REVERSING LABS

Hack In The Box 2012, Amsterdam

TitanEngine 3.0 – Return of the Titan and the exile of PE malformation

AGING BUSINESS OF SECURITY

Maturing Code

- PE has been on Windows for 18 Years now
- Optional features
- Backward compatibility
- Deprecated functionality
- Allowed values
- Point release and bug fixes

Multiple Specifications

Negative Testing

SDLC

SOFTWARE DOCUMENTATION

Always behind

Incorrectly translated

Inaccurate by design

- Developers are asked how should spec function?
- They may not remember how it functions

Spirit of the release 1 year later? 5 years later?

Zero bugs = Perfectly documented

- Who bug fixes documentation?
- Who proof reads documentation for technical errors?

BRIEF HISTORY OF PECOFF

What is PECOFF?

- Microsoft migrated to the PE format with the introduction of the Windows NT 3.1 in 1993
- The Portable Executable (PE) format is a file format for executables, object code and DLLs, used in 32-bit and 64-bit versions of Windows operating systems
- The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code

Where can you find it?

- Microsoft Windows / Windows CE / Xbox
- Extensible Firmware Interface (EFI)
- ReactOS
- WINE

WHAT IS A MALFORMATION?

Malformations

- Malformations are simple or complex modifications
- File format data and/or layout are modified
- Unusual form is not inside the boundaries permitted by the file format documentation but is still considered valid from the standpoint of tools that parse them.
- Malformation purpose is either breaking or omitting tools from parsing the malformed format correctly.

Simple malformations

- Require single field or data table modifications

Complex malformations

- Require multiple fields or data tables modifications

WHAT DOES IT AFFECT?

Security consequences

Malformations can have serious consequences

- Breaking unpacking systems
- Remote code execution
- Denial of service
- Sandbox escape

PE file format validation is hard!

- Due to its complexity many things can work in multiple ways achieving the same result
- Backward compatibility is very important and even though operating system loader evolves it still has to support obsolete compilers and files that are most definitely not compliant with the PECOFF docs

PE MALFORMATIONS

Previous published work on the PE subject

Constant Insecurity – Pericin/Vuksan [BH LV 2011]

PE Specification vs PE Loader - Alexander Liskin [SAS 2010]

PE Format as Source of Vulnerability - Ivan Teblin [SAS 2010]

Doin' The Eagle Rock - Peter Ferrie, Virus Bulletin, March 2010

Fun With Thread Local Storage (part 3) - Peter Ferrie, July 2008

Fun With Thread Local Storage (part 2) - Peter Ferrie, June 2008

Fun With Thread Local Storage (part 1) - Peter Ferrie, June 2008

TITANENGINE

Open source library for PE file processing

- Version 1.0
  - Historic version, purely dynamic file processing centered
- Version 2.0
  - Presented at BlackHat USA 2009
  - Total rewrite from ASM to C
  - Many improvements in the field of dynamic file processing
- Version 3.0
  - Presenting here at Hack In the Box 2012
  - Total rewrite to C++
  - Purely static file processing centered

TITANENGINE 3.0

Made with the following problems in mind

- Processing strange, malformed and damaged PE files
- Detecting malformations and damaged files
- Repairing damaged files in file preprocessing
- Extremely quick PE file processing

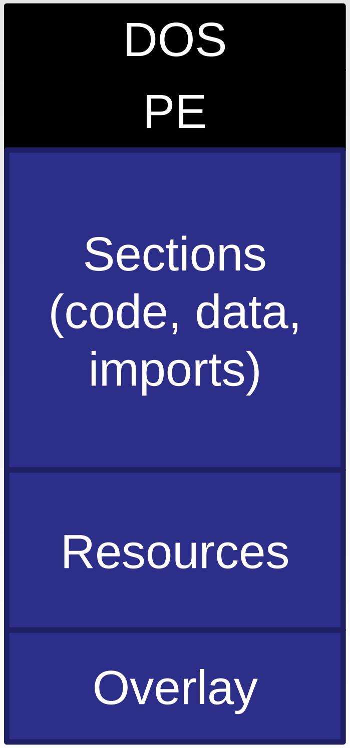
Full support for static file processing

- Easy to use interface for data reading/writing
- Large number of decompression algorithms included
- Ability to generate dynamic decrypters on the fly
- Ability to revert import name hashes back to strings

SIMPLE PECOFF MALFORMATIONS

GENERAL PE FORMAT LAYOUT

PE file format layout



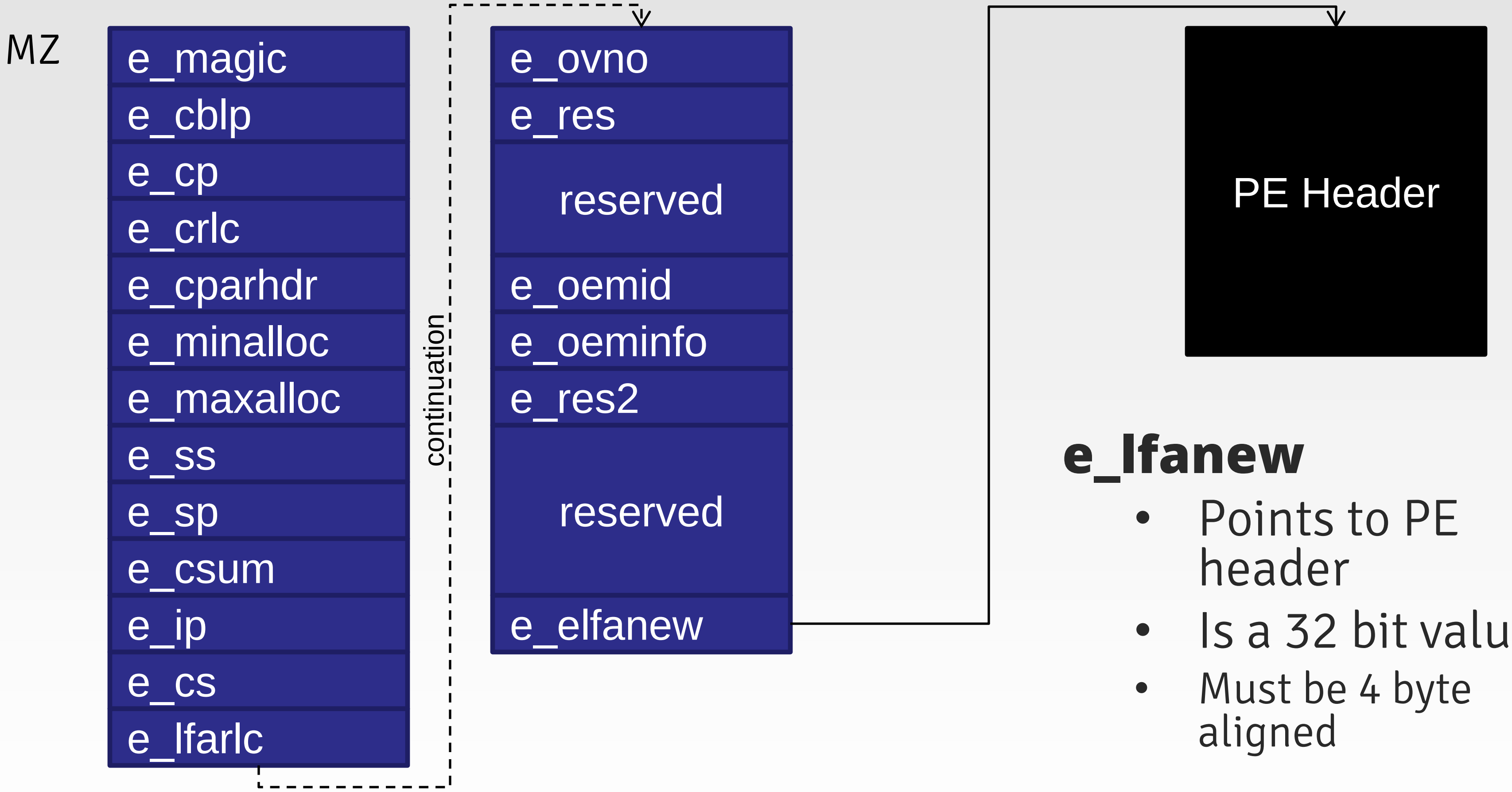
Traditional layout

Top level description

- **DOS header**
  - “MZ” & e_lfanew
- **PECOFF header**
  - COFF file header
  - Optional header
- **Sections**
  - Code, data, imports, exports, resources…
- **Overlay**
  - Appended file data

DOS HEADER

DOS header layout



MZ

e_magic
e_cblp
e_cp
e_crlc
e_cparhdr
e_minalloc
e_maxalloc
e_ss
e_sp
e_csum
e_ip
e_cs
e_lfarlc

