Binary File Formats

Reverse Engineering – Binary File Formats

Albert López - newlog@overflowedminds.net

Goals

What can you expect after this module?

Goals

1. Understand how binaries store information
2. Learn how to get that information by yourself
3. Get a high level idea about how to go from binary to process

Goals

1. Understand how binaries store information
2. Learn how to get that information by yourself
3. Get a high level idea about how to go from binary to process

This module is completely hands-on!

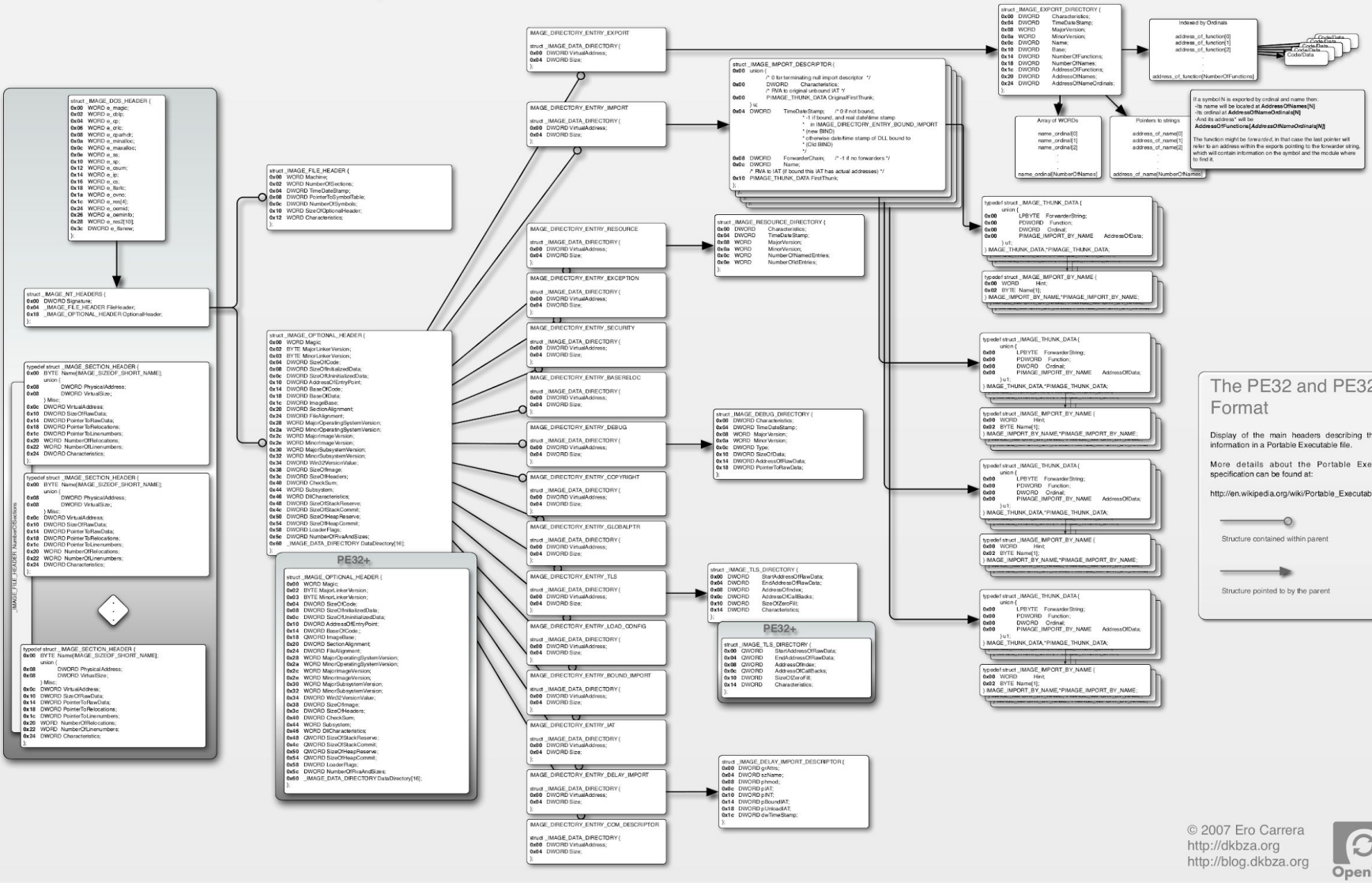
¡**Hero mode**!

Goals

All the code for this module can be found here:
https://github.com/newlog/binary_fun

Goals

Portable Executable Format Layout



Binary Files

The Basics

Binary Files → The Basics

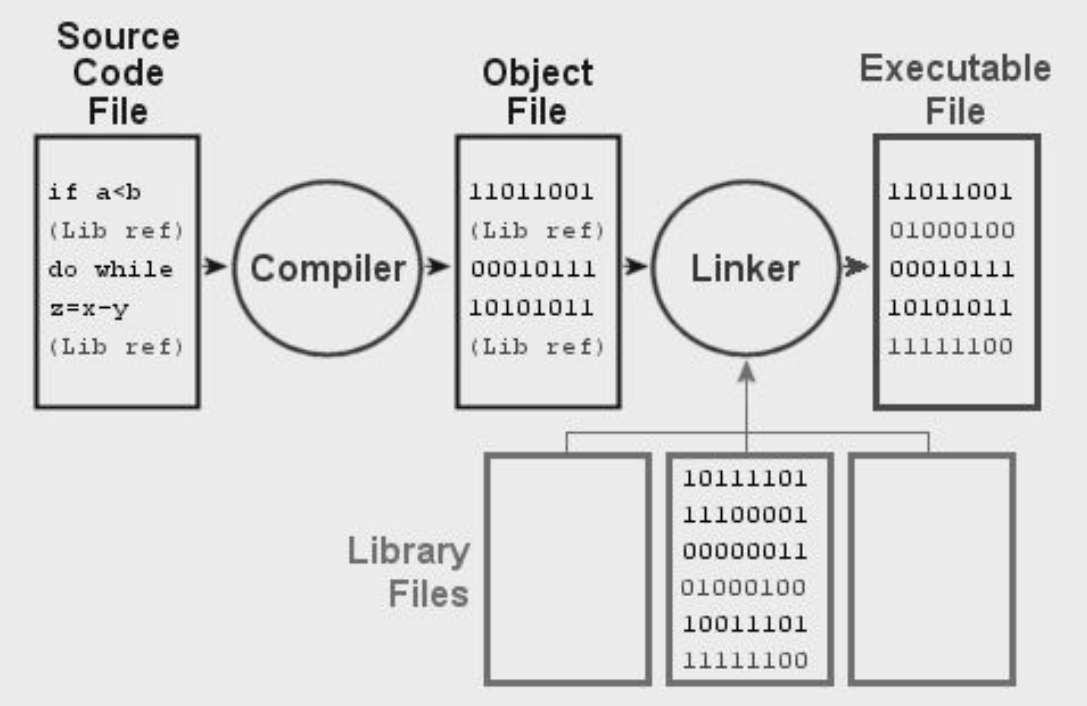
"An **executable file** is a regular file that describes how to initialize a new execution context"[1].

Basically, it's a container with all the information the operating system needs to spawn a process.

As any file, a binary file follows a specific structure. Their structure defines if they are an ELF binary, a PE binary, a Mach-o, etc.

[1] Understanding the Linux Kernel, 2nd Edition, Program Execution, page 808

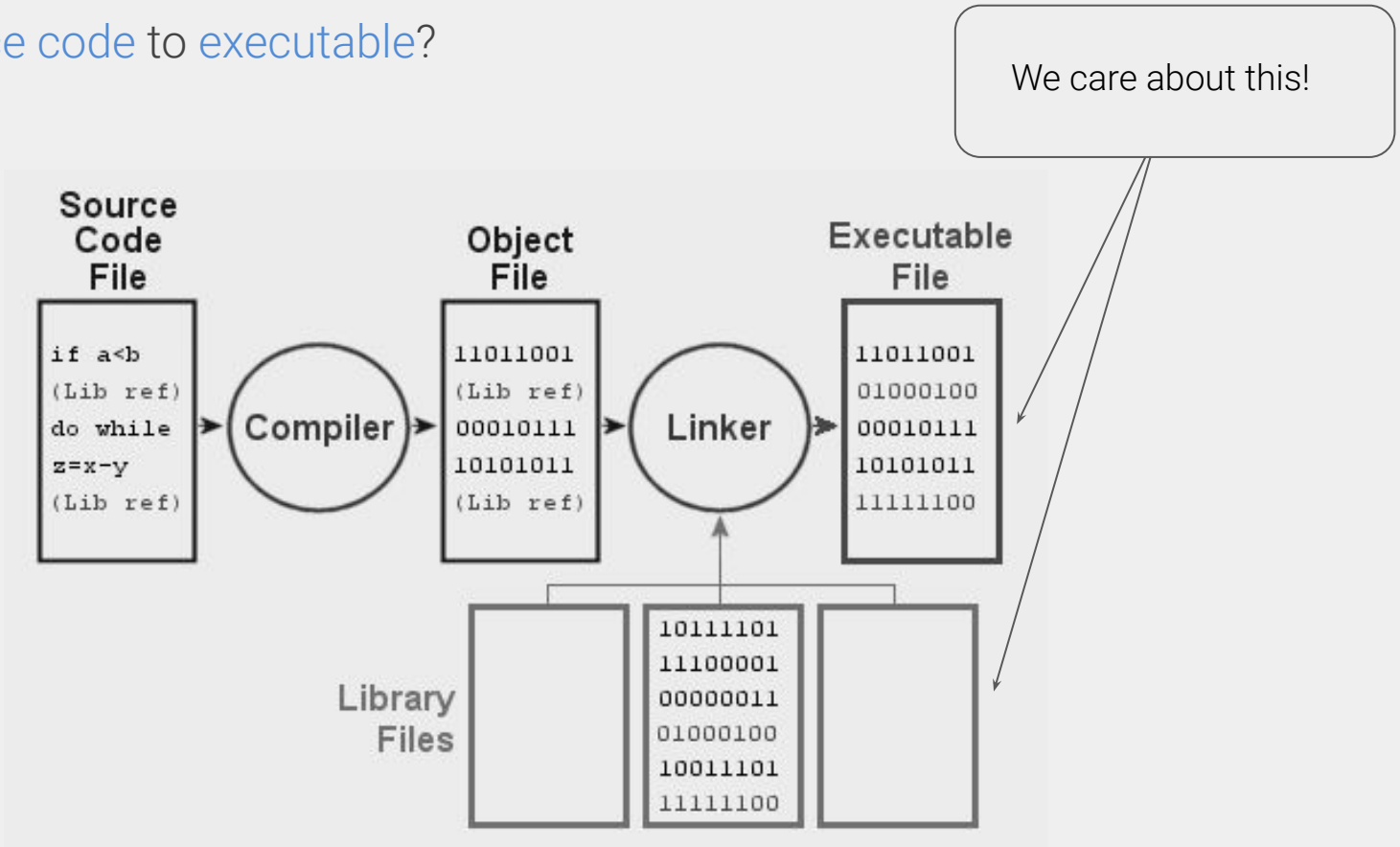
- From source code to executable?



Finally, the *loader*[1] reads the executable file and prepares it to be run.

[1] In Microsoft Windows 7 and above, the loader is the LdrInitializeThunk function contained in ntdll.dll

- From source code to executable?



Finally, the *loader*[1] reads the executable file and prepares it to be run.

[1] In Microsoft Windows 7 and above, the loader is the LdrInitializeThunk function contained in ntdll.dll

- **Executables** (.exe on Windows, no suffix on Linux):

The common binaries you are used to execute. They can either contain only the binary instructions that the developer generated or both the binary instructions and the libraries it depends on. Depending on if the binary is **dynamically linked** or **statically linked**, respectively.

- **Dynamic/Shared Libraries** (.dll on Windows, .so on Linux):

These are executables thought to be dynamically loaded as a dependency for regular executables at execution time. On Windows they are called Dynamically Linked Libraries and on Linux, Shared Objects.

- **Static Libraries** (.lib on Windows, .a on Linux):

These libraries are embedded in the executable at compile time. These are not ELFs/PEs, but just an archive containing multiple object files.

PE Binary Format

Binary Files → PE Binary Format

A PE (Portable Executable) file is the binary file format used by the Microsoft Windows operating system.

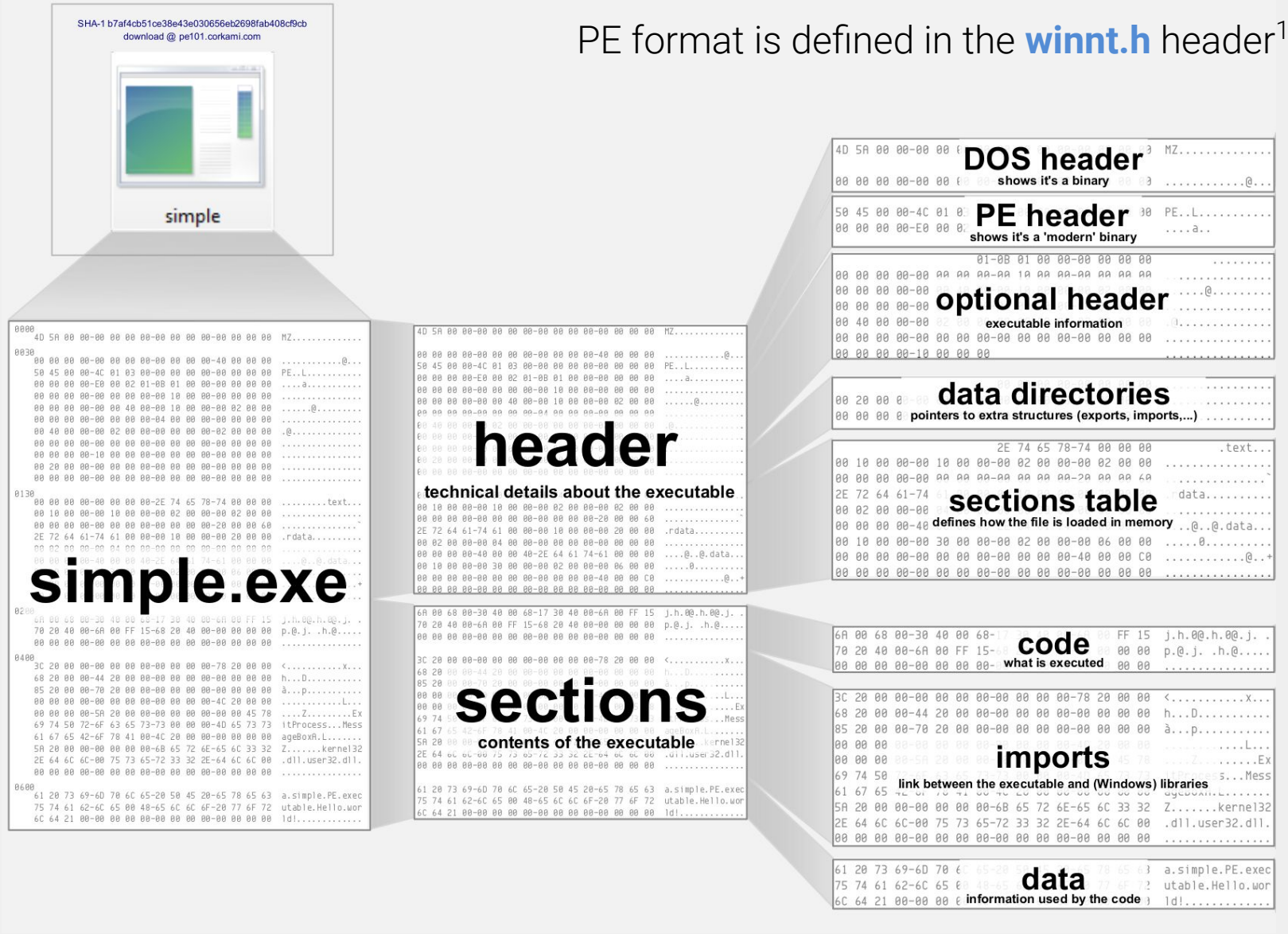
Common files that respond to this format are:

- .exe (common executable)
- .dll (dynamic library)
- scr (screen saver)
- sys (driver)
- (...)

PE files can contain instructions for the following architectures:

- IA32
- IA-64
- x86-64
- ARM
- (...)

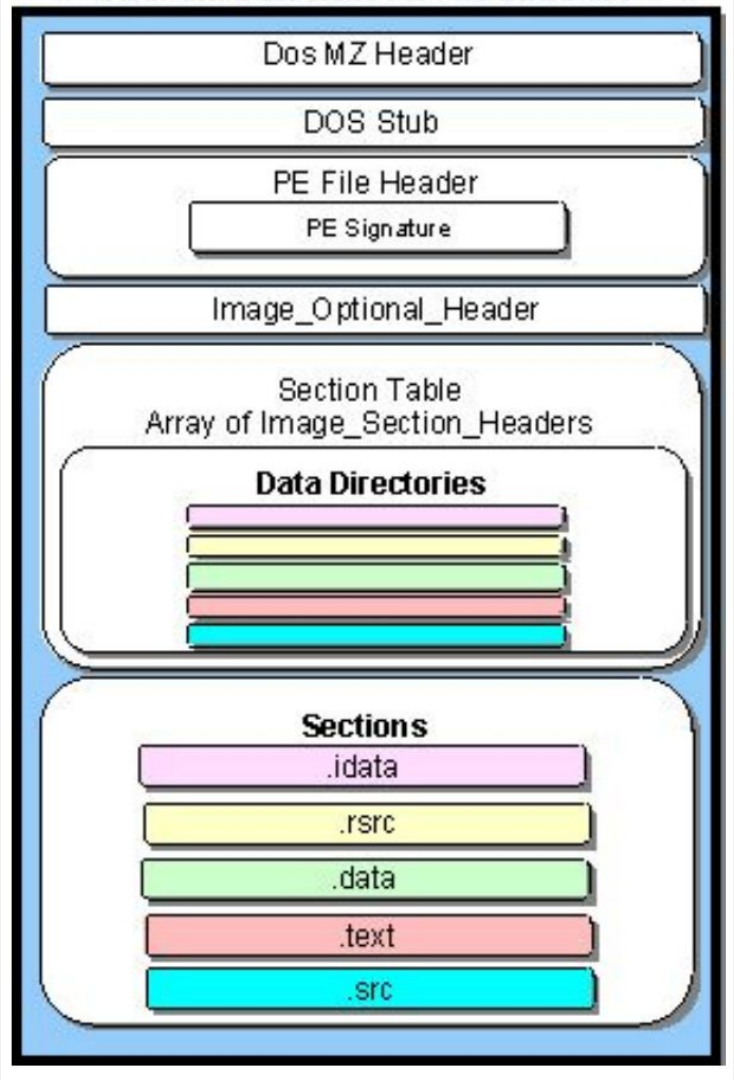
Binary Files → PE Binary Format



PE format is defined in the **winnt.h** header[1]

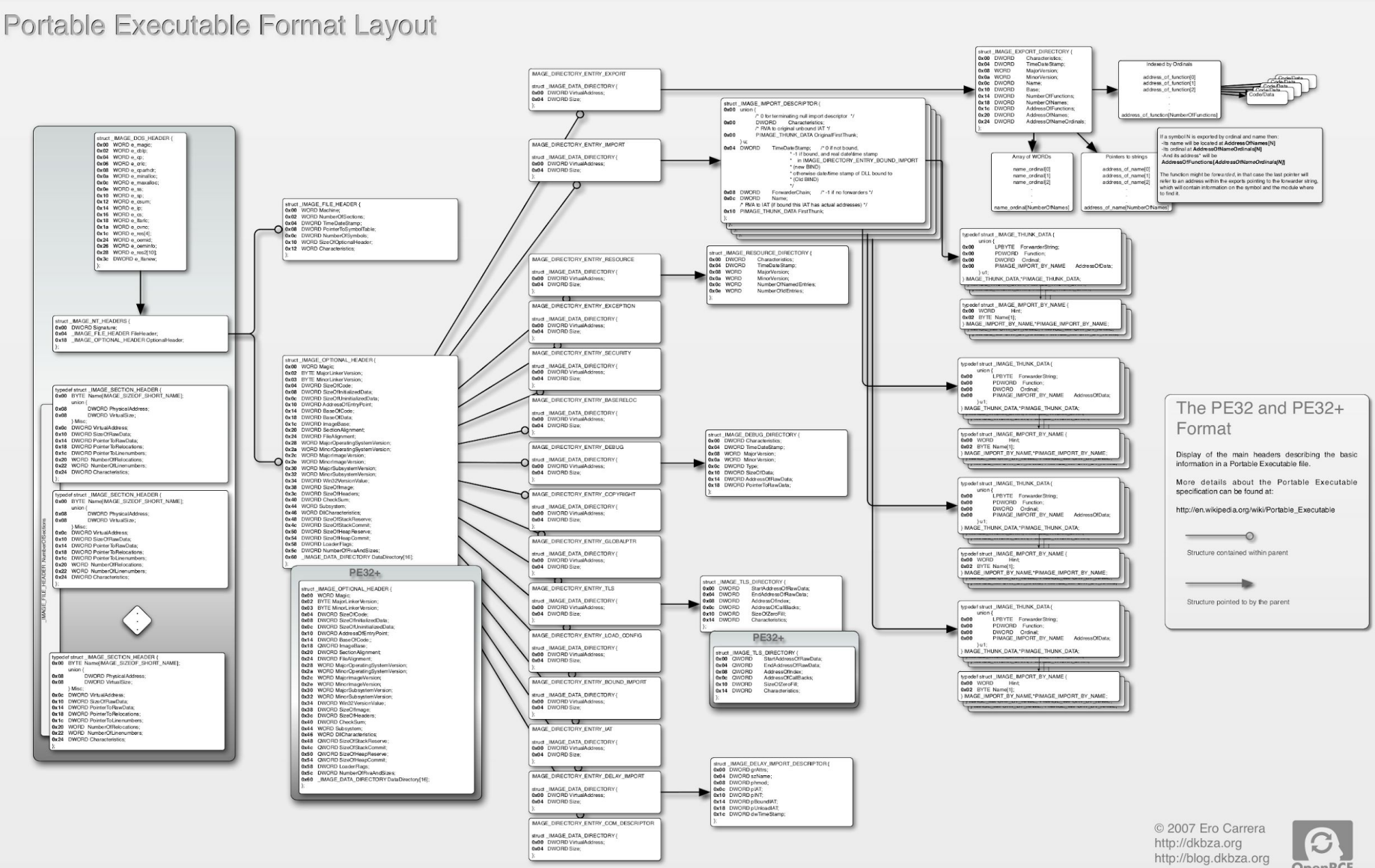
[1] https://github.com/wine-mirror/wine/blob/master/include/winnt.h

Binary Files → PE Binary Format



Binary Files → PE Binary Format

This is our Treasure Map!

