Structural Analysis of PE Files

In the development cycle, source code is converted to an object file by a compiler, and then a linker combines object files and necessary libraries to create an executable file (e.g., `.c` → `.o` → `.exe`). The Portable Executable (PE) format is the data structure standard for executables, object code, and linked libraries (`.exe` , `.dll` , `.sys` , etc.) on Windows operating systems. Its primary purpose is to hold the necessary information for the Windows Operating System loader to map the file into memory and execute the program.

1.1 File Structure.

The PE file is organized into two primary regions: headers and sections. The headers contain metadata required for the loader, while the sections contain the actual executable code, data, and resources (such as icons and images). The PE file structure is designed to maintain backward compatibility with MS-DOS and begins with components from the older file format.

DOS Header and Stub. This initial part consists of the MS-DOS Header and the MS-DOS Stub Program.

- *MS-DOS Header*: The MS-DOS Header is a 64-byte structure located at the very beginning of the PE file. It is the first structure the operating system examines. The first two bytes of this header contain the DOS Magic Number, which is `0x5AAD` (or "MZ" in ASCII), serving as a signature to identify it as an MS-DOS-compatible executable. The final 4-byte field of the DOS Header, known as `e_lfanew` , is a pointer to the beginning of the actual NT Headers. This offset is essential for the Windows loader to bypass the DOS portion and locate the modern PE structure. It's important to note that integer values within the PE file format, including `e_lfanew` , are stored in little-endian byte order.
- *MS-DOS Stub Program*: Immediately following the MS-DOS Header is the MS-DOS Stub Program. This is a small, legitimate 16-bit MS-DOS executable, which is the code that is executed if the file is run in an MS-DOS environment. By default, this stub is simple assembly code that typically prints a message like "This program cannot be run in DOS mode" and then exits, ensuring that the file does not crash legacy systems. The size of the stub is variable, but it commonly includes the ASCII string for the message it prints.

To analyze the byte-level structure of a PE file, tools like Hex Edit or HxD are typically used to examine the contents of the headers and sections.

PE Header. The PE Header is the core structure used by the Windows loader to manage the executable. It is formally known as the `IMAGE_NT_HEADERS` structure and is located at the file offset specified by the `e_lfanew` field in the preceding MS-DOS Header. This structure is composed of three main sub-structures:

(1) *Signature* - `IMAGE_NT_HEADER` : It is a fixed 4-byte signature with the value `0x50450000` which translates to "PE\0\0" in ASCII. This marks the beginning of the Windows NT-compatible executable structure, confirming the file is a Portable Executable.

(2) *COFF File Header* - `IMAGE_FILE_HEADER` : This structure contains basic, machine-independent characteristics of the file.

- `Machine` : Specifies the target CPU architecture for which the executable was built (e.g., `0x14C` for Intel i386/32-bit, or `0x8664` for AMD64/64-bit).
- `NumberOfSections` : An integer indicating the count of sections (like `.text` , `.data` , `.rsrc`) that follow the headers. This dictates the size of the subsequent Section Header Table.
- `TimeDataStamp` : A timestamp indicating the data and time the file was created by the linker. This field is often manipulated by malware authors as an anti-analysis technique.
- `Characteristics` : A set of flags describing key attributes of the file, such as whether it is a DLL, whether it is an executable image, or whether it is large address aware.

(3) *Optional Header* - `IMAGE_OPTIONAL_HEADER` : Despite its misleading name, the Optional Header is mandatory for executable files and contains the majority of the information the OS loader needs. It differs between 32-bit and 64-bit files. Crucial fields within the Optional Header include:

- `Magic` : A value that identifies the PE format type: `0x10B` for PE32, and `0x20B` for PE32+.
- `AddressOfEntryPoint` : This is a Relative Virtual Address (RVA) pointing to the first instruction of the program to be executed, which is the official starting point for the Windows loader.
- `ImageBase` : The preferred base address in virtual memory where the Windows loader attempts to load the executable. The default is typically `0x00400000` for executables.
- `BaseOfCode` / `BaseOfData` : RVAs indicating the start of the code and data sections in memory.
- `SectionAlignment` / `FileAlignment` : These fields specify the alignment boundaries for sections in memory and on disk, respectively. The alignment padding is critical for the loader to map sections correctly.
- `Subsystem` : Identifies the necessary environment for the executable.

Sections. Immediately following the Optional Header is the Section Header Table (also known as the Section Table). This table contains an array of structures, with the number of entries dictated by the `NumberOfSections` field in the COFF File Header. Each entry in the Section Header Table is 40 bytes long and acts as a descriptor for a single section of the executable. Its fields specify how the section is laid out on disk and how it should be mapped into memory by the loader. The most critical fields for analysis are:

- `Name` : An 8-byte field that identifies the section (e.g., `.text` , `.data`).
- `VirtualSize` : The actual size in bytes of the section data when loaded into memory.
- `VirtualAddress` : The Relative Virtual Address of the section's starting address when loaded into memory. This is the offset relative to the preferred `ImageBase` .
- `SizeOfRawData` : The size in bytes of the initialized data on disk.

- `PointerToRawData` : The file pointer to the section's data on disk.
- `Characteristics` : Flags that define the section's attributes, such as whether it is executable, readable, or writable.

