Undocumented PECOFF

REVERSING LABS

Contents

Overview

The Portable Executable (PE) format is a file format for executables, object code and DLLs. It is used in 32-bit and 64-bit versions of Windows operating systems. The term "portable" refers to format's versatility within numerous environments of operating system software architecture. The PE format is a data structure that encapsulates necessary information so that Windows OS loader can manage wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments.

PE is a modified version of the Unix COFF file format. PE/COFF is an alternative term in Windows development.

On Windows NT operating systems, PE currently supports the IA-32, IA-64, and x86-64 (AMD64/Intel64) instruction set architectures (ISAs). Prior to Windows 2000, Windows NT (and thus PE) supported the MIPS, Alpha, and PowerPC ISAs. Because PE is used on Windows CE, it continues to support several variants of the MIPS, ARM (including Thumb), and SuperH ISAs.

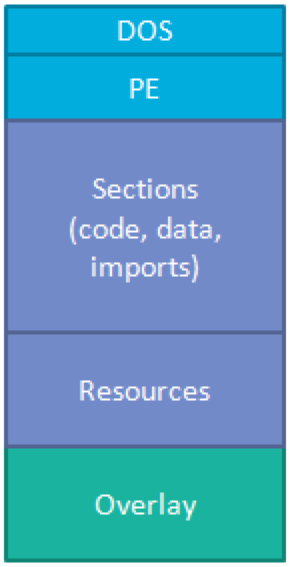
One constant challenge of modern security will always be the difference between published and implemented specifications. Evolving projects, by their very nature, open up a host of exploit areas and implementation ambiguities that cannot be fixed. As such, complex documentation such as that for PECOFF opens up a window of opportunity for misinterpretation and mistakes in parsing implementation.

This document will focus on all aspects of PE file format parsing that leads to undesired behavior or prevents security and reverse engineering tools from inspecting malformated files due to incorrect parsing. Special attention will be given to differences between PECOFF documentation and the actual implementation done by the operating system loader. With respect to these differences it will describe the existence of files that can't possibly be considered valid from a documentation standpoint but which are still correctly processed and loaded by the operating system. These differences and numerous design logic flaws can lead to PE processing errors that have serious and hardly detectable security implications. Effects of these PE file format malformations will be compared against several reverse engineering tools, security applications and unpacking systems.

Due to the nature of this document it is recommended that it's read in parallel with the official Microsoft PECOFF documentation.

Introduction

General Portable Executable (PE) format file layout can be described with the following graphical representation.



This is the traditional way of representing PECOFF files, as a union of multiple structure types that are interlinked by defined rules. File starts with the DOS header at file offset zero which is also considered to be a valid executable image in DOS mode. This header is identified by the “MZ” signature. The default DOS header for the PE32 and PE32+ files only displays the message that the file cannot be executed in DOS mode. Present in the DOS header structure at offset 0x3C is the field named e_lfanew which holds a 32 bit pointer to the PE header offset.

PE header is identified with the signature “PE\0x00\0x00”. This header describes the file properties and it is parsed by the operating system loader during the file load process. This table starts with COFF header and is followed by the optional header and the section table data.

More details about the specific fields and the structure layout can be found in the Microsoft PECOFF documentation.

Malformations

File format malformations represent special case conditions that are introduced to the file layout and specific fields in order to achieve undesired behavior by the programs that are parsing it.

DOS & PE Header

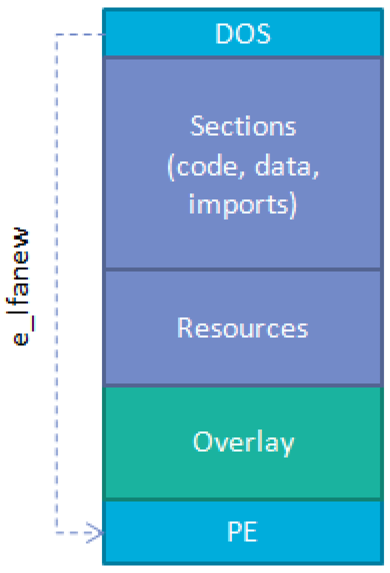
Every PE file starts with the DOS header. Along with the header signature the only other field that is used by the operating system loader is the value e_lfanew located at the offset 0x3C from the beginning of the file. That value indicates the offset of the PE header relative to the file start. The field itself is a 32 bit number which indicates that the starting offset can be anywhere in the first 4 GB of the file, provided that the address is aligned to a 4 byte boundary.