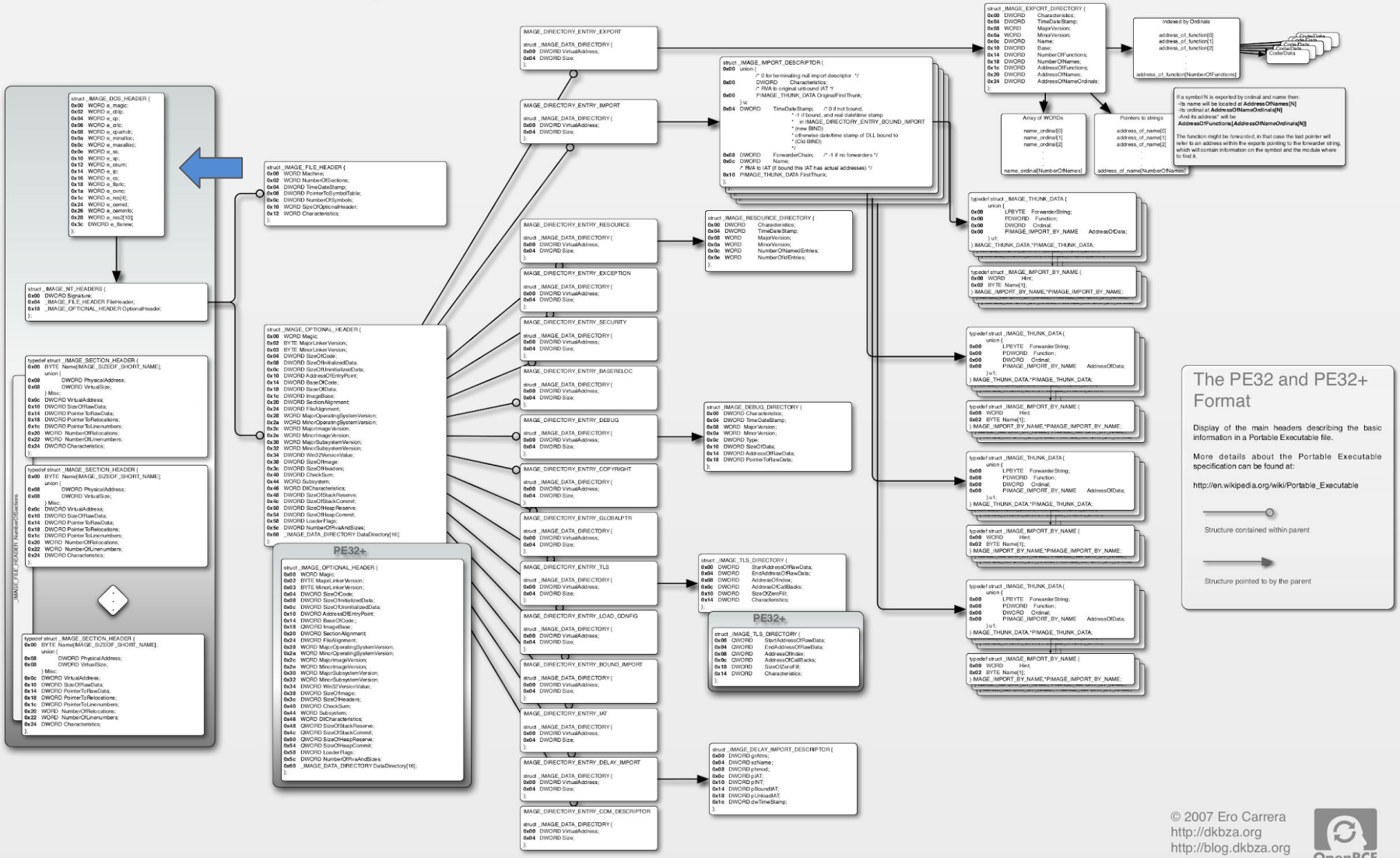


DOS/MZ Header

IMAGE_DOS_HEADER

Binary Files → PE Binary Format

Portable Executable Format Layout



The DOS Header → Generic information about the binary

```
typedef struct _IMAGE_DOS_HEADER {
    WORD   e_magic;      /* 00: MZ Header signature */
    WORD   e_cblp;       /* 02: Bytes on last page of file */
    WORD   e_cp;         /* 04: Pages in file */
    WORD   e_crlc;       /* 06: Relocations */
    WORD   e_cparhdr;    /* 08: Size of header in paragraphs */
    WORD   e_minalloc;   /* 0a: Minimum extra paragraphs needed */
    WORD   e_maxalloc;   /* 0c: Maximum extra paragraphs needed */
    WORD   e_ss;         /* 0e: Initial (relative) SS value */
    WORD   e_sp;         /* 10: Initial SP value */
    WORD   e_csum;       /* 12: Checksum */
    WORD   e_ip;         /* 14: Initial IP value */
    WORD   e_cs;         /* 16: Initial (relative) CS value */
    WORD   e_lfarlc;     /* 18: File address of relocation table */
    WORD   e_ovno;       /* 1a: Overlay number */
    WORD   e_res[4];     /* 1c: Reserved words */
    WORD   e_oemid;      /* 24: OEM identifier (for e_oeminfo) */
    WORD   e_oeminfo;    /* 26: OEM information; e_oemid specific */
    WORD   e_res2[10];   /* 28: Reserved words */
    DWORD e_lfanew;      /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Let’s build a tool to read the MZ header!

1.dos_header.py

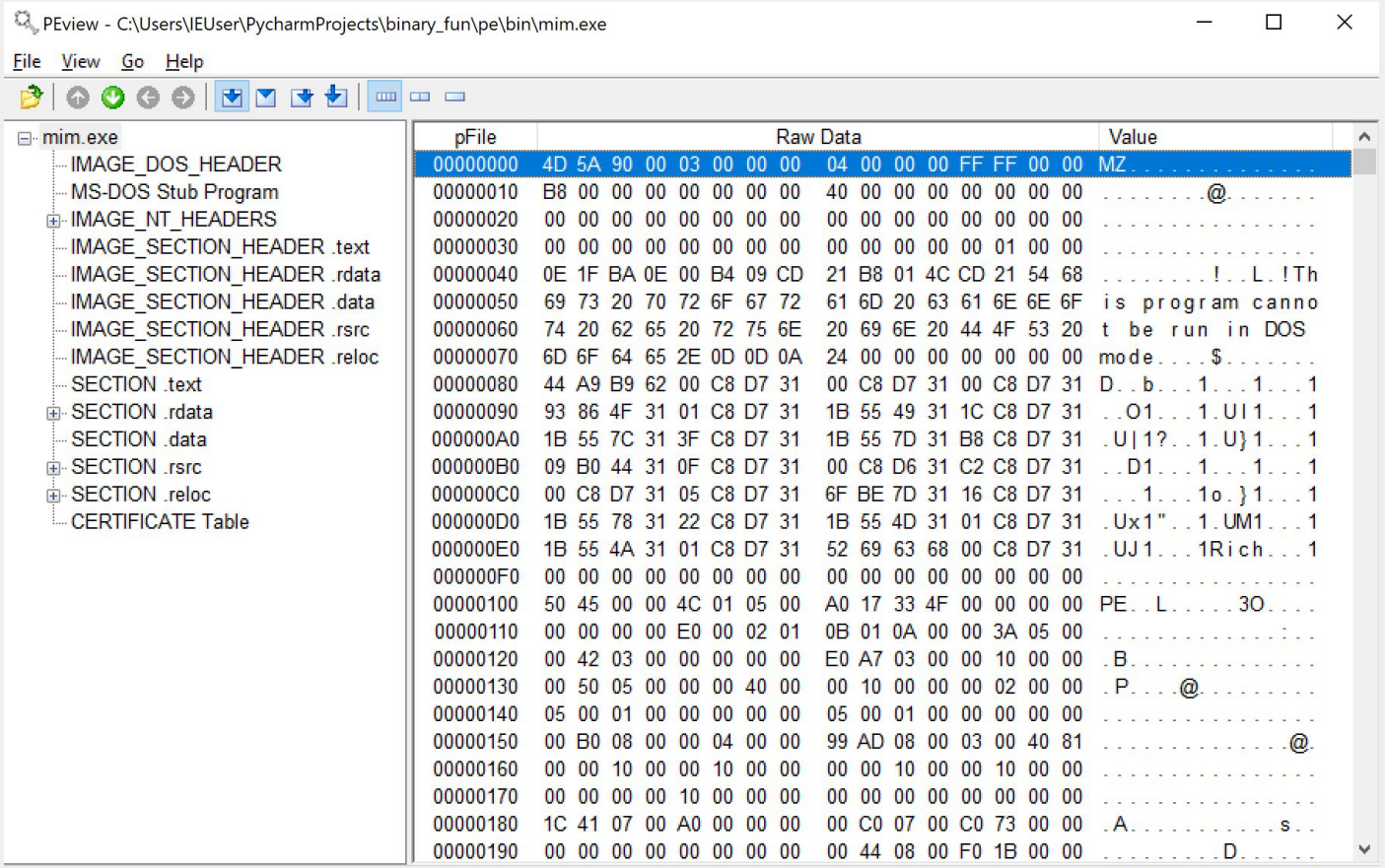
Binary Files → PE Binary Format

```
class ParsePE32(object):

    def __init__(self):
        self.bin_contents = None

    def execute(self, bin_path):
        self.bin_contents = FileUtils.read_file(bin_path)
        if self.bin_contents:
            self.parse_pe32()

    def parse_pe32(self):
        mz_header_string = self.get_mz_header_contents()
        logging.info('mz_header: {}'.format(mz_header_string))

    def get_mz_header_contents(self):
        image_dos_header_offs = 0
        e_magic_offs = 0
        # format string: <: big endian. h: 2 bytes (1 word)
        mz_header_bytes = struct.unpack_from('>h', self.bin_contents, image_dos_header_offs + e_magic_offs)
        mz_header_string = self._long_to_string(mz_header_bytes[0], 2)
        return mz_header_string

    @staticmethod
    def _long_to_string(value, length):
        # https://gph.is/1KjihQe (https://stackoverflow.com/questions/3673428/convert-int-to-ascii-and-back-in-python)
        return ''.join(chr((value >> 8 * (length - byte - 1)) & 0xFF) for byte in range(length))


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    if len(sys.argv) == 2:
        parser = ParsePE32()
        parser.execute(sys.argv[1])
    else:
        logging.error('Usage: {} <file_path>'.format(sys.argv[0]))
```

Binary Files → PE Binary Format

Obviously, there are tools that already allow you to get this information!

For example, **PEView**[1]

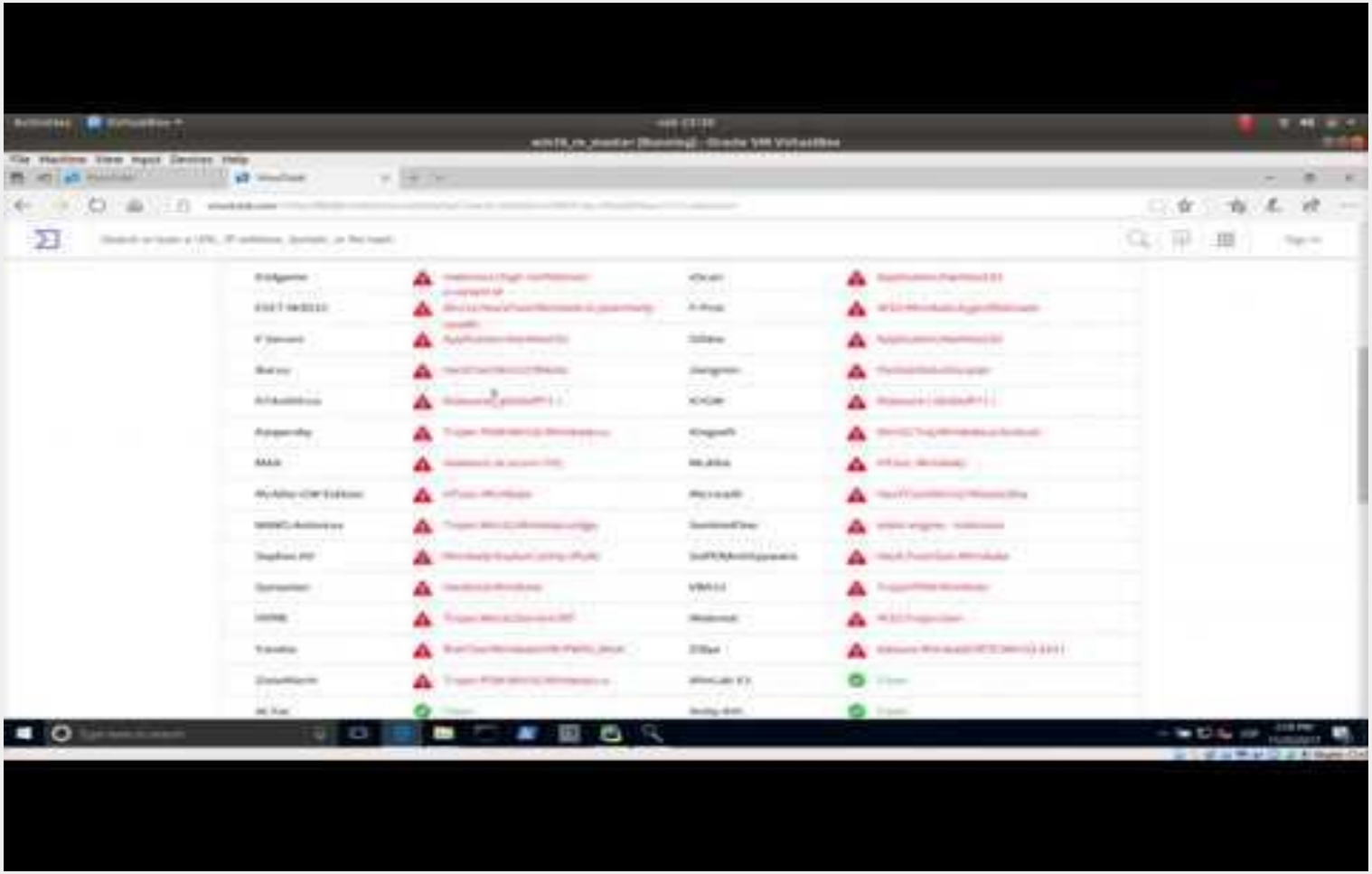


[1] http://wjradburn.com/software/

Only by knowing this we can already play a little bit...

tamper_bin.py

Let's get some fun! Let's modify a binary to lower detection rate with super dumb modification!



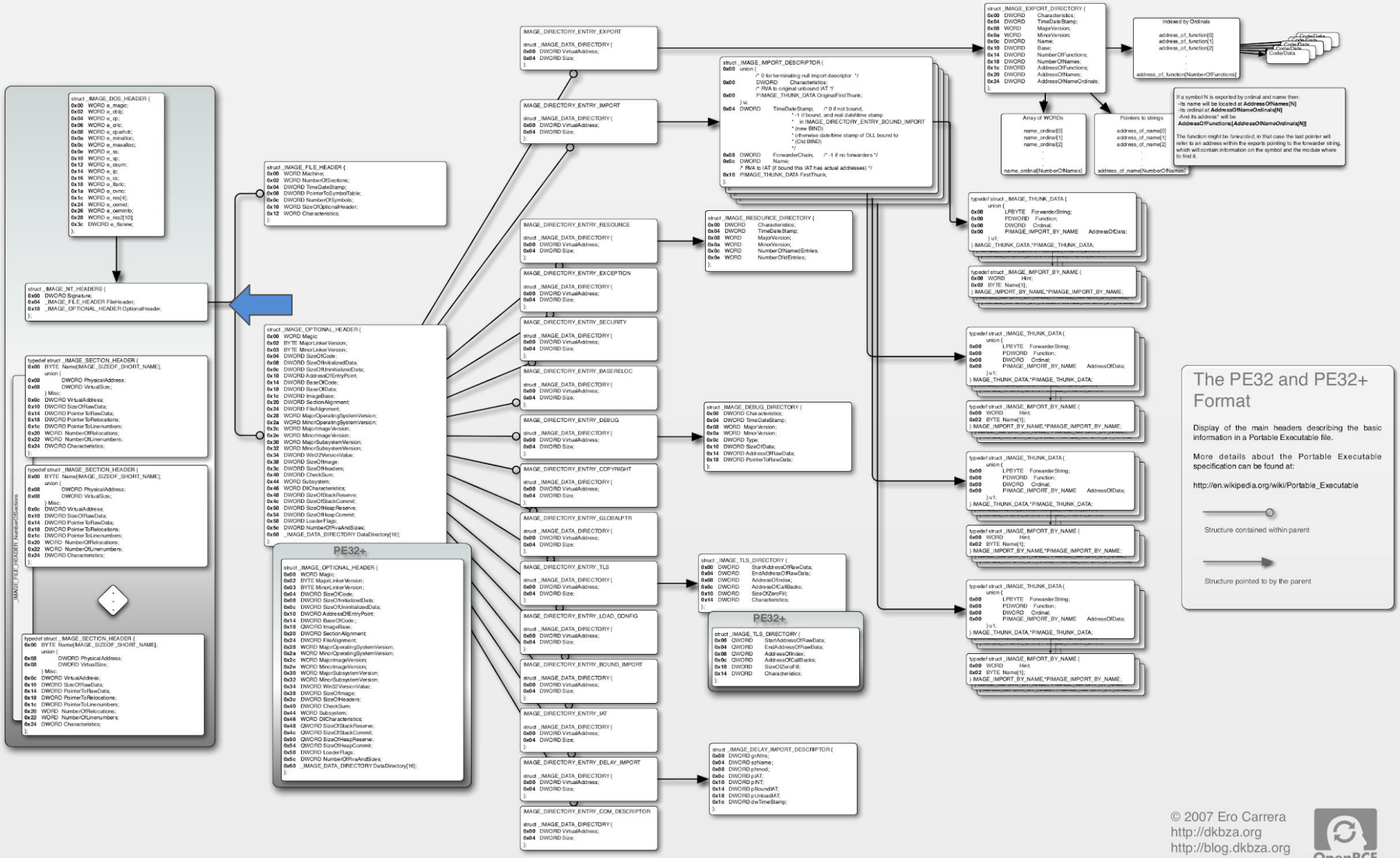
~**10** AVs got fooled by that simple trick.
Unfortunate...

PE Header

IMAGE_NT_HEADERS

Binary Files → PE Binary Format → IMAGE_NT_HEADERS

Portable Executable Format Layout



The PE Header → Important and specific information about the binary structure

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD Signature; /* "PE"\0\0 */       /* 0x00 */
  IMAGE_FILE_HEADER FileHeader;         /* 0x04 */
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;      /* 0x18 */
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

- **Signature**

Specifies whether this file is a PE file or not.

- **File Header**

Specifies the ISA (and PE format), when was it linked (compiled), number of sections, etc

- **Optional Header**

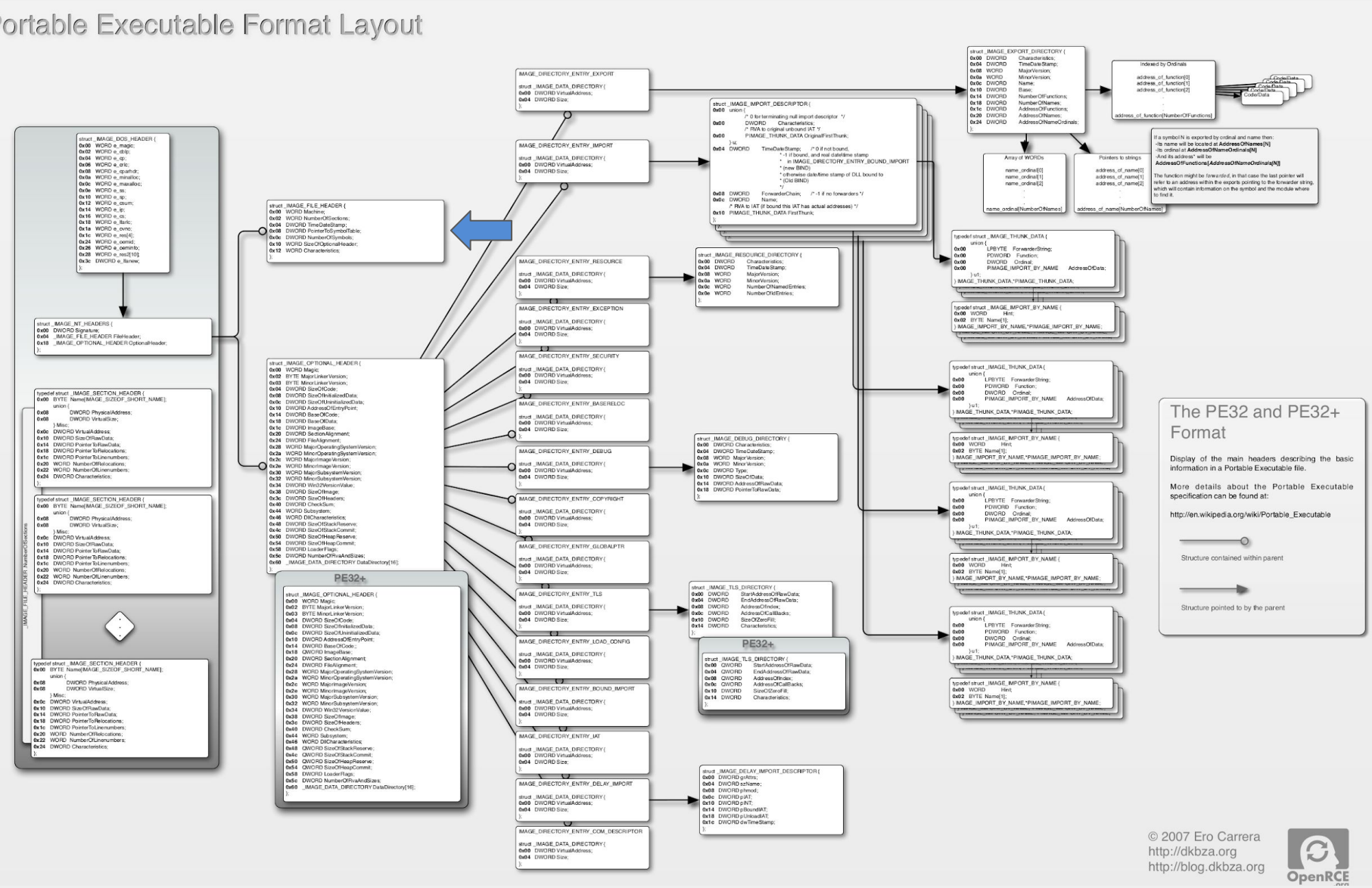
All the remaining information for handling the PE. “Most important” structure.

IMAGE_FILE_HEADER and IMAGE_OPTIONA_HEADER are both embedded in the parent IMAGE_NT_HEADERS structure (they are not pointers).

File Header

IMAGE_NT_HEADERS.FILE_HEADER

Binary Files → PE Binary Format → IMAGE_NT_HEADERS.FileHeader



The File Header

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

- **Machine**: Supported architectures
- **TimeDateStamp**: When was the file created
- **Characteristics**: Other information such as whether the binary is a DLL or not.

FileHeader.Machine

It specifies what type of architecture does this binary support.

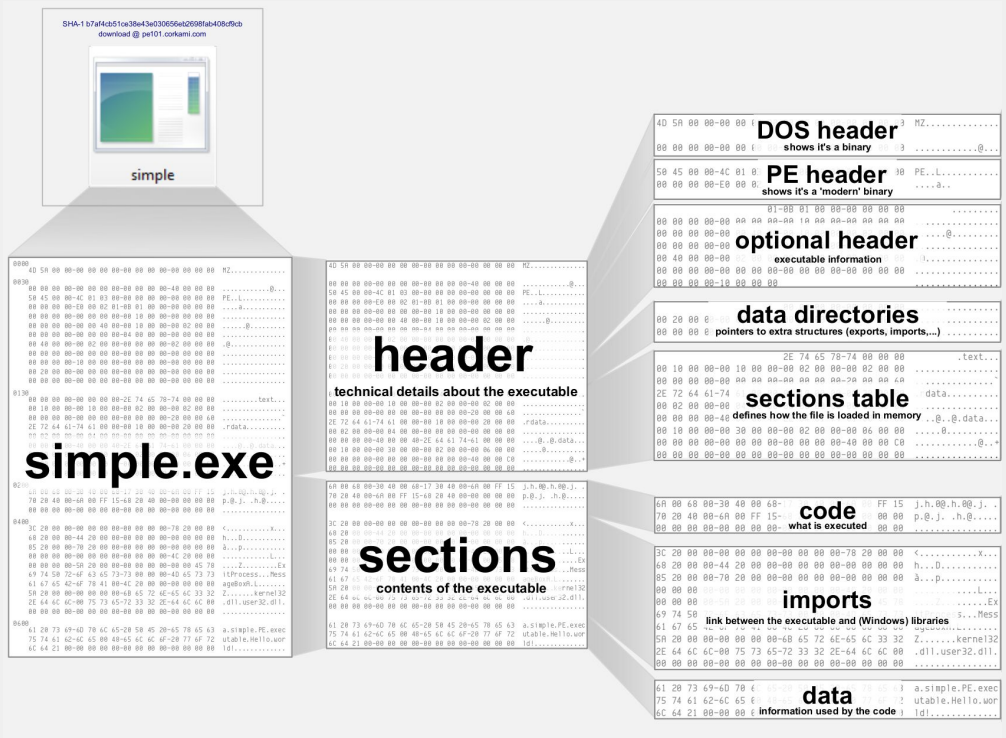
PE binaries support more than 23 types of architectures! x86, x86_64, x64, MIPS, ARM, Matsushita AM33... ¯_(ツ)_/¯

- PE32/x86/i386 → 0x14C
- PE32+/x86_64/AMD64 → 0x8664
- IA64/Itanium Processor → 0x200
- ARM Thumb-2 little Endian → 0x1C4

FileHeader.NumberOfSections

A section is what actually contains the content of the binary. Code and data.

This field specifies how many sections does the binary contain (just after the OptionalHeader structure) so they can be correctly mapped into memory.



FileHeader.TimeDateStamp

It specifies when was this binary **linked**. Usually, linking process happens at compile time, so it can be used to know when the binary was created. Unix timestamp since epoch (midnight January 1st 1970).

This field can be used for **Attribution**. The **pseudo-science** of guessing what *threat actor* created what malware.

1. Today company A gets compromised.
2. After forensic analysis, implant is found in compromised asset
3. Implant is found to ping back to Kim Jong-un home's NAS
4. Implant TimeDateStamp is null.
5. Implant is analyzed and is found to share binary blobs with previous found malware.
6. Previous found malware TimeDateStamp dates 10 years ago

Conclusion: We know Kim Jong-un is using malware functionalities as old as 10 years → Which might mean they are clumsy as hell.

Let’s build a tool to know when a PE was compiled!

2.pe_header.py, 3.compile_time.py

FileHeader.Characteristics

