_40._0_4_ = 0x8000004;
201     local_c8.__align = (long)FUN_0014ad20;
202     sigaction(7, (sigaction *)&local_c8, (sigaction *)0x0);
203 }
204 _DAT_00150910 = param_2;
205 FUN_0010f440(real_main);
206 if (DAT_00150888 != 0) goto LAB_00111b81;
207 goto LAB_00111b0c;
208 }
209 local_d0 = (void *)0x0;
210 local_e0 = 0;
211 local_d8 = pthread_attr_getstack(&local_c8, &local_d0, &local_e0);
212 pvVar2 = local_d0;
213 if (local_d8 != 0) goto LAB_00111bc2;
214 local_d4 = pthread_attr_destroy(&local_c8);

```

Overly complex “main”
with initializers, file
checks and signal
checks

Actual main is here