While the PE header usually follows the DOS header it might not be the case as that isn’t defined by the specification as a strict rule in respect to the file layout. Moving the location of the PE header leads to several specific malformation cases which have the potential of affecting security and reverse engineering tools.

Self-destructing PE header

Since PE header's location is indicated by the e_lfanew field it can be found at an arbitrary 4 byte aligned address in the file. One of the possible locations for the PE header is outside the memory mapped by the system loader. This is possible because during the loading process PE header is read from the disk and it is only committed to memory if it is physically located in the part of the file that needs to be available in memory. Therefore if the PE header is located in what is considered the overlay of the file it will not be committed in memory. The same effect can be achieved if the PE header has been placed in its own section whose raw size is zero.

This kind of malformation has the potential of breaking security and reverse engineering tools that parse the PE header from memory. Suggested workaround is to always parse the PE header from disk.

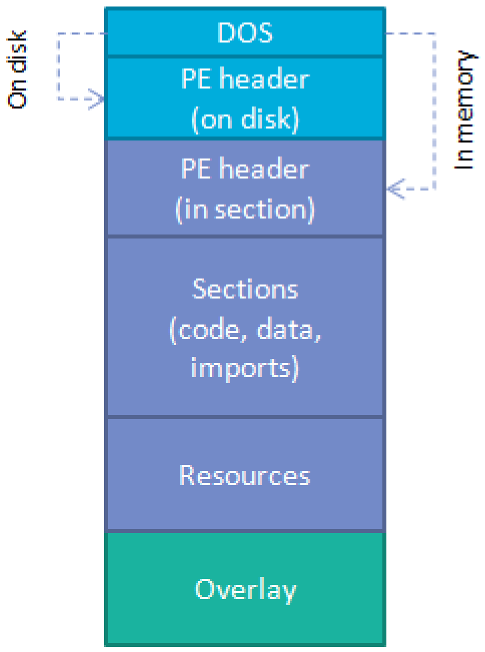


Dual PE header

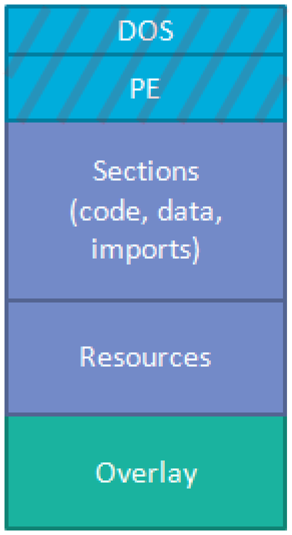
Because of the way PE file format is designed its sections have two arbitrary positions, one on disk and one in memory. Those locations are aligned to file and section alignment respectively. Since the location of the PE header is defined by the e_lfanew field it is possible to overlap the physical and virtual location so that they become two separate entities. One read from the disk during the file loading process and the second available in memory and parsed by the operating system loader on request by API calls.

To achieve this effect assume that the physical location of the PE header is at the address 0x1000. At the same virtual address we assign a new section which at its start has the new PE header which will be used by the loaded file. PE header read from disk is used to initialize the file loading process while the second PE header available in memory is used to process on request information such as resources and exports.

This kind of malformation has the potential of breaking security and reverse engineering tools that parse the PE header from memory.



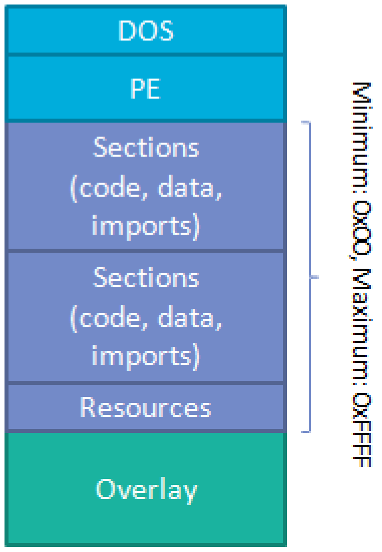
Writable PE header



By default the PE header has the read and execute attribute and if the DEP has been enabled the header becomes read only. Making the PE header writable is in some cases necessary for a specific PE header malformation to work. To achieve this one of the following two options are available:

a) If there’s a file with a normal single PE header, file values of fields FileAlignment and SectionAlignment can be set to the same value which must be lesser or equal to 0x200 but non zero. This will make the operating system loader apply the full access to the header.
b) If the PE header has been placed in its own section via the double PE header method, then that section can have write attributes enabled.