continuation

e_ovno
e_res
reserved
e_oemid
e_oeminfo
e_res2
reserved
e_elfanew

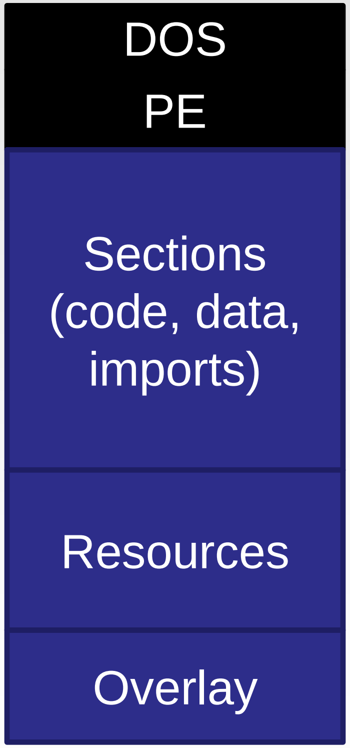
PE Header

e_lfanew

- Points to PE header
- Is a 32 bit value
- Must be 4 byte aligned

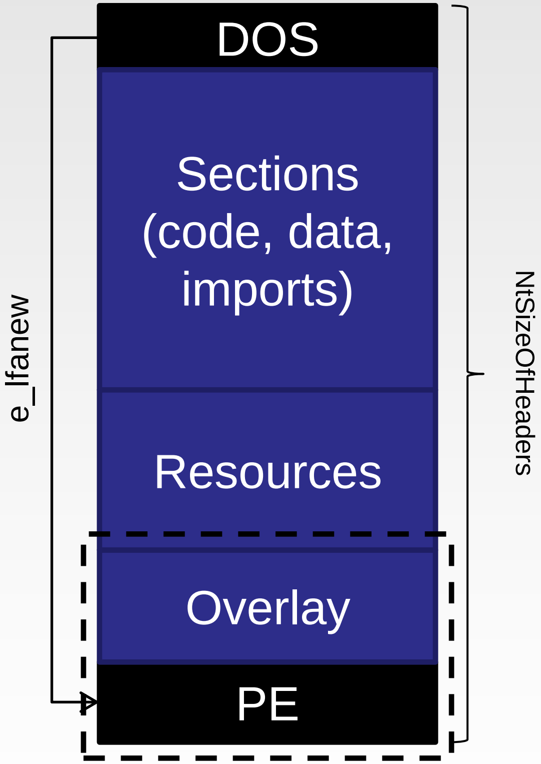
DOS HEADER | E_LFANEW

PE file format layout



Traditional layout

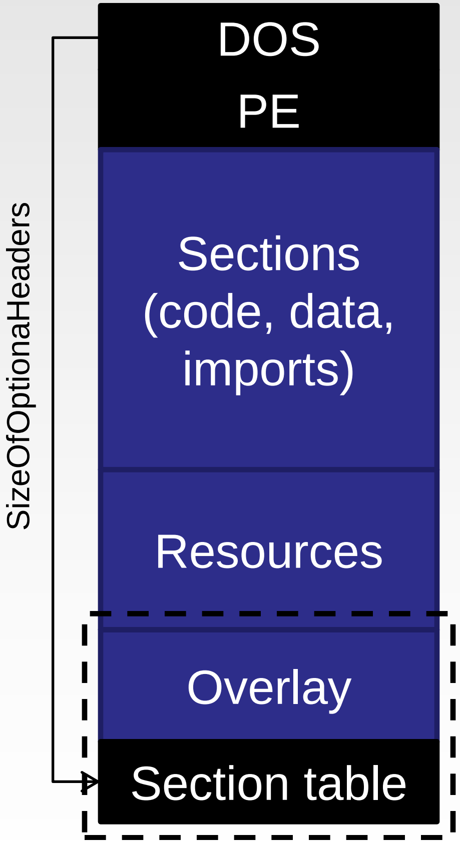
PE file malformation



PE HEADER | SIZEOFOPTIONALHEADERS

demo

PE file format layout



SizeOfOptionalHeaders

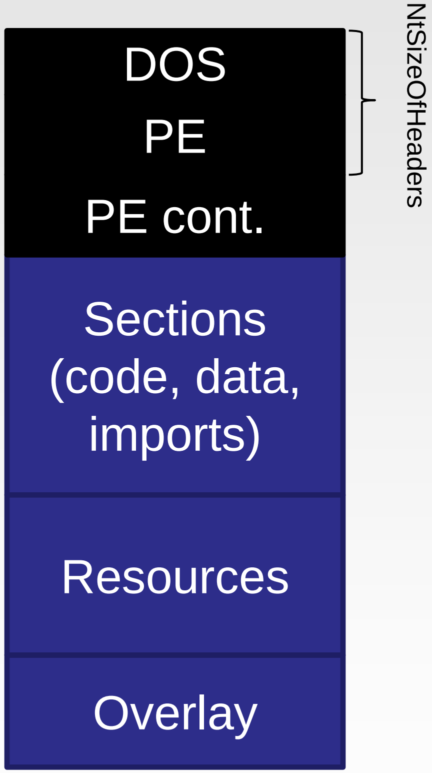
The size of the optional header, which is required for executable files but not for object files.

Issues with SizeOfOptionalHeaders

Since the field that allows us to move the section table is a 16 bit field the maximum distance that we can move the table is just 0xFFFF. This doesn't limit the maximum size of the file as the section table doesn't need to be moved to the overlay for this to work, just the region of physical space which isn't mapped in memory.

PE HEADER | NTSIZEOFHEADERS

PE file format layout



NtSizeOfHeaders

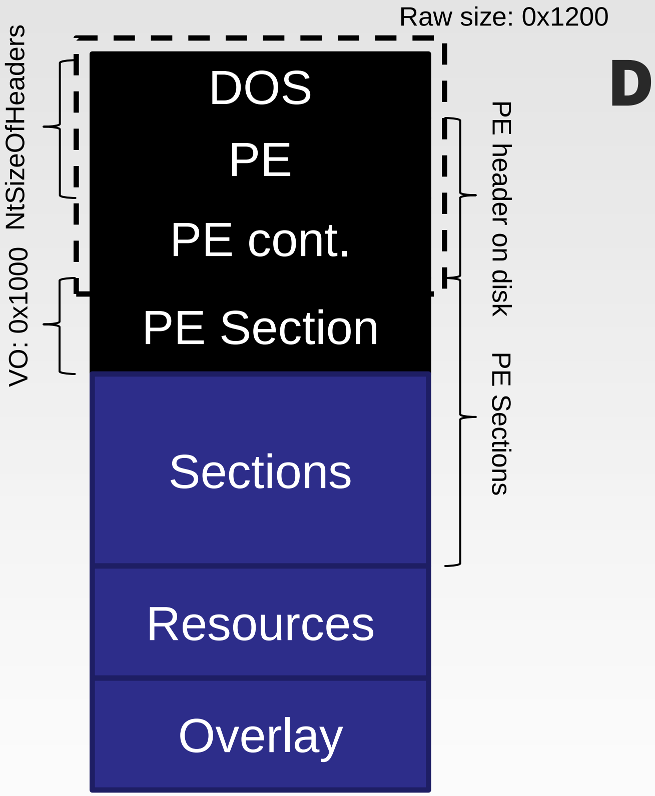
- Is meant to determine the PE header physical boundaries
- It also implicitly determines the virtual start of the first section

Issues with NtSizeOfHeaders

- It isn’t rounded up to FileAlignment
- Only the part of the PE header up until and including FileAlignment field needs to be inside the specified range
- Regardless of the specified header size the rest of the header is processed from disk
  - But not all of it!

PE HEADER | NTSIZEOFHEADERS

PE file malformation



Dual PE header malformation

e_lfanew : 0xF80

NtSizeOfHeaders : 0x1000

Effectively truncating part of the PE header containing data tables

FirstSectionRO: 0x1200

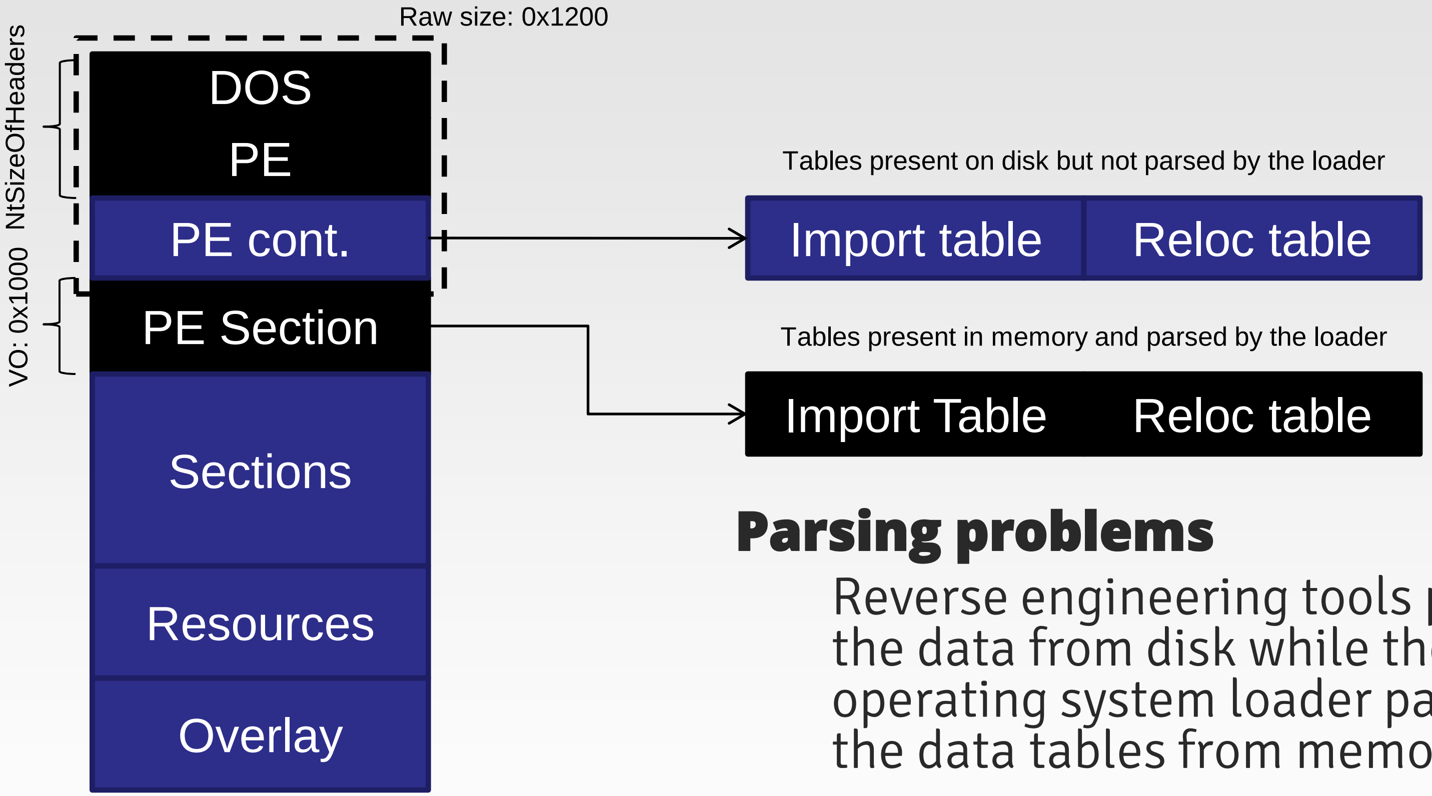
FirstSectionVO: 0x1000

At the start of the section we store the continuation of the PE header containing data tables (e.g. imports are different and parsed from memory and not from disk by the loader)

PE HEADER | DUAL DATA TABLES

demo

PE file malformation

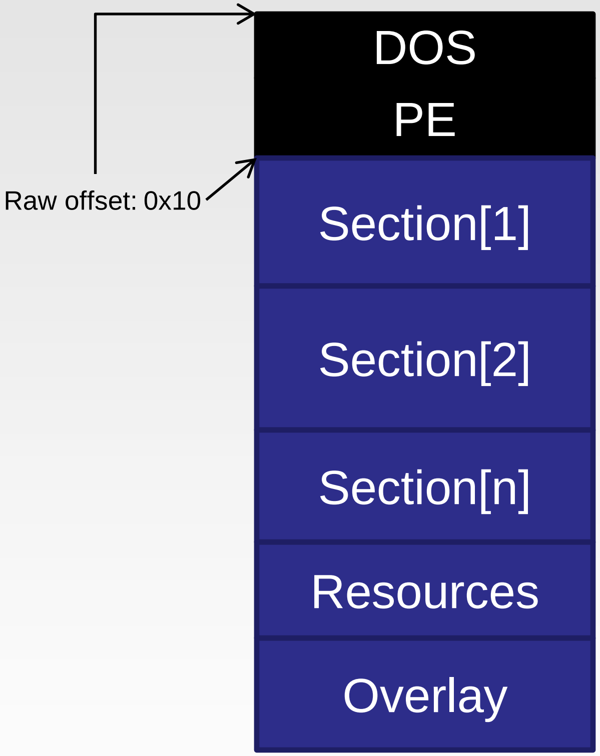


Parsing problems

Reverse engineering tools parse the data from disk while the operating system loader parses the data tables from memory.