```
#define IMAGE_FILE_RELOCS_STRIPPED      0x0001 /* No relocation info */
#define IMAGE_FILE_EXECUTABLE_IMAGE     0x0002
#define IMAGE_FILE_LINE_NUMS_STRIPPED   0x0004
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED  0x0008
#define IMAGE_FILE_AGGRESIVE_WS_TRIM    0x0010
#define IMAGE_FILE_LARGE_ADDRESS_AWARE  0x0020
#define IMAGE_FILE_16BIT_MACHINE        0x0040
#define IMAGE_FILE_BYTES_REVERSED_LO    0x0080
#define IMAGE_FILE_32BIT_MACHINE        0x0100
#define IMAGE_FILE_DEBUG_STRIPPED       0x0200
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP      0x0400
#define IMAGE_FILE_NET_RUN_FROM_SWAP    0x0800
#define IMAGE_FILE_SYSTEM               0x1000
#define IMAGE_FILE_DLL                  0x2000
#define IMAGE_FILE_UP_SYSTEM_ONLY       0x4000
#define IMAGE_FILE_BYTES_REVERSED_HI    0x8000
```

- IMAGE_FILE_DEBUG_STRIPPED → Debugging information removed from PE?
- IMAGE_FILE_DLL → Is this PE a DLL?

Let’s build a tool to know PE:

- Architecture
- Type
- Number of Sections

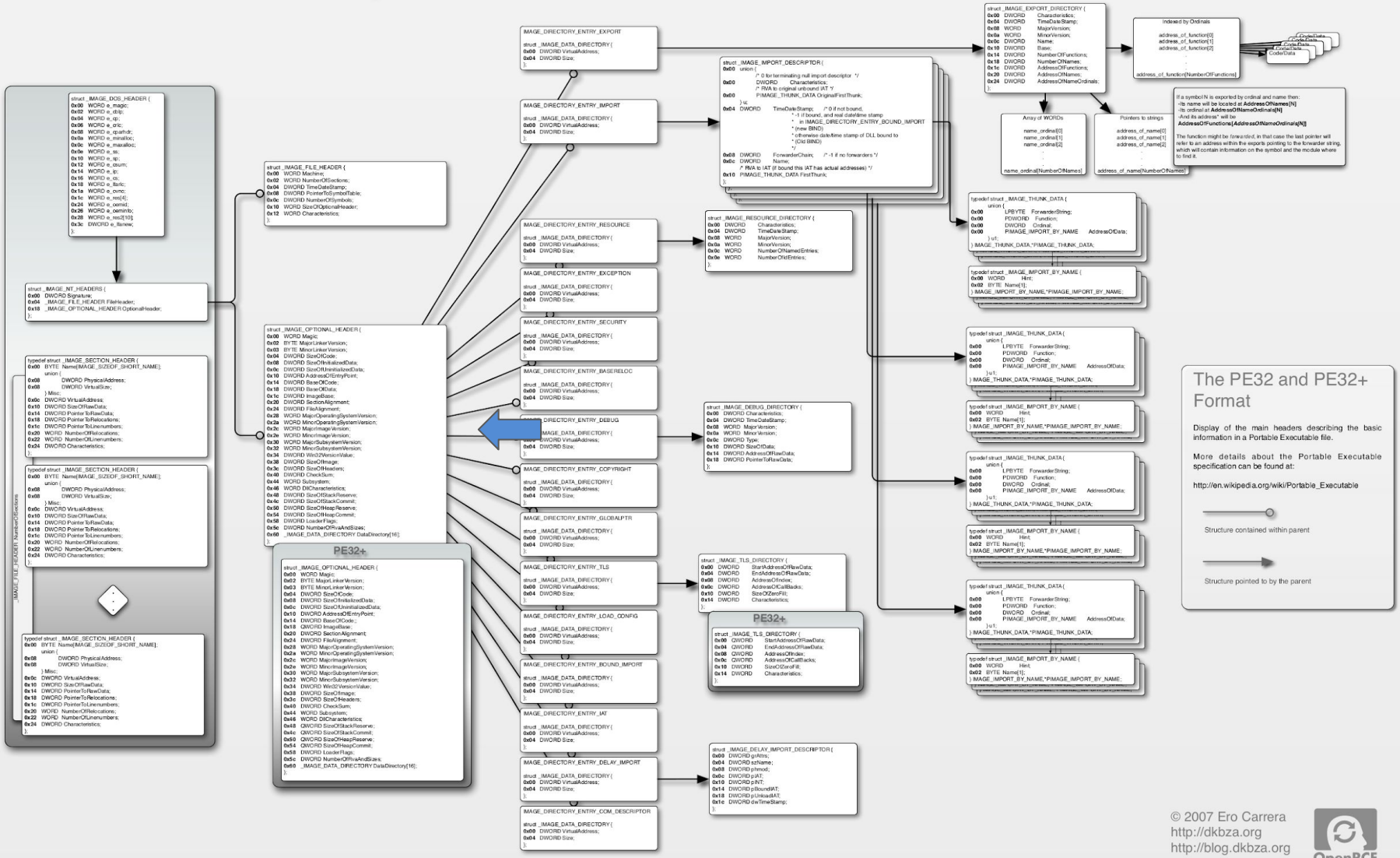
4.pe_arch.py, 5.pe_type.py, 6.number_of_sections.py

Optional Header

IMAGE_NT_HEADERS.OPTIONAL
_HEADER

Binary Files → PE Binary Format → IMAGE_NT_HEADERS.OptionalHeader

Portable Executable Format Layout



Binary Files → PE Binary Format → IMAGE_NT_HEADERS.OptionalHeader

The Optional Header

```
typedef struct _IMAGE_OPTIONAL_HEADER {

  /* Standard fields */

  WORD  Magic; /* 0x10b or 0x107 */     /* 0x00 */
  BYTE  MajorLinkerVersion;
  BYTE  MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;           /* 0x10 */
  DWORD BaseOfCode;
  DWORD BaseOfData;

  /* NT additional fields */

  DWORD ImageBase;
  DWORD SectionAlignment;              /* 0x20 */
  DWORD FileAlignment;
  WORD  MajorOperatingSystemVersion;
  WORD  MinorOperatingSystemVersion;
  WORD  MajorImageVersion;
  WORD  MinorImageVersion;
  WORD  MajorSubsystemVersion;         /* 0x30 */
  WORD  MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;                      /* 0x40 */
  WORD  Subsystem;
  WORD  DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;             /* 0x50 */
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
  /* 0xE0 */
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
  WORD  Magic; /* 0x20b */
  BYTE MajorLinkerVersion;
  BYTE MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;
  DWORD BaseOfCode;
  ULONGLONG ImageBase;
  DWORD SectionAlignment;
  DWORD FileAlignment;
  WORD MajorOperatingSystemVersion;
  WORD MinorOperatingSystemVersion;
  WORD MajorImageVersion;
  WORD MinorImageVersion;
  WORD MajorSubsystemVersion;
  WORD MinorSubsystemVersion;
  DWORD Win32VersionValue;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;
  WORD Subsystem;
  WORD DllCharacteristics;
  ULONGLONG SizeOfStackReserve;
  ULONGLONG SizeOfStackCommit;
  ULONGLONG SizeOfHeapReserve;
  ULONGLONG SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

- **Magic**

Determines if the executable is PE32 or PE32+. Depending on this field the file will be parsed one way or another.

- 0x10B → PE32
- 0x20B → PE32+

Until this point, all code we wrote worked both for PE32 and PE32+.

- **SizeOfImage**

The size (in bytes) of the image, including all headers, as the image is loaded in memory.

- **AddressOfEntryPoint**

Specifies the ***offset to add to the address of the image base → RVA***

Image base address + offset point to the first byte of the first instruction that will be executed once the *loader* maps the binary to memory.

The entry point function is optional for DLLs. When no entry point is present, this member is zero.

- **ImageBase**

The **preferred** address of the first byte of the PE when loaded into memory; always multiple of 64K.

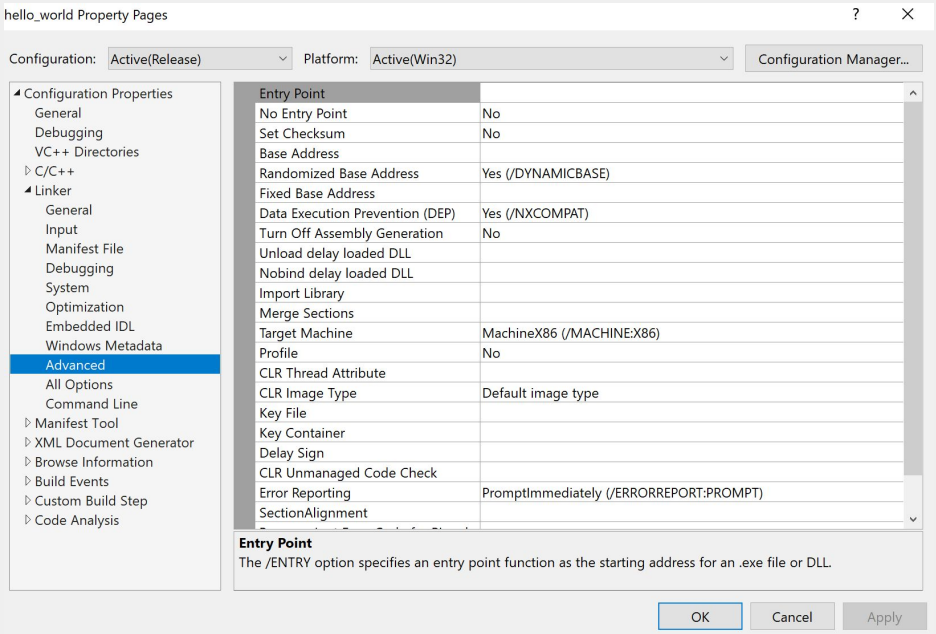
- The default for PE32 DLLs is 0x10000000, for PE32+ is 0x180000000.
- The default for PE32 executables is 0x00400000, for PE32+ is 0x140000000.

Offset for **PE32 is 0x1C, for PE32+ 0x18.** First difference between PE and PE+.

- **DllCharacteristics**

Information about the PE such as ASLR, NX or SEH support.

- IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE (ASLR) → 0x0040
- IMAGE_DLLCHARACTERISTICS_NX_COMPAT (DEP) → 0x0100
- IMAGE_DLLCHARACTERISTICS_ NO_SEH (SafeSEH) → 0x0400



- **DataDirectory**

An array of structures defining VirtualAddress and Size for multiple tables that are important in a PE (such as the Import Address Table, Export Address Table, etc).

This structure allows the loader to get the addresses for those tables.

The VirtualAddress of each entry will point to a given PE sections.

Let’s build a tool to know PE:

- Type
- ImageBase
- Address of Entry Point
- Security Features

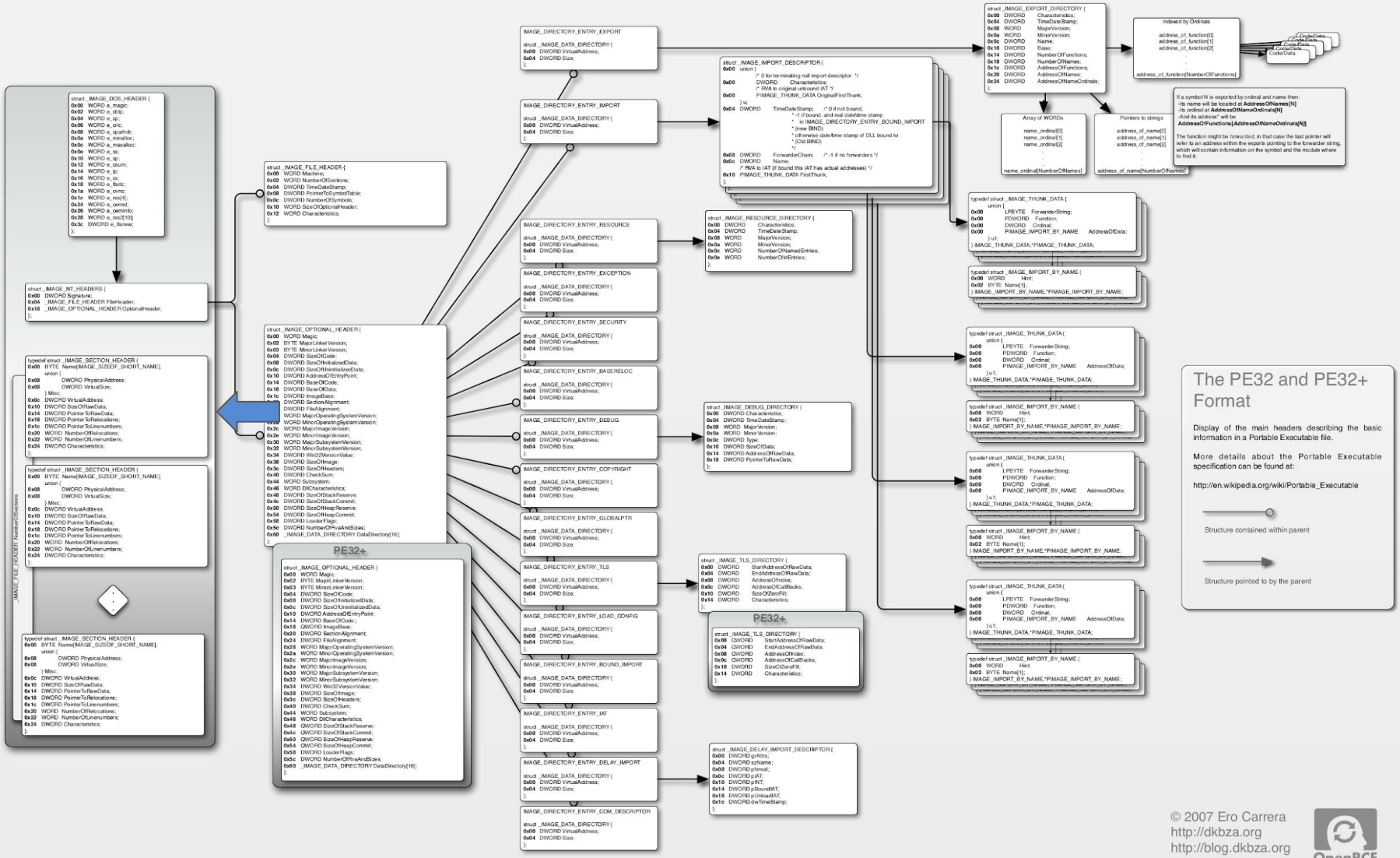
7.pe_optional_header_parsing.py

Section Headers

IMAGE_SECTION_HEADER

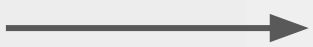
Binary Files → PE Binary Format → IMAGE_SECTION_HEADERS

Portable Executable Format Layout



Section Headers

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```



This field will be interpreted one way or another depending on the file type:

- VirtualSize → Executable
- PhysicalAddress → Object file

The section headers define the sections. **The sections are containers with certain permissions for binary data.**

There are as many sections as FileHeader.NumberOfSections

In the PE, the section headers are just right after the Optional Header.

In the PE, the section headers are listed in order by their RVA (from lower to higher)

- **Name**

This is the name of the section.

- **VirtualSize**

This is the size of the section contents[1]. It's the size the section will have once mapped into memory.

- **VirtualAddress**

The RVA (Relative Virtual Address) where the loader will map the section contents. The absolute virtual address is computed as ImageBase address + VirtualAddress.

When compiling with Microsoft tools, the first section defaults to an RVA of 0x1000.

[1] Before rounding up to the FileAlignment size.

- **SizeOfRawData**

Size of the section contents on disk. This can differ from VirtualSize because of OptionalHeaders.FileAlignment[1]. Some padding can be added between sections in disk.

- **PointerToRawData**

File offset pointing to the contents of the section.

- **Characteristics**

A bunch of properties for the section such as access permissions (rwx).

[1] The default value is 0x200 bytes, probably to ensure that sections always start at the beginning of a disk sector (which are also 0x200 bytes in length).

Common Sections in a PE

- **.text:** Where all the instructions are located. This section cannot be written into.
- **.data:** Initialized global or static variables.
- **.rdata:** Read-only data on such as strings and constants.
- **.bss:** Uninitialized global or static variables.
- **.rsrc:** Resource information such as icons or images.
- **.reloc:** Relocation information. Fixups when things are not loaded to preferred address
- **.pdata:** Debugging and exception handling information.
- **.idata & .edata:** Import/Export directory. Info about imported/exported functions.

Let’s build a tool to know PE:

- Number of Sections
- Section Header Information such as:
  - Name
  - Virtual Size
  - Virtual Address
  - Size of Raw Data
  - Pointer to Raw Data
  - Characteristics

8.pe_image_section_headers.py

Imports

What are the imports?

Imports are the functions implemented by external libraries (DLLs) that a given executable uses.

Using the functions implemented in external libraries allows an executable to remain as small as possible.

The code that is executed by your binary is not always part of the executable itself.

Basically, an import is defined by the DLL name from which the code is executed, an identifier of the function (name or number) and its address inside the DLL.

When are imports used? → **Statically Linked** vs **Dynamically Linked**

Imports are used when you executable is **dynamically** linked.

If your executable is **statically** linked → At compile time, the linker will look for all the functions that your code use, it will resolve those references and embed the code for those functions inside the binary.