Section number limits


DOS
PE
Sections (code, data, imports)
Sections (code, data, imports)
Resources
Overlay
Minimum: 0x00, Maximum: 0xFFFF

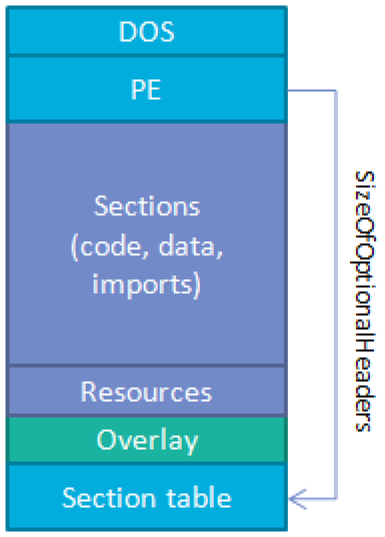
PE files have arbitrary section numbers; however it is assumed that the number of possible sections that a file can consist of is within a range from one to 96 as stated by the PECOFF documentation. Due to difference between the documentation and the implementation the actual limit on the number of sections varies depending on the operating system loader version. Most notably 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. This change has been introduced in Windows Vista and has propagated to later versions.

These limits can prove to be problematic for security and reverse engineering tools. If the SectionNumber is zero file addresses might not be properly converted as the entire file has been placed inside headers. In the other case tools might require too much memory allocated to handle all of the section data or they might have imposed lower limits for maximum section number which would prevent the file from being parsed correctly.

SizeOfOptionalHeaders

In the case of self destructing PE header e_lfanew has been used to position the header inside the overlay. The same logic can be applied to make only the section table nonexistent in memory. However this time it isn't e_lfanew that is moving the entire PE header to an arbitrary position in overlay, it is the SizeOfOptionalHeader which only moves the section table while keeping the rest of the header at the usual location at the start of the file. 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.

This kind of malformation has the potential of breaking security and reverse engineering tools that parse the PE header from memory as the section table might not be present in memory. Suggested workaround is to always parse the PE header from disk.

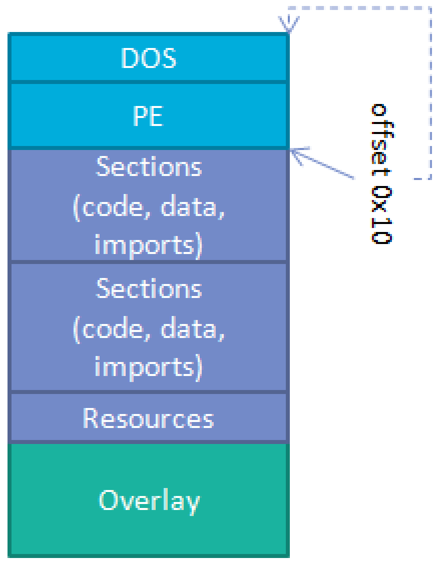


FileAlignment

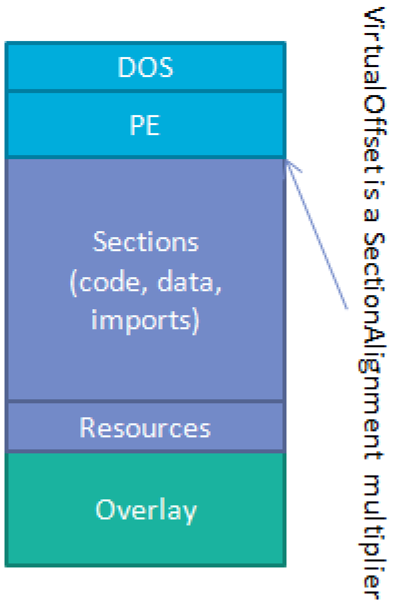
FileAlignment is the alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512 or 0x200. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment.