PE HEADER | FILE ALIGNMENT

PE file malformation



nSPack

FileAlignment

The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512.

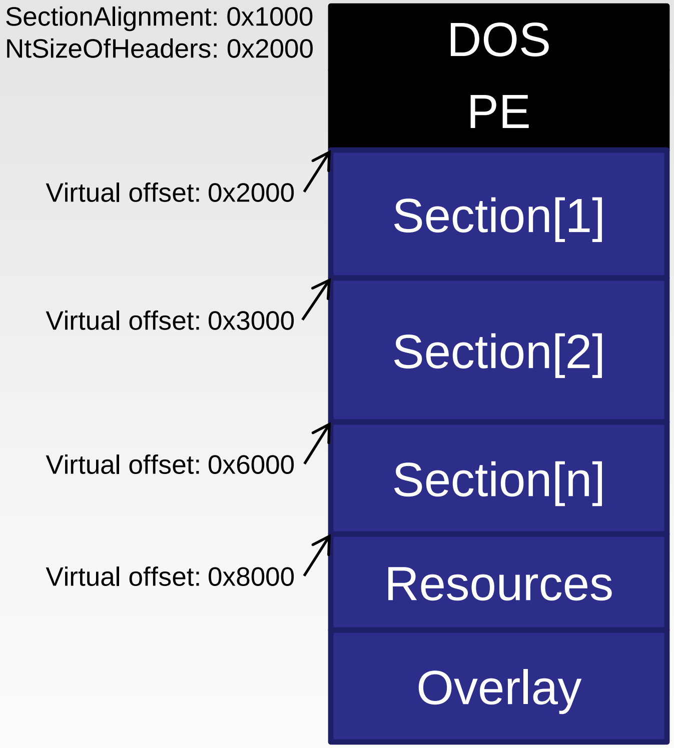
FileAlignment issues

Because of the conditions set by the PECOFF documentation whose excerpt is stated above we can safely assume that the value of FileAlignment can be hardcoded to 0x200.

Raw start of the sections is calculated by the formula (section_offset / 0x200) * 0x200

PE HEADER | SECTION ALIGNMENT

PE file layout



SectionAlignment

SectionAlignment is the alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture or a greater value which is the multiplier of the default page size.

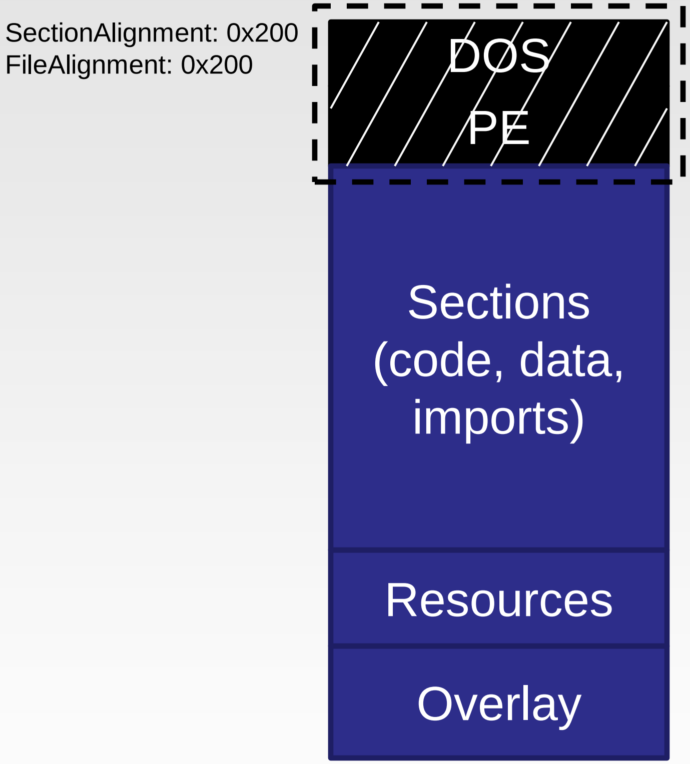
SectionAlignment issues

While every section must start as the multiplier of SectionAlignment the first section doesn't always start at the address which is equal to the value of SectionAlignment. Virtual start of the first section is calculated as the rounded up SizeOfHeaders value. That way header and all subsequent sections are committed to memory continuously with no gaps in between them.

PE HEADER | WRITABLE HEADERS

demo

PE file layout



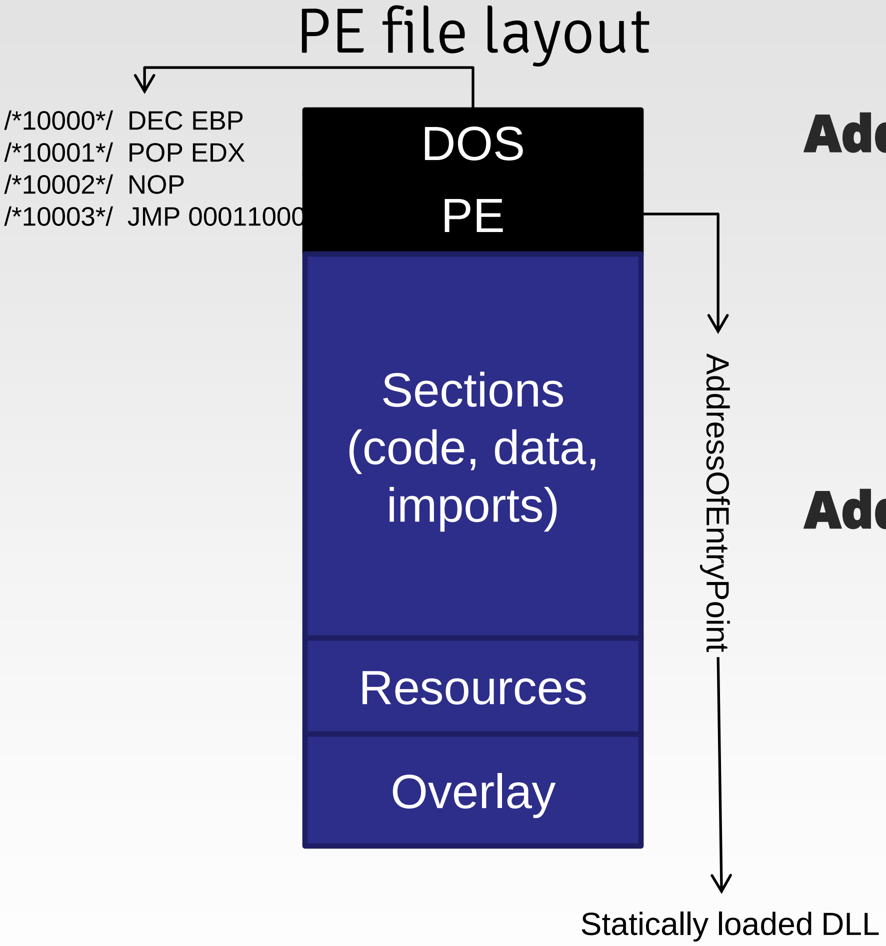
DOS/PE headers

By default the PE header has read and execute attributes set. If DEP has been turned on the header has read only attributes.

SectionAlignment / FileAlignment issues

If the values of FileAlignment and SectionAlignment have been set to the same value below 0x1000 the header will become writable. Typical value selected for this purpose is 0x200.

PE HEADER | ADDRESSOFENTRYPOINT



PE file layout

```
/*10000*/  DEC EBP
/*10001*/  POP EDX
/*10002*/  NOP
/*10003*/  JMP 00011000
```

AddressOfEntryPoint

The address of the entry point is relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.

AddressOfEntryPoint issues

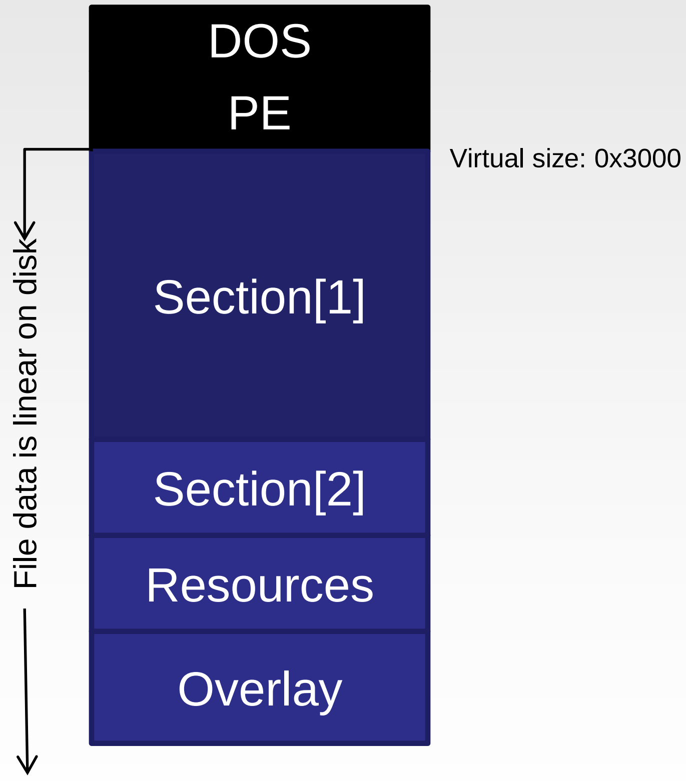
This excerpt from the PECOFF documentation implies that the entry point is only zero for DLLs with no entry point and that the entry point must reside inside the image. Neither of these two statements is true.