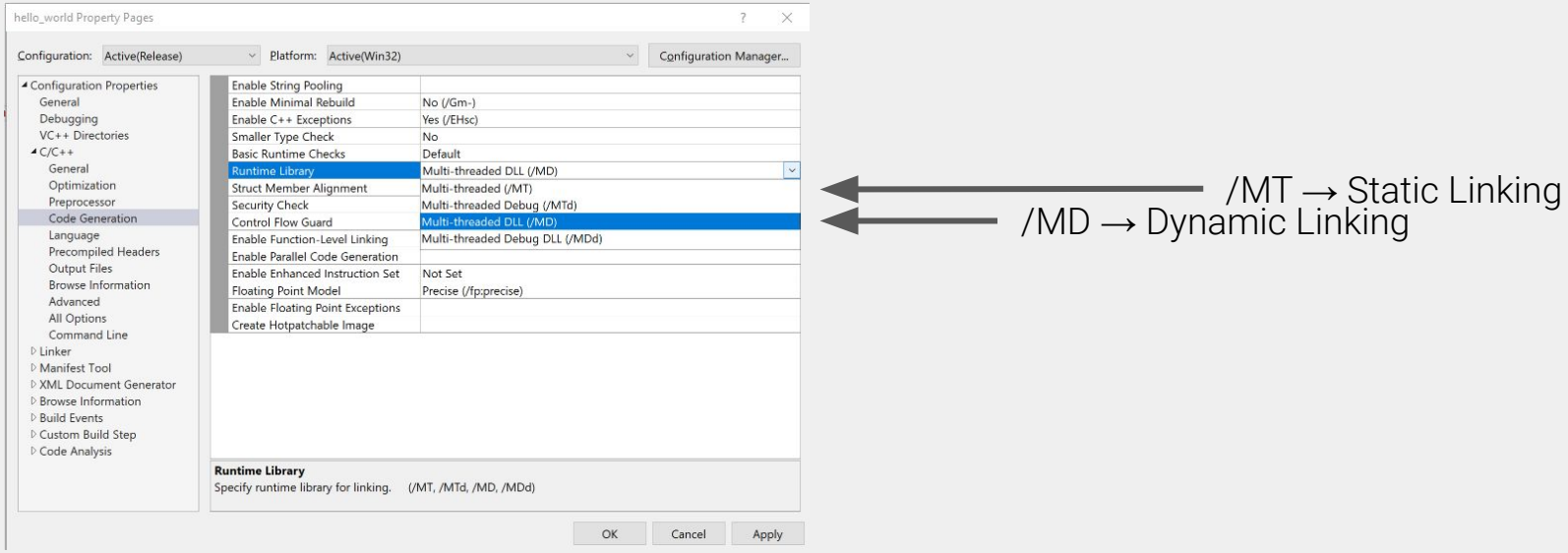
Binaries Statically Linked:

- Easy to deploy → No dependencies needed to be resolved at run time
- Harder to RE → All code is embedded with no readable symbols
- Big binaries (smallest binary can become 10 times bigger)
- Update trouble → Vuln in a DLL you need to re-deploy whole executable

Binaries Dynamically Linked:

- The opposite from above ;)

How to dynamically or statically link?



/MT → Static Linking
/MD → Dynamic Linking

hello_world.exe	11/25/2017 12:42 PM	Application	11 KB
hello_world_statically_linked.exe	12/4/2017 12:34 AM	Application	112 KB

Messed up thing of the day!

```
→ 2.binary_file_formats rabin2 -i hello_world.exe
[Imports]
ordinal=001 plt=0x140002080 bind=NONE type=FUNC name=VCRUNTIME140.dll_memset
ordinal=002 plt=0x140002088 bind=NONE type=FUNC name=VCRUNTIME140.dll___C_specific_handler
ordinal=001 plt=0x140002160 bind=NONE type=FUNC name=api-ms-win-crt-stdio-l1-1-0.dll___stdio_common_vfprintf
ordinal=002 plt=0x140002168 bind=NONE type=FUNC name=api-ms-win-crt-stdio-l1-1-0.dll___acrt_iob_func
ordinal=003 plt=0x140002170 bind=NONE type=FUNC name=api-ms-win-crt-stdio-l1-1-0.dll___p__commode
ordinal=004 plt=0x140002178 bind=NONE type=FUNC name=api-ms-win-crt-stdio-l1-1-0.dll__set_fmode
ordinal=001 plt=0x1400020c8 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__cexit
ordinal=002 plt=0x1400020d0 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__crt_atexit
ordinal=003 plt=0x1400020d8 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll_terminate
ordinal=004 plt=0x1400020e0 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__seh_filter_exe
ordinal=005 plt=0x1400020e8 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__set_app_type
ordinal=006 plt=0x1400020f0 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__initialize_onexit_table
ordinal=007 plt=0x1400020f8 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__register_thread_local_exe_atexit_callback
ordinal=008 plt=0x140002100 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll___p___argv
ordinal=009 plt=0x140002108 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll___p___argc
ordinal=010 plt=0x140002110 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__c_exit
ordinal=011 plt=0x140002118 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__exit
ordinal=012 plt=0x140002120 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll_exit
ordinal=013 plt=0x140002128 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__initterm_e
ordinal=014 plt=0x140002130 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__initterm
ordinal=015 plt=0x140002138 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__get_initial_narrow_environment
ordinal=016 plt=0x140002140 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__initialize_narrow_environment
ordinal=017 plt=0x140002148 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__configure_narrow_argv
ordinal=018 plt=0x140002150 bind=NONE type=FUNC name=api-ms-win-crt-runtime-l1-1-0.dll__register_onexit_function
ordinal=001 plt=0x1400020b8 bind=NONE type=FUNC name=api-ms-win-crt-math-l1-1-0.dll___setusermatherr
ordinal=001 plt=0x1400020a8 bind=NONE type=FUNC name=api-ms-win-crt-locale-l1-1-0.dll__configthreadlocale
ordinal=001 plt=0x140002098 bind=NONE type=FUNC name=api-ms-win-crt-heap-l1-1-0.dll__set_new_mode
ordinal=001 plt=0x140002000 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentProcessId
ordinal=002 plt=0x140002008 bind=NONE type=FUNC name=KERNEL32.dll_RtlLookupFunctionEntry
ordinal=003 plt=0x140002010 bind=NONE type=FUNC name=KERNEL32.dll_RtlVirtualUnwind
ordinal=004 plt=0x140002018 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleHandleW
ordinal=005 plt=0x140002020 bind=NONE type=FUNC name=KERNEL32.dll_IsDebuggerPresent
ordinal=006 plt=0x140002028 bind=NONE type=FUNC name=KERNEL32.dll_InitializeSListHead
ordinal=007 plt=0x140002030 bind=NONE type=FUNC name=KERNEL32.dll_GetSystemTimeAsFileTime
ordinal=008 plt=0x140002038 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentThreadId
ordinal=009 plt=0x140002040 bind=NONE type=FUNC name=KERNEL32.dll_RtlCaptureContext
ordinal=010 plt=0x140002048 bind=NONE type=FUNC name=KERNEL32.dll_QueryPerformanceCounter
ordinal=011 plt=0x140002050 bind=NONE type=FUNC name=KERNEL32.dll_IsProcessorFeaturePresent
ordinal=012 plt=0x140002058 bind=NONE type=FUNC name=KERNEL32.dll_TerminateProcess
ordinal=013 plt=0x140002060 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentProcess
ordinal=014 plt=0x140002068 bind=NONE type=FUNC name=KERNEL32.dll_SetUnhandledExceptionFilter
ordinal=015 plt=0x140002070 bind=NONE type=FUNC name=KERNEL32.dll_UnhandledExceptionFilter

42 imports
```

```
→ 2.binary_file_formats rabin2 -i hello_world_statically_linked.exe
[Imports]
ordinal=001 plt=0x140011000 bind=NONE type=FUNC name=KERNEL32.dll_RtlCaptureContext
ordinal=002 plt=0x140011008 bind=NONE type=FUNC name=KERNEL32.dll_RtlLookupFunctionEntry
ordinal=003 plt=0x140011010 bind=NONE type=FUNC name=KERNEL32.dll_RtlVirtualUnwind
ordinal=004 plt=0x140011018 bind=NONE type=FUNC name=KERNEL32.dll_UnhandledExceptionFilter
ordinal=005 plt=0x140011020 bind=NONE type=FUNC name=KERNEL32.dll_SetUnhandledExceptionFilter
ordinal=006 plt=0x140011028 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentProcess
ordinal=007 plt=0x140011030 bind=NONE type=FUNC name=KERNEL32.dll_TerminateProcess
ordinal=008 plt=0x140011038 bind=NONE type=FUNC name=KERNEL32.dll_IsProcessorFeaturePresent
ordinal=009 plt=0x140011040 bind=NONE type=FUNC name=KERNEL32.dll_QueryPerformanceCounter
ordinal=010 plt=0x140011048 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentProcessId
ordinal=011 plt=0x140011050 bind=NONE type=FUNC name=KERNEL32.dll_GetCurrentThreadId
ordinal=012 plt=0x140011058 bind=NONE type=FUNC name=KERNEL32.dll_GetSystemTimeAsFileTime
ordinal=013 plt=0x140011060 bind=NONE type=FUNC name=KERNEL32.dll_InitializeSListHead
ordinal=014 plt=0x140011068 bind=NONE type=FUNC name=KERNEL32.dll_IsDebuggerPresent
ordinal=015 plt=0x140011070 bind=NONE type=FUNC name=KERNEL32.dll_GetStartupInfoW
ordinal=016 plt=0x140011078 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleHandleW
ordinal=017 plt=0x140011080 bind=NONE type=FUNC name=KERNEL32.dll_RtlUnwindEx
ordinal=018 plt=0x140011088 bind=NONE type=FUNC name=KERNEL32.dll_GetLastError
ordinal=019 plt=0x140011090 bind=NONE type=FUNC name=KERNEL32.dll_SetLastError
ordinal=020 plt=0x140011098 bind=NONE type=FUNC name=KERNEL32.dll_EnterCriticalSection
ordinal=021 plt=0x1400110a0 bind=NONE type=FUNC name=KERNEL32.dll_LeaveCriticalSection
ordinal=022 plt=0x1400110a8 bind=NONE type=FUNC name=KERNEL32.dll_DeleteCriticalSection
ordinal=023 plt=0x1400110b0 bind=NONE type=FUNC name=KERNEL32.dll_InitializeCriticalSectionAndSpinCount
ordinal=024 plt=0x1400110b8 bind=NONE type=FUNC name=KERNEL32.dll_TlsAlloc
ordinal=025 plt=0x1400110c0 bind=NONE type=FUNC name=KERNEL32.dll_TlsGetValue
ordinal=026 plt=0x1400110c8 bind=NONE type=FUNC name=KERNEL32.dll_TlsSetValue
ordinal=027 plt=0x1400110d0 bind=NONE type=FUNC name=KERNEL32.dll_TlsFree
ordinal=028 plt=0x1400110d8 bind=NONE type=FUNC name=KERNEL32.dll_FreeLibrary
ordinal=029 plt=0x1400110e0 bind=NONE type=FUNC name=KERNEL32.dll_GetProcAddress
ordinal=030 plt=0x1400110e8 bind=NONE type=FUNC name=KERNEL32.dll_LoadLibraryExW
ordinal=031 plt=0x1400110f0 bind=NONE type=FUNC name=KERNEL32.dll_GetStdHandle
ordinal=032 plt=0x1400110f8 bind=NONE type=FUNC name=KERNEL32.dll_WriteFile
ordinal=033 plt=0x140011100 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleFileNameA
ordinal=034 plt=0x140011108 bind=NONE type=FUNC name=KERNEL32.dll_MultiByteToWideChar
ordinal=035 plt=0x140011110 bind=NONE type=FUNC name=KERNEL32.dll_WideCharToMultiByte
ordinal=036 plt=0x140011118 bind=NONE type=FUNC name=KERNEL32.dll_ExitProcess
ordinal=037 plt=0x140011120 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleHandleExW
ordinal=038 plt=0x140011128 bind=NONE type=FUNC name=KERNEL32.dll_GetCommandLineA
ordinal=039 plt=0x140011130 bind=NONE type=FUNC name=KERNEL32.dll_GetCommandLineW
ordinal=040 plt=0x140011138 bind=NONE type=FUNC name=KERNEL32.dll_GetACP
ordinal=041 plt=0x140011140 bind=NONE type=FUNC name=KERNEL32.dll_HeapFree
ordinal=042 plt=0x140011148 bind=NONE type=FUNC name=KERNEL32.dll_HeapAlloc
ordinal=043 plt=0x140011150 bind=NONE type=FUNC name=KERNEL32.dll_CompareStringW
ordinal=044 plt=0x140011158 bind=NONE type=FUNC name=KERNEL32.dll_LCMapStringW
ordinal=045 plt=0x140011160 bind=NONE type=FUNC name=KERNEL32.dll_GetFileType
ordinal=046 plt=0x140011168 bind=NONE type=FUNC name=KERNEL32.dll_FindClose
ordinal=047 plt=0x140011170 bind=NONE type=FUNC name=KERNEL32.dll_FindFirstFileExA
ordinal=048 plt=0x140011178 bind=NONE type=FUNC name=KERNEL32.dll_FindNextFileA
ordinal=049 plt=0x140011180 bind=NONE type=FUNC name=KERNEL32.dll_IsValidCodePage
ordinal=050 plt=0x140011188 bind=NONE type=FUNC name=KERNEL32.dll_GetOEMCP
ordinal=051 plt=0x140011190 bind=NONE type=FUNC name=KERNEL32.dll_GetCPInfo
ordinal=052 plt=0x140011198 bind=NONE type=FUNC name=KERNEL32.dll_GetEnvironmentStringsW
ordinal=053 plt=0x1400111a0 bind=NONE type=FUNC name=KERNEL32.dll_FreeEnvironmentStringsW
ordinal=054 plt=0x1400111a8 bind=NONE type=FUNC name=KERNEL32.dll_SetEnvironmentVariableA
ordinal=055 plt=0x1400111b0 bind=NONE type=FUNC name=KERNEL32.dll_SetStdHandle
ordinal=056 plt=0x1400111b8 bind=NONE type=FUNC name=KERNEL32.dll_GetStringTypeW
ordinal=057 plt=0x1400111c0 bind=NONE type=FUNC name=KERNEL32.dll_GetProcessHeap
ordinal=058 plt=0x1400111c8 bind=NONE type=FUNC name=KERNEL32.dll_FlushFileBuffers
ordinal=059 plt=0x1400111d0 bind=NONE type=FUNC name=KERNEL32.dll_GetConsoleCP
ordinal=060 plt=0x1400111d8 bind=NONE type=FUNC name=KERNEL32.dll_GetConsoleMode
ordinal=061 plt=0x1400111e0 bind=NONE type=FUNC name=KERNEL32.dll_HeapSize
ordinal=062 plt=0x1400111e8 bind=NONE type=FUNC name=KERNEL32.dll_HeapReAlloc
ordinal=063 plt=0x1400111f0 bind=NONE type=FUNC name=KERNEL32.dll_CloseHandle
ordinal=064 plt=0x1400111f8 bind=NONE type=FUNC name=KERNEL32.dll_SetFilePointerEx
ordinal=065 plt=0x140011200 bind=NONE type=FUNC name=KERNEL32.dll_CreateFileW
ordinal=066 plt=0x140011208 bind=NONE type=FUNC name=KERNEL32.dll_WriteConsoleW
ordinal=067 plt=0x140011210 bind=NONE type=FUNC name=KERNEL32.dll_RaiseException

67 imports
```

Messed up thing of the day!

Dynamically Linked:

- Contains all the C Runtime imports (e.g. printf function)
- 42 imports

Statically Linked:

- It contains imports!
- It contains **MORE** imports than dynamically linked!! (67)

- Imports are only shown for explicitly imported functions by your binary
- kernel32.dll will always be linked by design (even if it only imports GetProcAddress).
- printf → kernel32 WriteFile:
  - Given that when statically linking we do not have a reference to the C Runtime library, and its code is embedded into the PE, the binary now needs to dynamically link against other functions not specified in the binary itself when dynamically linking.

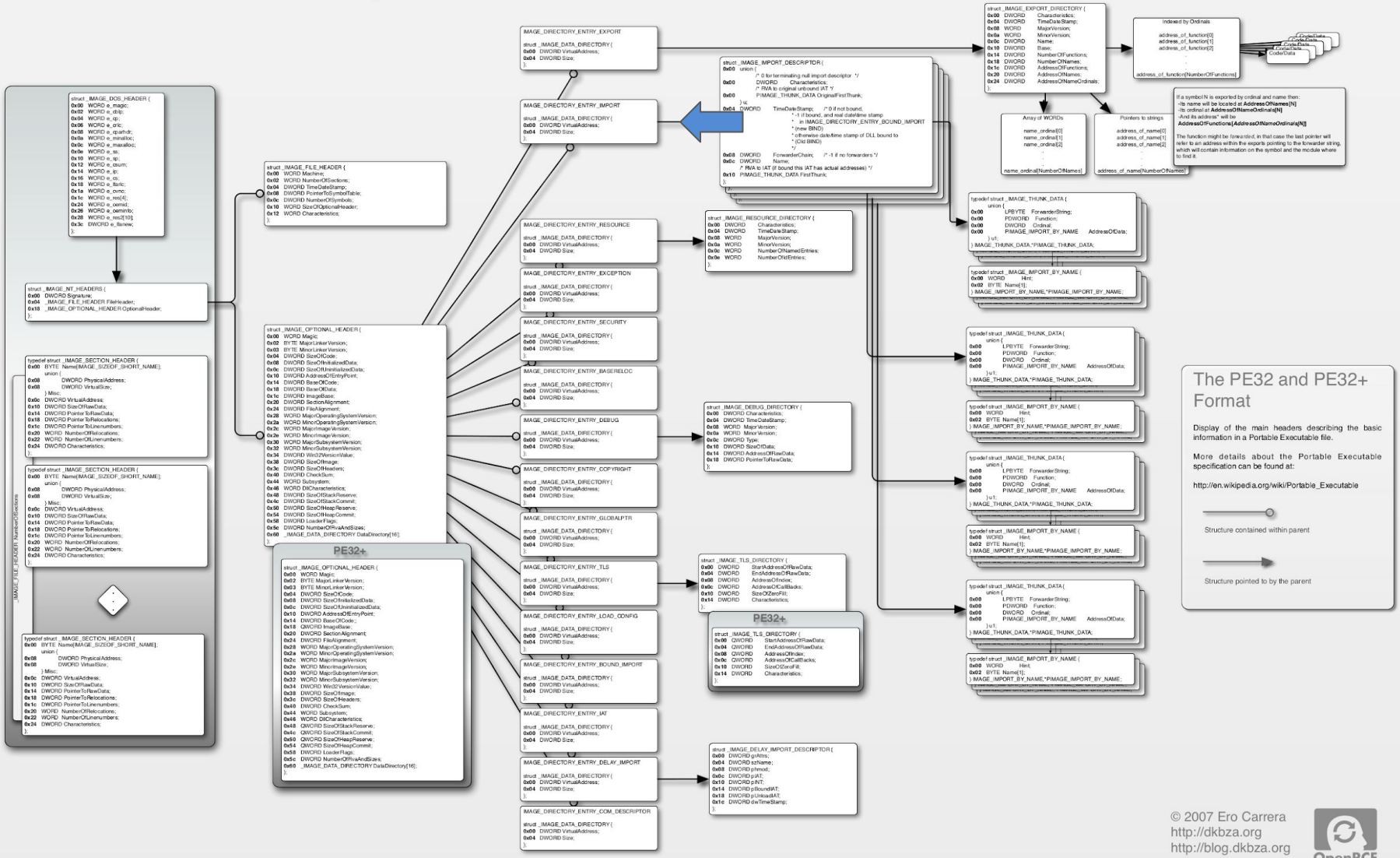
https://reverseengineering.stackexchange.com/questions/2070/can-i-statically-link-not-import-the-windows-system-dlls