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. Because if the value is less than that, it is rounded up to it and if it's greater than that, then it is its own multiplier.

The usage of FileAlignment is demonstrated by the following figure. If the section physically starts at the offset 0x10 then the following formula is used to calculate its real start: **(file_offset / 0x200) * 0x200** which in this case results to 0x00. Therefore the real start for first section of this file is at the beginning of the file which overlaps with the first section and the DOS/PE header in this case.


DOS
PE
Sections (code, data, imports)
Sections (code, data, imports)
Resources
Overlay
offset 0x10

SectionAlignment

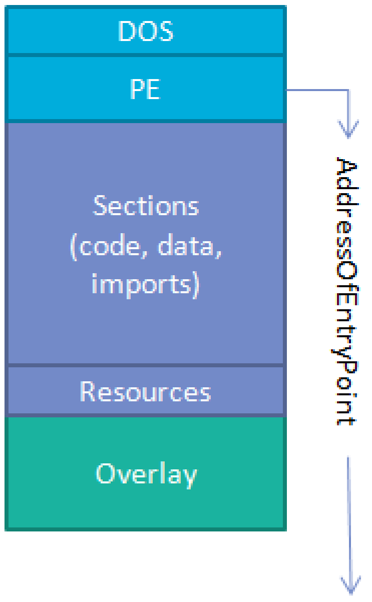

DOS
PE
Sections (code, data, imports)
Resources
Overlay
VirtualOffset is a SectionAlignment multiplier

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.

While every section must start at the 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.

Total size needed to commit all sections is set in the NtSizeOfImage variable. That value is also rounded up.

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.

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.

Entry point can be zero for any file and if that is the case the file execution starts in the DOS header executing the first two bytes which always translate to DEC EBP and POP EDX instructions. However this can be prevented by DEP if the header isn't made writable.

Additionally entry point can reside outside the file. Since the value of AddressOfEntryPoint is a relative virtual address any address can be put in its place. For example to have the file start its execution inside kernel32.ExitProcess one would need to enter that APIs address minus the ImageBase into the AddressOfEntryPoint field. For this malformation to work the module, in which the execution will take place, must be loaded on the static address. This can be achieved with a DLL that doesn't get rebased.