PE HEADER | SECTION DATA

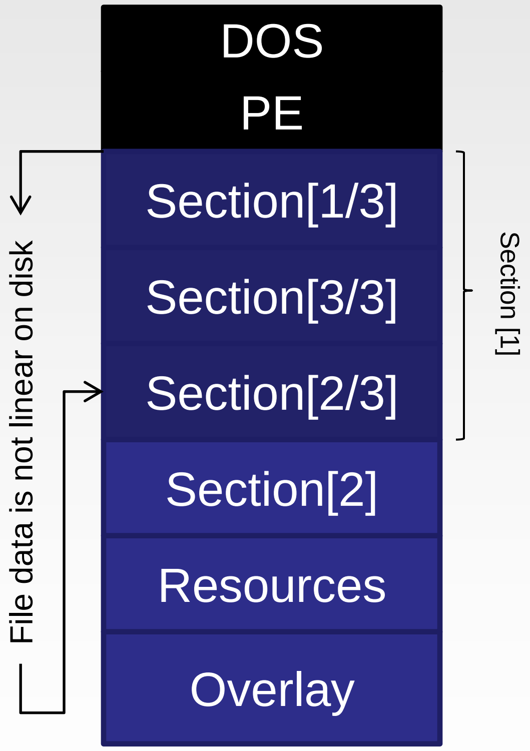
Layout problem with writing static unpackers

demo

PE file disk layout



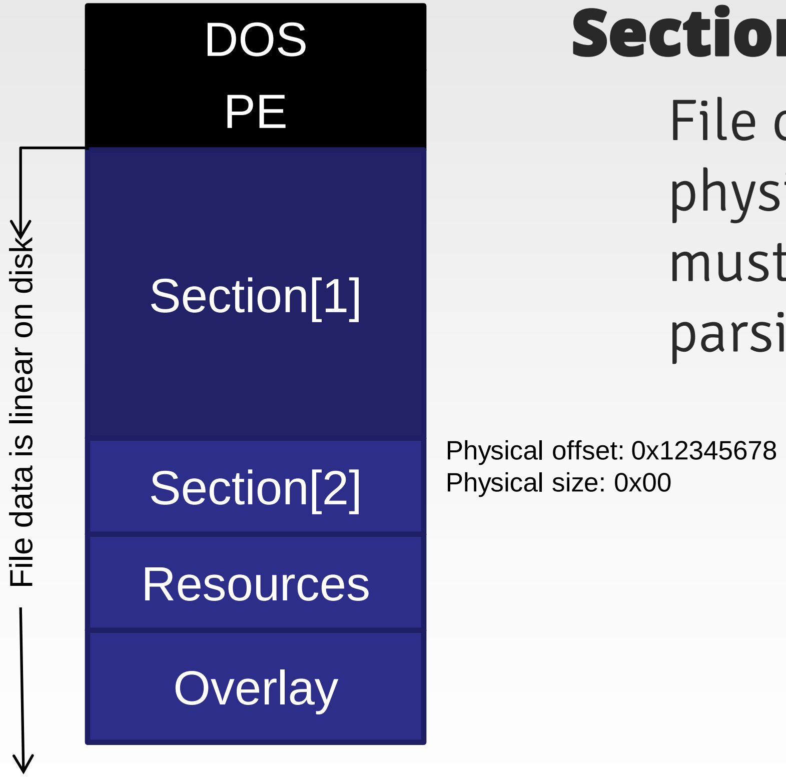
Section data shuffling



PE HEADER | SECTION DATA

demo

PE file disk layout

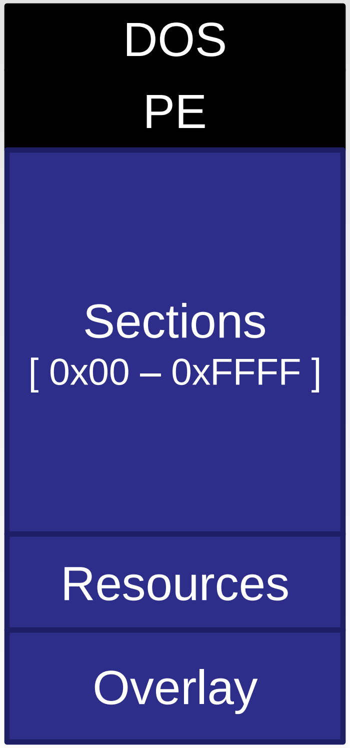


Section data

File can have sections that physically do not exist on disk. This must be taken into account when parsing and validating PE images.

PE HEADER | SECTION NUMBER

PE file layout



SectionNumber

PE files have arbitrary section numbers; however it is assumed that the number of possible sections that a file can consist of is within a range from one to 96 as stated by the PECOFF documentation.

SectionNumber issues

The latest implementations allow for this limit to be expanded to the range from zero sections to the maximum value allowed by the 16 bit field SectionNumber which is 0xFFFF.

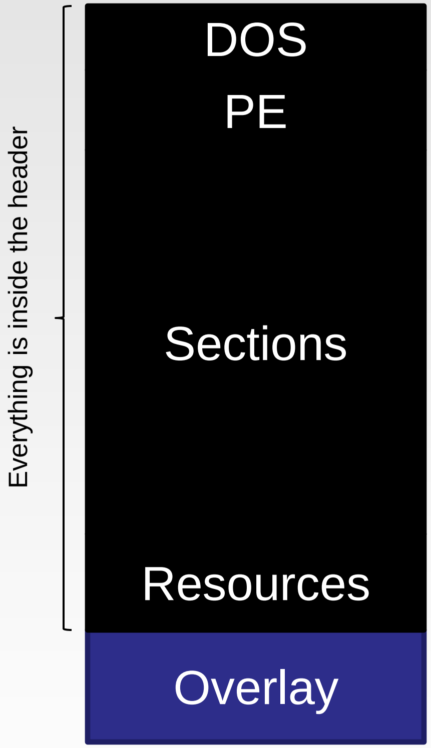
Huge number of sections is problematic for many reverse engineering and security tools

No sections is even more problematic!

PE HEADER | SECTION NUMBER

demo

Zero section PE file layout



Making a zero section file

- File must be converted to flat memory model in which all relative virtual addresses are equivalent to their physical counterparts
- Section table is removed and the number of section is set to zero
- NtSizeOfHeaders is set to the physical size of the mapped memory
- NtSizeOfImage is set to equal or grater value than NtSizeOfHeaders
- FileAlignment and SectionAlignment are set to same value 0x200 to make the header writable

TITANENGINE 3.0

Features

- Static PE file format processing functionality
  - Ability to read, modify and create new PE files
  - Ability to read, modify and create individual PE tables
- Support for decompressing large number of formats
- Support for building custom dynamic decrypters
- Support for import hash to original name reverting
- PE file format validation, malformation detection, damage assessment and recovery

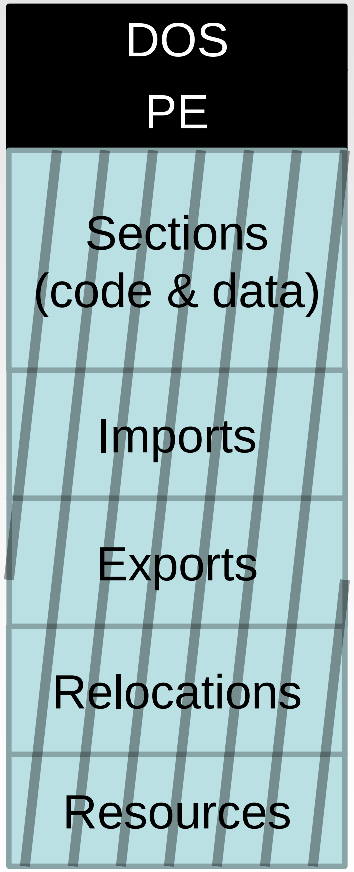
Workshop package download

http://www.reversinglabs.com/download/HITB.zip

CREATING A NEW PE32 FILE

demo

PE file format layout



Memory data layout

Creating a new PE32 file

titan_create_file API is used to create a new PE32/PE32+ file in memory. Once created this file can be filled with code and PE tables that link to that code. Additionally overlay data can be appended to the end of the file.

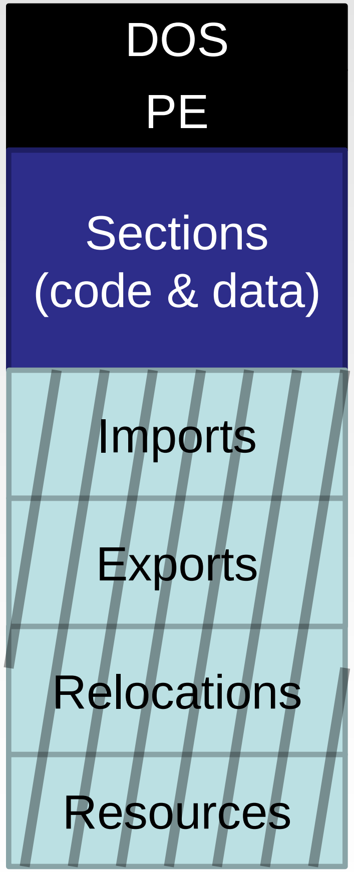
No sections exist at this time and they must be added before storing data at that location.

Default PE header can be accessed and the parameters can be changed at any time.

ADDING A CODE & DATA SECTIONS

demo

PE file format layout



Memory data layout

Adding a code section

titan_add_new_section API is used to create a section inside the PE header. Initially section can have any size. Based on the data inside the section its physical size is reduced to a minimum aligned to FileAlignment.

Last section can always be increased by writing past its end but writing must start with the current section limits.

titan_set_content API is used to write data to any part of the PE file.

titan_set_pe_header API is used to update the PE header data. Once we write data to our newly created section we want to move the AddressOfEntry point to the start of our code section.