Import Directory

IMAGE_DIRECTORY_ENTRY_IMPORT

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_IMPORT

Portable Executable Format Layout



Data Directory Reminder

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8
#define IMAGE_DIRECTORY_ENTRY_TLS             9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11
#define IMAGE_DIRECTORY_ENTRY_IAT             12
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14
```

The **OptionalHeader** contains an array of **DataDirectory** structs.

Here we discuss the **second entry** of that array.

Import Descriptor

Before explaining the import descriptor format, let's clarify **how to get to it**!

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

- **VirtualAddress**

The RVA to the structure once the binary has been loaded into memory!! (on execution)

Through the VirtualAddress (RVA) we can get the Import Descriptor information. But we need to **transform this RVA to a file offset.**

- **Size**

Gives the size in bytes in memory. Do not use this for file parsing! Use size of IMAGE_IMPORT_DESCRIPTOR struct.

RVA to File Offset

File Offset = RVA - SectionHeader.VirtualAddress + SectionHeader.PointerToRawData

Requirement:

- Know to what section the VirtualAddress belongs.

Notes:

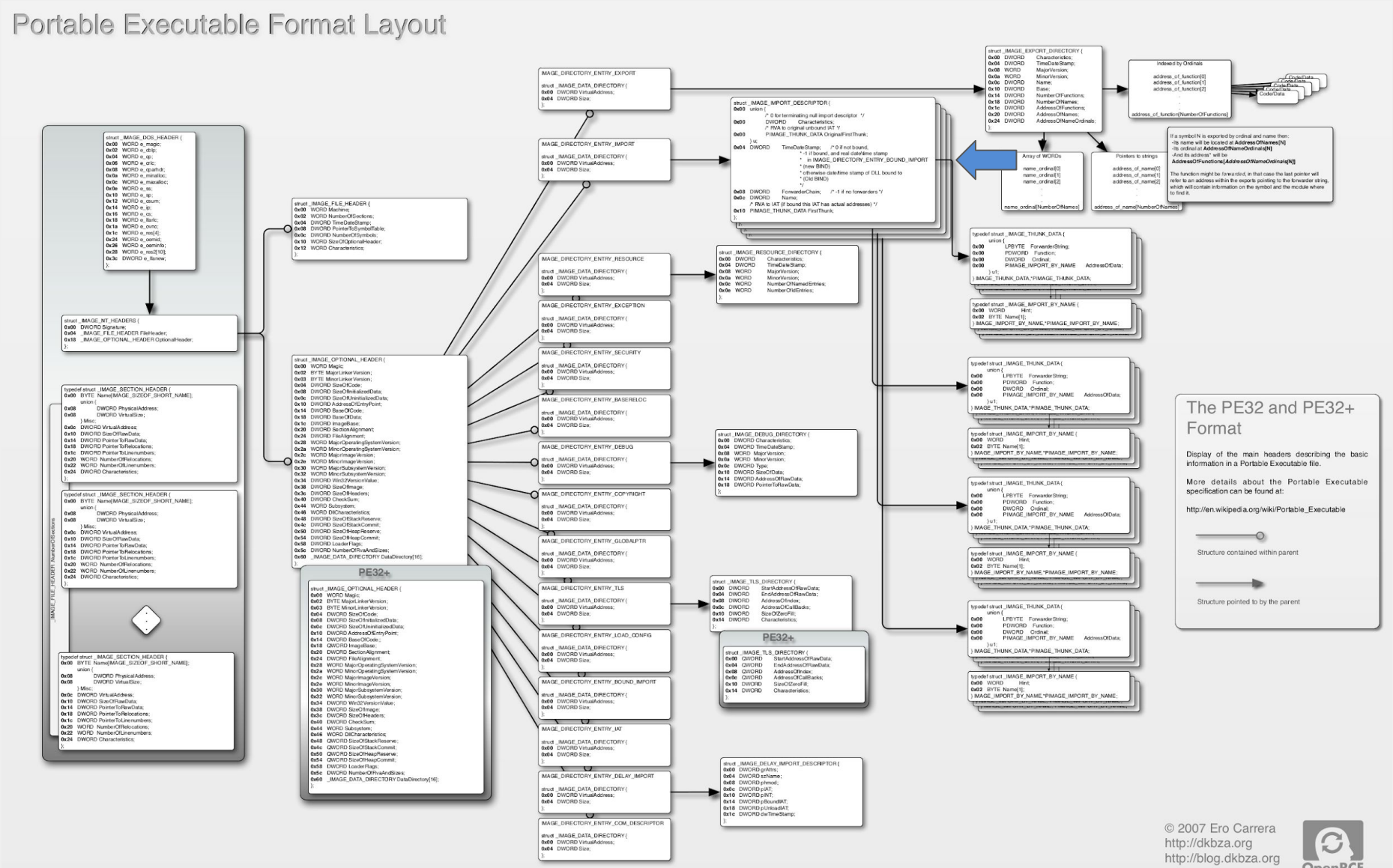
OptionalHeader.FileAlignment and **OptionalHeader.SectionAlignment** should be taken into consideration for this conversion, but we will ignore it. You can get more information here:

- https://github.com/erocarrera/pefile/blob/master/pefile.py#L5609
- https://github.com/erocarrera/pefile/blob/master/pefile.py#L5588
- https://github.com/erocarrera/pefile/blob/master/pefile.py#L1051

Import Descriptor

IMAGE_IMPORT_DESCRIPTOR

Binary Files → PE Binary Format → IMAGE_IMPORT_DESCRIPTOR



Import Descriptor

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
        union {
                DWORD   Characteristics; /* 0 for terminating null import descriptor  */
                DWORD   OriginalFirstThunk;     /* RVA to original unbound IAT */
        } DUMMYUNIONNAME;
        DWORD   TimeDateStamp;  /* 0 if not bound,
                                 * -1 if bound, and real date\time stamp
                                 *    in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
                                 * (new BIND)
                                 * otherwise date/time stamp of DLL bound to
                                 * (Old BIND)
                                 */
        DWORD   ForwarderChain; /* -1 if no forwarders */
        DWORD   Name;
        /* RVA to IAT (if bound this IAT has actual addresses) */
        DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

The Import Directory points to a linked list of these structures

There's one of these structures per each imported DLL

Import Descriptor

- **Name**

Name of the DLL that is imported.

- **OriginalFirstThunk**

RVA to the INT (Import Name Table). The INT is an array with the names of the functions that are imported for the given DLL.
Points to an **IMAGE_THUNK_DATA** structure.

- **FirstThunk**

RVA to the IAT (Import Address Table). The IAT is an array with the addresses of the functions that are imported for the given DLL.
Points to an **IMAGE_THUNK_DATA** structure.

Let’s build a tool to know PE:

- Import Descriptor Address
- Imported DLL Names

9.pe_import_descriptor_va.py, 10.pe_imported_dlls.py

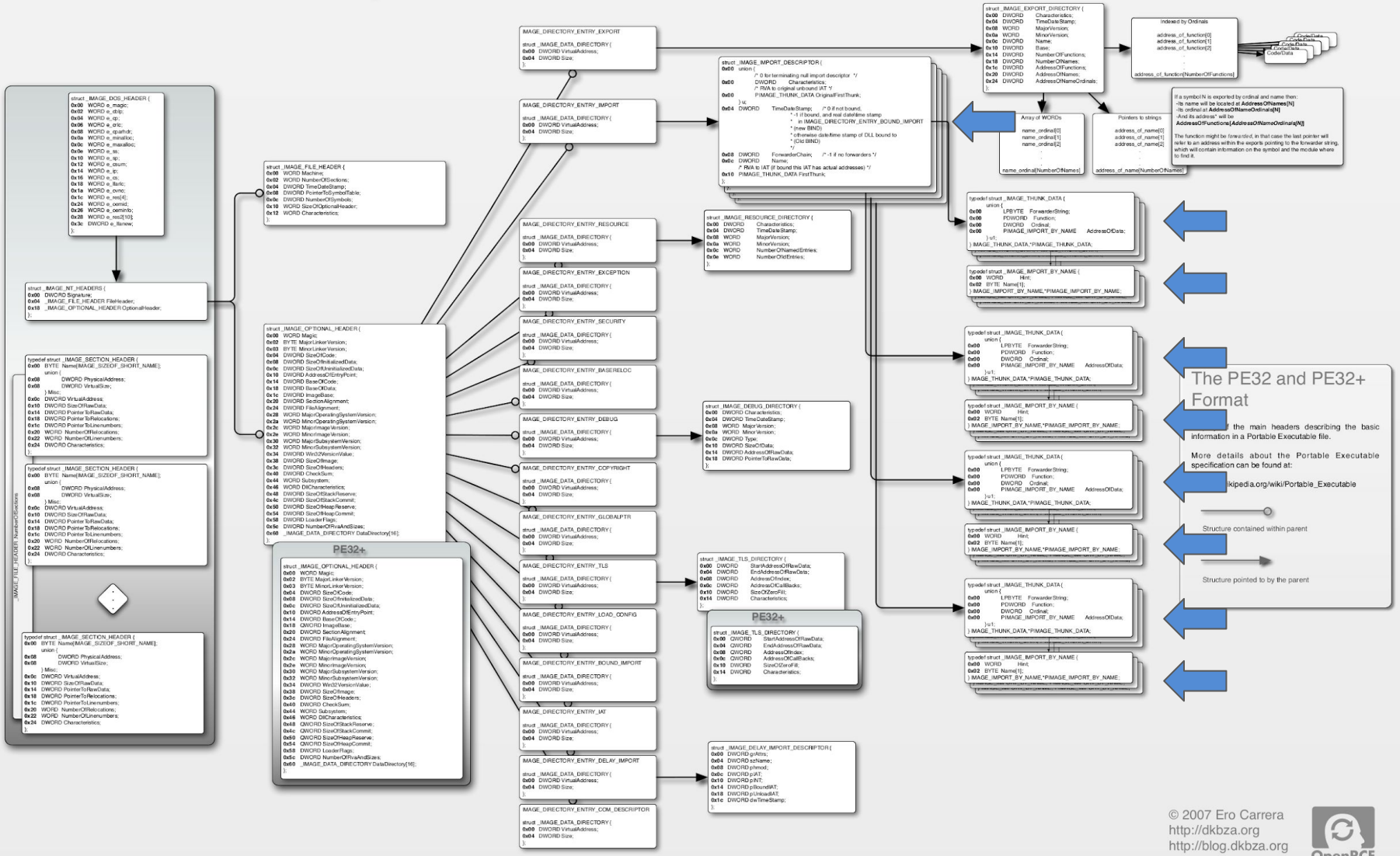
Let’s build a tool to know PE:

- Imported DLL Names for a PE32+

10.pe_imported_dlls.py (in 64/ directory)

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_IMPORT

Portable Executable Format Layout



Binary Files → PE Binary Format → IMAGE_THUNK_DATA

```
typedef struct _IMAGE_THUNK_DATA32 {
        union {
                DWORD ForwarderString;
                DWORD Function;
                DWORD Ordinal;
                DWORD AddressOfData;
        } u1;
} IMAGE_THUNK_DATA32,*PIMAGE_THUNK_DATA32;
```

```
/* Import thunk */
typedef struct _IMAGE_THUNK_DATA64 {
        union {
                ULONGLONG ForwarderString;
                ULONGLONG Function;
                ULONGLONG Ordinal;
                ULONGLONG AddressOfData;
        } u1;
} IMAGE_THUNK_DATA64,*PIMAGE_THUNK_DATA64;
```

Depending on who points to this structure, **OriginalFirstThunk or FirstThunk**, the structure/union will be interpreted as the **INT or IAT**, respectively.

- If pointed by the **OriginalFirstThunk (INT)**, the union is interpreted as the **AddressOfData** which is an address (RVA) that **points to an IMAGE_IMPORT_BY_NAME.**

- If pointed by the **FirstThunk (IAT)**, the union is interpreted as the **Function** which is an address (RVA) where the address of the import will eventually be set (load time).

```
/* Import name entry */
typedef struct _IMAGE_IMPORT_BY_NAME {
        WORD    Hint;
        BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

- **Hint**

Exported function number of the imported DLL. (e.g. this is the function number 10 of all the exported functions by this DLL).[1]

- **Name**

Name of the imported function.

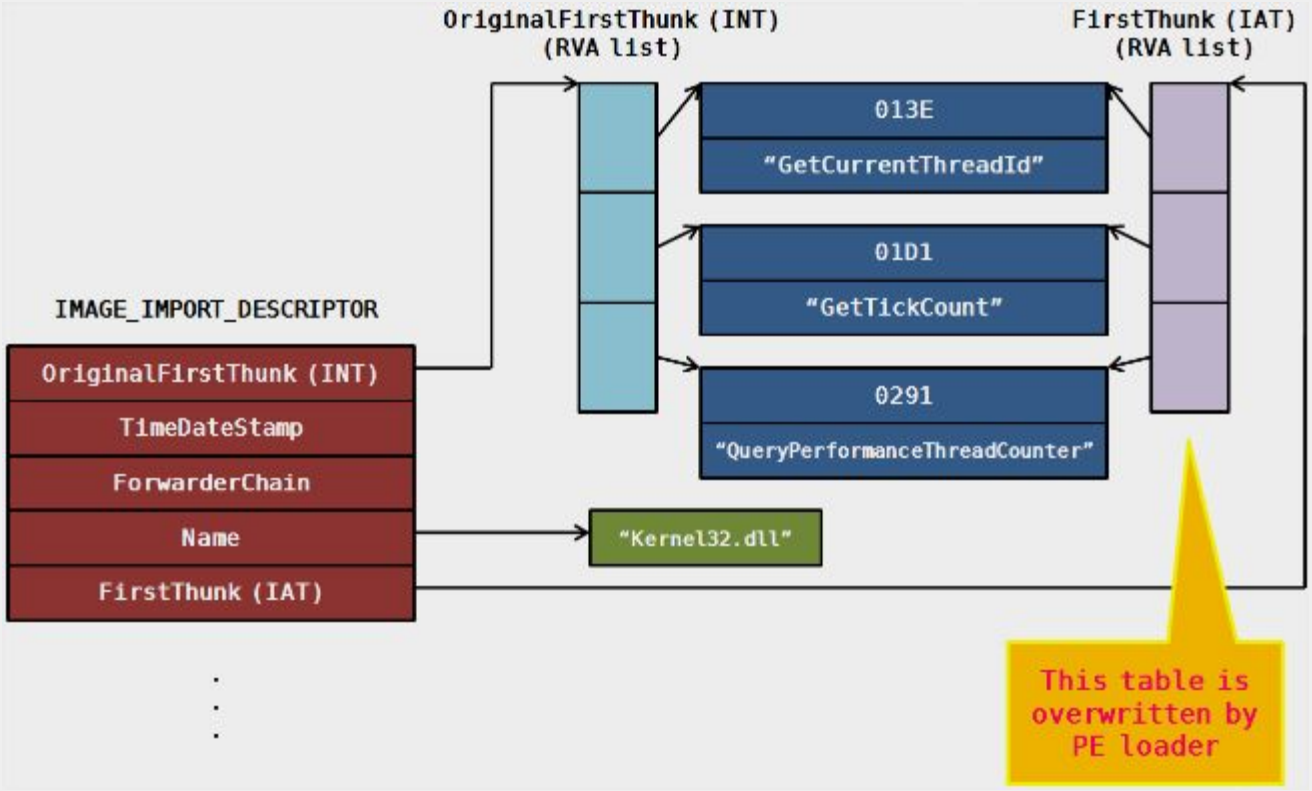
[1] This does not to be correct, is just a starting point to start looking for the imported function in the imported DLL

Let’s build a tool to know PE:

- Imported Functions from DLLs

11.pe_imported_functions_via_int.py

Import Name Table/OriginalFirstThunk vs Import Address Table/FirstThunk



At load time, the addresses for each imported function in the INT get resolved and the IAT is built in memory so it does not point to the hint/names table anymore.

Let’s build a tool to know PE:

- Imported Functions from DLLs

12.pe_imported_functions_via_iat.py

There's still some more stuff about imports that we will happily ignore, basically:

- **Bound Imports**

An optimization where the IAT is filled at linking time instead of load time.
The data directory entry that defines it is IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT

- **Delayed Imports**

Libraries are loaded once their functions are used (lazy loading).

There's a new Delay Load IAT that contains pointers to code snippets that will load the DLL. That code snippet will later overwrite Delay Load IAT entry to point to newly loaded function in the DLL so this loading process does not happen again.

The data directory entry that defines it is IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT.

Exports

What are the exports?

Exports are the functions that a given library exposes so other binaries can use them.

DLLs export many functions that can be used by other binaries so those binaries don't have to implement that functionality themselves.

Basically, an export is defined by:

- An address to the exported function.
- The name of the exported function.
- An identifier (ordinal) of the exported function.[1]

[1] A binary can import an exported function by id (ordinal) instead of name

What are the exports?

In source code, a function can be exported as follows:

```
__declspec(dllexport) void __cdecl Function1(void);
```

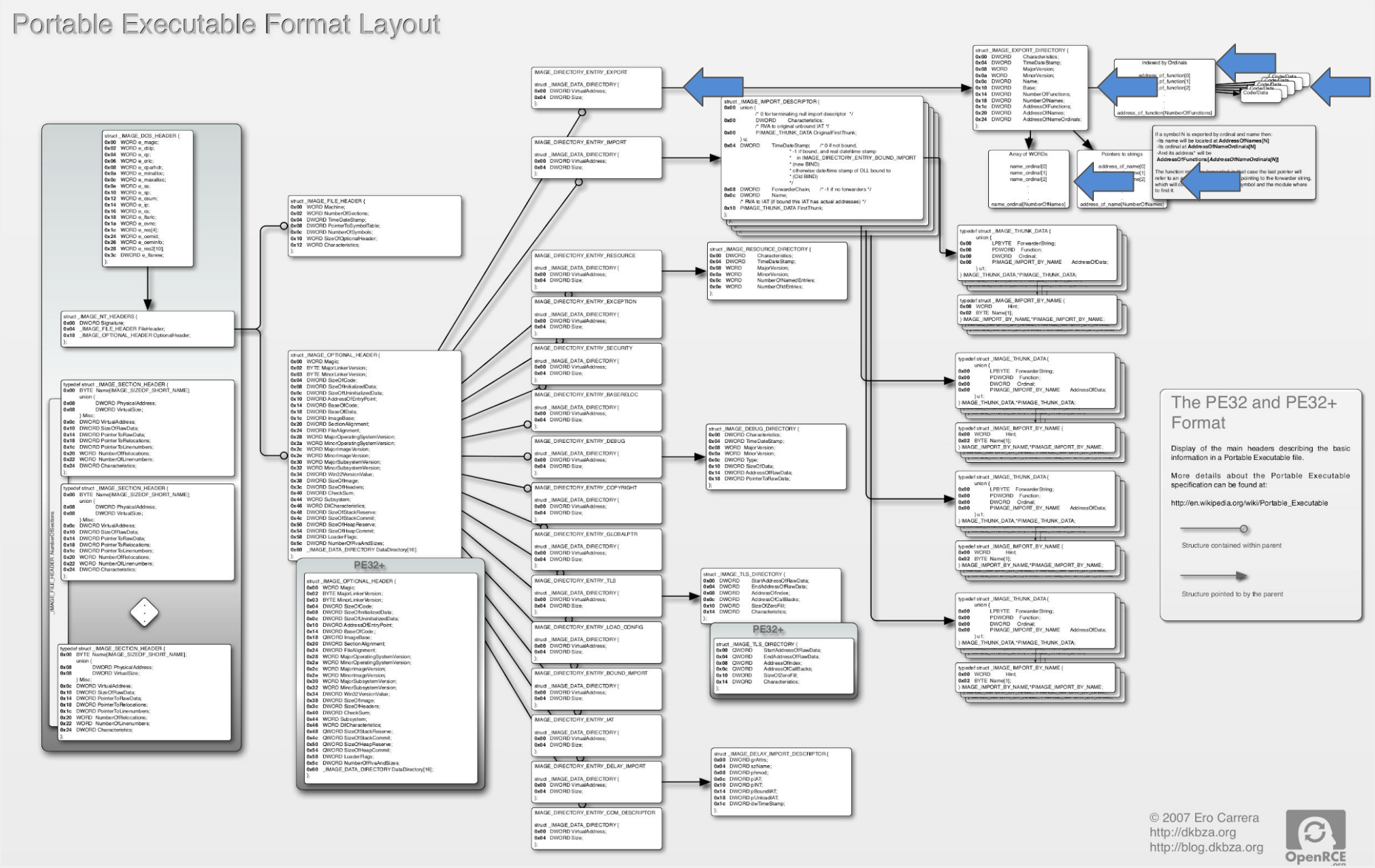
- __declspec(dllexport) → Identifies this function as exportable.
- __cdecl → Identifies the calling convention (more on this in following modules)

If you want to export functions by "ordinal"[1], a .def file needs to be defined specifying the ordinal for each exported function:

```
LIBRARY   BTREE
EXPORTS
  Insert   @1
  Delete   @2
  Member   @3
  Min   @4
```

[1] An ordinal is an id that uniquely identifies the exported function

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_EXPORT



Export Descriptor

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
        DWORD   Characteristics;
        DWORD   TimeDateStamp;
        WORD    MajorVersion;
        WORD    MinorVersion;
        DWORD   Name;
        DWORD   Base;
        DWORD   NumberOfFunctions;
        DWORD   NumberOfNames;
        DWORD   AddressOfFunctions;
        DWORD   AddressOfNames;
        DWORD   AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

- **TimeDateStamp**

A unix timestamp defining when the DLL was compiled.[1] It is also used to identify DLL version.

For example, bound imports (IAT built at compile time) use this TimeDateStamp on runtime to check if the correct DLL exists in the system where the binary is being executed in order to validate if the pre-built IAT is correct.

- **NumberOfFunctions**
- **NumberOfNames**

Number of functions exported

Number of functions exported **by name**

This number will differ if the DLL forces to export by ordinal.

[1] This field might not change just on DLL compilation but when there were changes that affected the exported functions that might break compatibility with previous versions of that DLL.

- **AddressOfFunctions**

It contains an address (RVA) that points to an array of addresses (RVAs) for the exported functions. It contains NumberOfFunction entries.

This array is known as the Address Export Table (EAT).

The index to access this array is computed using the AddressOfNameOrdinals array and the Base. More on this later.

- **Base**

Starting ordinal number for the exported functions. Usually set to 1. This field is used to access the AddressOfFunctions array.

To know the index you have to use to access AddressOfFunctions array you need to:

ArrayIndex = ExportedFunctionOrdinal - Base

- **AddressOfNames**

It contains an address (RVA) that points to an array of addresses (RVAs) for the names of the exported functions. It contains NumberOfNames entries.

This array of names is ordered alphabetically.

This array is known as the Address Export Table (ENT).

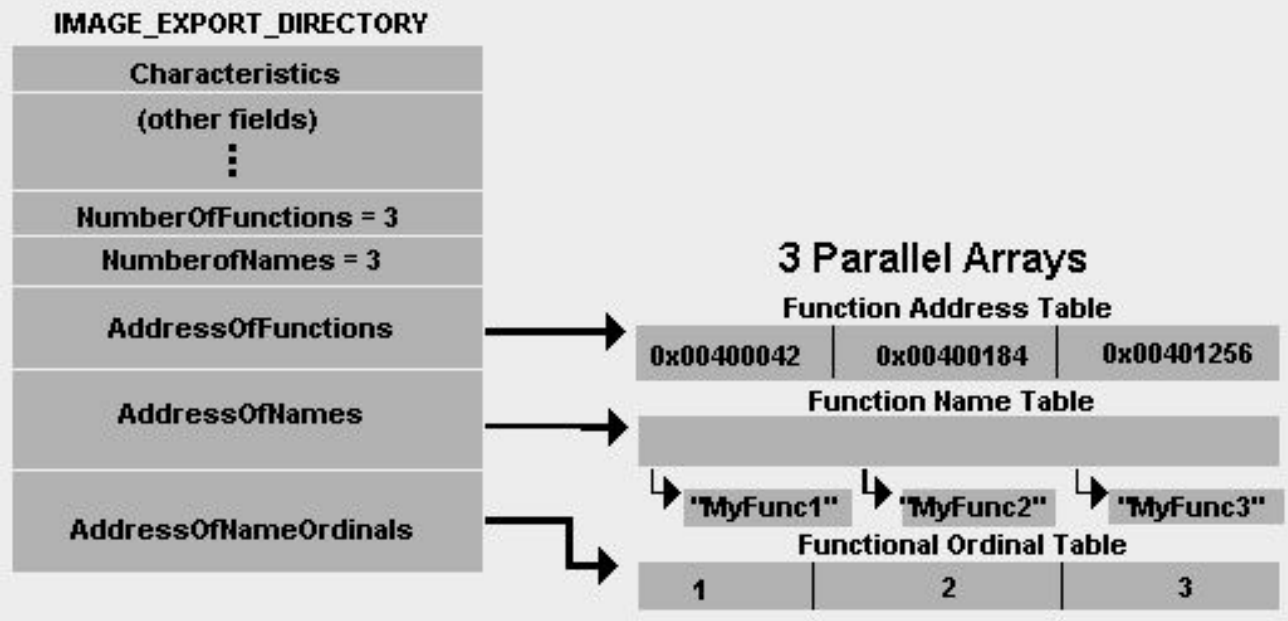
- **AddressOfNameOrdinals**

It contains an address (RVA) that points to an array of indexes that should be used to access the AddressOfFunctions array when functions are imported/exported by name.

More on this later.