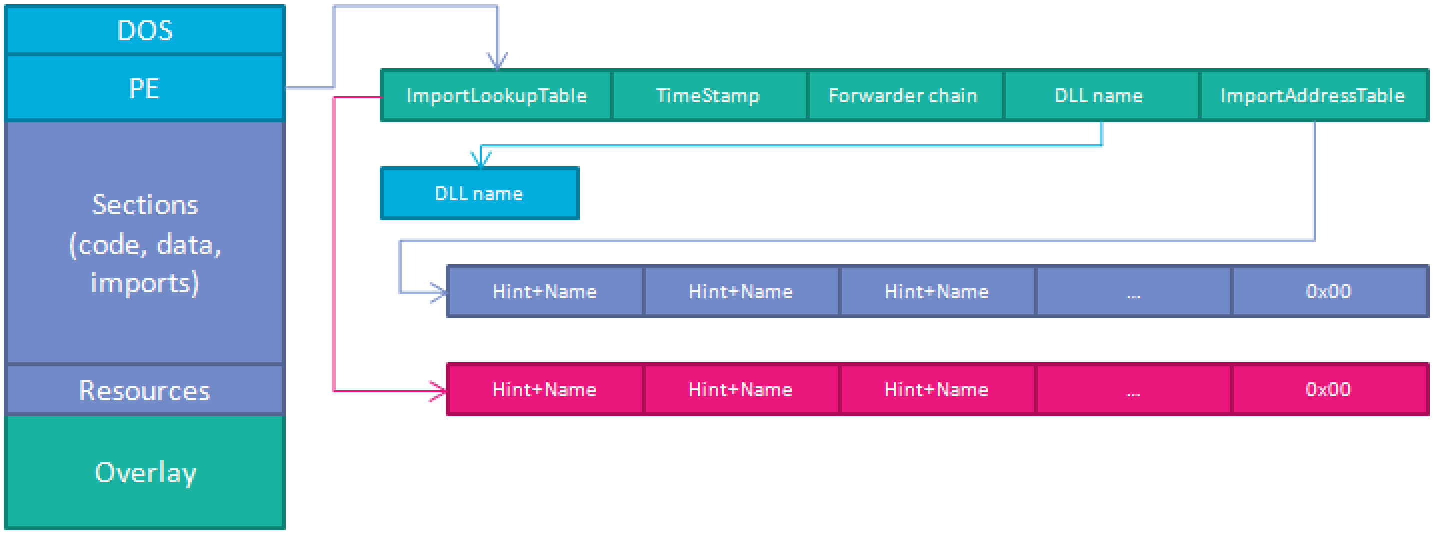
Optional header data directories

In addition to the DOS/PE header fields specific data directories can be a part of a malformation that has an impact on security solutions. Data in these directories can be invalid but the file still could be considered as valid from the operating system loaders perspective. Furthermore these data directories could also carry code payload.

Import table

Every PE file that imports symbols has an import table, however not every PE file imports symbols therefore not every PE file has an import table. Those files that do have it use it to load all DLL files necessary for the application to function correctly.

Import table consists of three parts, the import directory table, import address table and the import lookup table. They are linked together to provide the structure needed to link the application code with the appropriate DLL functions. Relationship between these tables can be presented with the following graph.



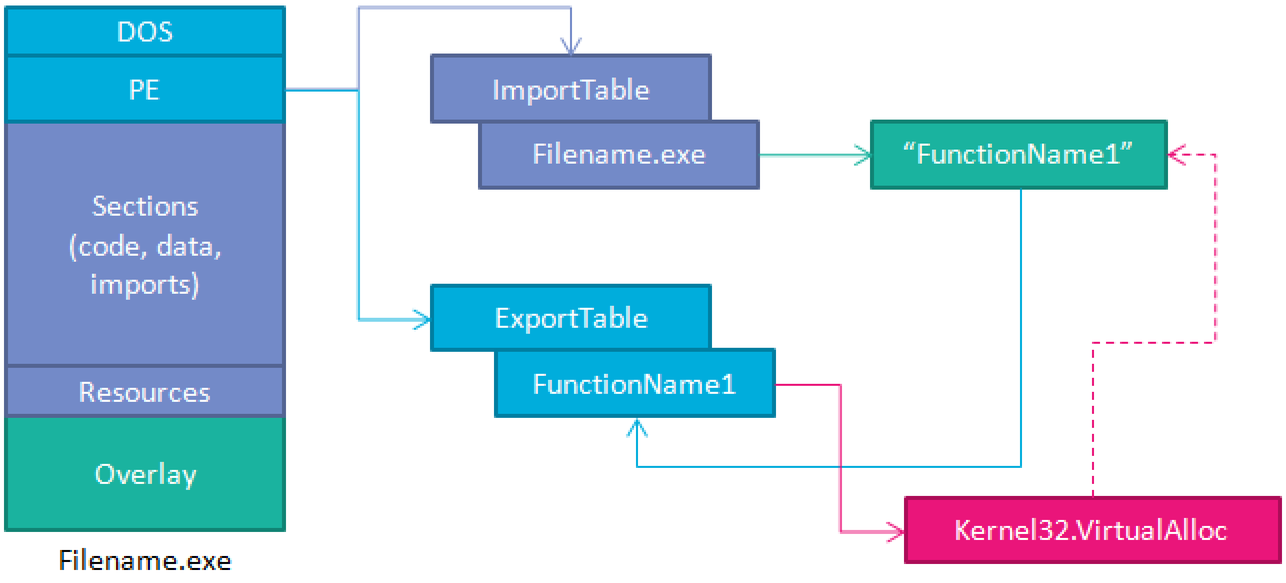
Here the import lookup table is a read only copy of the import address table and because of that its existence is optional. The import address table which will hold all the function pointers once the DLL has been loaded and because of that its existence is not optional. Both of those tables are a zero terminated array of API function name pointers or ordinal numbers.

Operating system loader will parse this table, load the necessary DLL files and fill the import address table with the function pointers. This happens for every entry inside the import directory table. However, if the import address table is empty or consists only of a single zero entry then that import directory will be skipped without loading the DLL associated with that import directory. Since the DLL file won't be loaded the DLL name is irrelevant and may in fact be a file that doesn't exist on the system. This can be used to omit dynamic analysis solutions such as emulators and sandboxes which could parse this information and decide to skip the file.