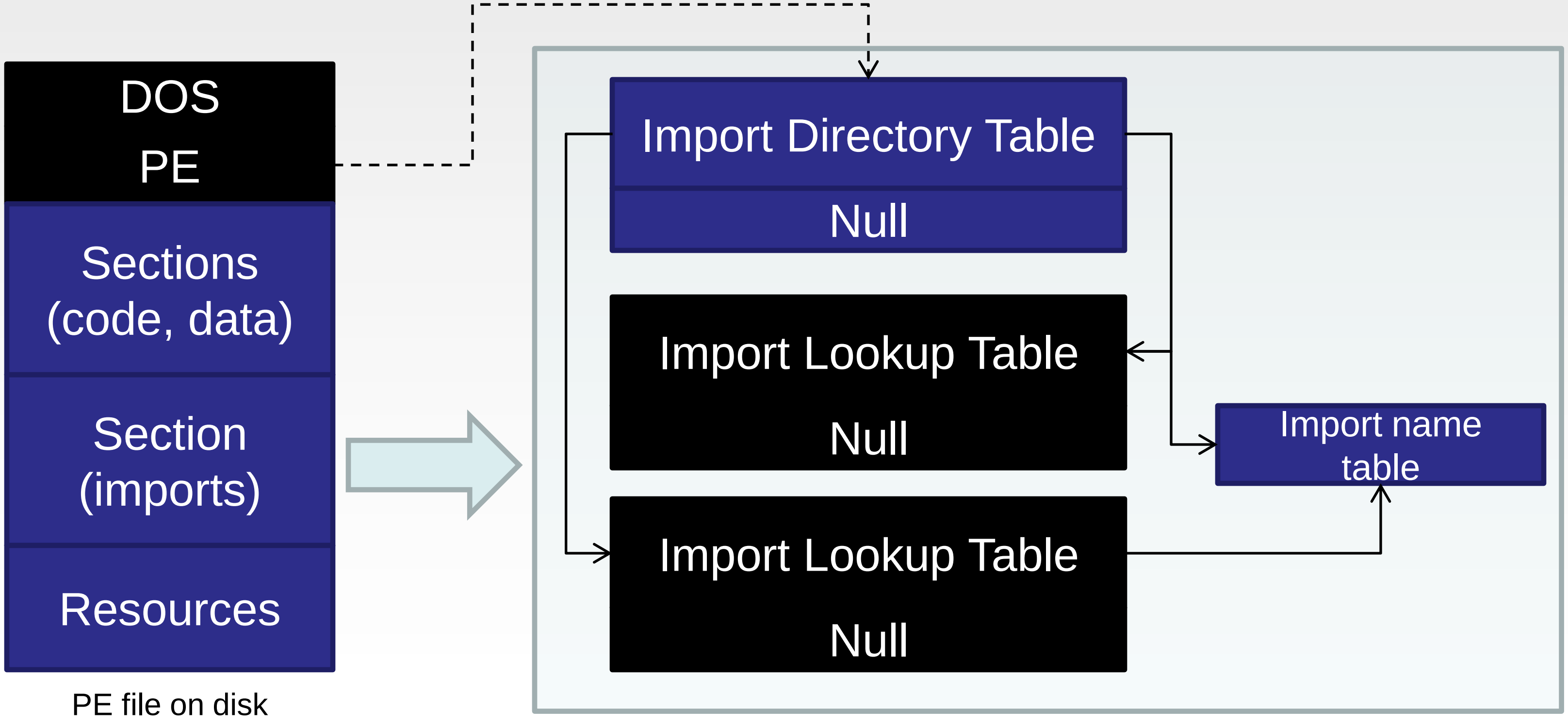
COMPLEX PECOFF MALFORMATIONS

PE HEADER | IMPORT TABLE

Import table overview

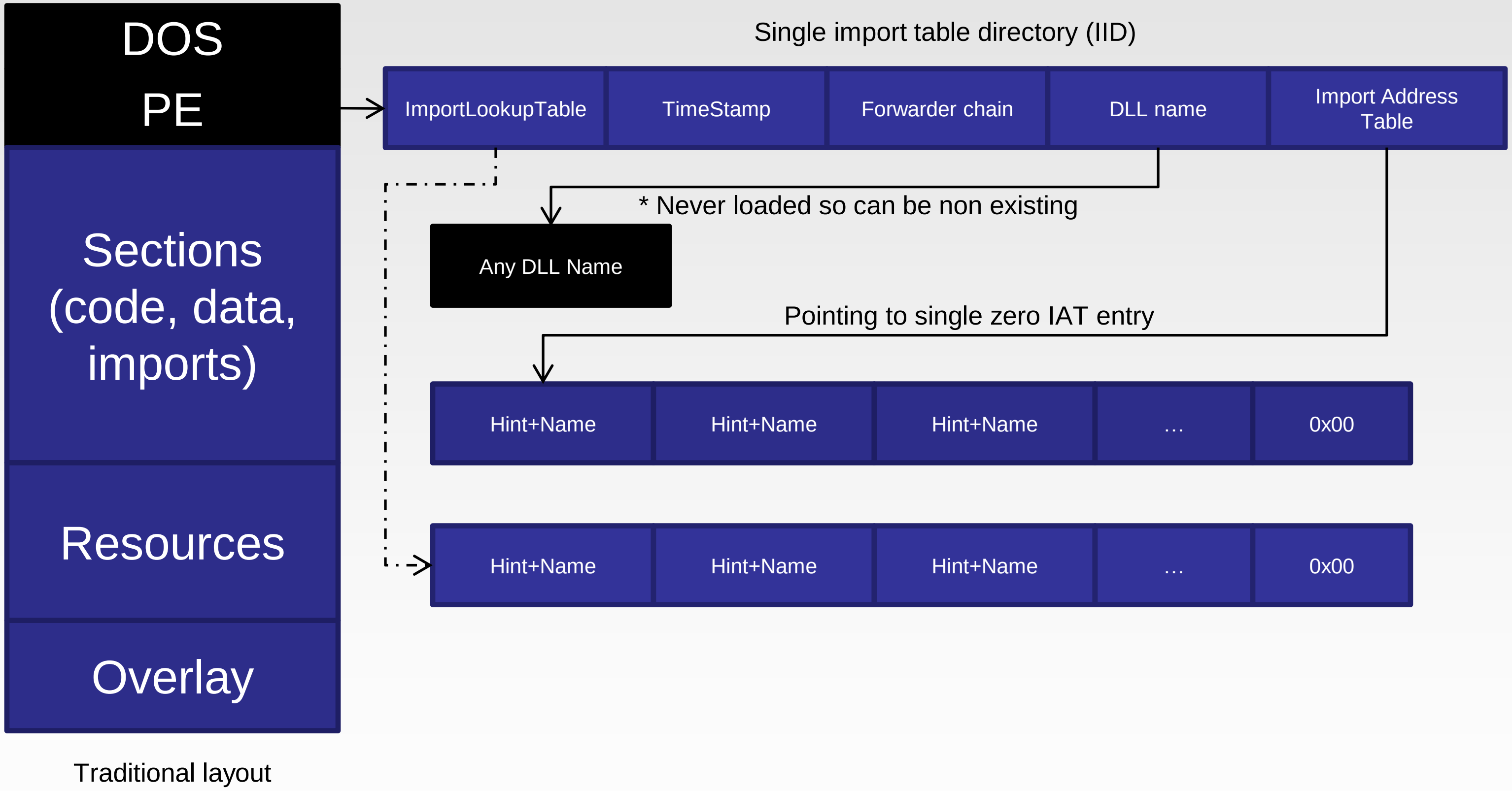
PE files that import symbols statically have an import table

Import table consists of names of dynamic link libraries and function names and/or function ordinal numbers



PE HEADER | IMPORT TABLE

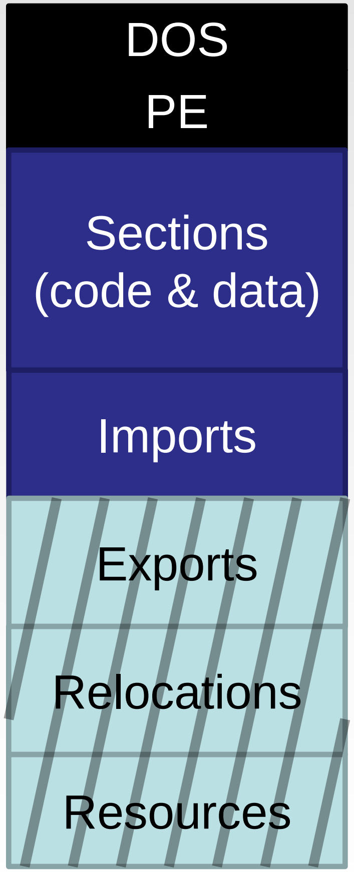
Dummy import table entries



ADDING AN IMPORT TABLE

demo

PE file format layout



Memory data layout

Adding an import table

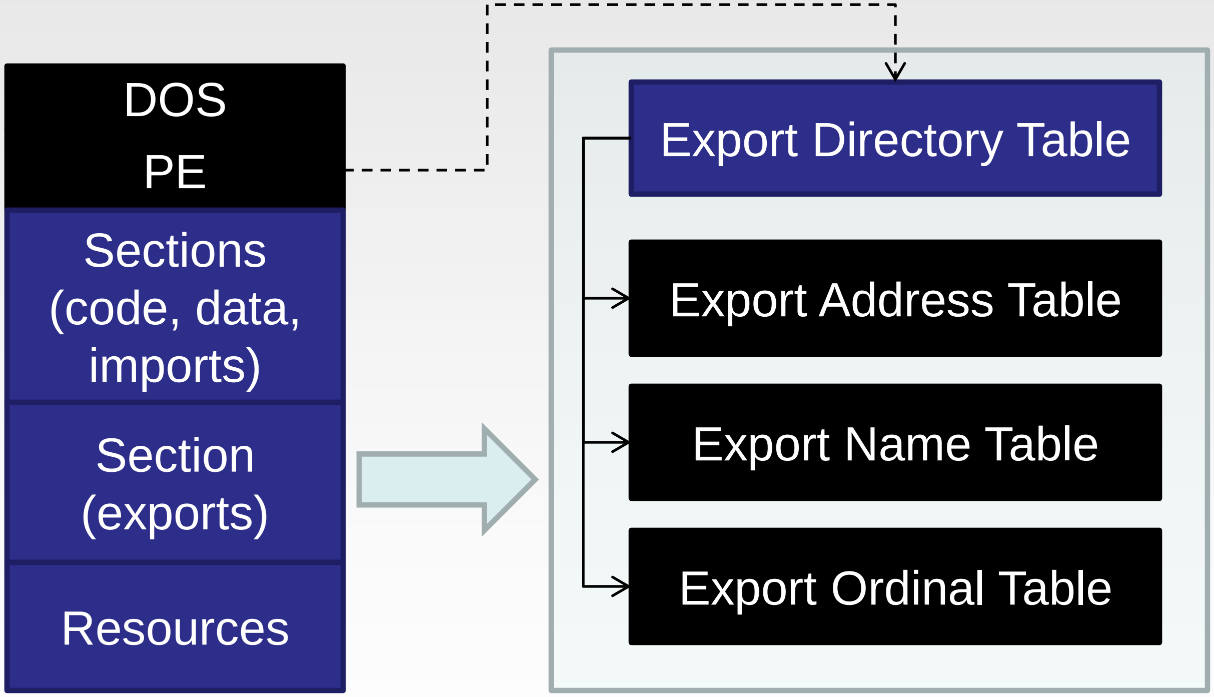
- **titan_add_import_library** API is used to add new imported DLL file.
- **titan_add_import_function** API is used to add APIs to all DLLs added with **titan_add_import_library**.
- **titan_add_stolen_import_info** API is used to connect the calls and jumps within the code section with the IAT which is yet to be created. This is optional and only used because we chose to add import table data before creating a section that will hold the IAT.
- **titan_write_import_table** API is used to write the import table data we pushed to the engine to the specified location. For this PE data table we added a new section.