A key difference to note is that `VirtualSize` often differs from `SizeOfRawData` . This occurs because sections must be aligned to specific boundaries in memory and on disk. When loaded into memory, sections are aligned to the value specified by `SectionAlignment` . On disk, sections are aligned to the value specified by `FileAlignment` . The raw data on disk may exclude sections of trailing zero-fill bytes that are included in memory to meet alignment

Name	**Purpose**	**Key Attributes**
`.text`	Contains the main executable code (machine instructions).	Executable and Readable.
`.data`	Holds initialized, writable data (e.g., global variables with initial values).	Readable and Writable.
`.rdata`	Contains read-only, initialized data (e.g., string literals, constant data structures, import/export directory information).	Readable only.
`.bss`	Represents uninitialized data. This section exists conceptually in memory but typically has a `SizeOfRawData` of zero on disk, as it is zeroed out by the OS loader.	Readable and Writable.
`.rsrc`	Stores resources used by the application (e.g., icons, bitmaps, menus, version information). The data is often structured as a complex tree.	Readable. This section is a common target for manipulation and can be easily modified without affecting execution flow.
`.idata`	Holds the Import Address Table (IAT) and Import Name Table (INT); used for dynamic linking. The IAT is patched at load time to point to functions in DLLs' Export Address Tables (EATs).	Readable (and sometimes Writable).
`.edata`	Contains the Export Directory (Export Address Table, EAT), listing functions a module makes available. *Note: Executables typically omit this section unless they export functions (e.g., as DLLs do).*	Readable.
`.reloc`	Contains base relocation information. This is used by the loader to modify addresses in the code and data if the preferred `ImageBase` is already occupied when the file is loaded.	Readable.

Table 1 / Summary of common PE (Portable Executable) section characteristics, detailing their purpose and memory protection attributes as used in Windows executables and DLLs.

requirements or to define the BSS section. While section names are generally conventional, they can be customized by the linker or modified by malware to evade detection.

Linking and Execution. The way an executable acquires the necessary code from libraries is managed through two main methods: static linking and dynamic linking.

- *Static Linking*: Static linking is performed by the linker at compile time. It embeds all the library functions a module may call directly into the final executable file. The resulting executable is self-contained and does not rely on external library files at runtime. Windows static libraries typically use the `.lib` extension. This process is generally considered inefficient in terms of size and memory usage, as the entire code for every used library function is immediately loaded into memory upon execution, leading to larger file sizes.

- *Dynamic Linking*: Dynamic linking is performed at runtime. It only includes references to the required library functions in the executable. The actual function code is stored in separate files called Dynamic-Link Libraries (DLLs). The program loads these external functions only when they are needed during execution. To achieve this, the executable uses specialized tables to resolve the addresses of functions imported from DLLs.

The Import Address Table (IAT) is a crucial component in dynamic linking for PE files, pointing to the external functions required by the executable. Initially, the IAT contains placeholders, dummy addresses or hints about the functions' names and the DLL they belong to. The Windows loader is responsible for finding the required DLL at runtime, calculating the actual virtual memory address of each function, and then writing this resolved address into the corresponding entry in the IAT. Once populated, the executable can reference these library functions straightforwardly by jumping to the address stored in the IAT entry. The IAT is generally considered a compulsory structure for most PE executables.

The Export Address Table (EAT) is contained within DLLs. The EAT acts as a public manifest, describing which functions the DLL makes available for use by other executables. It lists the names of these functions along with their corresponding RVAs within the DLL. This information is what the Windows loader uses to populate the importing executable's IAT. The EAT information may be referenced by the Resource Directory or the `.rdata` section.

Relocation. Relocation is the mechanism used by the Windows loader to adjust memory addresses within a PE file when it cannot be loaded at its preferred `ImageBase` address. Every PE file has a preferred base virtual address defined in the Optional Header. If multiple modules in the same process space try to load into the same memory region, a collision occurs. When a collision prevents an image from loading at its preferred address, the loader assigns it a different, available address (the actual base address). Since the executable's code contains many hard-coded addresses relative to the preferred base address, these addresses must be updated to be valid with respect to the actual base address. The PE file's `.reloc` section contains a list of all locations in the code and data that need to be fixed up. The loader uses this information to perform the necessary address changes. While most address resolution occurs during the linking phase, relocation of base addresses is a key step that happens during the execution phase. For efficient memory access, each section must start at a virtual address that is a multiple of the `SectionAlignment` value, typically 4096 bytes.

Packing. Packing is a technique where the executable's code and data are compressed or encrypted to obscure the original content. The primary goal is to make the executable's information not statically visible to analysts or traditional signature-based antivirus scanners. It serves both legitimate purposes and, frequently, malicious purposes. A packed executable contains a small "stub" program. Upon execution, this stub is the first part of the code to run. Its sole job is to unpack the original code into a newly allocated area of memory before transferring execution control to the original entry point. From a PE file structure perspective, packing often introduces several anomalies that can aid in detection:

- *High Entropy*: The packed sections will have a high degree of randomness, which is characteristic of compressed or encrypted data.
- *Atypical Section Sizes/Names*: Packed files often have very large `SizeOfRawData` compared to `VirtualSize` for certain sections, or they may use unusual, non-standard section names.
- *Manipulated Entry Point*: The `AddressOfEntryPoint` points to the unpacker stub, not the original code.

While effective at obfuscation, packing is detectable. Tools like PeiD or other packer-detection utilities look for the unique characteristics and signatures left behind by specific packers. Detecting a packer is often the first step in dynamic analysis, as it tells the analyst that the file must be unpacked before the true malicious code can be examined.