Import name table

Both import address and import lookup table entries point to import name table which consists of two values, hint and a zero terminated ASCII function name. It is implied by the PECOFF documentation that the import name should consist only of printable ASCII characters while that isn't the case. In fact any byte stream can be considered a valid import name as long as the stream is zero terminated. Furthermore these strings aren't limited in size and can be extremely long. This can be used by the files that import functions from their own export tables to cloak the name of such functions, and depending on the nature of string parsing in security and reverse engineering tools that could lead to undesirable effects.

Quite similar to this but even more dangerous is the possibility of imports being forwarded through the same file's export table. The following graph explains that concept.



The file is importing functions from its own export table and those functions are just forwarders to the real API. Since the forwarder mechanism has been designed in a way that the forwarder string consists of both DLL name and the function name or the ordinal number of the necessary function no import information is lost. Because of that the operating system loader can take that information to load the necessary DLL and locate the appropriate function inside it. This is an effective way of cloaking imports and circumventing security solutions which don't store the original name of the application on which the file using this method to hide imports becomes depended on.

Additionally it is possible to create an infinite parsing loop if the forwarder is pointing back to itself.

Import name hint

Both import address and import lookup table entries point to import name table which consists of two values, hint and a zero terminated ASCII function name. Hint is an index into the export name pointer table. A match is attempted first with this value. If it fails, a binary search is performed on the DLL's export name pointer table.

This behavior model enables even better import clocking because since there's no need for binary search and since the hint is used to check the export name first all exports can have the same name. This can additionally confuse security and reverse engineer tools as it might not be obvious on the first look which function is being used where. Couple with the fact that function names might not be printable ASCII characters this makes this PECOFF feature even more important from the analysis standpoint.

Import directory layout

The import information begins with the import directory table, which describes the remainder of the import information. The import directory table contains address information that is used to resolve fixup references to the entry points within a DLL image. The import directory table consists of an array of import directory entries, one entry for each DLL to which the image refers. The last directory entry is empty (filled with null values), which indicates the end of the directory table. Because of that it is possible that the last entry isn't present in the file but the file is still considered as valid with the correct import table. This can only occur if the last valid import directory is located at the physical but not the virtual end of the file. Achieving this is easy because the sections are virtually rounded up to multiplier of SectionAlignment but physically can have any size.

This kind of malformation has the potential of breaking security and reverse engineering tools that parse the PE import table from disk as it can be incomplete.

Complex file malformations

Complex file malformations involve modifications of multiple code and data directory fields in order to make the malformation work correctly.

Zero section PE file

Carefully manually crafted file is able to omit security and reverse engineering tools from parsing it correctly. However modifications done for this malformation still reside in the boundaries permitted by the operating system image loader, but not the PECOFF documentation. Following Portable executable fields are affected by malformation:

- PE32.FileAlignment
- PE32.SectionAlignment
- PE32.NumberOfSections
- PE32.SizeOfHeaders

Neither of fields by itself is responsible for malformation, it’s the combination of them all that make the effect work across all Windows NT systems.

Issue comes from the ability to move certain parts of PE header and data so that they overlap with other features of the file format. Several possible combinations of different values for selected fields leads to different possible implementations that result in the same or similar effect.

Effect produced by modifications introduced to the file is a section-less file that contains all code and data inside the PE header. Making such file involves solving numerous problems related to the file layout and enabling the actual execution of the code stored inside the read only header. As sections themselves are used as a base for address conversion calculations any program that relies on number of sections not being null in order to get the correct data location will not be able to resolve the file content. However file can’t simply be merged to fulfill all the requirements and still be considered as a valid PE image.