PE HEADER | EXPORT TABLE

Export table overview

PE files can also export symbols that other PE files import

PE files can export functions and variables

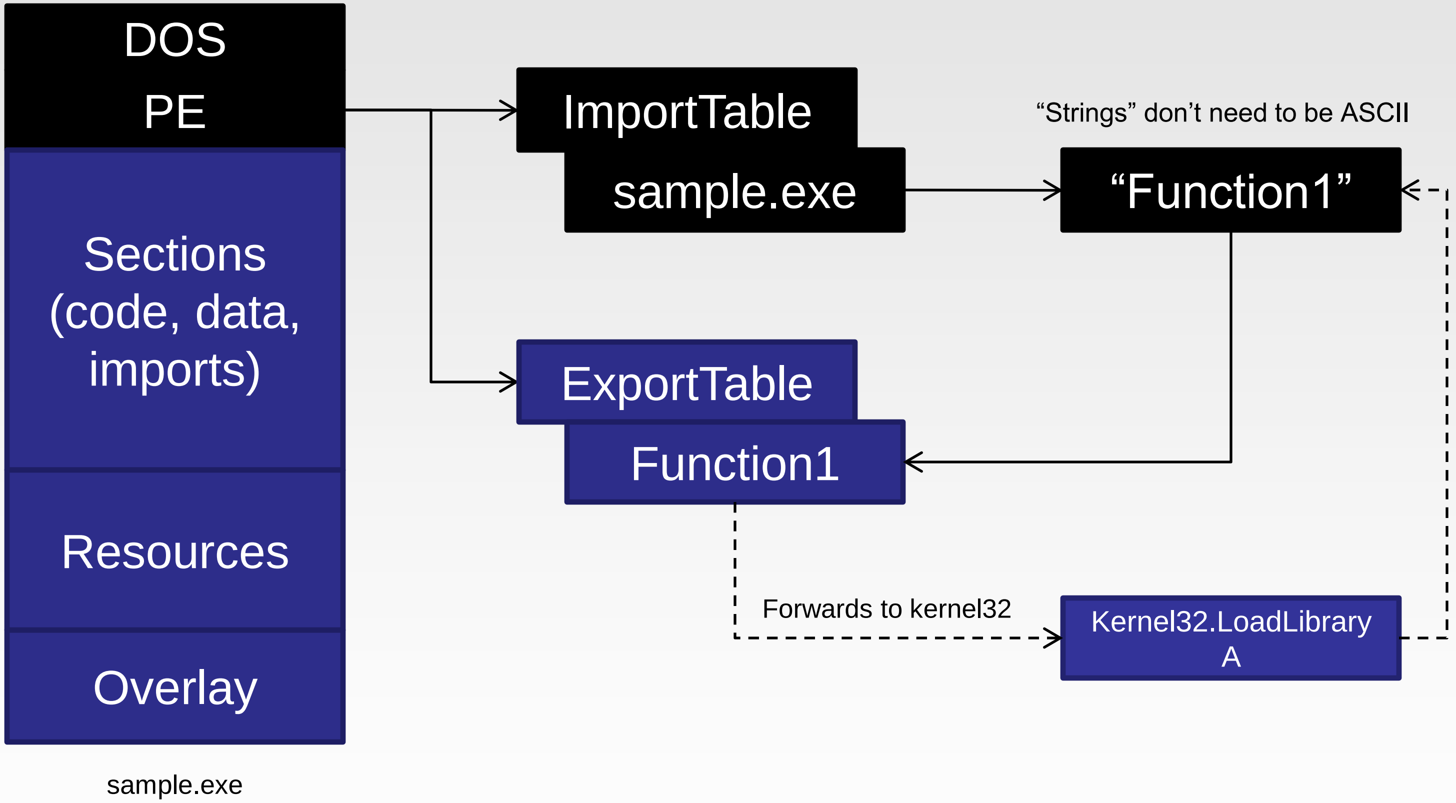


PE file on disk

PE HEADER | IMPORT & EXPORT TABLE

demo

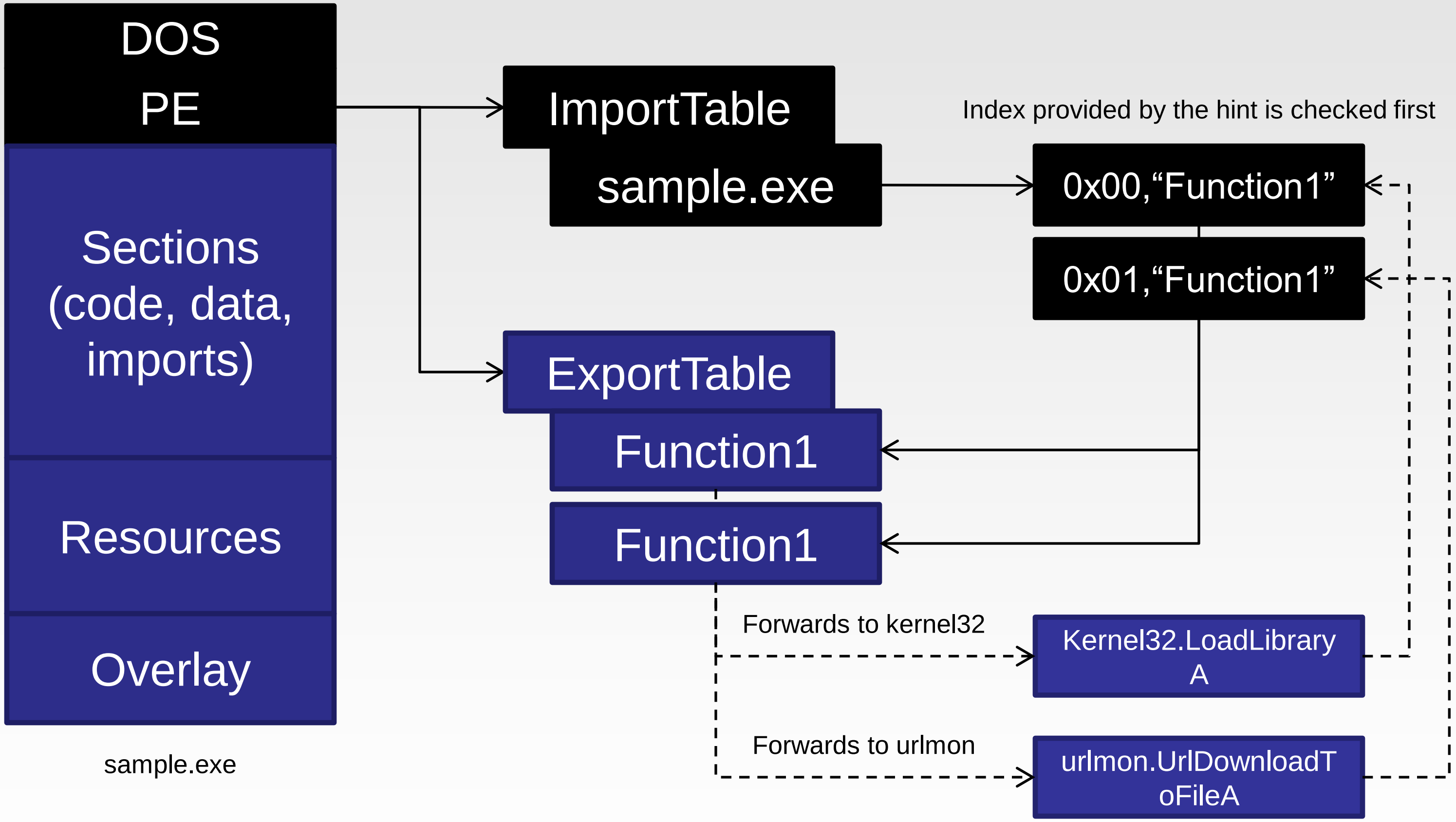
Import obfuscation



PE HEADER | IMPORT & EXPORT TABLE

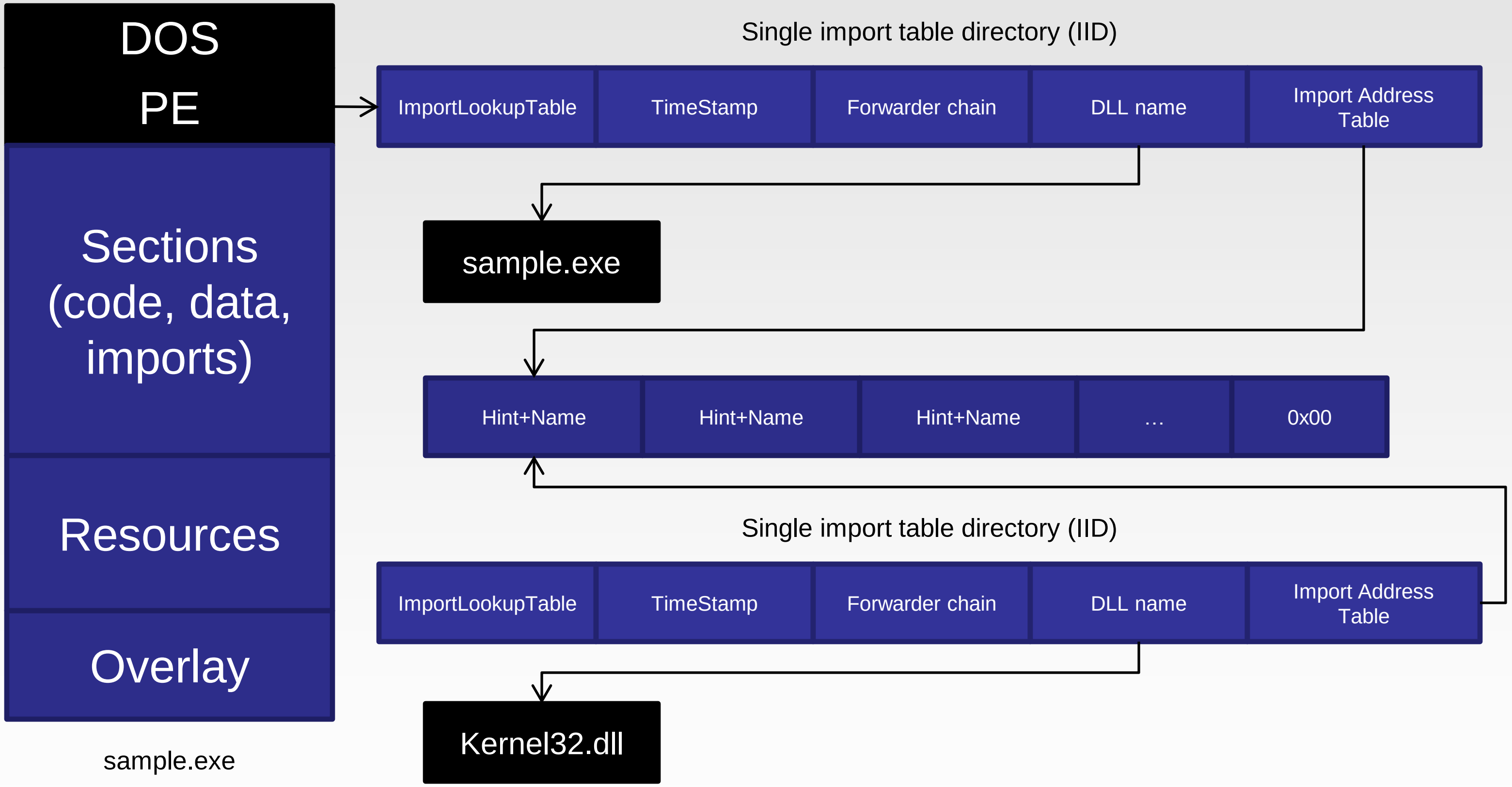
demo

Import obfuscation with hint



PE HEADER | IMPORT & EXPORT TABLE

Rebuilding data with exports

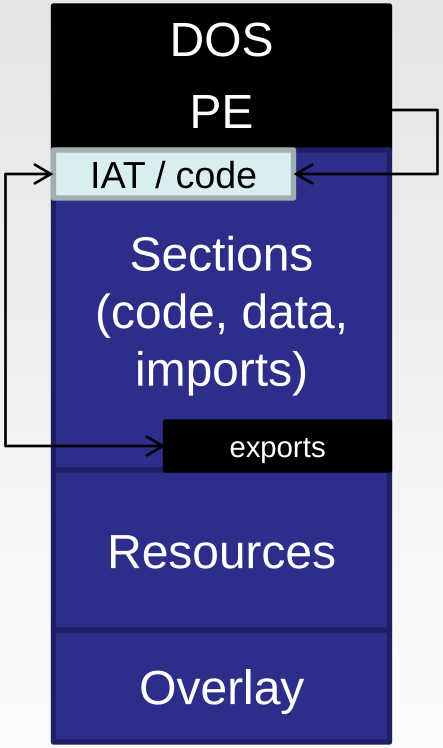


sample.exe

PE HEADER | IMPORT & EXPORT TABLE

demo

Rebuilding code with exports



sample.exe

Rebuilding code from exports

File imports functions from its own export table.

Export table doesn’t hold the valid pointers, it holds data that will be written to the import table.

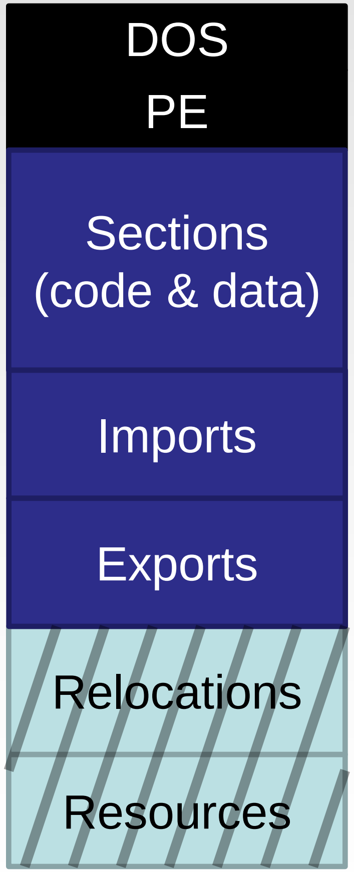
Import table pointers are stored at the original code location (e.g. entry point)

Once file is loaded its import table is filled with the original code which in turn executes after that normally.

ADDING AN EXPORT TABLE

demo

PE file format layout



Memory data layout

Adding an export table