Address Resolution Algorithm for Export

- **Export by Ordinal**

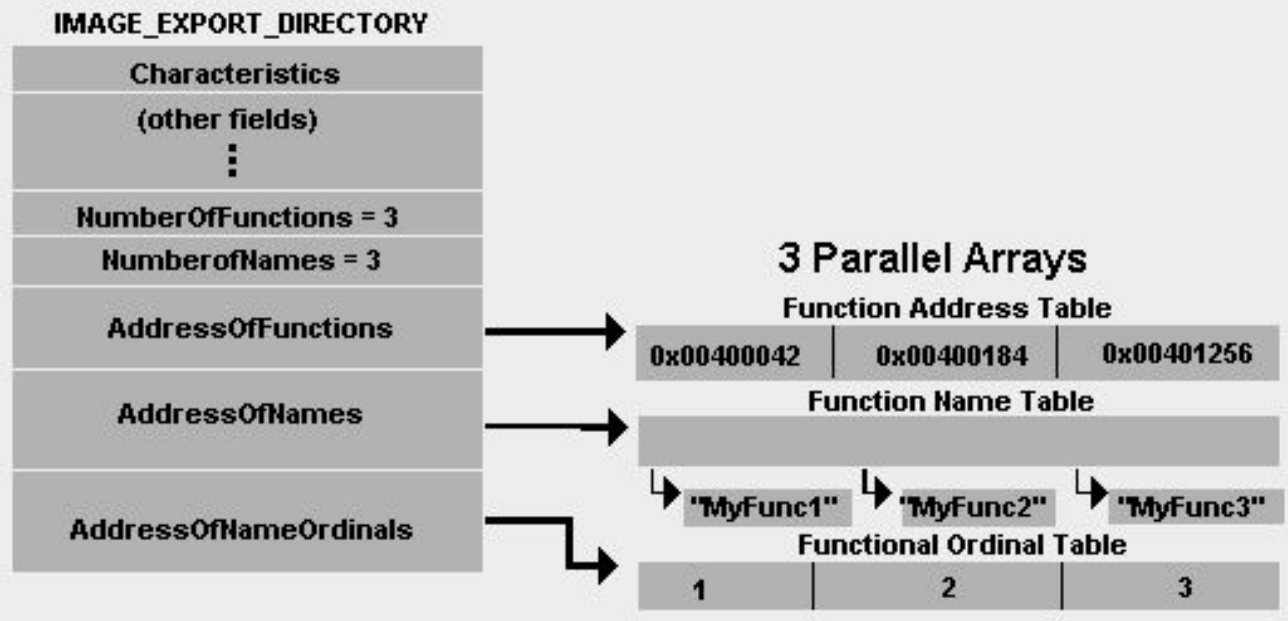


Assume export with ordinal 2 needs to be imported

1. AddressOfFunctions Index → index = 2 - Base
2. Export Address → AddressOfFunctions[index]

Address Resolution Algorithm for Export

- **Export by Name**



Assume export with name “MyFunc2” needs to be imported

1. Find index in AddressOfNames → Index = 1
2. Find ordinal in AddressOfNameOrdinals → AddressOfNameOrdinals[1] = 2
3. AddressOfFunctions Index → index = 2 - Base
4. Export Address → AddressOfFunctions[index]

Let’s build a tool to know PE:

- Exported Functions by name

14.pe_exported_functions_by_name.py

There's still some more stuff about imports that we will happily ignore, basically:

- **Forwarded Exports**

An exported function is not part of the DLL but part of another DLL (which will need to also be loaded).

Basically, the RVAs in the AddressOfFunctions array do not point to the .text section in the DLL, but to a string specifying DLLName.ExportedFunctionName.

Exported functions can be defined with the following line of code:

- #pragma comment(linker, "/export:<function to export>=<DLL filepath>.<exported function>")

This is used for:

- Backward compatibility reasons (one DLL is refactored and a function moved out)
- ~~Lazy~~ malware writers for DLL injection (Stuxnet, Symantec report, page 37).

Resources

What are the resources?

Resources are files that can be embedded into PEs.

A PE might embed a kernel driver that will be deployed when the PE gets executed.

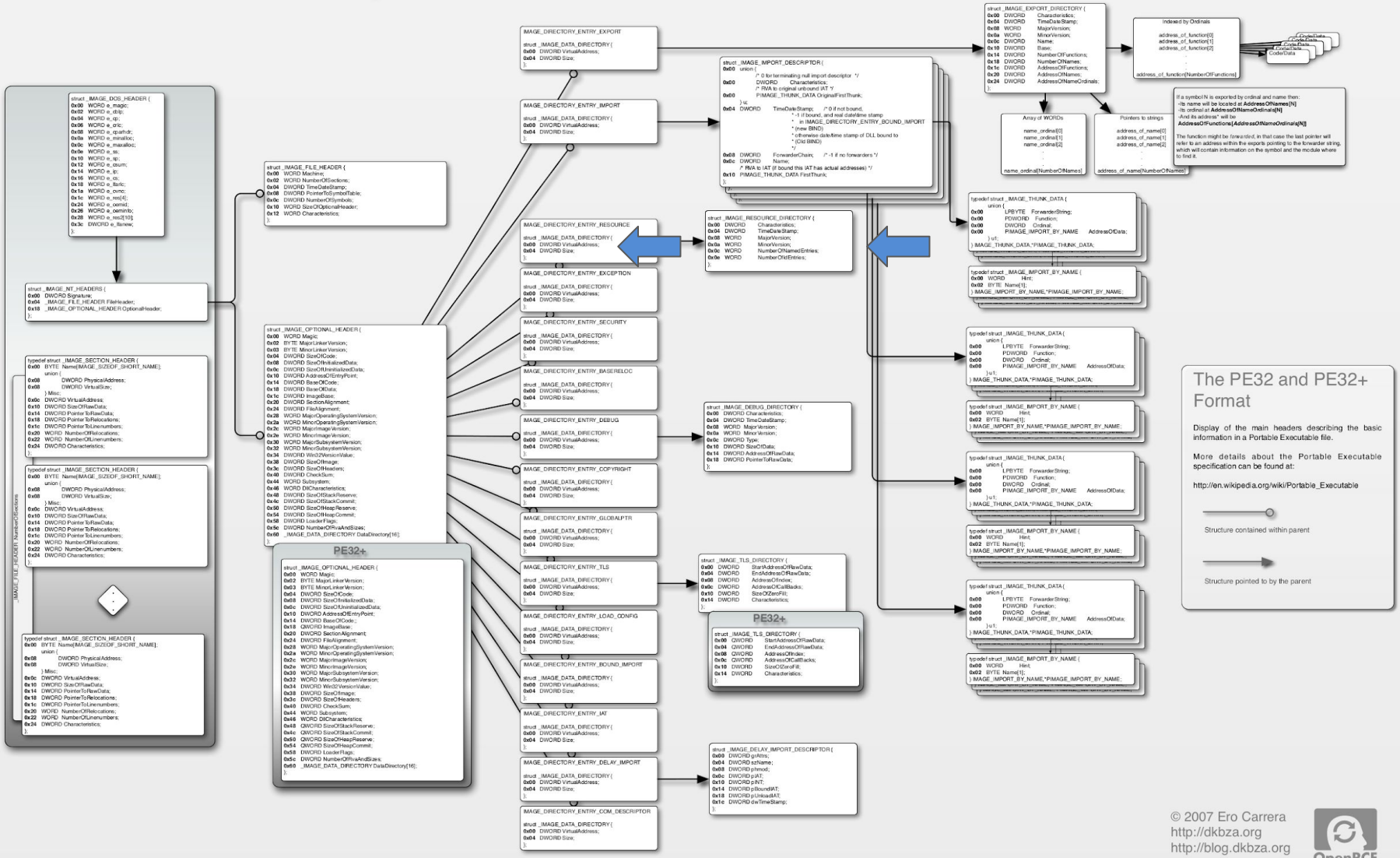
Other uses are for malware to embed “”hidden”” malicious code (exploits, stage 2 binaries, etc). Some examples that do that are Stuxnet or Locky.

We will have to deal with:

- Resource Directories
- Resource Directory Entries
- Resource Data Entries

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_RESOURCE

Portable Executable Format Layout



Resource Directory

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
        DWORD   Characteristics;
        DWORD   TimeDateStamp;
        WORD    MajorVersion;
        WORD    MinorVersion;
        WORD    NumberOfNamedEntries;
        WORD    NumberOfIdEntries;
        /*  IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]; */
} IMAGE_RESOURCE_DIRECTORY,*PIMAGE_RESOURCE_DIRECTORY;
```

Resources can have name or just ids.

- **NumberOfNamedEntries**: Specify how many resources with name exist.
- **NumberOfIdEntries**: Specify how many resources without a name exist.

NumberOfNamedEntries + NumberOfIdEntries IMAGE_RESOURCE_DIRECTORY_ENTRY structures follow the IMAGE_RESOURCE_DIRECTORY structure.

Resource Directory

Resources can have name or just ids.

- **NumberOfNamedEntries**: Specify how many resources with name exist.
- **NumberOfIdEntries**: Specify how many resources without a name exist.

After the Resource Directory there are (NumberOfNamedEntries + NumberOfIdEntries) Resource Directory Entries

The named entries go before the entries identified by IDs.

Resource Directory Entry

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
#ifdef BITFIELDS_BIGENDIAN
                        unsigned NameIsString:1;
                        unsigned NameOffset:31;
#else
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
#endif
                } DUMMYSTRUCTNAME;
                DWORD   Name;
#ifdef WORDS_BIGENDIAN
                WORD    __pad;
                WORD    Id;
#else
                WORD    Id;
                WORD    __pad;
#endif
        } DUMMYUNIONNAME;
        union {
                DWORD   OffsetToData;
                struct {
#ifdef BITFIELDS_BIGENDIAN
                        unsigned DataIsDirectory:1;
                        unsigned OffsetToDirectory:31;
#else
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
#endif
                } DUMMYSTRUCTNAME2;
        } DUMMYUNIONNAME2;
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Resource Directory Entry

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
#ifdef BITFIELDS_BIGENDIAN
                        unsigned NameIsString:1;
                        unsigned NameOffset:31;
#else
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
#endif
                } DUMMYSTRUCTNAME;
                DWORD   Name;
#ifdef WORDS_BIGENDIAN
                WORD    __pad;
                WORD    Id;
#else
                WORD    Id;
                WORD    __pad;
#endif
        } DUMMYUNIONNAME;
        union {
                DWORD   OffsetToData;
                struct {
#ifdef BITFIELDS_BIGENDIAN
                        unsigned DataIsDirectory:1;
                        unsigned OffsetToDirectory:31;
#else
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
#endif
                } DUMMYSTRUCTNAME2;
        } DUMMYUNIONNAME2;
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Let's simplify this mess

Resource Directory Entry

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
                };
                DWORD   Name;
                WORD    Id;
                WORD    __pad;
        };
        union {
                DWORD   OffsetToData;
                struct {
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
                };
        };
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Resource Directory Entry

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
                };
                DWORD   Name;
                WORD    Id;
                WORD    __pad;
        };
        union {
                DWORD   OffsetToData;
                struct {
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
                };
        };
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

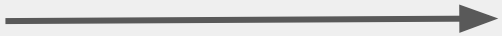
How the heck is this interpreted?

Resource Directory Entry

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
                };
                DWORD   Name;
                WORD    Id;
                WORD    __pad;
        };
        union {
                DWORD   OffsetToData;
                struct {
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
                };
        };
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

This is like if we had **only two** fields:

- **Name**
- **OffsetToData**



Depending on their MSB, they will mean one thing or another

Understanding the Resource Directory Entry

If: Name[31] = 1 → It's an identifier.
If: Name[31] = 0 → It's a RVA to a BSTR[1]. Only the least significant 31 bits count.

If: OffsetToData[31] = 1 → RVA to another IMAGE_RESOURCE_DIRECTORY
If: OffsetToData[31] = 0 → RVA to **IMAGE_RESOURCE_DATA_ENTRY**

The **IMAGE_RESOURCE_DATA_ENTRY** is the structure that **points to the raw data**

All RVAs are relative to the start of the resource DataDirectory
(usually the beginning of the .reloc section)

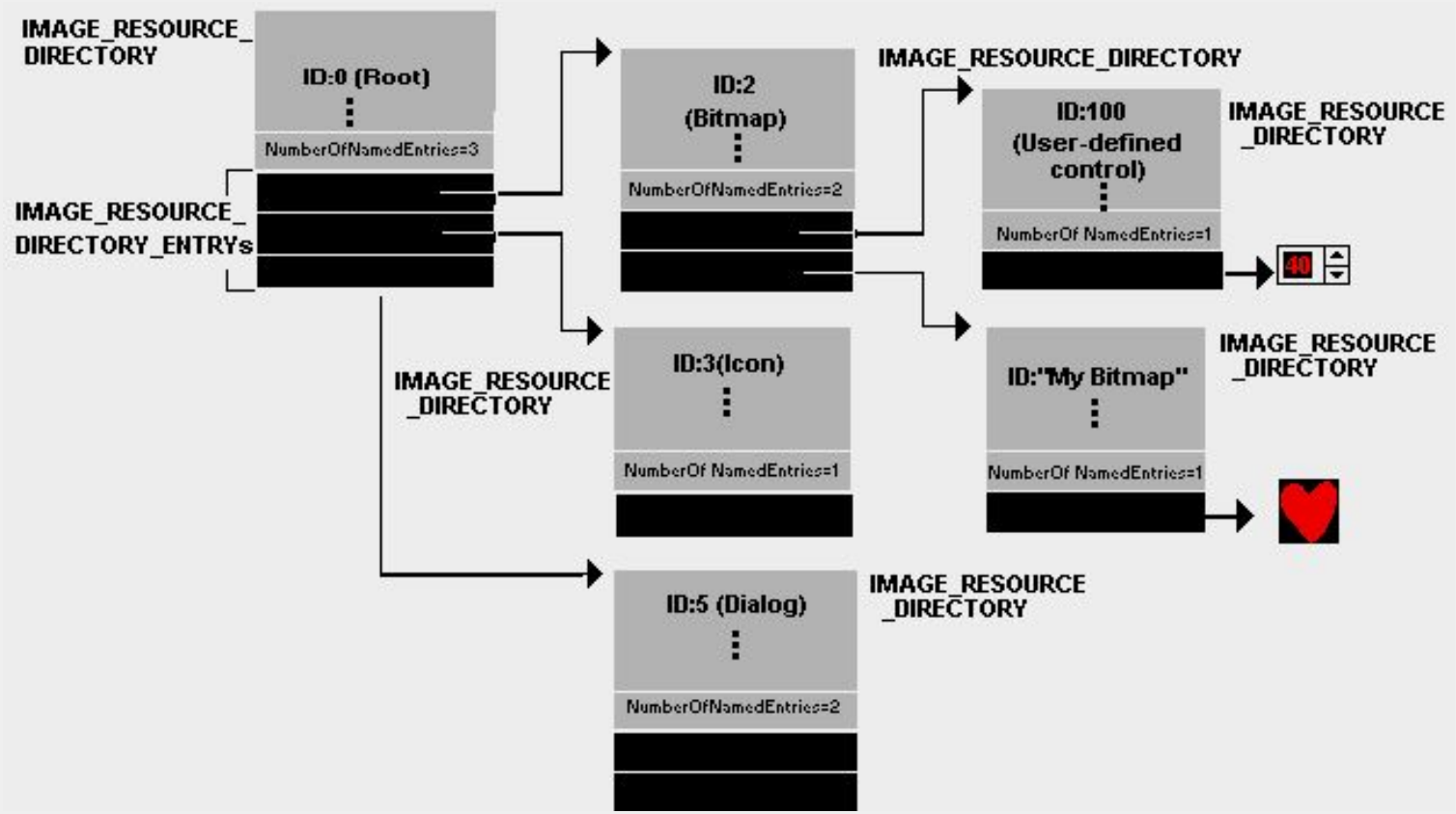
1 A BSTR is a IMAGE_RESOURCE_DIR_STRING_U structure which contains a WORD with the length and a WCHAR array with a unicode string.

Resource Data Entry

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
        DWORD   OffsetToData;
        DWORD   Size;
        DWORD   CodePage;
        DWORD   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY,*PIMAGE_RESOURCE_DATA_ENTRY;
```

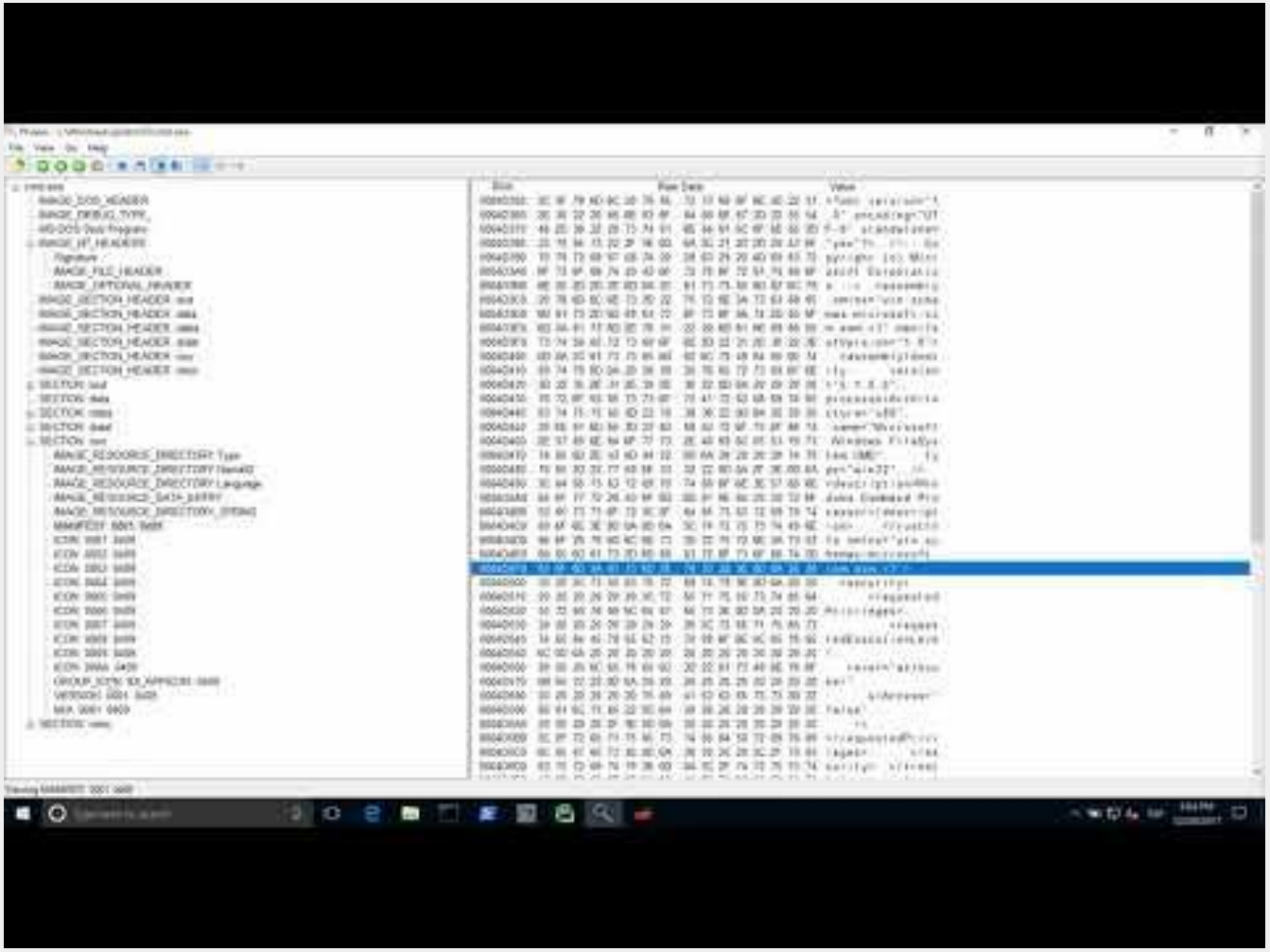
- OffsetToData: RVA from .reloc to the raw contents of the resource.
- Size: Size in bytes of the raw resource contents.

Overview



By design, the root Resource Directory Entry will never point to a Resource Data Entry, it will always point to another Resource Directory.

Let's not develop a tool for this, let's get some rest :)



Thread Local Storage

What is the Thread Local Storage (TLS)?

The Thread Local Storage (TLS) is a mechanism used to allocate variables not shared among threads.

Which is not too interesting if it wasn't because...

TLS also provides the capability of executing functions before the binary's entrypoint gets executed! These are commonly known as *TLS callbacks*

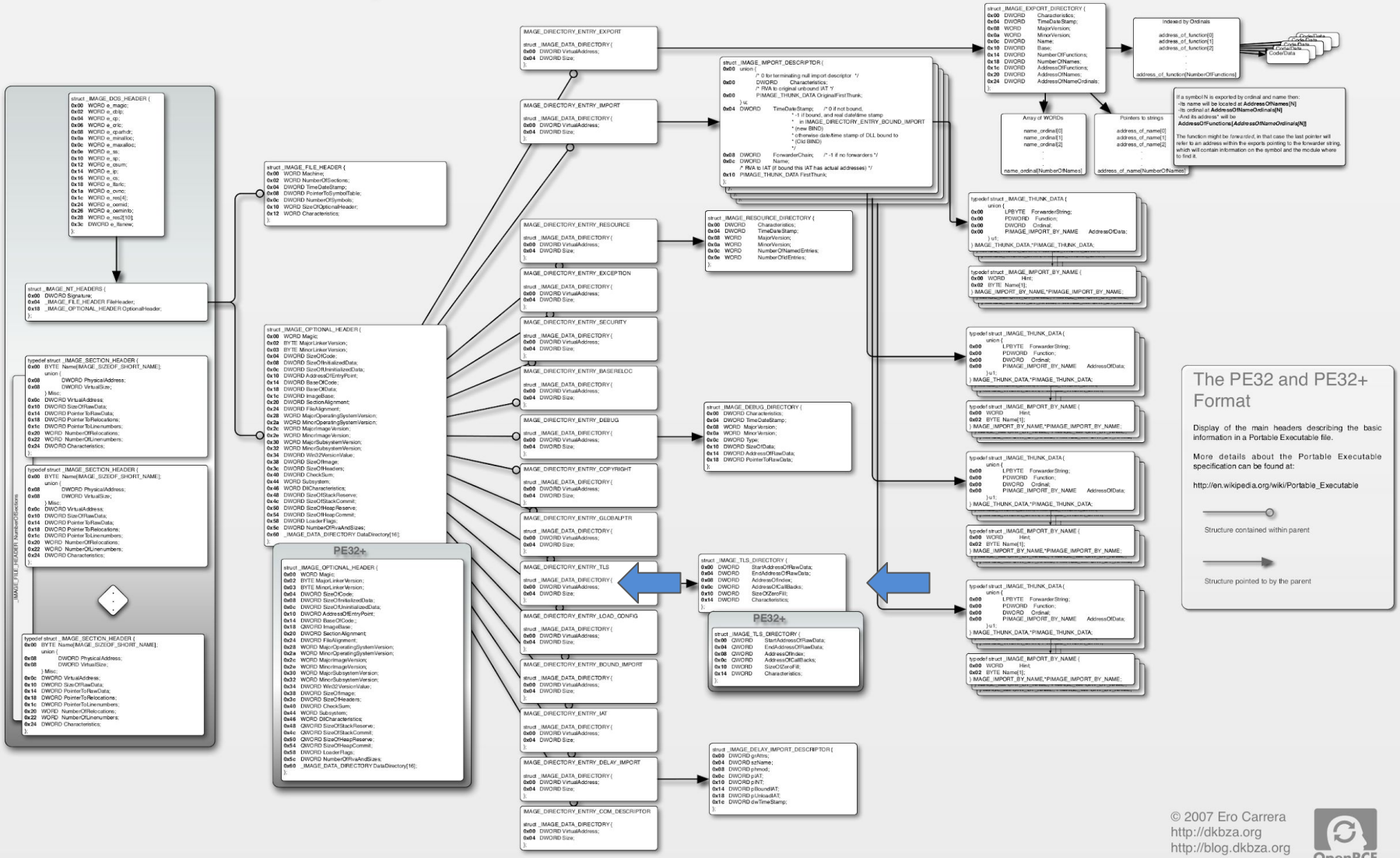
This is a neat technique to harden the static and dynamic analysis of a malware sample. Samples such as Locky ransomware uses it.