This must be performed by the following steps:

a) File must be expanded so that the relative virtual addresses and physical ones match each other. At this point file still has sections but their raw offset is equal to the virtual ones. This implies that the file size is equal to the value of NtSizeOfImage filed. This kind of file has the simplest address conversion possible as the relative virtual addresses are equal to physical ones. Having a file layout made like this is a very important step as the operating systems must be able to locate the correct data and execute code at the right location.

b) Once the section layout is linear sections can be “merged” by setting the section number to zero and erasing the section table from the PE header.

c) After section merging following PE fields must be corrected:
   a. SizeOfHeaders should be set to be a value equal or greater than NtSizeOfImage. This modification is necessary to force the operating system to load the entire file content in the memory reserved for PE file format headers.
   b. FileAligment set to value lesser or equal to 0x200 (excluding zero).
   c. SectionAlignment set to the same value as FileAlignment. To circumvent DEP we have to set these fields to the same value as that will make the PE header writable and able to execute the file normally.

Combination of all actions listed above leads to creation of files with no sections. That kind of file would contain all data and code inside the header. As a result of bad address conversion calculation it is possible that security and reverse engineering might be affected and not display all features of the malformated file.

File encryption via relocations

Carefully manually crafted file is able to achieve detection evasion is all cases because it is declared as damaged or not processed correctly by most affected products. However modifications done for this malformation still reside in the boundaries permitted by the operating system image loader, but not the PECOFF documentation. Following Portable executable fields are affected by malformation:

- PE32. OriginalEntryPoint
- PE32. ImageBase
- PE32.RelocationTable

Neither of fields by itself is responsible for evasion, it’s the combination of them all that make the vulnerability work across all Windows NT systems.

Starting from the AddressOfEntryPoint modification evasion is achieved. This field is set to NULL which makes the operating system loader execute the file from the first DOS header byte. When disassembled that code looks like this:

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

This serves as a redirection to the original entry point and it is just a part of the malformation. As the jump itself is relative it doesn't matter on which base address the program is loaded, correct EP will always be called. Stack is corrupted by the POP EDX instruction but it is irrelevant for purpose of the small testing program. It is also noteworthy that DEP would prevent execution of this kind of executable at this first step but entry point redirection is optional with little to no effect on the vulnerability itself.

Second part of the malformation is relocation of the file to base address 0x00010000. This base address has been chosen carefully because it is very important on which base address the program will be loaded. Since Windows uses ASLR to randomize image base loading we want to have it behave in the predictable fashion in order trigger the decryption process correctly. By having the file relocated to specific base address we can insure that it is always loaded there even thought it should be relocated by the system. To make sure that the image must be (but isn't effectively) relocated on each start we set the ImageBase value in the header to NULL or an address which resides in the kernel space (for example 0x80000000). This doesn't break the file even though it should by the PECOFF standard, and instead it comes with surprising side effects. And they are:

- PE32.ImageBase can only be set to NULL if the file has a relocation table, valid or not but present. This is surprising as NULL isn't the multiplier of 65k as stated in the PECOFF documentation requirements.
- Files are always allocated at the same place in memory, on base address 0x00010000. ASLR behaves predictably for such cases and always performs the same way by assigning the same base address to the image.

Third and final part of the vulnerability is the use of the relocation table to obfuscate data. Normally loader uses this data to relocate the file to new base address and since we are forcing the system to always relocate the data we can use the relocation table to have the system modify any part of portable executable file each time it starts. Therefore we can use the relocation table to effectively decrypt data and have it reliable enough to perform each time in predictable fashion. Most notably we can encrypt the import table of the file and have the loader decrypt it just before it needs to use it. That is how a simple encrypted "URLCownloadFileA" becomes "URLDownloadFileA" at the right time, just before the system uses it. It is only the complexity of the relocation table that indicates encryption / obfuscation severity.

Argumentatively relocation table can always be used to obfuscate code and data but one must insure that the file is always relocated in a predictable fashion for obfuscations to work correctly. Simple relocations cannot achieve this if the code and data are being randomly relocated in memory. It is only in this kind of described scenario in which the relocation table can be used to encrypt / obfuscate data.

Document revision history

Latest version: http://pecoff.reversinglabs.com

Revision 1.0 – Initial draft

Revision 1.1 – Changes based on input from Peter Ferrie

References

Microsoft Portable Executable and Common Object File Format Specification – version 8.2
"PE format as source of vulnerabilities" - Ivan Teblin [presentation from Caro 2009 or SAS 2010]
Doin' The Eagle Rock - Peter Ferrie, Virus Bulletin, March 2010, page 4-6
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

Copyright and legal notices