titan_init_export_data API is used to set basic export table parameters such as the ordinal base the and module name.

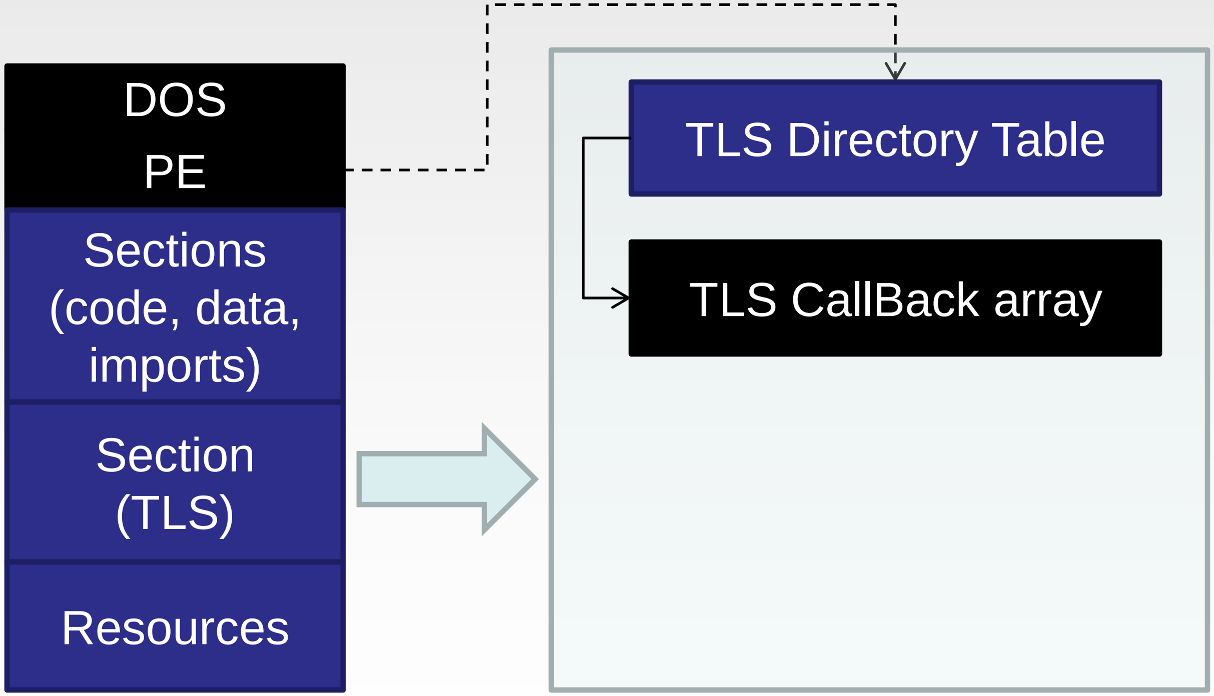
titan_add_export_function API is used to add new exported functions to the export table. Forwarders can also be added with a separate API.

titan_write_export_table API is used to write the export table data we pushed to the engine to the specified location. For this PE data table we added a new section.

PE HEADER | TLS TABLE

Thread local storage table overview

TLS is a special storage class that Windows support in which a data object is not an automatic (stack) variable, yet is local to each individual thread that runs the code. Thus, each thread can maintain a different value for a variable declared by using TLS.

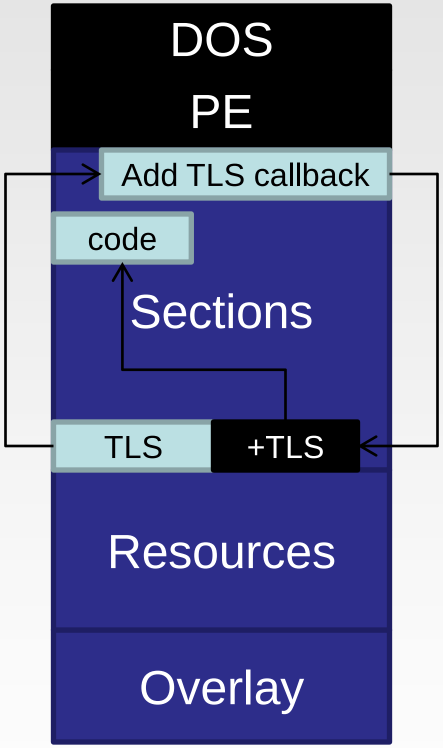


PE file on disk

PE HEADER | TLS TABLE

demo

Dynamic callbacks



Dynamic callback table generation

TLS callback array is processed from memory so it is possible that its content is modified from the first callback.

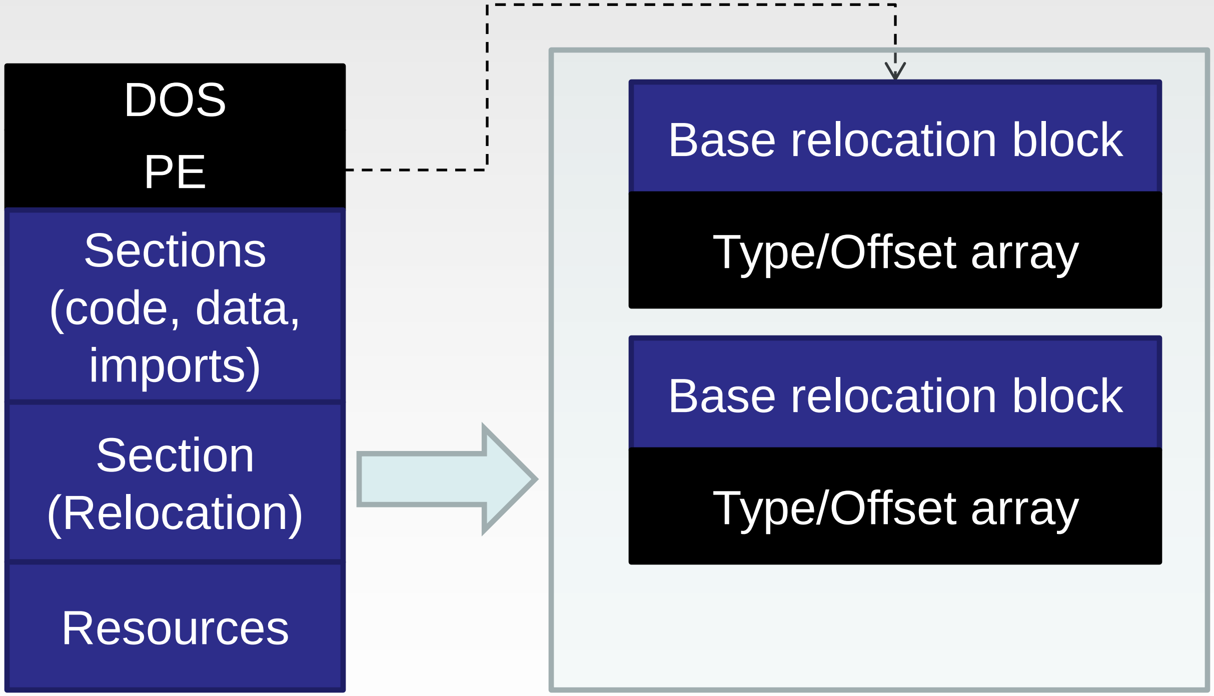
TLS callback array can be overlapped with import table so that code which gets executed is outside image.