TLS callback are executed each time that a thread is created (even the one from the main process) or a thread is about to exit (even the one from the main process).

Usually stored in the .rdata section.

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_RESOURCE

Portable Executable Format Layout



TLS Directory

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD   StartAddressOfRawData;
    DWORD   EndAddressOfRawData;
    DWORD   AddressOfIndex;
    DWORD   AddressOfCallBacks;
    DWORD   SizeOfZeroFill;
    DWORD   Characteristics;
} IMAGE_TLS_DIRECTORY32, *PIMAGE_TLS_DIRECTORY32;
```

```
typedef struct _IMAGE_TLS_DIRECTORY64 {
    ULONGLONG   StartAddressOfRawData;
    ULONGLONG   EndAddressOfRawData;
    ULONGLONG   AddressOfIndex;
    ULONGLONG   AddressOfCallBacks;
    DWORD       SizeOfZeroFill;
    DWORD       Characteristics;
} IMAGE_TLS_DIRECTORY64, *PIMAGE_TLS_DIRECTORY64;
```

- **StartAddressOfRawData**
- **EndAddressOfRawData**

It contains two addresses (Virtual Address, not relative!) indicating the start and the end of the space reserved for the specific variables for this thread (if any).

- **AddressOfCallbacks**

It contains an address (Virtual Address, not relative!) pointing to an arrays of pointers. Each pointer points to a TLS callback function.

A null pointer (a DWORD/QWORD set to 0) defines the end of the array.

Let’s build a tool to know PE:

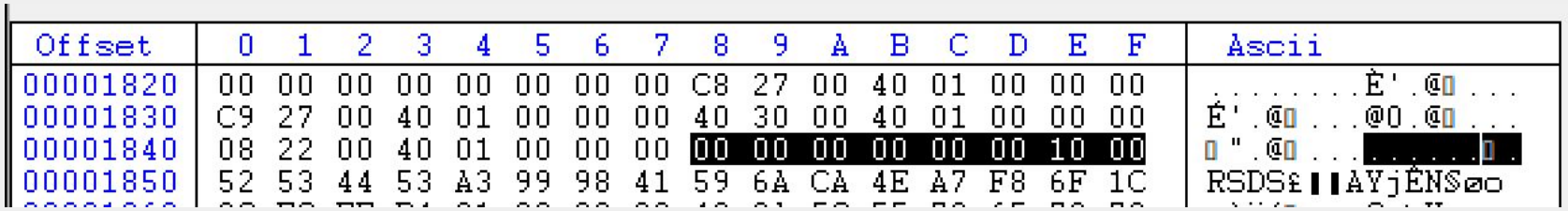
- TLS Callbacks

16.pe_thread_local_storage.py

PE32+ Problems

It looks like to know what is the last function callback (null pointer) for PE32+ we cannot rely on reading a QWORD and comparing it to 0.

Looks like we need to read a QWORD and if the LSB DWORD is 0, then we can assume that is the last callback. **(I need to confirm this…)**



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00001820	00	00	00	00	00	00	00	00	C8	27	00	40	01	00	00	00	........È'.@...
00001830	C9	27	00	40	01	00	00	00	40	30	00	40	01	00	00	00	É'.@....@0.@....
00001840	08	22	00	40	01	00	00	00	00	00	00	00	00	00	10	00	."..@..........
00001850	52	53	44	53	A3	99	98	41	59	6A	CA	4E	A7	F8	6F	1C	RSDS£..AYjÊN§øo

At file offset 0x1840 we have the right callback VA.

However, in bold you have the last callback pointer (it should have been all 0s). AS can be seen, we do not have a full QWORD set to 0, the second MSB is set to 10.

This is not either the base address, which is 0x140000000

PE32+ Problems

It looks like to know what is the last function callback (null pointer) for PE32+ we cannot rely on reading a QWORD and comparing it to 0.

Looks like we need[illegible] 0, then we can assume that is the[illegible]

Offset	0	1	Ascii
00001820	00	00	........È'.@▯...
00001830	C9	27	É'.@▯...@0.@▯...
00001840	08	22	▯".@▯...▯▯▯▯▯▯▯.
00001850	52	53	RSDS£▮▮AYjÉNS@o

At file offset 0x1840 we have the right callback VA.

However, in bold you have the last callback pointer (it should have been all 0s). AS can be seen, we do not have a full QWORD set to 0, the second MSB is set to 10.

This is not either the base address, which is 0x140000000

Relocations

What are relocations?

A relocation is an address fix that the loader applies to a binary when it is loaded into memory.

Executables and libraries have a pre-defined (base) address where they should be loaded (the linker sets them). However, what happens if:

- More than one library has the same predefined base address?
- ASLR is enabled?

The loader needs to address that situation. This is where the relocations table come into play.

The relocations table contain all the addresses that need to be fixed if the binary or library could not be loaded to its preferred base address.

What needs to be relocated?

Basically, any code that the compiler has generated that contain absolute virtual addresses. Examples of that might be addresses for strings, global variables, etc.

In x86, relocations (or fixups) are a common thing. With the introduction of x64, compilers generated more PIC (Position Independent Code) binaries.

PIC binaries reference other instructions or data using relative offsets instead of absolute addresses. Therefore, the use of relocations is avoided or, at least, reduced.

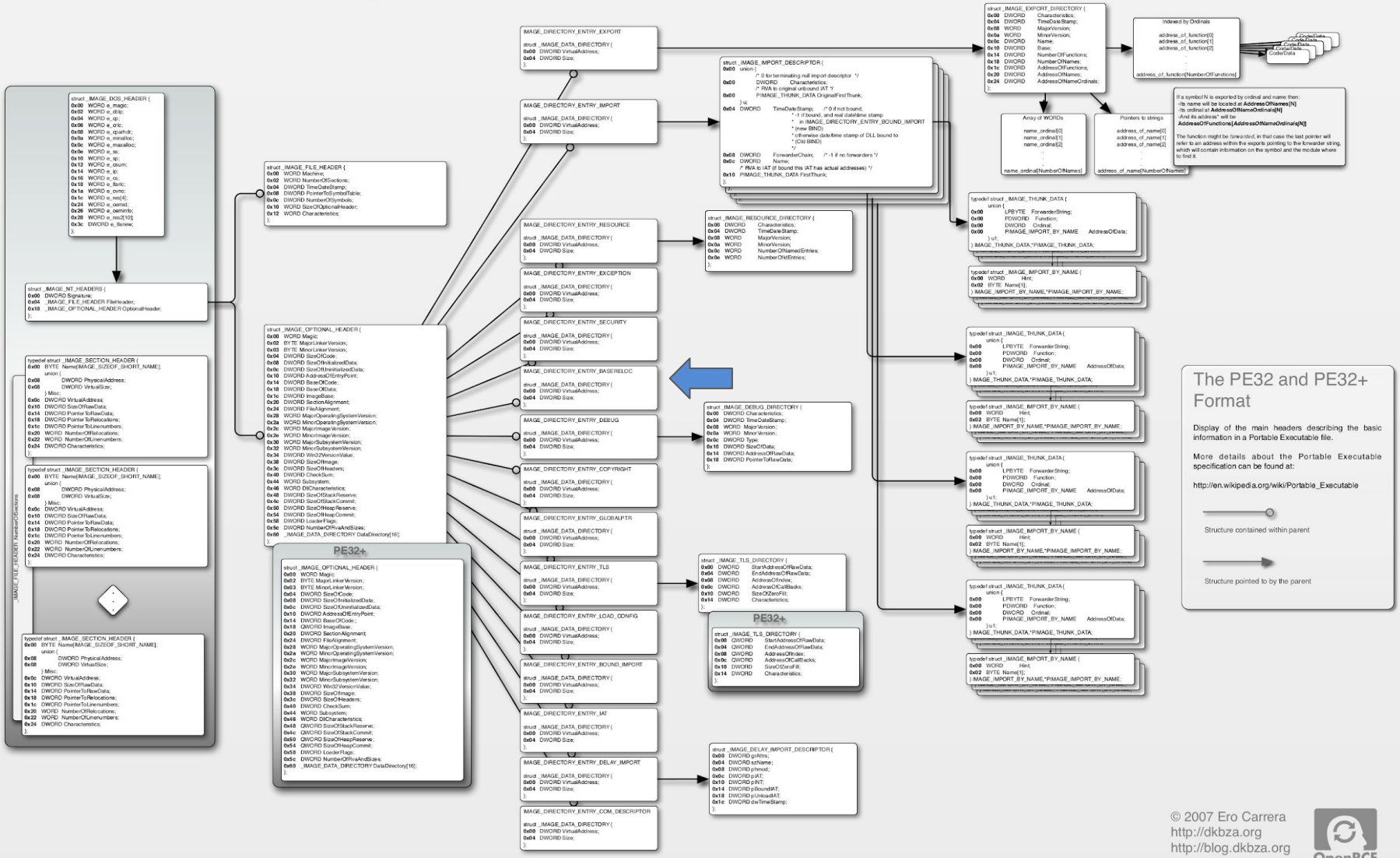
More information on the topic:

- PIC on x86
- PIC on x64

You can double-check that by comparing relocs for a PE and a PE32+ binary.

Binary Files → PE Binary Format → IMAGE_DIRECTORY_ENTRY_BASERELOC

Portable Executable Format Layout



Relocation Directory

```
typedef struct _IMAGE_BASE_RELOCATION
{
        DWORD   VirtualAddress;
        DWORD   SizeOfBlock;
        /* WORD TypeOffset[1]; */
} IMAGE_BASE_RELOCATION,*PIMAGE_BASE_RELOCATION;
```

VirtualAddress
SizeOfBlock
Relocation 1
Relocation 2
(...)
VirtualAddress
SizeOfBlock
Relocation 1
(...)

The IMAGE_DIRECTORY_ENTRY_BASERELOC VirtualAddress points to an array of IMAGE_BASE_RELOCATIONs.

This structure is not shown in Ero's diagram. it is pointed by the resource directory.

- **VirtualAddress**

This is a RVA that should be added to each relocation value for this block of relocations.

- **SizeOfBlock**

Size of the relocations block that follows.

SizeOfBlock = sizeof(IMAGE_BASE_RELOCATION) + <all relocations for this block> * 2

VirtualAddress
SizeOfBlock
Relocation 1
Relocation 2
(...)
VirtualAddress
SizeOfBlock
Relocation 1
(...)

Number of relocations per block:

(SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD) →

(SizeOfBlock - 8) / 2

How are relocations applied?

- **VirtualAddress**

This is a RVA that should be added to each relocation value for this block of relocations.

e.g

In address <VirtualAddress1> + <relocation1> the loader needs to apply a delta

In address <VirtualAddress1> + <relocation2> the loader needs to apply a delta

In address <VirtualAddress2> + <relocation3> the loader needs to apply a delta

VirtualAddress 1
SizeOfBlock 1
Relocation 1
Relocation 2
(...)
VirtualAddress 2
SizeOfBlock 2
Relocation 3
(...)

The **delta** is the difference between the preferred base address and the address where the binary was really loaded.

delta = loaded base addr - preferred addr

Relocation Format

A relocation has the length of a WORD (16 bits).

The bottom 12 bits → are a relocation offset,
(what is added to VirtualAddress for its block)

The high 4 bits → are a relocation type.

There are two types of relocations:

VirtualAddress 1
SizeOfBlock 1
Relocation 1
Relocation 2
(...)
VirtualAddress 2
SizeOfBlock 2
Relocation 3
(...)

IMAGE_REL_BASED_ABSOLUTE = 0 → Relocations used for padding
IMAGE_REL_BASED_HIGHLOW = 3 → Regular relocation.

How many blocks / Blocks limit?

Unfortunately, there isn't a field specifying how many relocation blocks exist.

One approach to tackle that is:

End of Blocks ?

IMAGE_DIRECTORY_ENTRY_BASERELOC.VirtualAddress

+

IMAGE_DIRECTORY_ENTRY_BASERELOC.Size

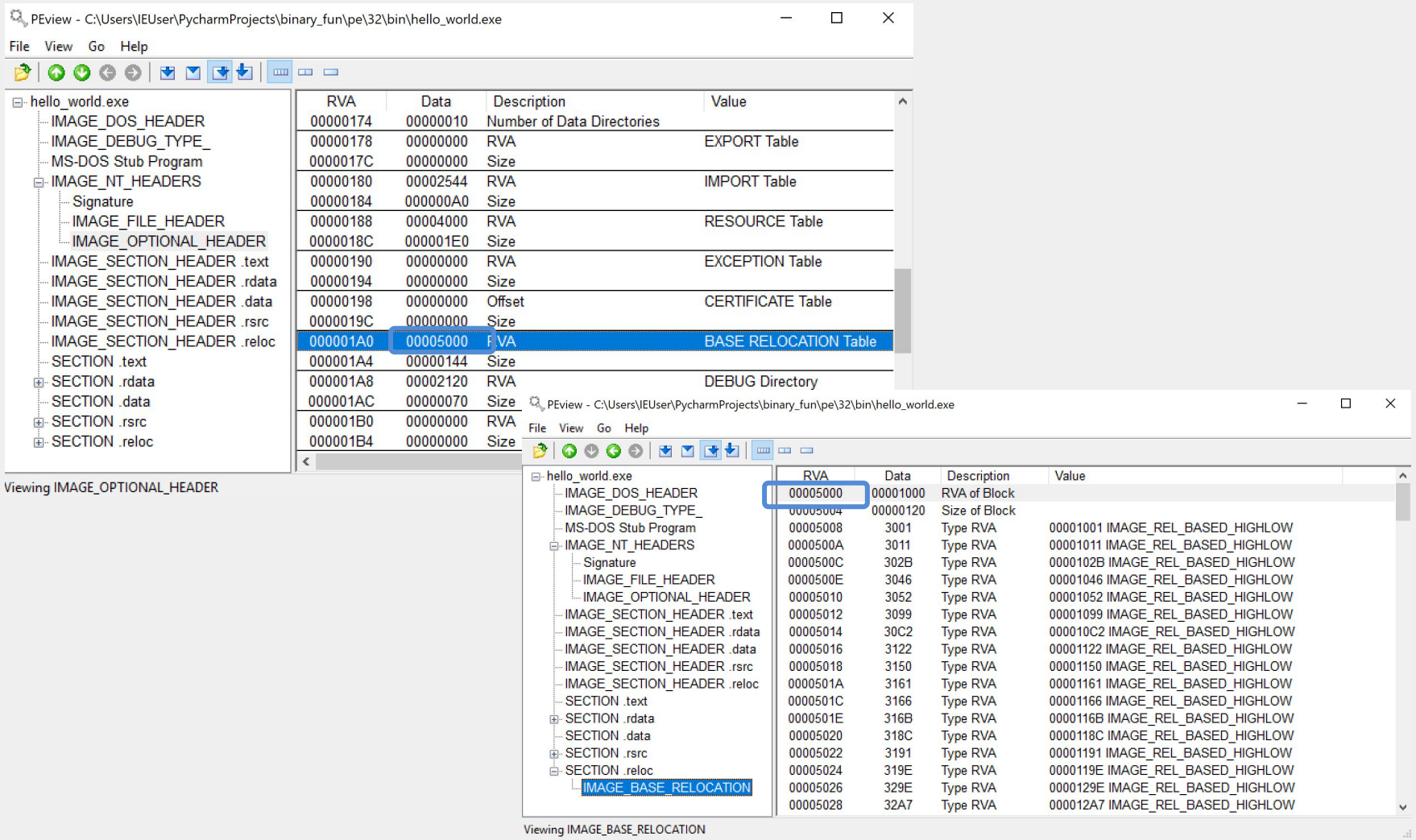
==

New Block Address

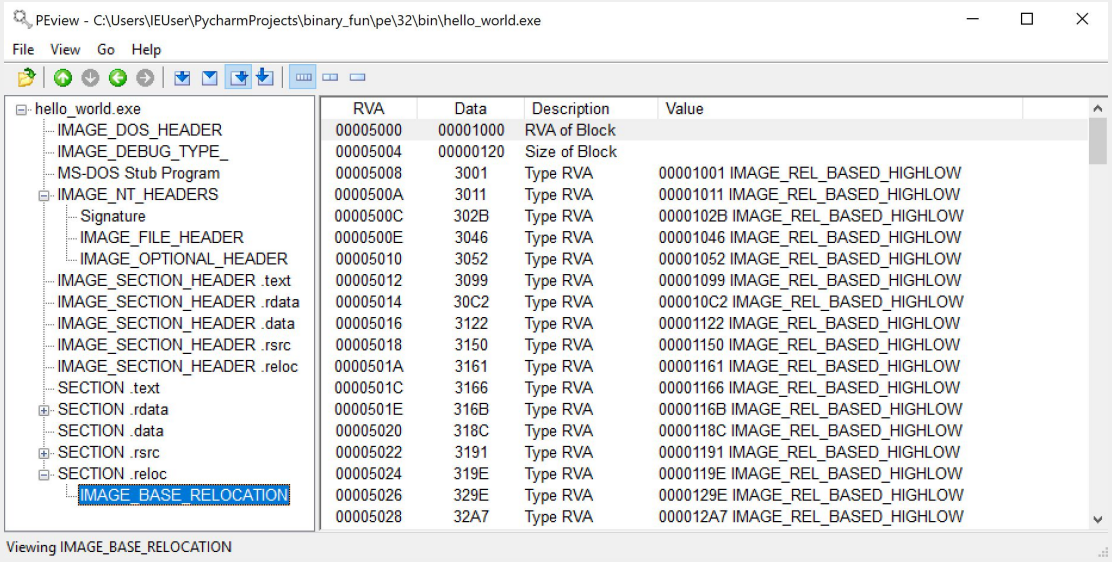
VirtualAddress 1
SizeOfBlock 1
Relocation 1
Relocation 2
(...)
VirtualAddress 2
SizeOfBlock 2
Relocation 3
(...)

Example

Relocations are usually inside the .reloc section:



Example



- In VA 0x1000+0x1 a relocation would need to be applied.
- In VA 0x1000+0x11 a relocation would need to be applied.
- In VA 0x1000+0x2B a relocation would need to be applied.
- (...)

Let’s build a tool to know PE:

- Relocations

15.pe_relocations.py

Now we have a deep understanding about
PE internal structures and capabilities.

As an analyst this will help you to:

- Understand how the tools you use work
- Improve the tools when they don't work for you
- Analyze wrongly crafted PEs
- Understand hooking techniques (IAT hooking)
- Understand advanced import resolving techniques
- Understand DLL Injection techniques (EAT Forwarding)
- Understand/write binary (un)packing algorithms
- Give you the base for any PE trickery used out there
- Etc

ELF Binary Format

Binary Files → ELF Binary Format

An ELF (Executable and Linkable Format) file is the binary file format used in many unix platforms such as:

- Linux
- OpenBSD
- Dreamcast
- Playstation 2...4
- Windows 10[1]
- Android[2]
- etc!

[1] Through the Windows Subsystem for Linux.
[2] As shared libraries ".so"

An ELF (Executable and Linkable Format) file is the binary file format used in many unix platforms such as:

- Linux
- OpenBSD
- Dreamcast
- Playstation 2...4
- Windows 10[1]
- Android[2]
- etc!

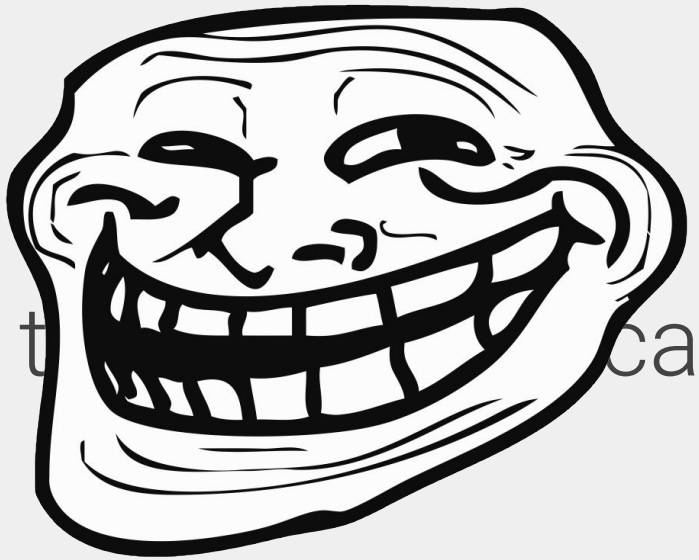
Let's start dissecting the full specification!

[1] Through the Windows Subsystem for Linux.
[2] As shared libraries ".so"

An ELF (Executable and Linkable Format) file is the binary file format used in many unix platforms such as:

- Linux
- OpenBSD
- Dreamcast
- Playstation 2...4
- Windows 10[1]
- Android[2]
- etc!

Let's start dissecting t cation!



[1] Through the Windows Subsystem for Linux.
[2] As shared libraries ".so"

Now you've been empowered to tackle that on your own!

As you can see, format is similar. ELF header, sections, relocations, etc.

ELF Specification

Introduction 1-1
File Format 1-1
Data Representation 1-2
Character Representations 1-3

ELF Header 1-4
ELF Identification 1-6

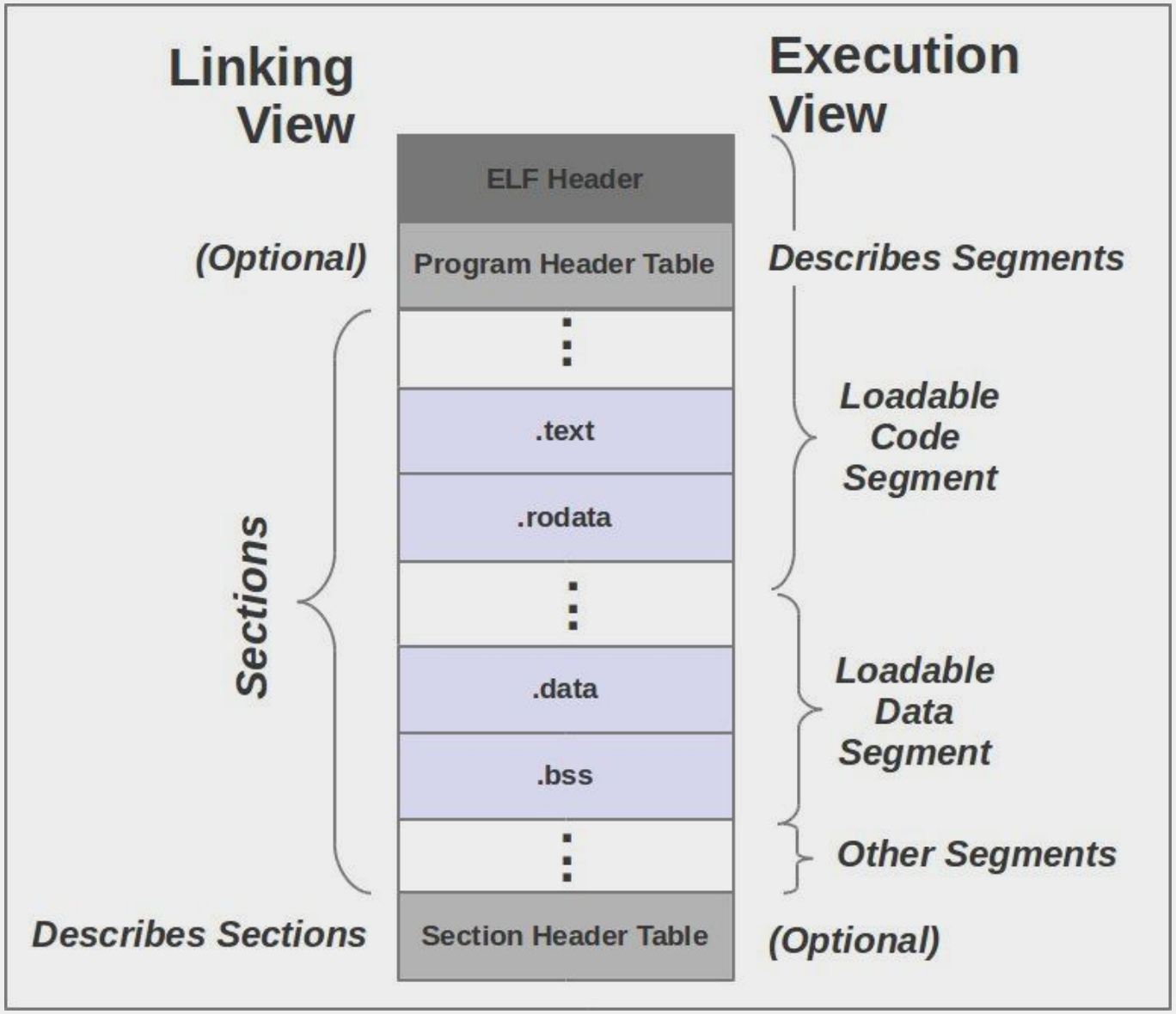
Sections 1-9
Special Sections 1-15

String Table 1-18

Symbol Table 1-19
Symbol Values 1-22

Relocation 1-23

Headers


Linking View
Execution View
ELF Header
(Optional)
Program Header Table
Describes Segments
.text
.rodata
Loadable Code Segment
Sections
.data
.bss
Loadable Data Segment
Other Segments
Describes Sections
Section Header Table
(Optional)

Aside from other **headers**, ELF files contain

- **Sections** (.text, .rodata, .bss, etc):

1. They contain the raw data to be loaded to memory.
2. They contain information used by the linker that won't have to be mapped to memory (e.g. relocations)

- **Segments** (they don't have names):

1. They contain the VA where they will be loaded on execution.
2. They contain the access permission in memory

A segment can contain multiple sections.

A section might not be mapped into memory.

ELF Header

ELF Header

- e_ident: File identifier (“ELF”)
- e_type: Type of ELF (library, executable, …)
- **e_entry**: Image base in memory (entry point).
- e_phoff: File offset to program header.
- e_shoff: File offset to section header.
- **e_phentsize**: Program header entry size (all entries have same size)
- **e_phnum**: Number of program header entries.
- **e_shentsize**: Section header entry size (all entries have same size)
- **e_shnum**: Number of section header entries

(...)

Program Header

Program Header Table (Segments)

An array of entries.
Size (always the same) and number of structs are defined in ELF header.

- p_type: Type of segment[1]. Only PT_LOAD segments will be loaded to memory.
- p_offset: File offset where the segment data starts.
- p_vaddr: Virtual address where segment will be loaded.
- p_filesz: Size of the segment in the file.
- p_memsz: Size of the segment in memory.
- p_flags: Access permission in memory: read, write, execute

(...)

[1] In Windows sections do not have type

Section Header

Section Header Table (sections)

An array of entries.
Size (always the same) and number of structs are defined in ELF header.

- sh_name: Name of the section[1].
- sh_type: Type of the section. Type SHT_PROGBITS is the generic one.
- sh_addr: Virtual address where the section will be loaded (if it's loaded).
- sh_offset: File offset where the section is stored.
- sh_size: Size of the section.

(...)

[1] This field is an index to what is known as a String Table. String tables contain null terminated strings.

But wait, Albert, didn't you say segments contain multiple sections? Where's all that information?

In fact, ELF files do not contain that information.

It is a loader script that specifies what sections are merged in what segments.

$ ld --verbose