TLS callback array can be overlapped with linked import & export table so that the executed code is still in the same image.

PE HEADER | RELOCATION TABLE

Relocation table overview

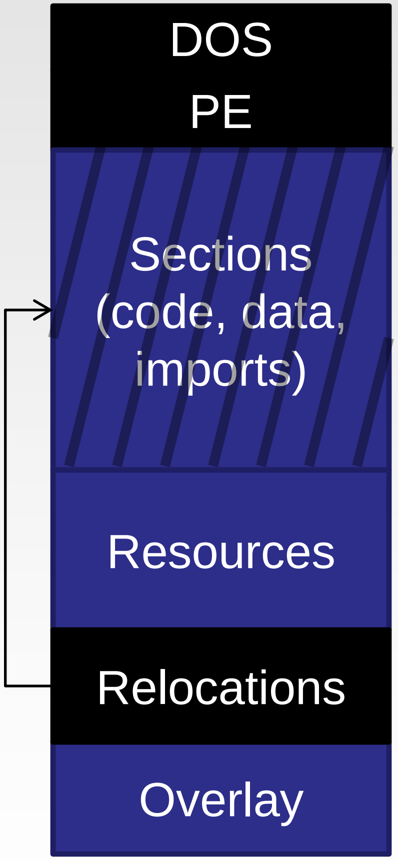
Base relocation table is used by the operating system loader to rebase the file in memory if the PE file needs to load on the base address which is different from its default one which is specified by the ImageBase PE header field.



PE file on disk

PE HEADER | RELOCATION TABLE

Decryption via relocations



Decryption via relocations

To be able to decrypt the content correctly the file always needs to be loaded through relocation process on the same base address. That way the decryption key wont change and the data will be decrypted correctly every time.

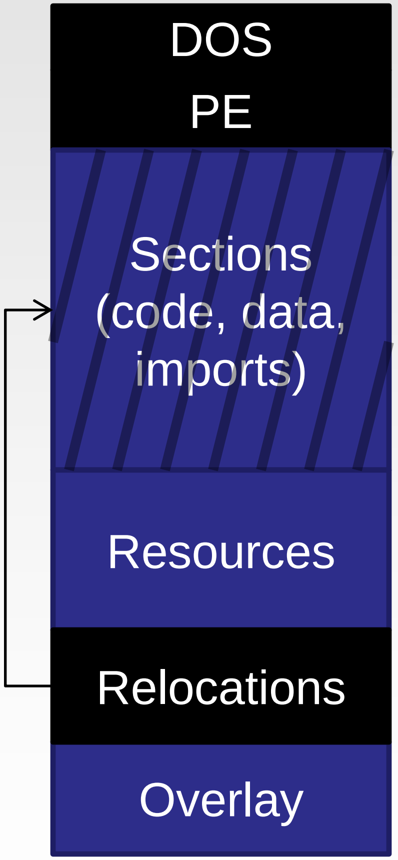
Pre Windows 7 SP1: If the file has an ImageBase 0x00 it will always be loaded on the base address 0x10000.

Post Windows 7 SP1: If the file has a base address inside kernel memory it will always be loaded on the base address 0x10000.

PE HEADER | RELOCATION TABLE

demo

Decryption via relocations



Decryption via relocations

Every byte of selected section is encrypted with forward addition encryption. The value added is the value that the operating system loader will subtract when relocating the file.

New relocation table is created with four entries per page so that decryption is performed for every byte in reverse.

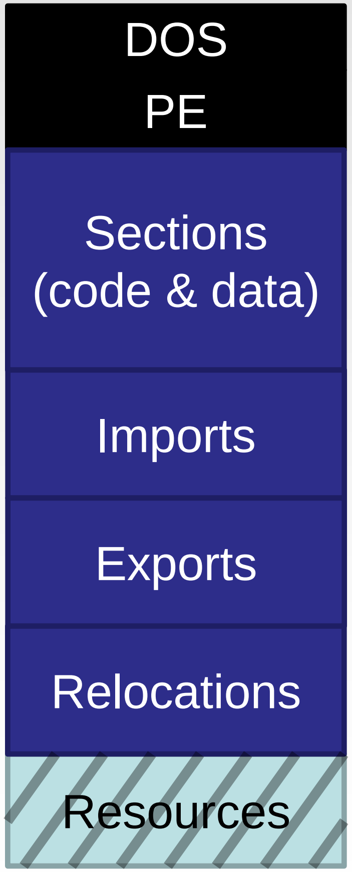
Every DWORD inside the selected section is processed four times.

Scary? First malware (LeRock) using it was detected last year. Its behavior was described by Peter Ferrie in VirusBulletin.

ADDING A RELOCATION TABLE

demo

PE file format layout



Memory data layout

Adding a relocation table

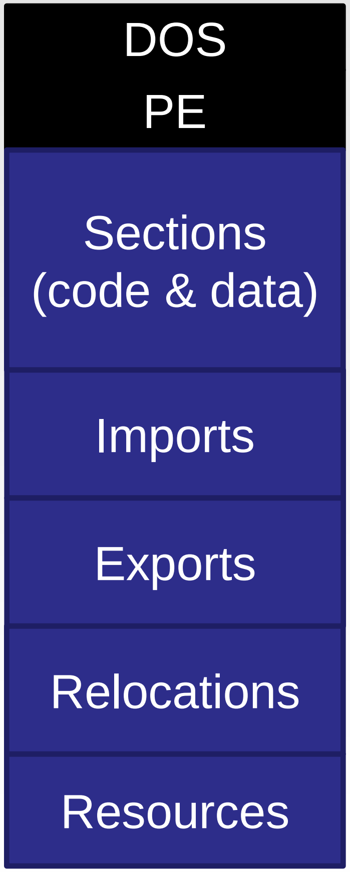
titan_add_base_relocation API is used to add addresses from code and data sections which need to be relocated. Optionally the engine can relocate these addresses while its rebuilding the relocation table. This can be used if the relocations have not yet been applied to the specified address.

titan_write_relocation_table API is used to write the relocation table data we pushed to the engine to the specified location. For this PE data table we added a new section.

ADDING A RESOURCE TABLE

demo

PE file format layout



Memory data layout

Adding a resource table

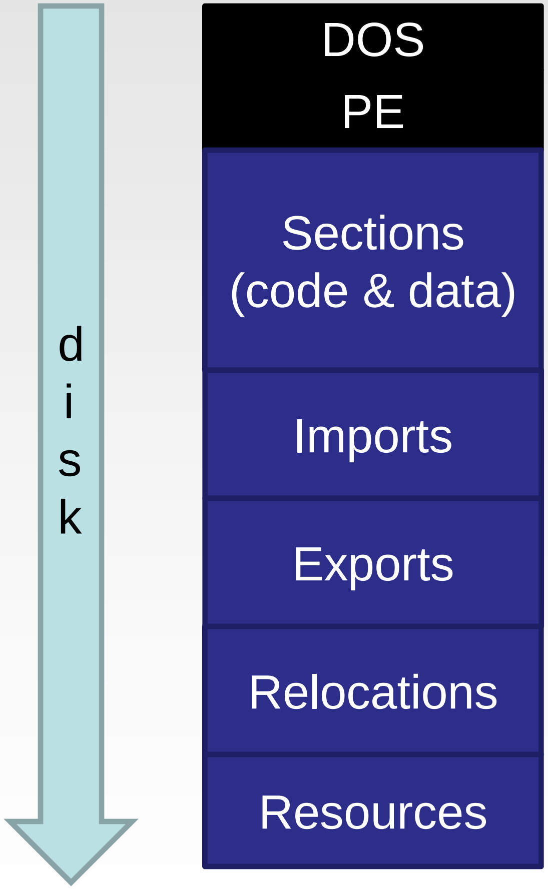
titan_add_resource_data API is used to add new resources to the file. Every resource is defined with its name, type, language, code page and data. Based on this data the resource tree is constructed.

titan_write_resource_table API is used to write the resource table data we pushed to the engine to the specified location. For this PE data table we added a new section.

EXPORTING THE PE32 FILE TO DISK

demo

PE file format layout



Memory data layout

Exporting the created file

titan_export_file API is export the current state of the PE header and the sections from memory to disk. When exported file is reconstructed and its section content physical size is minimized so that sections with no data take-up no space on disk. Sections which have data will be scanned from the back for the first non NULL byte. That size is then aligned with FileAlignment and used to write the data on disk.

DETECTING MALFORMATIONS

DETECTING MALFORMATIONS

- **PE file validations**
  - **Headers**
    - Disallow files which have headers outside the NtSizeOfHeaders
    - Disallow files which have too big NtSizeOfOptionalHeaders field value
    - Disallow files which have entry point outside the file
  - **Sections**
    - Disallow files with zero sections
  - **Imports**
    - String validation
    - Disallow table reuse and overlap
  - **Exports**
    - Disallow multiple entries with the same name
    - Disallow entries which have invalid function addresses
  - **Relocations**
    - Block files which utilize multiple relocations per address
  - **TLS**
    - Disallow files whose TLS callbacks are outside the image

FINAL THOUGHTS

on PE file format malformations

PE is riddled with possibilities for malformation and we can’t always predict them all or design our tools to be aware of all of them

Malformations can lead to serious consequences such application crashes, buffer and integer overflows

Everyone implements their own PE parser which makes it impossible to say whether or not a product is affected by a malformation and if so by which ones

Unified document published by ReversingLabs is available at http://pecoff.reversinglabs.com and will help you test your product’s resilience to malformations (RL will maintain this document)

Validate_tool as a part of the TitanEngine 3.0 SDK can be used to validate PE files, detect damaged or malformed ones and optionally correct the detected damage if that is possible.

REVERSING LABS | Reverse Engineering & Software Protection

THANK YOU!

May 25, 2012