```
.text           :
{
  *(.text.unlikely .text.*_unlikely .text.unlikely.*)
  *(.text.exit .text.exit.*)
  *(.text.startup .text.startup.*)
  *(.text.hot .text.hot.*)
  *(.text .stub .text.* .gnu.linkonce.t.*)
  /* .gnu.warning sections are handled specially by elf32.em.  */
  *(.gnu.warning)
}
```

This means that segment .text will merge sections .text.unlikely, .stub, etc.

But wait, Albert, didn't you say segments don't have names?

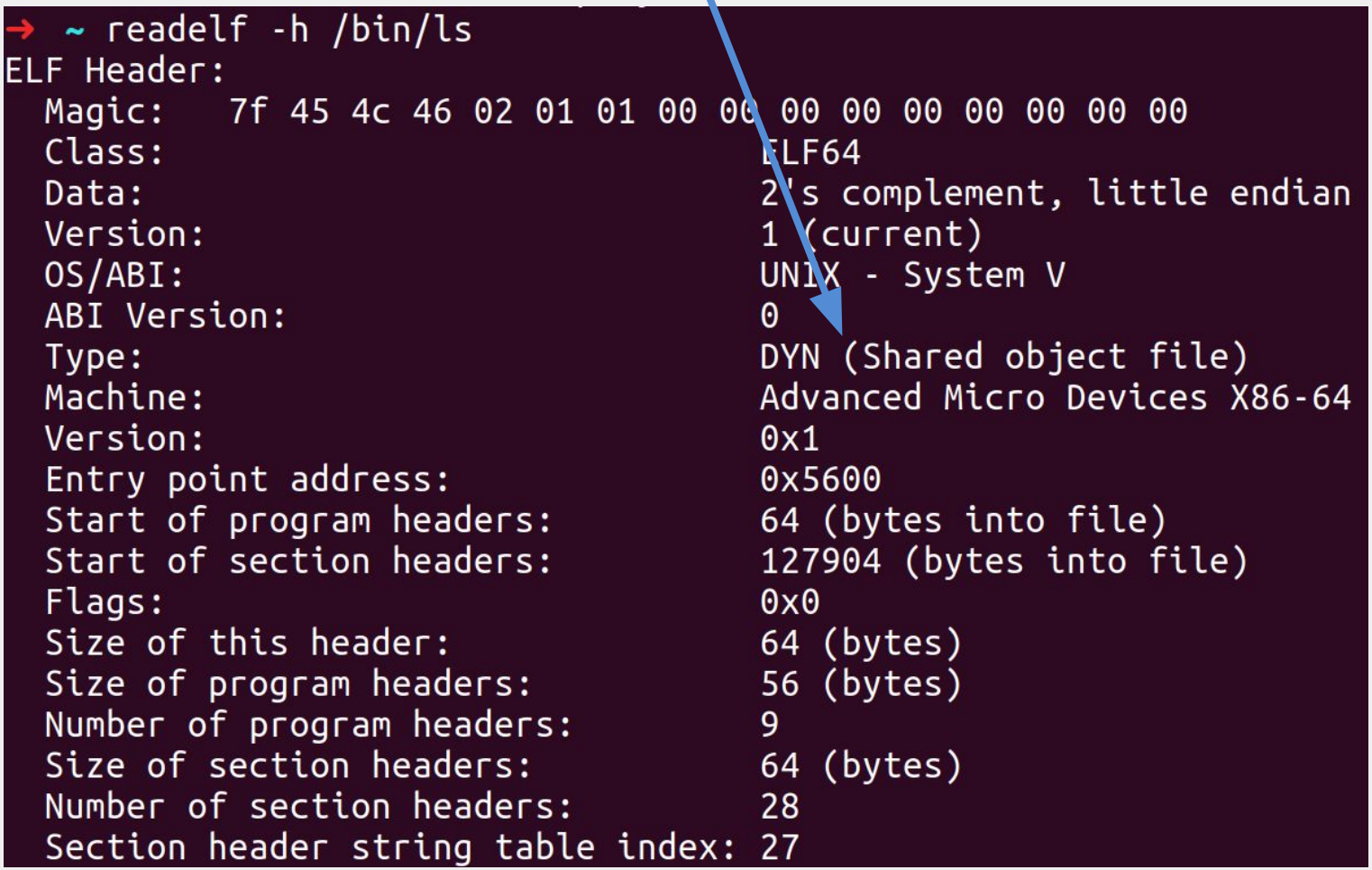
Yeah, that's still **true**!

I guess the loader uses those names as variables for the script.

ELF Headers Analysis

Let's check the ELF header

¯_(ツ)_/¯ → It does not happen in Debian GNU/Linux 8 or Ubuntu 16.04.3 LTS[1]



[1] I discovered GCC compiles things as PIE (Position Independent Executables) by default nowadays, that is interpreted as shared objects, thus the e_type ELF header field set to 0x3.

Let's check the Program Header (Segments)

```
→ ~ readelf -l /bin/ls

Elf file type is DYN (Shared object file)
Entry point 0x5600
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000001d930 0x000000000001d930  R E    0x200000
  LOAD           0x000000000001e030 0x000000000021e030 0x000000000021e030
                 0x0000000000001238 0x0000000000002418  RW     0x200000
  DYNAMIC        0x000000000001ea58 0x000000000021ea58 0x000000000021ea58
                 0x0000000000000200 0x0000000000000200  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x000000000001a5d4 0x000000000001a5d4 0x000000000001a5d4
                 0x0000000000000834 0x0000000000000834  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x000000000001e030 0x000000000021e030 0x000000000021e030
                 0x0000000000000fd0 0x0000000000000fd0  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.ve
dr .eh_frame
   03     .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .data.rel.ro .dynamic .got
```

This segment contains the path of the loader that will be used.

Only these two segments will be loaded to memory

Segment 0 contains no section
Segment 1 contains .interp section
(...)

Let's check the Section Headers

```
→ ~ readelf -S /bin/ls
There are 28 section headers, starting at offset 0x1f3a0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000000254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000000274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000298  00000298
       00000000000000ec  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000000388  00000388
       0000000000000d98  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000001120  00001120
       0000000000000661  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000001782  00001782
       0000000000000122  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000000018a8  000018a8
       0000000000000070  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000001918  00001918
       0000000000001350  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000002c68  00002c68
       00000000000009f0  0000000000000018  AI       5    23     8
  [11] .init             PROGBITS         0000000000003658  00003658
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000003670  00003670
       00000000000006b0  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000003d20  00003d20
       0000000000000020  0000000000000008  AX       0     0     8
  [14] .text             PROGBITS         0000000000003d40  00003d40
```

```
  [22] .dynamic          DYNAMIC          000000000021ea58  0001ea58
       0000000000000200  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         000000000021ec58  0001ec58
       00000000000003a8  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         000000000021f000  0001f000
       0000000000000268  0000000000000000  WA       0     0     32
  [25] .bss              NOBITS           000000000021f280  0001f268
       00000000000011c8  0000000000000000  WA       0     0     32
  [26] .gnu_debuglink    PROGBITS         0000000000000000  0001f268
       0000000000000034  0000000000000000           0     0     4
  [27] .shstrtab         STRTAB           0000000000000000  0001f29c
       0000000000000101  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Imports & Exports

Imports and exports are symbols. In ELF files, symbols are defined inside **Symbol Tables**. The symbol tables are contained in the sections of type SHT_DYNSYM or SHT_SYMTAB.

The **.dynsym** and **.symtab** symbol tables contain all file symbols, however, the only one that matters on run-time is the **.dynsym**[1].

- **How to know if symbols are imports or exports?**

Close to the truth answer: If the symbol is undefined, then it is an import. It is an export otherwise.

You'll go nuts answer: "hackers who've read this far, have probably gone apoplectic. Sorry about that."

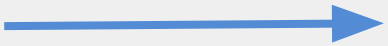
[1] To know more about why both symbol table exist, you can read this.

Binary Files → ELF Binary Format → Imports & Exports

```
→ 2.binary_file_formats readelf -s hello_world_elf64

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

These undefined symbols are the imports

These symbols might not be mapped to memory on execution



```
Symbol table '.symtab' contains 60 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004002b8     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400318     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400356     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000400360     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400380     0 SECTION LOCAL  DEFAULT    9
    10: 00000000004003b0     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004003c8     0 SECTION LOCAL  DEFAULT   11
    12: 00000000004003e0     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000400400     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000400574     0 SECTION LOCAL  DEFAULT   14
    15: 0000000000400580     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000400590     0 SECTION LOCAL  DEFAULT   16
    17: 00000000004005c8     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000600e10     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000600e18     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000600e20     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000600ff0     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000601000     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000601020     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000601030     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000000000     0 SECTION LOCAL  DEFAULT   25
    26: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    27: 0000000000400430     0 FUNC    LOCAL  DEFAULT   13 deregister_tm_clones
    28: 0000000000400460     0 FUNC    LOCAL  DEFAULT   13 register_tm_clones
    29: 00000000004004a0     0 FUNC    LOCAL  DEFAULT   13 __do_global_dtors_aux
    30: 0000000000601030     1 OBJECT  LOCAL  DEFAULT   24 completed.7632
    31: 0000000000600e18     0 OBJECT  LOCAL  DEFAULT   19 __do_global_dtors_aux_fin
    32: 00000000004004d0     0 FUNC    LOCAL  DEFAULT   13 frame_dummy
    33: 0000000000600e10     0 OBJECT  LOCAL  DEFAULT   18 __frame_dummy_init_array_
    34: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello_world.c
    35: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    36: 00000000004006b4     0 OBJECT  LOCAL  DEFAULT   17 __FRAME_END__
    37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    38: 0000000000600e18     0 NOTYPE  LOCAL  DEFAULT   18 __init_array_end
    39: 0000000000600e20     0 OBJECT  LOCAL  DEFAULT   20 _DYNAMIC
    40: 0000000000600e10     0 NOTYPE  LOCAL  DEFAULT   18 __init_array_start
    41: 0000000000400590     0 NOTYPE  LOCAL  DEFAULT   16 __GNU_EH_FRAME_HDR
    42: 0000000000601000     0 OBJECT  LOCAL  DEFAULT   22 _GLOBAL_OFFSET_TABLE_
    43: 0000000000400570     2 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    44: 0000000000601020     0 NOTYPE  WEAK   DEFAULT   23 data_start
    45: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
    46: 0000000000601030     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    47: 0000000000400574     0 FUNC    GLOBAL DEFAULT   14 _fini
    48: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    49: 0000000000601020     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    50: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    51: 0000000000601028     0 OBJECT  GLOBAL HIDDEN    23 __dso_handle
```

Resolving Imports

For ELF files, import resolution work similar to Windows delayed imports.

This means that an import address is first resolved when that imported function is used for the first time.

This resolution involves the **PLT (Procedure Linkage Table)**, the **GOT (Global Offset Table)** and the linker.

- The GOT is a data structure that is updated with the import addresses once they get resolved.

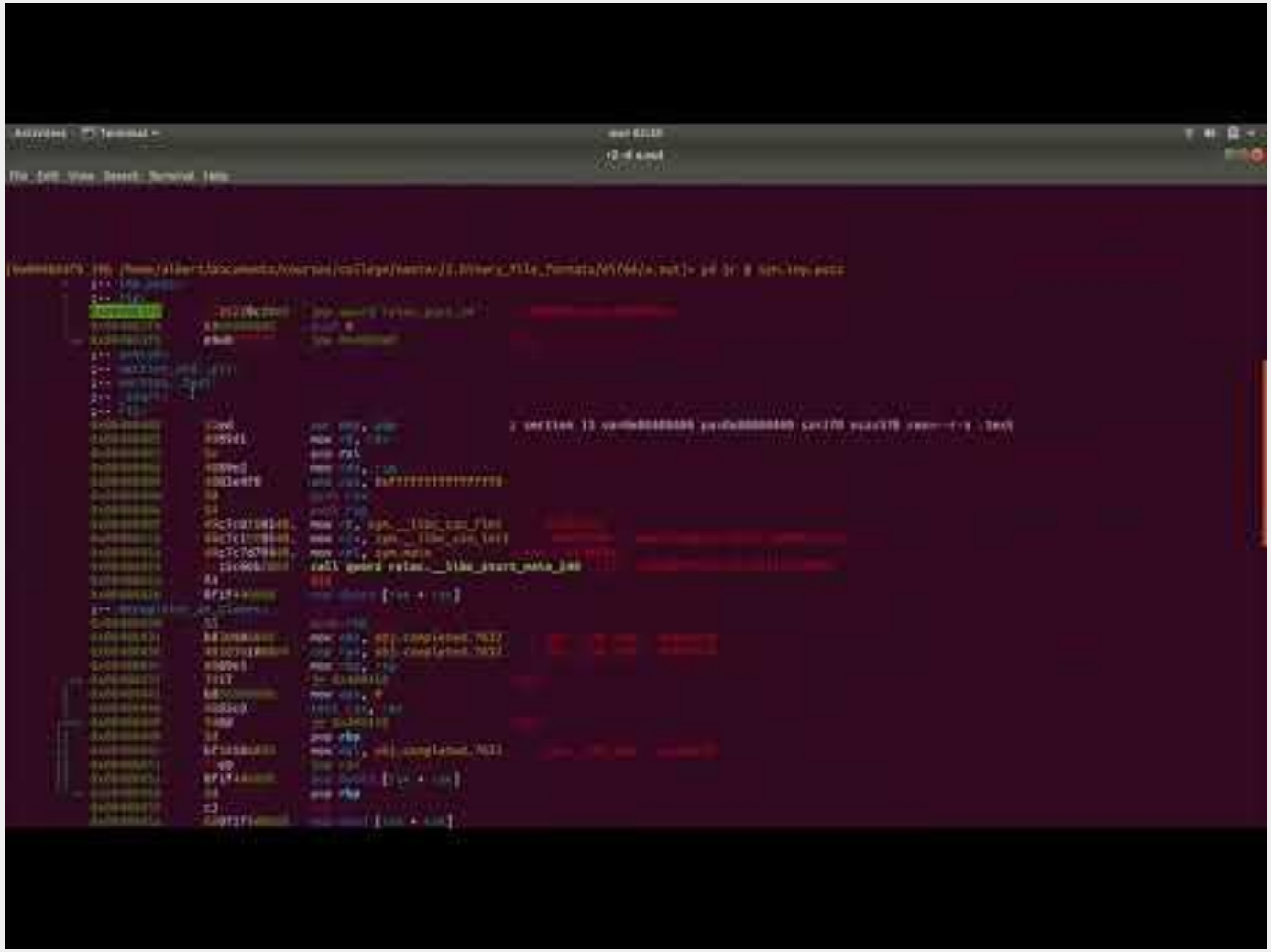
- The PLT is a structure contains code to handle the steps to resolve delayed imports (with the help of the linker).

Import Address Resolution Theory

1. The program code calls to an unresolved import
2. The program code first jumps to the PLT
3. The PLT contains code that will try to retrieve the address of the import from the GOT
4. The GOT will not contain the address of this import because it's its first call
5. The GOT contains the address of a different part of the PLT
6. This part of the PLT will call the linker with the appropriate info so it can update the GOT with the correct resolved address
7. The code will jump to the correct import address

1. Next time the import is called, the GOT will contain the resolved import address and the PLT code will directly jump to it.

Import Address Resolution Demo



ELF vs PE

ELF vs PE

- **Some Differences**

- ELF does not have TLS Callbacks
- ELF does not have resources. Some workarounds here.
- ELF function importing is always lazy (like in PE delayed imports)
- PE do not have the concept of segments vs sections
- ELF use string tables for section names/sym instead of embedding into struct
- ELF most of the time works with absolute VA whereas PE works with RVA
- ELF specifies what program should link the file (.intpr section)
- (...)

- **Similarities**

- Files are divided into chunks that get loaded into memory
- Work with the concept of file offsets and virtual addresses
- Work with start address + size (instead of e.g. start addr and end addr)
- Use relocations/fixups
- (...)

Appendices

Optional Header PE32+ Offsets

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
0  WORD  Magic; /* 0x20b */
2  BYTE MajorLinkerVersion;
3  BYTE MinorLinkerVersion;
4  DWORD SizeOfCode;
8  DWORD SizeOfInitializedData;
12  DWORD SizeOfUninitializedData;
16  DWORD AddressOfEntryPoint;
20  DWORD BaseOfCode;
24  ULONGLONG ImageBase;
32  DWORD SectionAlignment;
36  DWORD FileAlignment;
40  WORD MajorOperatingSystemVersion;
42  WORD MinorOperatingSystemVersion;
44  WORD MajorImageVersion;
(...)
46  WORD MinorImageVersion;
48  WORD MajorSubsystemVersion;
50  WORD MinorSubsystemVersion;
52  DWORD Win32VersionValue;
56  DWORD SizeOfImage;
60  DWORD SizeOfHeaders;
64  DWORD CheckSum;
68  WORD Subsystem;
70  WORD DllCharacteristics;
72  ULONGLONG SizeOfStackReserve;
80  ULONGLONG SizeOfStackCommit;
88  ULONGLONG SizeOfHeapReserve;
96  ULONGLONG SizeOfHeapCommit;
104  DWORD LoaderFlags;
108  DWORD NumberOfRvaAndSizes;
110  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

Homework

Homework

1. **Finish all the exercises in this module.**

    1. Record a video doing a code walk-through and executing your code.
    2. Upload the code somewhere (github, bitbucket)

1. **Do something useful not explained in this module.**

Examples:

1. Parse some interesting ELF structures.

or

1. Enumerate exported functions from a DLL by ordinal.

or

1. Attach to a process and parse IAT in memory.

or

4. List all binaries with TLS callbacks in your system

or

5. Parse resources and dump them

Bibliography

Bibliography

- PE File Format by Microsoft
- More on the PE Structure by Microsoft
- Microsoft Portable Executable and Common Object File Format Specification
- PE File Structure
- Open Security Training, Life of Binaries

Helpful Links:

- https://github.com/wine-mirror/wine/blob/master/include/winnt.h
- https://stackoverflow.com/questions/15960437/how-to-read-import-directory-table-in-c
- https://stackoverflow.com/questions/9955744/getting-offset-in-file-from-rva
- http://www.sunshine2k.de/reversing/tuts/tut_rvait.htm

Index of Figures

- Slide 7, http://www.aboutdebian.com/compile.htm
- Slide 69, http://reversecore.com/23
- Slide 83, 84, 101, A Tour of the Win32 Portable Executable File Format
- Slide 130, ELF Sections & Segments and Linux VMA Mappings